North-Holland

HIGH PERFORMANCE COMPUTING: Technology, Methods and Applications

ADVANCES IN PARALLEL COMPUTING

10

J.J. Dongarra, L. Grandinetti,
G.R. Joubert, J. Kowalik
Editors

HIGH PERFORMANCE COMPUTING: Technology, Methods and Applications

ADVANCES IN PARALLEL COMPUTING

VOLUME 10

Bookseries Editors:

Gerhard R. Joubert
(Managing Editor)
Aquariuslaan 60
5632 BD Eindhoven, The Netherlands

Udo Schendel
Institut für Mathematik I
Freie Universität Berlin
Germany

ELSEVIER
Amsterdam - Lausanne - New York - Oxford - Shannon - Tokyo

HIGH PERFORMANCE COMPUTING: Technology, Methods and Applications

Edited by

J.J. Dongarra
Knoxville, TN
U.S.A.

L. Grandinetti
Rende (Cosenza)
Italy

G.R. Joubert
Eindhoven
The Netherlands

J. Kowalik
Seattle, WA
U.S.A.

1995

ELSEVIER
Amsterdam - Lausanne - New York - Oxford - Shannon - Tokyo

ELSEVIER SCIENCE B.V.
Sara Burgerhartstraat 25
P.O. Box 211, 1000 AE Amsterdam, The Netherlands

ISBN: 0 444 82163 5

© 1995 Elsevier Science B.V. All rights reserved

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher, Elsevier Science B.V., Copyright & Permissions Department P.O. Box 521, 1000 AM Amsterdam, The Netherlands.

Special regulations for readers in the U.S.A. - This publication has been registered with the Copyright Clearance Center Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923. Information can be obtained from the CCC about conditions under which photocopies of parts of this publication may be made in the U.S.A. All other copyright questions, including photocopying outside of the U.S.A., should be referred to the publisher,Elsevier Science B.V.

No responsibility is assumed by the publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

This book is printed on acid-free paper.

Printed in The Netherlands.

Editorial Preface

The idea of editing this book was conceived at the Advanced Workshop "High Performance Computing: Technology and Applications," held in Cetraro, Italy on June 27-29, 1994, and hosted by the Department of Electronics, Informatics and Systems (DEIS) of the University of Calabria. The purpose of the Workshop was to discuss the current status of and trends in high performance computing, and to identify the most critical issues related to this field of computer science. The editors decided that the book would include the best Workshop papers and additional invited contributions by several outstanding specialists in the field who had no opportunity to attend the workshop. The result of the project is this volume containing twenty-five papers arranged in the four major sections: General, Hardware Systems, Programming and Software, and Applications.

We define High Performance Computing (HPC) as an integrated computing environment that includes hardware, software, algorithms, programming tools and visualization, and enables solving compute intensive problems. HPC has contributed very significantly to solving large-scale problems in many areas of science and engineering, such as: computational fluid dynamics, structures, electromagnetics, weather forecast and climate evolution, etc. Newly emerging areas of the HPC applications include: medical sciences, transportation, financial operations, advanced human-computer interface such as virtual reality, collaborative computing and computational chemistry. The Application section of this volume contains papers that present practical uses of HPC in these emerging areas of application.

The book is recommended for technical, as well as, management oriented individuals who have a need for or would like to know more about the cutting edge HPC technology that has two major thrusts: achieving the maximal absolute computing performance, and providing the most cost effective and affordable computing for science, industry and business.

We thank all contributors for their cooperation and Ms. Eefke Smit from Elsevier for her support and advice which have made this book possible.

Editors
April, 1995

This Page Intentionally Left Blank

CONTENTS

Editorial Preface v

I. GENERAL 1

J. Kowalik and L. Grandinetti
Preface. About High Performance Computing and This Book 3

J. Dongarra
Heterogeneous Network-Based Computing Systems 5

O. A. McBryan
Data Communication and High Performance Computing 17

J. Kowalik
Scalability of Parallel Systems: Efficiency Versus Execution Time 39

J. Sobolewski
Architectural Features Affecting Total Scalability of Parallel Systems 49

II. HARDWARE SYSTEMS 61

J-L. Baer
Trends in High-Performance Computer Architecture 63

F. Baetke
The CONVEX Exemplar SPP1000 and SPP1200 - New Scalable Parallel Systems with a Virtual Shared Memory Architecture 81

A. Nodomi et al
Hardware Performance of the VPP500 Parallel Supercomputer 103

L. Davis
The Balance of Power - A Brief History of Cray Research Hardware Architectures 121

T. Agerwala et al
Designing a Scalable Parallel System: the IBM SP2 133

III. PROGRAMMING AND SOFTWARE 157

M. Pantano and H. Zima
An Integrated Environment for the Support of Automatic Compilation 159

J. Choi et al
The Design and Implementation of the Reduction Routines in ScaLAPACK 177

W. Gentzsch
Software Tools for Parallel Computers and Workstation Clusters 203

A. Grimshaw
Mentat-Applying the Object-Oriented Paradigm to Parallel Processing 223

K. K. Yue and D. J. Lilja
Parallel Loop Scheduling for High Performance Computers 243

W. Gropp and E. Lusk
The MPI Message-Passing Interface Standard: Overview and Status 265

J. Schaeffer and D. Szafron
Software Engineering Considerations in the Construction of Parallel Programs 271

IV. APPLICATIONS 291

C. Diegert
A Role for High Performance Computing in Surgery 293

D. P. Bertsekas et al
Parallel Shortest Paths Methods for Globally Optimal Trajectories 303

A. Joubert
Financial Applications and HPF 317

H. Sowizral
Virtual Reality and High Performance Computing 337

K. Sharma
Collaborative Computing 351

R. Asenjo et al
Sparse Block and Cyclic Data Distributions for Matrix Computations 359

G. Falkemeier and G. Joubert
Parallel Image Compression with JPEG for Multimedia Applications 379

M. F. Guest et al
High Performance Computational Chemistry; NWChem and Fully Distributed Parallel Applications 395

I. GENERAL

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Preface

About High Performance Computing and This Book

Janusz S. Kowalik

Boeing Computer Services, P.O. Box 24346, MS 7L-44, Seattle, WA 98124-0346, USA
and
Department of Computer Science and Engineering, University of Washington, FR-35, Seattle, WA 98195

Lucio Grandinetti

Department of Electronics, Informatics and Systems, University of Calabria, DEIS, 87036 RENDE (COSENZA), Italy

High Performance Computing (HPC), synonymous with Supercomputing, originated in the late 1970s and has spread dramatically in the current decade. Initially limited to applications in defense industry and several high technology industries, HPC is now entering other sectors of business, industry and science. In some industries the yearly growth of supercomputer cycles reaches over 100%.

Computer manufacturing industry has responded to this demand by offering a variety of computer architectures ranging from traditional vector processors to assemblies of RISC processors.

Successful manufacturers have understood the importance of software for high performance computers and their cost effectiveness. They have moved carefully to new architectures making sure that the cost of porting the existing software or developing new parallel software is economically acceptable. Less successful manufacturers either have offered exotic architectures requiring software discontinuity or continued to produce very expensive custom made supercomputers.

From the technology point of view, the 1990s have brought significant, almost revolutionary developments. One of the most visible developments is the architectural advancement in the processor technology. In few years the new RISC processors have reached the performance levels of vector supercomputers. RISC-based computers have incorporated the ability to process concurrently several instructions using multiple pipelined functional units. This combined with fast clocks, large data caches and out-of-order instruction processing have made RISC processors exceptionally suitable as the building blocks for cost effective new generation parallel computers.

We have also seen a significant progress in the software area. The software tool called Parallel Virtual Machine (PVM) allows a heterogeneous collection of workstations and supercomputers to function as a single high performance parallel machine. Network computing has been accepted by the global computing community and used successfully for solving large scale problems in science, industry and business.

Of similar importance is a standard portable message-passing library called Message Passing Interface (MPI). The library contains routines to be called from Fortran or C programs. Its adoption by both users and implementers will provide the parallel programming community with the portability and features needed for creating parallel applications.

The emergency of parallel processing and its dominant programming style, called the data parallelism, has influenced the development of new programming language standards. One of them is the High Performance Fortran (HPF) standard that specifies a set of additions to Fortran. These programming constructs help the compiler with the data partitioning process. HPF is an example of the evolutionary transition step towards parallel programming. It facilitates developing parallel programs and at the same time it maintains the similarity to the classic sequential programming style.

As mentioned earlier High Performance Computing is entering a new phase as a practical, cost effective tool. Today HPC is used in a wide variety of applications outside the classic fields such as structural analysis, oil exploration, computational fluid dynamics, atmospheric sciences and defense applications. The new applications include: medicine, computational chemistry, financial applications, transportation, virtual reality and many others.

We have designed this volume having in mind the HPC developments and trends described above. The book begins with the General section that includes the subjects of network computing, and scalability of parallel systems. The second section is on Hardware Systems - it presents descriptions of several state-of-the-art parallel computers: CONVEX EXEMPLAR SPP1000, FUJITSU VPP500, CRAY Computers and IBM SP2. It also contains a paper on Trends in High Performance Architecture. The Programming and Software section contains seven contributions covering wide spectrum of software issues: automatic compilation, linear algebra libraries, workstation cluster software tools, object-oriented paradigms for parallel processing, loop scheduling, MPI and software engineering.

In the last section, Applications, the book presents eight areas of high performance computing applications. They represent a sample of newly emerging applications in: surgery, finance, virtual reality, collaborative computing, image compression, and computational chemistry. Additional two papers present new computing techniques for shortest path methods for trajectories and sparse matrices.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Heterogeneous Network-Based Concurrent Computing Systems

Jack J. Dongarra [a] *

[a] Oak Ridge National Laboratory and University of Tennessee

1. Introduction

Parallel processing, the method of having many small tasks solve one large problem, has emerged as a key enabling technology in modern computing. The past several years have witnessed an ever-increasing acceptance and adoption of parallel processing, both for high-performance scientific computing and for more "general-purpose" applications, was a result of the demand for higher performance, lower cost, and sustained productivity. The acceptance has been facilitated by two major developments: massively parallel processors (MPPs) and the widespread use of distributed computing.

MPPs are now the most powerful computers in the world. These machines combine a few hundred to a few thousand CPUs in a single large cabinet connected to hundreds of gigabytes of memory. MPPs offer enormous computational power and are used to solve computational Grand Challenge problems such as global climate modeling and drug design. As simulations become more realistic, the computational power required to produce them grows rapidly. Thus, researchers on the cutting edge turn to MPPs and parallel processing in order to get the most computational power possible.

The second major development affecting scientific problem solving is *distributed computing*. Distributed computing is a process whereby a set of computers connected by a network are used collectively to solve a single large problem. As more and more organizations have high-speed local area networks interconnecting many general-purpose workstations, the combined computational resources may exceed the power of a single high-performance computer. In some cases, several MPPs have been combined using distributed computing to produce unequaled computational power.

The most important factor in distributed computing is cost. Large MPPs typically cost more than $10 million. In contrast, users see very little cost in running their problems on a local set of existing computers. It is uncommon for distributed-computing users to realize the raw computational power of a large MPP, but they are able to solve problems several times larger than they could using one of their local computers.

Common between distributed computing and MPP is the notion of message passing.

*This work was supported in part by DOE under contract number DE-AC05-84OR21400 and by National Science Foundation Grant number ASC-ASC-9005933.

In all parallel processing, data must be exchanged between cooperating tasks. Several paradigms have been tried including shared memory, parallelizing compilers, and message passing. The message-passing model has become the paradigm of choice, from the perspective of the number and variety of multiprocessors that support it, as well as in terms of applications, languages, and software systems that use it.

The Parallel Virtual Machine (PVM) [5] system described in this chapter uses the message-passing model to allow programmers to exploit distributed computing across a wide variety of computer types, including MPPs. A key concept in PVM is that it makes a collection of computers appear as one large *virtual* machine, hence its name.

2. Heterogeneous Network Computing

In an MPP, every processor is exactly like every other in capability, resources, software, and communication speed. Not so on a network. The computers available on a network may be made by different vendors or have different compilers. Indeed, when a programmer wishes to exploit a collection of networked computers, he may have to contend with several different types of heterogeneity:

- architecture,
- data format,
- computational speed,
- machine load, and
- network load.

The set of computers available can include a wide range of architecture types such as 386/486 PC class machines, high-performance workstations, shared-memory multiprocessors, vector supercomputers, and even large MPPs. Each architecture type has its own optimal programming method. In addition, a user can be faced with a hierarchy of programming decisions. The parallel virtual machine may itself be composed of parallel computers. Even when the architectures are only serial workstations, there is still the problem of incompatible binary formats and the need to compile a parallel task on each different machine.

Data formats on different computers are often incompatible. This incompatibility is an important point in distributed computing because data sent from one computer may be unreadable on the receiving computer. Message-passing packages developed for heterogeneous environments must make sure all the computers understand the exchanged data. Unfortunately, the early message-passing systems developed for specific MPPs are not amenable to distributed computing because they do not include enough information in the message to encode or decode it for any other computer.

Even if the set of computers are all workstations with the same data format, there is still heterogeneity due to different computational speeds. As an simple example, consider the problem of running parallel tasks on a virtual machine that is composed of one supercomputer and one workstation. The programmer must be careful that the supercomputer doesn't sit idle waiting for the next data from the workstation before continuing. The problem of computational speeds can be very subtle. The virtual machine can be composed of a set of identical workstations. But since networked computers can have several other users on them running a variety of jobs, the machine load can vary dramatically. The result is that the effective computational power across identical workstations can vary by an order of magnitude.

Like machine load, the time it takes to send a message over the network can vary depending on the network load imposed by all the other network users, who may not even be using any of the computers in the virtual machine. This sending time becomes important when a task is sitting idle waiting for a message, and it is even more important when the parallel algorithm is sensitive to message arrival time. Thus, in distributed computing, heterogeneity can appear dynamically in even simple setups.

Despite these numerous difficulties caused by heterogeneity, distributed computing offers many advantages:

- By using existing hardware, the cost of this computing can be very low.
- Performance can be optimized by assigning each individual task to the most appropriate architecture.
- One can exploit the heterogeneous nature of a computation. Heterogeneous network computing is not just a local area network connecting workstations together. For example, it provides access to different data bases or to special processors for those parts of an application that can run only on a certain platform.
- The virtual computer resources can grow in stages and take advantage of the latest computational and network technologies.
- Program development can be enhanced by using a familiar environment. Programmers can use editors, compilers, and debuggers that are available on individual machines.
- The individual computers and workstations are usually stable, and substantial expertise in their use is readily available.
- User-level or program-level fault tolerance can be implemented with little effort either in the application or in the underlying operating system.
- Distributed computing can facilitate collaborative work.

All these factors translate into reduced development and debugging time, reduced contention for resources, reduced costs, and possibly more effective implementations of an application. It is these benefits that PVM seeks to exploit. From the beginning, the PVM software package was designed to make programming for a heterogeneous collection of machines straightforward.

3. Trends in Distributed Computing

Stand-alone workstations delivering several tens of millions of operations per second are commonplace, and continuing increases in power are predicted. When these computer systems are interconnected by an appropriate high-speed network, their combined computational power can be applied to solve a variety of computationally intensive applications. Indeed, network computing may even provide supercomputer-level computational power. Further, under the right circumstances, the network-based approach can be effective in coupling several similar multiprocessors, resulting in a configuration that might be economically and technically difficult to achieve with supercomputer hardware.

To be effective, distributed computing requires high communication speeds. In the past fifteen years or so, network speeds have increased by several orders of magnitude (see Figure 1).

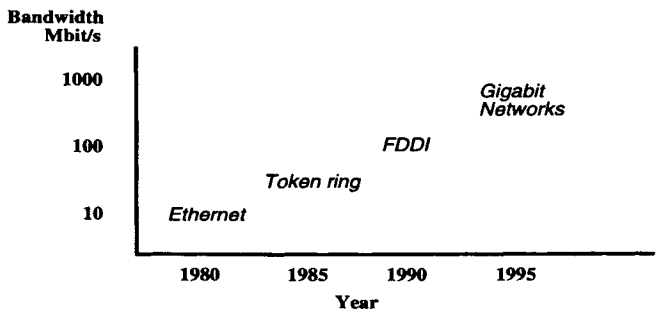


Figure 1. Networking speeds

Among the most notable advances in computer networking technology are the following:

- Ethernet – the name given to the popular local area packet-switched network technology invented by Xerox PARC. The Ethernet is a 10 Mbit/s broadcast bus technology with distributed access control.
- FDDI – the Fiber Distributed Data Interface. FDDI is a 100-Mbit/sec token-passing ring that uses optical fiber for transmission between stations and has dual counter-rotating rings to provide redundant data paths for reliability.

- HiPPI – the high-performance parallel interface. HiPPI is a copper-based data communications standard capable of transferring data at 800 Mbit/sec over 32 parallel lines or 1.6 Gbit/sec over 64 parallel lines. Most commercially available high-performance computers offer a HIPPI interface. It is a point-to-point channel that does not support multidrop configurations.

- SONET – Synchronous Optical Network. SONET is a series of optical signals that are multiples of a basic signal rate of 51.84 Mbit/sec called OC-1. The OC-3 (155.52 Mbit/sec) and OC-12 (622.08 Mbit/sec) have been designated as the customer access rates in future B-ISDN networks, and signal rates of OC-192 (9.952 Gbit/sec) are defined.

- ATM – Asynchronous Transfer Mode. ATM is the technique for transport, multiplexing, and switching that provides a high degree of flexibility required by B-ISDN. ATM is a connection-oriented protocol employing fixed-size packets with a 5-byte header and 48 bytes of information.

These advances in high-speed networking promise high throughput with low latency and make it possible to utilize distributed computing for years to come. Consequently, increasing numbers of universities, government and industrial laboratories, and financial firms are turning to distributed computing to solve their computational problems. One of the objectives of systems like PVM are to enable these institutions to use distributed computing *efficiently.*

4. PVM Overview

The PVM software provides a unified framework within which parallel programs can be developed in an efficient and straightforward manner using existing hardware. PVM enables a collection of heterogeneous computer systems to be viewed as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across a network of incompatible computer architectures.

The PVM computing model is simple yet very general, and accommodates a wide variety of application program structures. The programming interface is deliberately straightforward, thus permitting simple program structures to be implemented in an intuitive manner. The user writes his application as a collection of cooperating *tasks.* Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM message-passing primitives are oriented towards heterogeneous operation, involving strongly typed constructs for buffering and transmission. Communication constructs include those for sending and receiving data structures as well as high-level primitives such as broadcast, barrier synchronization, and global sum.

PVM tasks may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, any task in existence may start or stop other tasks or add or delete computers from the virtual machine. Any process may communicate and/or synchronize with any other. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.

Owing to its ubiquitous nature (specifically, the virtual machine concept) and also because of its simple but complete programming interface, the PVM system has gained widespread acceptance in the high-performance scientific computing community.

5. Other Packages

Several research groups have developed software packages that like PVM assist programmers in using distributed computing. Among the most well known efforts are P4 [1], Express [3], MPI [4], and Linda [2]. Various other systems with similar capabilities are also in existence; a reasonably comprehensive listing may be found in [7].

5.1. The p4 System

P4 [1] is a library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines. The p4 system supports both the shared-memory model (based on monitors) and the distributed-memory model (using message-passing). For the shared-memory model of parallel computation, p4 provides a set of useful monitors as well as a set of primitives from which monitors can be constructed. For the distributed-memory model, p4 provides typed send and receive operations and creation of processes according to a text file describing group and process structure.

Process management in the p4 system is based on a configuration file that specifies the host pool, the object file to be executed on each machine, the number of processes to be started on each host (intended primarily for multiprocessor systems), and other auxiliary information. An example of a configuration file is

```
# start one slave on each of sun2 and sun3
local 0
sun2  1  /home/mylogin/p4pgms/sr_test
sun3  1  /home/mylogin/p4pgms/sr_test
```

Two issues are noteworthy in regard to the process management mechanism in p4. First, there is the notion a "master" process and "slave" processes, and multilevel hierarchies may be formed to implement what is termed a *cluster* model of computation. Second, the primary mode of process creation is static, via the configuration file; dynamic process creation is possible only by a statically created process that must invoke a special o4 function that spawns a new process on the local machine. Despite these restrictions, a variety of application paradigms may be implemented in the p4 system in a fairly straightforward manner.

Message passing in the p4 system is achieved through the use of traditional **send** and **recv** primitives, parameterized almost exactly as other message-passing systems. Several variants are provided for semantics, such as heterogeneous exchange and blocking or nonblocking transfer. A significant proportion of the burden of buffer allocation and management, however, is left to the user. Apart from basic message passing, p4 also offers a variety of global operations, including broadcast, global maxima and minima, and barrier synchronization.

5.2. Express

In contrast to the other parallel processing systems described in this section, Express [3] toolkit is a collection of tools that individually address various aspects of concurrent computation. The toolkit is developed and marketed commercially by ParaSoft Corporation, a company that was started by some members of the Caltech concurrent computation project.

The philosophy behind computing with Express is based on beginning with a sequential version of an application and following a recommended development life cycle culminating in a parallel version that is tuned for optimality. Typical development cycles begin with the use of VTOOL, a graphical program that allows the progress of sequential algorithms to be displayed in a dynamic manner. Updates and references to individual data structures can be displayed to explicitly demonstrate algorithm structure and provide the detailed knowledge necessary for parallelization. Related to this program is FTOOL, which provides in-depth analysis of a program including variable use analysis, flow structure, and feedback regarding potential parallelization. FTOOL operates on both sequential and parallel versions of an application. A third tool called ASPAR is then used; this is an automated parallelizer that converts sequential C and Fortran programs for parallel or distributed execution using the Express programming models.

The core of the Express system is a set of libraries for communication, I/O, and parallel graphics. The communication primitives are akin to those found in other message-passing systems and include a variety of global operations and data distribution primitives. Extended I/O routines enable parallel input and output, and a similar set of routines are provided for graphical displays from multiple concurrent processes. Express also contains the NDB tool, a parallel debugger that uses commands based on the popular "dbx" interface.

5.3. MPI

The Message Passing Interface (MPI) [4] standard, whose specification was completed in April 1994, is the outcome of a community effort to try to define both the syntax and semantics of a core of message-passing library routines that would be useful to a wide range of users and efficiently implementable on a wide range of MPPs. The main advantage of establishing a message-passing standard is portability. One of the goals of developing MPI is to provide MPP vendors with a clearly defined base set of routines that they can implement efficiently or, in some cases, provide hardware support for, thereby

enhancing scalability.

MPI is not intended to be a complete and self-contained software infrastructure that can be used for distributed computing. MPI does not include necessities such as process management (the ability to start tasks), (virtual) machine configuration, and support for input and output. As a result, it is anticipated that MPI will be realized as a communications interface layer that will be built upon native facilities of the underlying hardware platform, with the exception of certain data transfer operations that might be implemented at a level close to hardware. This scenario permits the provision of PVM's being ported to MPI to exploit any communication performance a vendor supplies.

5.4. The Linda System

Linda [2] is a concurrent programming model that has evolved from a Yale University research project. The primary concept in Linda is that of a "tuple-space", an abstraction via which cooperating processes communicate. This central theme of Linda has been proposed as an alternative paradigm to the two traditional methods of parallel processing: that based on shared memory, and that based on message passing. The tuple-space concept is essentially an abstraction of distributed shared memory, with one important difference (tuple-spaces are associative), and several minor distinctions (destructive and nondestructive reads and different coherency semantics are possible). Applications use the Linda model by embedding explicitly, within cooperating sequential programs, constructs that manipulate (insert/retrieve tuples) the tuple space.

From the application point of view Linda is a set of programming language extensions for facilitating parallel programming. It provides a shared-memory abstraction for process communication without requiring the underlying hardware to physically share memory.

The Linda system usually refers to a specific implementation of software that supports the Linda programming model. System software is provided that establishes and maintains tuple spaces and is used in conjunction with libraries that appropriately interpret and execute Linda primitives. Depending on the environment (shared-memory multiprocessors, message-passing parallel computers, networks of workstations, etc.), the tuple space mechanism is implemented using different techniques and with varying degrees of efficiency. Recently, a new *system* scheme has been proposed, at least nominally related to the Linda project. This scheme, termed "Pirhana" [6], proposes a proactive approach to concurrent computing: computational resources (viewed as active agents) seize computational tasks from a well-known location based on availability and suitability. This scheme may be implemented on multiple platforms and manifested as a "Pirhana system" or "Linda-Pirhana system."

6. The PVM System

PVM (Parallel Virtual Machine) is a byproduct of an ongoing heterogeneous network computing research project involving the authors and their institutions. The general goals of this project are to investigate issues in, and develop solutions for, heterogeneous

concurrent computing. PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to to enable such a collection of computers to be used cooperatively for concurrent or parallel computation. Detailed descriptions and discussions of the concepts, logistics, and methodologies involved in this network-based computing process are contained in the remainder of the book. Briefly, the principles upon which PVM is based include the following:

- User-configured host pool: The application's computational tasks execute on a set of machines that are selected by the user for a given run of the PVM program. Both single-CPU machines and hardware multiprocessors (including shared-memory and distributed-memory computers) may be part of the host pool. The host pool may be altered by adding and deleting machines during operation (an important feature for fault tolerance).

- Translucent access to hardware: Application programs either may view the hardware environment as an attributeless collection of virtual processing elements or may choose to exploit the capabilities of specific machines in the host pool by positioning certain computational tasks on the most appropriate computers.

- Process-based computation: The unit of parallelism in PVM is a task (often but not always a Unix process), an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor.

- Explicit message-passing model: Collections of computational tasks, each performing a part of an application's workload using data-, functional-, or hybrid decomposition, cooperate by explicitly sending and receiving messages to one another. Message size is limited only by the amount of available memory.

- Heterogeneity support: The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one datatype to be exchanged between machines having different data representations.

- Multiprocessor support: PVM uses the native message-passing facilities on multiprocessors to take advantage of the underlying hardware. Vendors often supply their own optimized PVM for their systems, which can still communicate with the public PVM version.

The PVM system is composed of two parts. The first part is a daemon, called *pvmd3* and sometimes abbreviated *pvmd*, that resides on all the computers making up the virtual machine. (An example of a daemon program is the mail program that runs in the

background and handles all the incoming and outgoing electronic mail on a computer.) Pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wishes to run a PVM application, he first creates a virtual machine by starting up PVM. The PVM application can then be started from a Unix prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.

The second part of the system is a library of PVM interface routines. It contains a functionally complete repertoire of primitives that are needed for cooperation between tasks of an application. This library contains user-callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine.

The PVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload. Sometimes an application is parallelized along its functions; that is, each task performs a different function, for example, input, problem setup, solution, output, and display. This process is often called functional parallelism. A more common method of parallelizing an application is called data parallelism. In this method all the tasks are the same, but each one only knows and solves a small part of the data. This is also referred to as the SPMD (single-program multiple-data) model of computing. PVM supports either or a mixture of these methods. Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case.

The PVM system currently supports C, C++, and Fortran languages. This set of language interfaces have been included based on the observation that the predominant majority of target applications are written in C and Fortran, with an emerging trend in experimenting with object-based languages and methodologies.

The C and C++ language bindings for the PVM user interface library are implemented as functions, following the general conventions used by most C systems, including Unix-like operating systems. To elaborate, function arguments are a combination of value parameters and pointers as appropriate, and function result values indicate the outcome of the call. In addition, macro definitions are used for system constants, and global variables such as `errno` and `pvm_errno` are the mechanism for discriminating between multiple possible outcomes. Application programs written in C and C++ access PVM library functions by linking against an archival library (`libpvm3.a`) that is part of the standard distribution.

Fortran language bindings are implemented as subroutines rather than as functions. This approach was taken because some compilers on the supported architectures would not reliably interface Fortran functions with C functions. One immediate implication of this is that an additional argument is introduced into each PVM library call for status results to be returned to the invoking program. Also, library routines for the placement and retrieval of typed data in message buffers are unified, with an additional parameter indicating the datatype. Apart from these differences (and the standard naming prefixes — *pvm_* for C, and *pvmf* for Fortran), a one-to-one correspondence exists between the two

language bindings. Fortran interfaces to PVM are implemented as library stubs that in turn invoke the corresponding C routines, after casting and/or dereferencing arguments as appropriate. Thus, Fortran applications are required to link against the stubs library (`libfpvm3.a`) as well as the C library.

All PVM tasks are identified by an integer *task identifier* (TID) . Messages are sent to and received from tids. Since tids must be unique across the entire virtual machine, they are supplied by the local pvmd and are not user chosen. Although PVM encodes information into each TID the user is expected to treat the tids as opaque integer identifiers. PVM contains several routines that return TID values so that the user application can identify other tasks in the system.

There are applications where it is natural to think of a *group of tasks* . And there are cases where a user would like to identify his tasks by the numbers $0 - (p - 1)$, where p is the number of tasks. PVM includes the concept of user named groups. When a task joins a group, it is assigned a unique "instance" number in that group. Instance numbers start at 0 and count up. In keeping with the PVM philosophy, the group functions are designed to be very general and transparent to the user. For example, any PVM task can join or leave any group at any time without having to inform any other task in the affected groups. Also, groups can overlap, and tasks can broadcast messages to groups of which they are not a member. To use any of the group functions, a program must be linked with `libgpvm3.a`.

The general paradigm for application programming with PVM is as follows. A user writes one or more sequential programs in C, C++, or Fortran 77 that contain embedded calls to the PVM library. Each program corresponds to a task making up the application. These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from machines in the host pool. To execute an application, a user typically starts one copy of one task (usually the "master" or "initiating" task) by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and exchange messages with each other to solve the problem. Note that while the above is a typical scenario, as many tasks as appropriate may be started manually. As mentioned earlier, tasks interact through explicit message passing, identifying each other with a system-assigned, opaque TID.

7. Conclusions

It is important to realize that message passing systems, such as PVM, are not merely a software framework for network-based concurrent computing; they are an integrated methodology for concurrent, distributed, and parallel processing, and more importantly, it is an interface definition for portable application development. From the portability point of view, message-passing application implemented in one of these standards may be migrated not just from one machine (parallel or serial) to another but across different

collections of machines. Given that these standards are implemented so that they can operate within a parallel machine, across networks, and combinations thereof, significant generality and flexibility exists, and it is hard to imagine a computing environment where such a model would *not* be appropriate. From the point of view of performance, many of the systems deliver a significant proportion (of the order of 80 – 90%) of the capacity available from the underlying hardware, operating system, network, and protocols – and we expect to retain this characteristic as network and CPU speeds increase, and as protocol and OS software becomes more efficient. It should be pointed out, while on this topic, that in measuring the worth of systems such as PVM, comparisons of clusters vs. MPP's are inappropriate; rather, a meaningful metric is the value added by a system such as PVM to a given hardware environment, whether this is a multiprocessor or a collection of workstations. In terms of functionality, the PVM system currently supports an adequate suite of features, and with the integration of extensions described earlier, will be in a position to cater to a much larger realm of distributed and concurrent applications. Heterogeneous network-based concurrent computing systems like PVM are therefore likely to remain viable technologies for concurrent and distributed computing.

REFERENCES

1. R. Butler and E. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. Technical Report Preprint MCS-P362-0493, Argonne National Laboratory, Argonne, IL, 1993.
2. Nicholas Carriero and David Gelernter. LINDA in context. *Communications of the ACM*, 32(4):444–458, April 1989.
3. J. Flower, A. Kolawa, and S. Bharadwaj. The Express way to distributed processing. *Supercomputing Review*, pages 54–55, May 1991.
4. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
5. G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
6. David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *1992 International Conference on Supercomputing*, pages 417–427. ACM, ACM Press, July 1992.
7. Louis Turcotte. A survey of software environments for exploiting networked computing resources. Draft report, Mississippi State University, Jackson, Mississippi, January 1993.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Data Communication and High Performance Computing

Oliver A. McBryan†
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430, U. S. A.

1. Overview

The so called Grand Challenge problems are those scientific and engineering problems which, while not solvable with today's computers, are believed likely to yield to substantially more powerful supercomputers. Examples of such problems include detailed weather and climate modeling, turbulence simulation, aerodynamic design, and molecular modeling. It is generally believed that computers capable of delivering in excess of one teraflops (10^{12} floating point operations per second) will be needed for such solutions.

Rapid advances in microprocessor performance leave little doubt that it will be feasible to build Tflops (teraflops) supercomputers soon, perhaps within two years. As an example, Digital Equipment Corporation is already selling 600 Mflops (megaflops) processors for its Sable workstations, and commodity Gflops (gigaflops) processors are expected to be available within 1995. A 1000 node massively parallel processor (MPP) system constructed from such processors would in principle provide a Tflops capability. A main point of this paper is to argue that providing Tflops computing power is not sufficient for solving many Grand Challenge (GC) problems. Most GC problems involve major requirements for data communication in addition to processing power. There are of course some exceptions, such as Quantum Chromodynamics, where the main result is just a few numbers, and where there is also minimal input data required.

Data communication needs of many GC computations provide a discouraging view of our ability to effect such computations in the immediate future. Data communication takes several forms. For example, data must be transferred between each processor and its local memory (or a common shared memory), between the memories of different processors, and between processors and mass archival facilities, often at several levels. Data communication from processors to remote workstations may be essential for visualization. Also the

† Research supported in part by NSF Grand Challenges Application Group grant ASC-9217394 and by NASA HPCC Group grant NAG5-2218.

final archive sites for output data may be remote from the center where a computation is performed. In this paper we will demonstrate that raw processing power alone is not the primary limiting factor for solving certain important Grand Challenge problems, and we identify data output, as opposed to other forms of data communication, as the most severe limiting factor in many cases at the present time.

We will illustrate the critical role of data output by describing case studies of two HPC GC applications we are involved in. The first GC application, Turbulence Simulation, is described briefly in section 2, and then analyzed in detail in section 3. Section 4 deals with the second GC problem, Aeroelastic Design.

Our interdisciplinary group of researchers at the University of Colorado has begun work on dealing with data communication and storage issues in Grand Challenge computing, driven by the needs of the above, and other, examples. We have developed a local high performance ATM network and interfaced it to a wide area ATM network. These two networks encompass most of the facilities needed to solve GC problems - several supercomputers, visualization systems and mass storage. We will describe this network, which is intended to provide a first step towards solving these challenging communication issues, in sections 5-7, and present some measurements in section 8.

2. High Resolution Turbulence - a Case Study

We will explore limiting computational effects in high resolution CFD computations because they provide an excellent illustration of where the true limiting factors of large-scale computations are to be found. We will illustrate by examples of computations currently being developed at the University of Colorado as part of an NSF Grand Challenges Application Group (GCAG). This is a joint effort involving both computer scientists and astrophysicists, and includes also researchers at several other institutions.

Turbulence is found throughout the applications of fluid mechanics. Despite this the physics of turbulence is not well understood, and as a result our ability to qualitatively simulate turbulence remains poor. Understanding the phenomenon of turbulence, and determining computation methods for simulating turbulent flows, is therefore one of the great Grand Challenge problems.

There are many approaches to computational simulation of turbulence. These include field computations based on the Navier Stokes Equations, various forms of discrete simulation involving particles and vortices or vortex sheets, and phenomenological approaches such as large eddy simulation. Turbulence is

striking because of the large range of scales encountered, which in computational terms means that high resolution computations will be required, until we understand the phenomena involved much more deeply.

One standard approach therefore is to use very high resolution finite difference (or finite element) methods applied to the Navier-Stokes Equations [1], and by varying the resolution to explore the effects of including higher frequencies. Because turbulence is statistical in nature one cannot expect to observe deterministic behavior in such computations. Instead one expects to perform many similar computations and study statistical quantities such as energy spectra. The combination of the need to use high resolution, and the need to perform multiple experiments, make these computations especially challenging as we now illustrate.

3. Computational Limitations for High Resolution CFD

Our group is developing computational methods for turbulence targeted for the Tflops machines we expect to have available in several years. We are performing simulations on today's Gflops MPP supercomputers, but our planning is focused by the needs of the future machines. Much of our project is oriented to understanding the issues involved in designing such Tflops computations.

We utilize a combination of finite difference, spectral and multigrid methods applied to the Navier-Stokes Equations. In recent years 3D grids with as large as 256^3 grid points have been used for such computations on Gflops machines, as well as some limited experiments at 512^3 grid points. However extending the resolution from 256 to 1024 in each spatial dimension increases computational effort for a run by a factor of 4^4 or 256, which indicates the need for teraflops machines for full runs.

In a typical CFD simulation of this type there will be a requirement to follow a turbulent flow on a 1024^3 rectangular grid well into the turbulent "steady state". Such a computation should be expected to run for about 100,000 timesteps, because of the coupling of timestep lengths to spatial resolution. We will estimate the various aspects of such a computation in order to determine the critical limits for Tflops computation. Therefore we will analyze the following components of such a computation in detail: a) Memory needed; b) CPU rate and time; c) Internode communication; and d) Data output.

3.1. Memory Usage

A typical computation may involve perhaps 10 physical variables (pressure, temperature, velocity, ...) per grid point of the 1024^3 grid. Furthermore numerical methods will require the full solution at the previous timestep to be

remembered, so that in practice 20 variables must be stored per grid point. This dictates a total memory requirement of 160 GBytes (GB), assuming that 8 byte words are used to store quantities.

This number is only a factor of about 3 larger than the current main memories of the most powerful MPP systems such as the 2000 node Intel Paragon supercomputer at Sandia National laboratory. Given the rate of increase of memory density, and the resulting decrease in cost per byte, it is clear that memory is not in itself a limiting factor for this kind of computation. In fact, this computation could be performed with less memory by swapping from a fast disk array, although we would then need to examine the data rates to disk used for that purpose.

3.2. Computation Time

Total computation time is of critical interest to the scientist or engineer. A computation that takes too long to complete is not of great interest, particularly in an area where many similar simulations must be performed in order to arrive at statistical or optimization conclusions. Thus total computation time is a possible limiting factor for HPC in general. A good rule of thumb is that scientists appreciate turnaround on the order of 1 day or less. Certainly computations that run for months are not acceptable since waiting that long to discover there is a bug in software (which there invariably is) is not practical. We therefore estimate computation time for our proposed Tflops simulations.

Looking at a range of different CFD algorithms, a general conclusion suggests that about a hundred floating point operations are executed at each grid point per timestep by many codes. This is of course an approximate figure and might vary by a factor of two or more. Certain specialized algorithms are also not covered by the estimate. For example, the PPM algorithm developed by our Grand Challenge colleague Paul Woodward at University of Minnesota requires of order 3,000-5,000 operations per grid point - but in return provides a higher quality result.

Combining the grid size (10^9), the number of operations per grid point (10^2), and the number of timesteps (10^5), we conclude that a single run will require about 10^{16} floating point operations. To give a feel for what this means, we note that on a current generation vector processor or MPP, typically delivering of order 1 Gflops in actual computation, this would require 120 days to run. This assumes the user has a dedicated machine for all of that time, which is highly unlikely. In practical terms this computation will run for several years on a multi-user machine. On the other hand, on a dedicated Tflops (delivered performance) machine we can expect the computation to complete within 2.5 hours. Even on a multi-user system, restricted to a few users, one can expect turnaround time of order a day or two. Thus this is truly a Tflops Grand Challenge problem, but does not require significantly more than a Tflops to be

practical. Because, as previously indicated, Tflops processing power should be easily attainable within several years, we therefore do not regard CPU power as a serious limiting factor to performing these computations.

3.3. Internal Communication

Any Tflops machine in the near future will surely be a MPP. We must therefore ask whether the internal processor-processor communication rates within such machines could be primary limiting factors for our proposed computation.

We will assume that our Tflops machine consists of a thousand nodes, each rated at one Gflops, interconnected by a high speed communication network. The exact number of nodes is not critical, and will in any event be determined by the node processor speed (here assumed to be 1 Gflops) and the total processing power required (1 Tflops). The large grid will then be subdivided into a corresponding number of smaller subgrids, with each assigned to a distinct node. As a result each node will contain a sub-cube of approximately 2^{20} grid points.

Typical CFD computations using explicit methods, or implicit methods solved iteratively on regular rectangular grids, have a simple internal communication pattern. Each node will typically need to send the variables stored on the boundary of its subgrid to its six neighboring subgrids. The boundary of each subgrid has about 2^{16} points. This assumes a boundary depth of one grid point - higher order methods may increase that by a small factor such as two or three. We will assume a boundary depth of two is required. Furthermore all ten physical variables of interest will need to be exchanged at each boundary point. This results in about a 10 MBytes data exchange per timestep for each node.

Based on 100,000 timesteps performed in 2.5 hours of processor time, each timestep requires about 1/10th second. Clearly a basic requirement is that the internal communication should not slow down the computation. Fortunately it is possible to overlap communication with computation assuming the system supports asynchronous communication, as most do. Since the boundary exchange may certainly overlap the computation (except near the boundary) we see that an aggregate data rate of about 100 MBytes/sec bi-directional sustained between each processor and its 6 neighbors will suffice - i.e. about 20 MBytes/sec on each connection. Several current MPP systems already attain or exceed this value for internode communication. Such a rate can be achieved in fact even for distributed workstation clusters connected by an ATM network at OC-3 (155 Mbit/sec) rates or higher. It appears then that internal communication is unlikely to be a limiting factor for Tflops CFD computation.

3.4. Data Output

All of the value of an HPC computation such as we have described is contained in the output. In a few fields, e.g. Quantum Chromodynamics, the output amounts to as little as a single number. However in CFD, usually a substantial time history of the fluid flow is required. A time history will consist of a sequence of single timesteps, although typically not all of the timesteps that were computed. In our proposed experiment, each captured timestep requires dumping 10 physical parameters at each of the 1024^3 grid points. Thus a single timestep requires a total of approximately 100 GBytes of mass storage.

It is clearly impractical to save all timesteps. Suppose that we took a *very conservative* approach, and saved only every thousandth timestep. This would result in only 100 frame dumps per run. The total data generated in a run would then be 10 TBytes. All of that data must be deposited in a suitable mass storage facility before the computation can end, since the amount of data involved is too large to be buffered effectively.

In order that the computation not be slowed down by the output of data, it is essential that all of the data can be outputted in the time the computation requires. We will again assume that output can overlap computation completely. To generate 10 Tybtes of data in 2.5 hours of computation would require effective I/O rates to mass store of order 1 GByte/sec. Note that, given our assumptions, we are effectively allowing the output from a single timestep to be produced over the following 1000 timesteps. This is reasonable, provided we increase memory required by 50% in order to store the output data from that timestep in memory until the output is complete. Otherwise we would require data rates of a Tbyte/sec (100 GBytes in 1/10th second)! This is a nice example of how a very modest increase (50%) in one easily achieved aspect of hardware (memory) can reduce the needs in another much more unnatainable area (I/O) by a huge amount (1000 times).

Discussions with supercomputer center directors around the world [2] indicate that peak I/O rates to *mass store* (i.e. storage capable of holding many tens of TBytes) today are of order 2.5 MByte/sec. Thus we are off by almost three orders of magnitude from what would be required to sustain an efficient CFD simulation. There are certainly faster I/O channels than 2.5 MBytes/sec available on computers today (e.g. HiPPI channels) . Unfortunately mass storage systems capable of holding tens of TBytes are not directly writeable at these speeds.

Finally then we have determined a major obstacle towards completing our computation. Data output to mass store is seen to be the fundamental issue for HPC. In practice the problem is more severe than described, because researchers would wish to keep a larger number of frames of data - probably closer to every hundred timesteps.

3.5 Some Related Issues

It is far from clear that computations such as the one described above will be performed on computers that are adjacent to the mass storage facility. Every supercomputer will have some associated storage, but in many cases the output of such computations may be stored at other sites, possibly even in several sites. In some cases the user may not even be aware of the storage location, as there may be a national or distributed file system involved that provides a transparent interface. In these cases the bandwidth between the supercomputer and the remote storage site(s) becomes a real limiting factor. At the present time the fastest production interconnections between U.S. supercomputer centers are T3 lines running at 45 Mbit/sec. These lines carry the NSFNet backbone and are shared by tens of thousands of users. Even with a dedicated T3 line it would take 2.5 days to dump output from the computation described above to a remote mass storage facility, effectively slowing the computation by a factor of 24.

Most data output of the type described above is archived at runtime for later post processing. Researchers involved in post processing work will typically not be located at the mass storage site either, and then the bandwidth between their location and the mass storage facility becomes another critical factor. As an example, most research centers today have T1 connectivity to the internet, implying a 1.5 Mbit/sec data rate, which is shared by hundreds or thousands of researchers (and possibly students). Even with a dedicated 1.5 Mbit/sec line, it would still take 77 days to transfer the blocks of a typical 10 TBytes output run to a post processing site.

It would be misleading to suggest that external I/O is the only area of HPC algorithm design of importance. Clearly the underlying understanding of the basic physics, and the quality of the numerical algorithm chosen are critical. In fact a reasonable objection to the whole discussion to this point would be to complain that it is irrelevant - because no one needs 1000^3 grids. Surely we do not need that many degrees of freedom to describe the physics? With improved understanding we may learn how to compute turbulence without using 10^{10} degrees of freedom. In fact that is the goal of most turbulence simulation work. But for the time being we have little choice but to use a brute force approach.

Within each node of an MPP, substantial effort goes into optimizing the single-node performance of an algorithm [3]. Much of this effort is also related to I/O rates - in this case the I/O bandwidths between processor, registers, cache and memory. Viewed this way, we realize that an HPC computation is mostly an I/O exercise, taking place on 4 or more distinct levels ranging from registers to TByte mass stores!

Finally there is clearly a role for improved representations of output data. Compression can significantly reduce storage needs, especially if methods similar to the MPEG compression algorithms used in video compression could be utilized. However for high resolution turbulence studies, where phenomena are

developing dynamically on all length scales, data compression using standard methods is unlikely to qualitatively improve the situation, until such time as we have a deeper understanding of the relevant degrees of freedom. Of course at that point we will likely have opportunities to completely reformulate such computations in terms of the real degrees of freedom. Computations performed with more appropriate physical degrees of freedom would presumably result in much smaller output data.

4. Aeroelastic Design - A Second Case Study

In a second set of Grand Challenge computations, our group of researchers in the Aerospace Engineering and Computer Science Departments at University of Colorado are studying so-called Coupled Field problems. Coupled Field computations involve several distinct computational phases, and often these phases may use quite different numerical algorithms, and may perform best on different architectures. Furthermore, in many cases the coupling between the fields is localized and of lower dimensionality. Such problems are appropriate for heterogeneous computing - using several supercomputers of different architectures, with each chosen to optimize performance on one phase.

One coupled field problem we are working on is the aeroelastic simulation of aircraft [4]. Standard aircraft simulation is usually restricted to either a) finite element analysis of an aircraft structure or b) CFD analysis of the airflow over the aircraft, regarded as a rigid structure. In reality, aircraft are flexible structures that change shape under the pressure exerted by a flow. This shape change in turn modifies the flow, and thus the pressure. This time-dependent coupling of the fluid and structure modes defines the aeroelastic problem. Its effects are evident to any traveler who has seen the wings oscillating through an aircraft window - the aircraft is very obviously a flexible structure and not a rigid body. Ensuring against a catastrophic oscillation, which can lead for example to wing rupture, is a critical aspect of full aircraft simulation

The aeroelastic computation is a classic coupled field problem, involving two coupled fields: the structure and the fluid. When treated numerically a third field is also introduced - a moving mesh that interfaces cleanly between the structure and fluid. In practice, the structure and fluid dynamics are largely independent of each other, interacting only on a lower dimensional surface - the aircraft surface. The natural timescales for these two fields are also quite different - the fluid requires short timesteps while the structure can utilize longer ones.

The coupled atmospheric/ocean model for weather and climate modeling is another example of a coupled field problem. Here the atmospheric and oceanographic phases are handled differently, have different timescales, and interact only through the two-dimensional surface of the ocean. Timescales for the atmospheric and oceanic components are vastly different, and the

oceanographic problem involves a huge range of length scales compared to the atmospheric problem.

Such problems suggest a heterogeneous treatment in which each phase is computed on the optimal supercomputer hardware for that phase, and the exchange of information between the phases is accomplished by data exchange between the computers. We can analyze heterogeneous computing experiments in the same way that we analyzed the turbulence computations. We again find that external I/O - external to the individual supercomputers - becomes the primary bottleneck. Of critical importance is the data bandwidth and latency between the individual cooperating supercomputers, and for problem sizes we expect to run within the next few years, data rates of about 100 MBytes/sec are required [4]. Latency is somewhat less critical because of the large granularity of the processes running on the individual supercomputers. While each supercomputer may decompose the component field into sub domains of smaller granularity (e.g. on an MPP), individual nodes of one MPP do not need to interact with those on another supercomputer.

4.1. Most Computations are Heterogeneous

Even traditional single computer computations also require the use of multiple computational resources. For example, the previously described turbulence computations that will generate 10 TBytes of data in 2.5 hours when run on Tflops computers produce data which must be visualized both in real-time (for monitoring purposes) and in a post processing phase. Thus such computations involve three critical resources: the supercomputer, the mass storage facility and the visualization system. Typically these resources are not located in the same location, and the speed of the communication links between them becomes the limiting factor in the computation as mentioned earlier. Thus heterogeneous computing is seen to be the rule, rather than the exception.

5. Broadband Networking

High speed networking has been the focus of considerable ongoing research over the past few years. Early work developed high speed proprietary channel products, advanced to standards work on HiPPI (High Performance Parallel Interface) [5] and Fiber Channel [6], and continued with establishment of a number of gigabit networking test beds which integrated this work with the emergence of SONET and ATM [7]. In general, supercomputers were viewed as requiring networking speeds of 800 Mbps or higher for access to high speed peripherals and for cooperative work on the Grand Challenge problems [8]. By contrast, workstations and PC's were viewed as requiring only Ethernet networking at 10 Mbps.

While supercomputing may require higher bandwidth, few supercomputers can in fact effectively utilize such bandwidth. An informal

survey of national supercomputer center directors [2] yielded a maximum sustained capability to write to high volume (Tbyte) mass storage of 20 Mbps. Thus HiPPI and similar network protocols represent overkill in many situations at the present time. Simultaneously, in recent years it has been realized that standard communication links used to interconnect workstations both within local area networks (LAN's) - often 10 Mbps Ethernet - and between LAN's - often 1.5 Mbps T1 - are completely inadequate for many purposes. Examples include high quality visualization, desktop videoconferencing and multimedia access to video and audio. Particularly relevant is that such communication channels are often shared by tens or hundreds of users, reducing effective bandwidth far below 1 Mbps.

6. BATMAN: An ATM Network for Computational Science

BATMAN is a high-speed Wide Area Network (WAN) trial developed to deal with some of the issues addressed above [9]. BATMAN was constructed by US West as an ATM trial, and participants include the NCAR, NIST and NOAA government research laboratories in Boulder, Colorado, and the University of Colorado (CU) as well as several area companies, the local public library, a school system, and even the author's house, as a prototypical home on the future information highway.

BATMAN is based on ATM (Asynchronous Transfer Mode) networking and is implemented as a distributed set of five large ATM switches interconnected by a mesh of 155 Mbit/sec fiber connections. Each switch provides a full cross-bar connectivity between its connected circuits. The switches are distributed so that each is reasonably close to several participants. Participants are connected to a switch by one or two connections: a 150 Mbit/sec ATM line and/or an Ethernet line. At each site the incoming ATM line is then interfaced to local networks such as Ethernet, FDDI or ATM local area networks using routers, or in the some cases, by locating further ATM switches and an internal ATM network at that site.

Each of the supercomputers in the Boulder area (KSR-1 at CU, Intel Paragon at NOAA, CRAY YMP/8, CRAY T3D and Thinking Machines CM-5 at NCAR) are connected with FDDI or faster connections to BATMAN. Mass storage facilities at NCAR are also connected to BATMAN, as are visualization labs at each institution. Furthermore the Internet backbone is connected to BATMAN at NCAR, providing DS-3 rate (45 Mbits/sec) connectivity to the outside world from any ATM-connected site within BATMAN.

At the Computer Science Dept. of the University of Colorado we have built an ATM local area network based on Fore Systems ATM switches and interface cards. We have tested ATM-connected DEC, HP, SUN and SGI workstations within this LAN. We have also connected the LAN to the BATMAN WAN providing an interesting environment for both applications research and

network interoperability testing. Finally we are using the ATM LAN to begin construction of an ATM-connected cluster of multi-Gflops workstations on which a virtual shared memory environment will be supported.

A suite of 36 applications are now running on this network, including desktop video-conferencing, movies on demand and bit-mapped newspaper distribution, in addition to supercomputer applications such as those described earlier. While BATMAN is not a complete solution to the I/O limits posed in the earlier sections, it does provide a step in the right direction. Most importantly, because ATM is inherently scalable to very high speeds (such as 4.8 Gbits/sec) and because of its widespread adoption by communication companies, it may well provide much of the communication infrastructure needed to support Tflops supercomputing.

7. BATMAN Design and Applications

BATMAN is an acronym for **B**oulder **ATM** **A**ccess **N**ode. This name was natural as the intent was to create a distributed ATM network in which individual sites within Boulder, Colorado, would obtain access through nearby access nodes (ATM switches) rather than through direct access to a monolithic switch. Such an architecture is inherently more scalable, and is intended to serve as a model for larger scale metropolitan area deployment.

The basic design involves five Newbridge ATM switches, interconnected by OC-3 fiber. Each switch can accommodate up to eight ports, with a typical switch involving four OC-3 ATM ports and four 10 Mbps Ethernet ports. However in some cases 1.5 Mbps T1 ports, supporting a transparent Ethernet Bridge, are used instead.

This mix of communication capabilities in turn encourages development of a network that includes a range of institutions, each acquiring bandwidth appropriate to its needs. This has resulted in a remarkably diverse team of participants, and correspondinly led to extremely interesting network experiments.

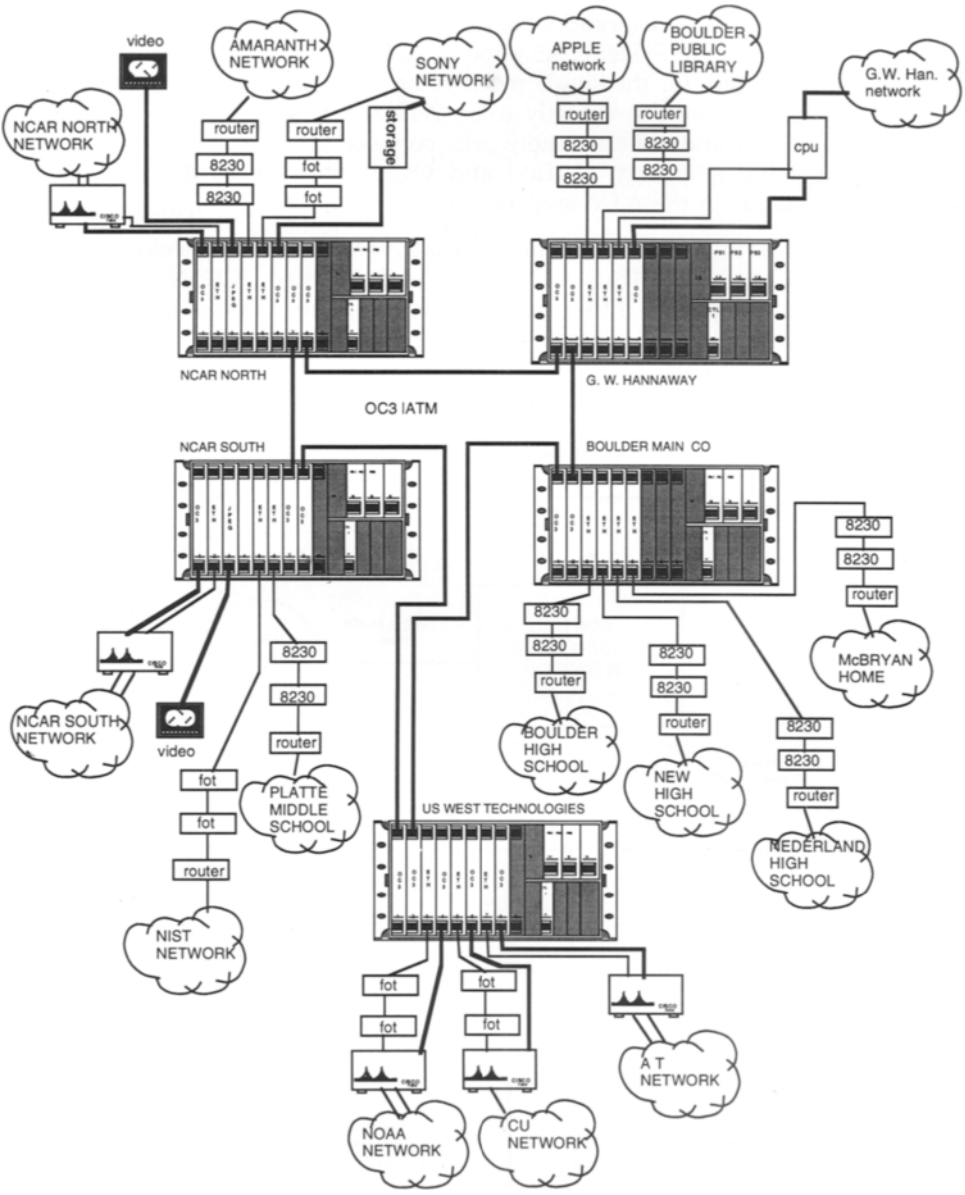
7.1. BATMAN Architecture and Participants

A complete list of BATMAN participants is provided in Table 1, and essentially spans the spectrum of educational, community, government and commercial users. Each of these participants is directly connected to a port on a BATMAN ATM switch using either OC-3 on single mode fiber, or high quality commercial T1 circuits. All of the ATM switches are in turn interconnected using single-mode fiber at OC-3 rates. For those institutions connected to a BATMAN switch by OC-3 single mode fiber, the destination point at the institution is a CISCO 7000 router. In these cases, an Ethernet connection to the switch is also provided for testing and backup.

Table 1: BATMAN PARTICIPATING INSTITUTIONS	
Coordinator:	U S WEST Advanced Technologies
Universities:	University of Colorado at Boulder
Government Laboratories:	National Center for Atmospheric Research National Institute for Standards and Technology National Oceanographic and Atmospheric Admin.
Industrial Laboratories:	Apple Electronic Media Laboratory SONY Recording Media Laboratory Amaranth Custom Photographic Laboratory GW Hannaway & Associates Laboratory
Civic Organizations:	Boulder Public Library
Public Schools:	Platt Middle School Boulder High School New High School Nederland Middle/High School
Private Residence:	McBryan Residence

The remaining institutions typically use a simpler router or a routing workstation as the point of contact. Despite the lower performance of these connections, observed performance is typically still very impressive due to the lack of other traffic on this private network.

Figure 1 shows a complete network diagram of BATMAN. The focus is on the five ATM switches, and the fiber (dark lines) interconnecting these and connecting to the various institutions. The university is shown at the bottom (CU Network) while the authors home is on the right along with schools attended by several of his children. The thin lines with an 8230 box located on them are T1 lines over which a bridged Ethernet is supported. The boxes marked "video" denote switch interfaces that contain JPEG compression cards for real-time video. Each small cloud at the endpoints of the network actually represents a complete internal network at the corresponding institution.



3/25/94 LGF

Fig 1: Boulder ATM Trial Network Diagram (BATMAN)

The University of Colorado has developed an internal ATM local area network based on a Fore Systems ASX-100 ATM switch. CU uses the Fore switch as the point of contact for the single mode fiber from the BATMAN switch. The Fore switch in turn connects directly to an ATM LAN network and to a CISCO 7000 router. FDDI and Ethernet networks connect to the router. The various workstations have a variety of Taxi and OC-3 interface cards, matched by corresponding cards in the ATM switch.

Figure 2 is a schematic of the internal ATM network developed within University of Colorado. Thus the small cloud denoted "CU NETWORK" in Figure 1 actually expands into Figure 2. Each of the workstations shown in Figure 2 has a native ATM interface.

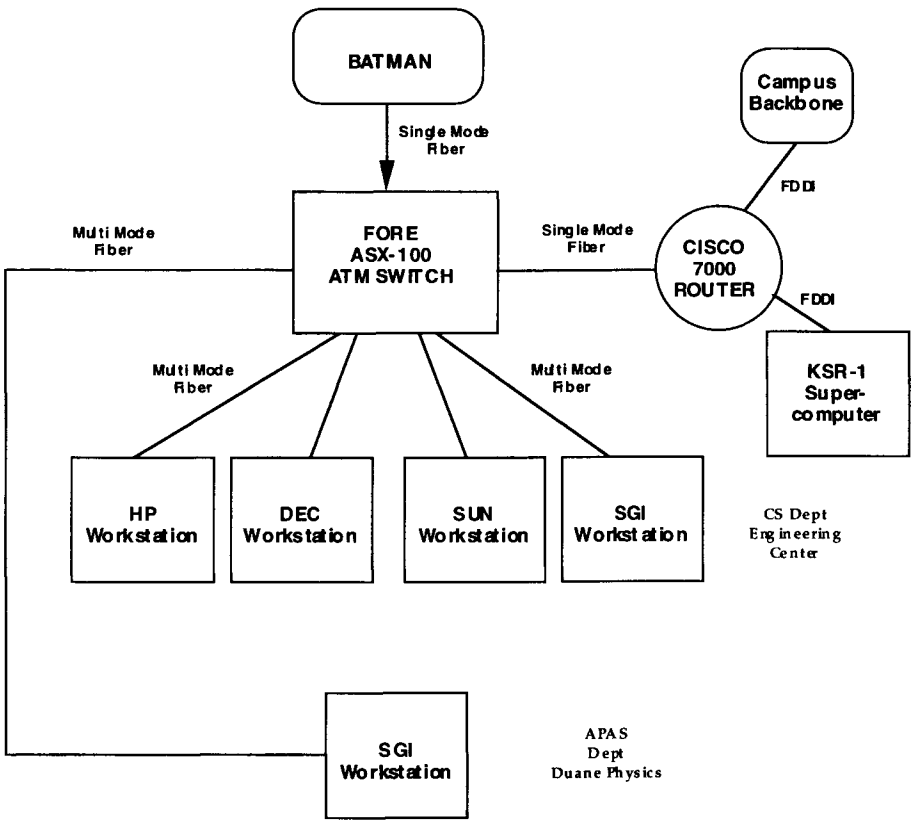


Fig 2: University of Colorado ATM Local Area Network

Static hardware routing throughout the BATMAN network is provided by a set of permanent virtual circuits (PVC's) established globally between the institutions. All systems connected directly to BATMAN belong to the internet **batman.net**. Software routing is provided from a central source at NCAR and utilizes RIP or OSPF running on individual routers. Because of interoperability issues involving WAN ATM switches, and a lack of completed standards from the ATM Forum, it is not possible to establish switched virtual circuits (SVC's) over such a WAN/LAN environment at this time.

BATMAN became operational in March 1994. For a more complete description of the BATMAN architecture as well as a full list of applications and results, we refer to the BATMAN World Wide Web site located at the URL: **http://www.cs.colorado.edu/~batman/Home.html**.

7.3. BATMAN Application Suite

A suite of 15 applications has been developed to exercise the network described above. These applications span the range from digital video servers to supercomputer access to mass storage. A complete list of applications is contained in Table 2. We will discuss applications 1, 3 and 8-10 in this paper along with performance of results obtained. The measurements reported here were performed by a group directed by McBryan. For more details of all performance measurements described here we refer to our paper [10].

Table 2: BATMAN APPLICATIONS

1. Super Computer access to mass storage
2. High-Density High-Speed Storage Access
3. Heterogeneous Processing using multiple supercomputers
4. Online Electronic Newspaper
5. NTSC video via JPEG to workstations
6. Desktop Videoconferencing
7. Satellite Access to ATM network using ARPA ACTS satellite
8. Desktop ATM -> Desktop ATM using ATM LAN
9. Desktop ATM -> remote Desktop FDDI
10. Desktop ATM -> remote Desktop ATM
11. School Access to government graphical databases
12. Desktop video conference between student and teachers
13. School Access to remote CD/ROM drives at Library
14. Family Information Server
15. Home Business on Information Highway

8. Performance over the BATMAN ATM Network

8.1. Supercomputer Access to Disk Farms via ATM

NCAR currently manages a large Mass Storage System (MSS) which is utilized by the University of Colorado and NOAA as storage for their supercomputers. Network access between the University of Colorado and NOAA supercomputers and the NCAR disk farm is normally provided by the existing TCP/IP internet in the area. This configuration has proven to be quite useful.

This existing internet is based upon T1 lines, limiting throughput to 1.5 Mbps or lower. Many of the data sets which the supercomputers wish to store are extremely large. The relatively slow speed of the T1-based internet is a bottleneck for the supercomputers which wish to store data. In practice, due to the fact that potentially thousands of other researchers are utilizing the same T1 line simultaneously, average observed bandwidths for file transfer fall below 100 Kbps at times. As a result, the MSS at NCAR is not as well used by external researchers as it might be if a faster network were available.

There are three obstacles to achieving high MSS data rate. First is the inherent limit set by the maximum data rate into the mass storage facility. This is currently 20 Mbps, and expected to grow fairly rapidly in coming years. The second is the data rate limitation imposed by the bandwidth of the interconnection. Here the jump from heavily loaded T1 bandwidth to dedicated ATM speed implies an enormous improvement. The third critical factor is the ability of the supercomputer to output data fast enough, which depends on its architecture and on design characteristics of its I/O sub-system.

As shown in Figure 1 the ATM Trial Network provided an alternate internet environment. This ATM-based internet was utilized to provide very fast internet access between the University of Colorado and NOAA supercomputers and the NCAR MSS. Unfortunately realities of the MSS interface to the network have shown that ATM data rates are not currently of great benefit for direct MSS access. Table 3 presents data rates from an ATM attached SGI workstation at University of Colorado to the MSS. The limiting factor proves to be the TCP/IP interface to the MSS where an IBM 3084 computer appears unable to process IP packets at faster rates.

Table 3: Data Rates to NCAR MSS	
File Size (Bytes)	**Data Rate (Mb/s)**
100000	1.952
1000000	.350
10000000	.515

As an alternate, but indirect, MSS strategy we have also explored the use of an HiPPI-attached proxy server to provide a faster gateway to the mass storage. In these experiments we utilized a CRAY YMP computer which has a HiPPI attachment to the MSS. External data was then transferred over BATMAN to the CRAY, after which it was downloaded to the MSS over HiPPI. Data rates of 6 Mbps were then achieved. A limiting factor in this case was the interface between the CRAY (which has no native ATM port) and BATMAN. A second limitation was that all date to be transferred had to be completely downloaded to the CRAY before any data could be transferred to the MSS. This use of a store-and-forward protocol rather than pipe lining was necessitated by the limited nature of the MSS HiPPI interface.

8.2. Heterogeneous Supercomputing

Concurrent heterogeneous processing (see section 4) over BATMAN has already been implemented between the Kendall Square KSR-1 supercomputer at the University of Colorado, the Intel Paragon supercomputer at NOAA, and the TMC CM-5 supercomputer at NCAR, and also involving SGI workstations for visualization. This concurrent processing was supported by the existing T1-based internet in the Boulder area. Since the T1-based internet was quite slow in comparison to the needs of the supercomputers, the resulting performance was too poor for practical use. The aeroelastic design model of section 4 was implemented and debugged over this network.

This T1-based configuration was converted over to the ATM/OC3c-based BATMAN internet which was created as part of the trial. Ideally the network would function like a very high speed backplane which connects all of the supercomputers together. Although 155 Mbps connectivity is not fast enough to truly emulate a very high speed backplane, it was hoped that upgrading from a T1-based internet to an OC3c-based internet would result in significant improvement to the point of practicality. Table 4 describes the three supercomputers and the visualization computer that we have used in these experiments, while Table 5 describes the results obtained in controlled measurements of inter-machine performance using two standard measurement tools: ftp and ttcp. In all cases, measurements are far below ATM peak rates, but are also generally far higher than those obtained over loaded T1 connections. The primary limiting factor at the present time is the poor quality of the FDDI

interfaces to each of the three supercomputers. All values represent maximum measurements and are in Mbps.

Table 4: Machines Involved in Supercomputer Access				
Name	**Location**	**Type**	**Software**	**I/O Interface**
lbsn	NCAR	TMC CM-5, Sun Sparc Front-end	SunOS 4.1.3	FDDI
mother	CU CS	Kendall Square Research KSR-1	OSF/1	FDDI
f16	NOAA	Intel Paragon	OSF/1 rel 1.0.4	FDDI
guano	CU CS	SGI Indy R4400	IRIX 5.2	ATM (Fore GIA-100)

Table 5: Performance Results for Supercomputer Access (All results in Mbps)			
Source	**Destination**	**ftp**	**ttcp**
lbsn (TMC CM-5)	guano (SGI Indy R4400)	6.61	7.00
guano (SGI Indy R4400)	lbsn (TMC CM-5)	**5.69**	**7.61**
mother (KSR KSR-1)	guano (SGI Indy R4400)	**5.68**	5.38
guano (SGI Indy R4400)	mother (KSR KSR-1)	9.60	12.99
f16 (Intel Paragon)	guano (SGI Indy R4400)	0.32	0.31
guano (SGI Indy R4400)	f16 (Intel Paragon)	2.16	2.00
mother (KSR KSR-1)	lbsn (TMC CM-5)	2.88	2.36
lbsn (TMC CM-5)	mother (KSR KSR-1)	3.04	2.27

8.3. High-Speed Access between Desktop Workstations

Perhaps as important as high-speed access to supercomputers is the need to improve access between workstations. Workstations provide the primary window from scientists to other researchers as well as to remote databases and

supercomputers. Collaboration environments that are in use across workstations include teleconferencing tools, shared whiteboards and multimedia browsers for the World Wide Web. Furthermore on-line access to remote CD-ROM databases is becoming an important activity.

Tables 6 and 7 present measurements of performance between workstation over the ATM LAN/WAN. Table 6 describes the individual machines used in these experiments, while Table 7 describes the performance measured. In all cases the limiting factors have again been in the work-station interfaces, but in this case we are seeing throughput as high as 50 Mbit/sec, which is a significant fraction of theoretical OC-3 bandwidth.

In one case we measured 132 Mbps between two DEC workstations with second generation ATM interface cards (not included in the tables). Most of our experiments however utilized first generation cards which are generally limited in performance to about 50 Mbps.

Table 6: Machines Involved in Workstation Access

Name	Location	Type	Software	I/O Interface
guano	CU CS	SGI Indy R4400	IRIX 5.2	ATM (Fore GIA-100)
joker	CU CS	HP 9000/735	HP-UX A.09.05	ATM (Fore HPA-200)
galatea	US West AT	SGI Indy R4000	IRIX 5.2	FDDI
kilt	CU CS	DEC Alpha 3000	OSF/1 v2.0	ATM (DEC OTTO)
nike	CU JILA	SGI Indy	IRIX 5.2	ATM (Fore HPA-200)

Table 7: Performance Results for Workstation Access (All results in Mbps)			
Source	**Destination**	**ftp**	**ttcp**
joker (HP 9000/735)	guano (SGI Indy R4400)	23.31	24.80
guano (SGI Indy R4400)	joker (HP 9000/735)	24.77	50.26
galatea (SGI Indy R4000)	guano (SGI Indy R4000)	1.18	0.89
guano (SGI Indy R4400)	galatea (SGI Indy R4000)	12.95	46.48
kilt (DEC 3000)	guano (SGI Indy 4400)	21.39	21.67
guano (SGI Indy 4400)	kilt (DEC 3000)	23.20	51.21
kilt (DEC 3000)	nike (SGI Indy 4000)	39.47	43.44
nike (SGI Indy 4000)	kilt (DEC 3000)	29.60	57.76

9. Conclusions

By examining the details of typical HPC computations, we have shown that data output rates represent the major obstacle to scaling HPC computations to the Tflops range. If past experience is to be a guide, we can safely conclude that manufacturers will concentrate on peak performance and internal I/O rates while largely ignoring the more significant external I/O issues. Unless pressure is put on the vendors, there is a real danger that Tflops machines will not deliver on expectations for this reason.

An extensive technical trial of ATM networking in Boulder, Colorado has shown that OC3-based ATM WANs can be effectively utilized by supercomputers to boost performance in the areas of heterogeneous processing and of remote access to data. ATM connectivity can also provide significant performance improvements for high speed connectivity between desktop workstations. Perhaps the strongest lesson learned is that the weakest link is indeed the weakest link in any networked application.

References

[1] F. Cattaneo, N.H. Brummell, J Toomre, A. Malagoli, and N.E. Hurlburt, Astrophys. Jou., 370, 282, 1992.

[2] O. McBryan, Informal Survey of NSF Supercomputer Center Directors, January 1994.

[3] C.F. Baillie and O. McBryan, *"Computer Science Aspects of Grand Challenge Computing"*, in Proceedings of High Performance Computing '94, ed. A. Tentner, pp. 247-251, The Society for Computer Simulation, San Diego, 1994.

[4] C. Farhat and O. McBryan, *"Heterogeneous Computing for Aeroelastic Computations"*, Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, SIAM, to appear.

[5] D. Tolmie, *"Gigabit Networking"*, IEEE LTS, May 1992, pp. 28-36.

[6] F. Vakil, *"Fibre Channel: A Technology Perspective"*, Bellcore Document TM-ARH-021211, April 7, 1992.

[7] K. Young, *"Gigabit Networking"*, Bellcore Digest, August 1992.

[8] C. Catlett, *"In Search of Gigabit Applications"*, IEEE Communications Magazine, April 1992, pp. 42-51.

[9] R.G. Foldvick and O. McBryan, *"BATMAN: A Wide Area ATM Network"*, Proceedings ICC '95, IEEE to appear.

[10] A. Aggarwal, A. Griff, O. McBryan and B. Reilly, *"Performance Measurements over ATM LAN and WAN networks"*", University of Colorado Tech Report, in preparation.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Scalability of Parallel Systems: Efficiency Versus Execution Time

J. S. Kowalik

Boeing Computer Services, P.O. Box 24346, MS 7L-44, Seattle, WA 98124-0346, USA

Abstract: Ideal parallel computer systems (architecture + class-of-algorithms) are linearly scalable. They allow to solve a series of increasingly larger problems by using more processors and maintain the machine efficiency and the execution time constant. Most of the practical parallel systems are nonlinearly scalable. For these systems we must choose either a constant execution time or a constant efficiency when scaling-up the problem and the machine.

Keywords: parallel computation, scalability, performance

1. INTRODUCTION

Often claims are made by hardware vendors and software/algorithms developers that their creations are scalable. Scalability is a very essential feature of well designed and implemented hardware or software. Intuitively we understand what it means that a computer system is scalable but there is no generally accepted precise definition of scalability or a well established body of knowledge that could help us to verify the aforementioned claims.

Computer users are interested in using scalable parallel machines for two reasons:

(a) to solve fixed-size problems in shorter time by increasing the machine size,

(b) to solve larger problems within acceptable time limits by larger machine configurations.

In this paper we briefly discuss the first case and focus on the second.

Figure 1 illustrates a typical benchmark for a parallel machine running a fixed-size problem. This example has been taken from a recent NASA Ames Research Center report (Bailey, Barszcz, Dagum and Simon [1994]). The figure shows the computational efficiency (speedup divided by the number of processors) as a function of p (the number of processors) for the NAS multigrid benchmark. To calculate this efficiency curve we assume that the 8-processor configuration of SP-1 is a fully efficient base system (efficiency = 1).

Figure 1 reveals a significant drop of efficiency when SP-1 is scaled-up from eight to sixty-four processors to solve faster the fixed-size benchmark problem. Such behavior is not unusual and all of the benchmark tests is the NASA report show similar trends, except the embarrassingly parallel benchmark. This deterioration of efficiency is of no surprise. It is not possible to reduce the execution time below certain minimum no matter how many processors are used. Increasing the number of processors beyond certain value may even be counterproductive and result in worse execution times.

The reader interested in this problem can consult Gupta and Kumar [1993] who describe a method for estimating the minimum parallel execution time for fixed-size problems. The dependence of parallel benchmarks both on machine size and on problem size is discussed in Morse [1994], Section 10.1. He summarizes the discussion by the two statements, (1) efficiency decreases with machine size, and (2) efficiency increases with problem size.

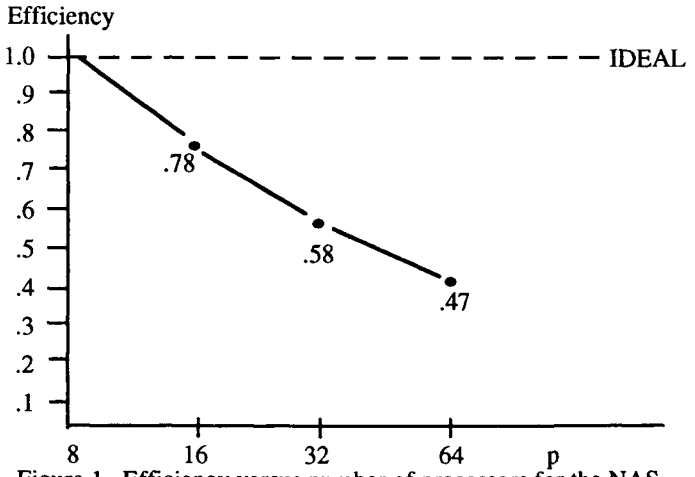


Figure 1. Efficiency versus number of processors for the NAS Multigrid Benchmark (3D Poisson PDE) and IBM SP-1.

Suppose now that we are not interested in minimizing the execution time for a fixed-size problem, but want to increase simultaneously the problem size and the machine size in order to solve a series of increasing in size problems within a limited time interval. A special case of the above goal is as follows: we have a small parallel machine configuration that solves efficiently a relatively moderate size problem of interest within a reasonable period of time. We are interested to know how large problems can be solved preserving the good performance (the efficiency and the execution time) on a larger machine with a higher number of processors. This scenario suggests a preliminary definition of scalability:

> A computer system (an architecture plus a class-of-algorithms) is scalable if increasing in size problems can be solved efficiently by using more processors.

The concept of efficiency is important since low values of efficiency mean wasted processing power and decreased cost/performance ratios.

In this definition we assume that the underlying solution algorithm remains unchanged as we scale-up the problem, and that the parallel computer is an assembly of identical processors. The issue of problem scaling merits further comments since large problems can be constructed in different ways. Singh, Hennessy and Gupta [1993] point out that computer scientists tend to think about algorithm complexity and problem size in terms of data set size. This may be correct for typical nonnumerical applications, but is too simplistic for most of the numerical analysis problems. In numerical applications we can scale problems by varying: grid size, time steps, accuracy of the model or numerical results, and other parameters. The choice of scaling method significantly influences the computational time and memory complexity of the scaled-up problem as well as the communication requirements. Singh, Hennessy and Gupta [1993] suggest that scaling has to take into account all significant sources of error and propose an intuitively good scaling principle: all sources of error should be scaled so that their error contributions are about equal. This approach to scaling for numerical problems leads to well balanced large-scale problems, correct conclusions about the execution times and useful assessment of the key architectural parameters, such as: size and speed of caches and main memory, communication requirements, etc.

Additional consideration should be given to the large problem scope that usually makes parallelization and scaling difficult (Kuck [1994]). By scope Kuck refers to the total amount of data as well as code, together with the complexity of their interaction. Scope is closely related to the notion of the system's differentiation, i.e. degree to which a system is composed of interacting parts that differ in structure and function. Some large computer programs integrate different parts that communicate and enhance each others objectives. This interaction typically reduces performance and scalability of parallel codes. The issue of scope and its impact on scalability is, at present, an open research subject.

In the preliminary definition of scalability we require that a scalable system's performance is efficient. In this paper, we use the term efficiency to mean the normalized speedup, i.e. the speedup divided by the number of processors. Implicitly we also require that the parallel execution time for increasingly larger problems does not grow too fast.

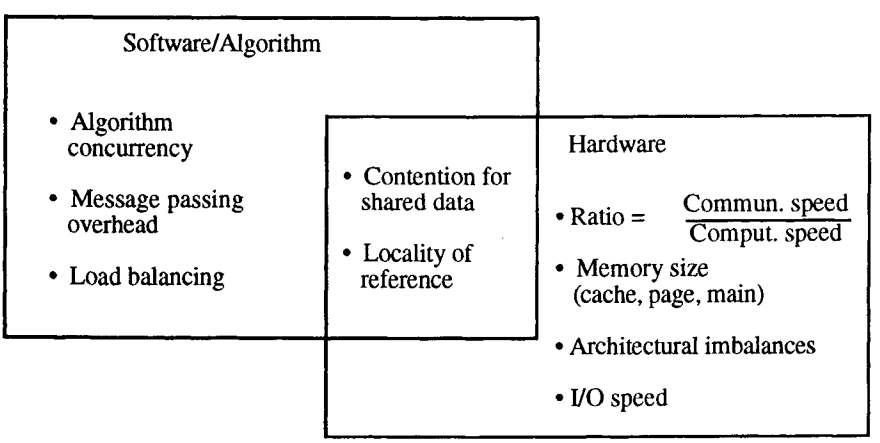
In Section 2 we discuss the main subject of the paper: efficiency versus execution time for scalable systems. Section 3 provides a simple example of a nonlinearly scalable system. Conclusions are in Section 4.

2. EFFICIENCY VERSUS EXECUTION TIME

We will make several simplifying assumptions about the system under consideration:

(a) single user system,
(b) homogeneous multiprocessor or multicomputer,
(c) out of many factors influencing scalability (see Table 1 and the paper by Sobolewski in this volume), we include in the total overhead: communication time, idle processors time and extra work required by parallel algorithm and code. Other factors are ignored.
(d) single algorithm or a class of algorithms with similar data structures and computation/ communication patterns.

Table 1
Factors influencing scalability



The notation used in this paper is similar to the one introduced by Gupta and Kumar [1993]:

$W(\underset{\sim}{N})$ - computational workload; the number of all operations needed to solve the problem,

$\underset{\sim}{N}$ - set of problem scaling parameters,

$T_0(p,W(\underset{\sim}{N}))$ - the total overhead time which is typically a function of p and W. It includes: communication, idle processor time and extra work required by the parallel algorithm,

$T_1(W(\underset{\sim}{N}))$ - sequential algorithm execution time which is $W(\underset{\sim}{N}) \cdot t_{AO}$ where t_{AO} is the average time required for executing one operation,

$T_p(p,W(\underset{\sim}{N}))$ - parallel execution time on a p-processor machine,

S_p - parallel system's speedup: - T_1/T_p,

E_p - parallel system's efficiency: - S_p/p.

The following two relationships will be the basis for our analysis:

$$T_p = \frac{T_1 + T_0}{p} \tag{1}$$

and

$$E_p = \frac{T_1}{p \cdot T_p} = \frac{1}{1 + \frac{T_0}{T_1}} = \frac{1}{1 + \frac{T_0}{W \cdot t_{A0}}} \tag{2}$$

Linear Scalability

Suppose that we expand the parallel computer size (the number of processors p) and the workload $W(\underset{\sim}{N})$ proportionally to p, and the system maintains its performance, i.e. constant efficiency and constant execution time. This can be achieved if:

(a) we add processors with constant workload increments (W and T_1 are linear functions of p),
(b) the combined effect of increasing p and W results in linear growth of the total overhead $T_0(p,W)$ with respect to p.

The linear growth of $T_1(W)$ and $T_0(p, W)$ implies that T_p = constant and E_p = constant (see equations (1) and (2)).

A parallel computer system that satisfies these requirements is linearly scalable. Linearly scalable system happen in real benchmark tests and applications. For example, Buzbee [1993] discusses four examples of linearly scalable systems:

(i) LU decomposition of a banded matrix when implemented on two hypercube machines, an NCUBE and Mark III.
(ii) multigrid technique for solving elliptic equation on an NCUBE,
(iii) QR factorization of a square matrix on an NCUBE,
(iv) the NCAR shallow-water benchmark on a CM-2.

A great advantage of a linearly scalable system is the possibility of solving increasingly larger problems in constant (or approximately constant) time by adding more processors proportionally to the increased workload. Ideally, such a parallel system is linearly scalable for a broad class of similar algorithms solving problems from various application domains.

In general, linear scalability is difficult to accomplish for most of the algorithms implemented on real machines. Usually the total overhead T_0 grows faster then any linear function of p. In Section 3 we discuss a simple example of a system whose growth of T_0 is p log p. In such a system it might be possible to maintain E_p = constant or T_p = constant, while solving larger problems on larger parallel machines, but it is not possible to maintain constant both E_p and T_p.

Nonlinear scalability

If we write (2) as

$$E_p = \frac{1}{1 + \frac{T_0(p, W)}{W \bullet \text{constant}}}$$

we may be able to use the equation

$$T_0(p,W) = W$$

and find the rate of growth for the workload W(p) such that $T_0(p,W(p))/W(p)$ remains constant as p increases and consequently keep E_p constant when the machine size p increases within a certain interval, say $p_1 \le p \le p_2$.

This rate of growth of W(p), which is required to maintain E_p constant, is called the isoefficiency function and has been introduced by Kumar and his coworkers; see for example, Grama, Gupta and Kumar [1993] and Gupta and Kumar [1993]. If a system has a nonlinear isoefficiency function then we call it nonlinearly scalable. Nonlinearly scalable systems have less favorable (faster) growth of the workload than systems that are linearly scalable. The problem workload may grow so fast with p that a system may lose scalability due to its memory limitation.

Another major defect of nonlinear scalability is that T_p increases with p, since both $T_1 = W \bullet t_{AO}$ and T_0 are growing faster than any linear function of p and $T_p=(T_1+T_0)/p$. We cannot anymore have $E_{p1} = E_{p2}$ and $T_{p1} = T_{p2}$, $p_2 > p_1$. The only way to achieve $T_{p1} = T_{p2}$ would be by slowing the growth of the workload. But in this case the efficiency E_p will decrease, $E_{p2} < E_{p1}$ (see equation (2)). When dealing with nonlinearly scalable systems we are faced with an unfortunate dilemma: either we sacrifice the execution time (let it grow) and maintain E_p = constant, or keep the execution time under control and let the efficiency E_p to drop. Table 2 summarizes the results of Section 2.

Table 2. Linear Versus Nonlinear Scalability

Linear Scalability	Nonlinear Scalability
There exists an interval $p_1 \leq p \leq p_2$ such that $E_{p1} = E_p = E_{p2}$ and $T_{p1} = T_p = T_{p2}$	There exists an internal $p_1 \leq p \leq p_2$ such that $E_{p1} = E_p = E_{p2}$ and $T_{p2} > T_{p1}$ or $T_{p1} = T_{p2}$ and $E_{p2} < E_{p1}$
Comment: The work per processor is constant as we expand the problem size and the machine. Memory increases linearly to accommodate larger problems.	Comments: (a) Systems with fast growing isoefficiency functions are less scalable than those with slow isoeffeciency functions. (b) Systems with fast growing isoefficiency functions may suffer from an excessive memory growth requirement or an unacceptable growth of the execution time.

3. A SIMPLE EXAMPLE

Consider computing vector dot products on a p-processor hypercube. Assume that the vectors length is N and each arithmetic operation requires one unit of time. Each processor gets 2N/p vector components to calculate a partial dot product. Then partial sums are added in 2 log p time (assume that one unit of time is required to communicate a number between any two adjacent processors).

For this system:

$$T_1(N) = W(N) = 2N \tag{5}$$

$$T_p(p, N) = \frac{2N}{p} + 2 \log p = \frac{T_1 + 2p \log p}{p} \tag{6}$$

$$T_0(p) = 2 \log p \tag{7}$$

$$S_p = \frac{1}{\frac{1}{p} + \frac{\log p}{N}} \tag{8}$$

$$E_p = \frac{1}{1 + \frac{p \log p}{N}} \tag{9}$$

The system is nonlinearly scalable and its isoefficiency rate function is p log p, i.e. the vector length N has to increase at the rate p log p to maintain E_p = constant.

For example, for N =64 and p = 4 the value of $E_4 = 0.89$. To keep E_p constant requires increasing N = 64 to $N^+ = N\frac{p^+ \log p^+}{p \log p} = 512$ when p = 4 increases to $p^+ = 16$ (see Figure 2).

The execution time increases from $T_4 = 36$ to $T_{16} = 72$. We may want to reduce the execution time by fixing N at 512 and adding more processors, but this reduces E_p. For example, if p = 32 and N = 512, the T_{32} =42 and $E_{32} = 0.76$. This efficiency versus execution time dilemma is illustrated in Figure 3.

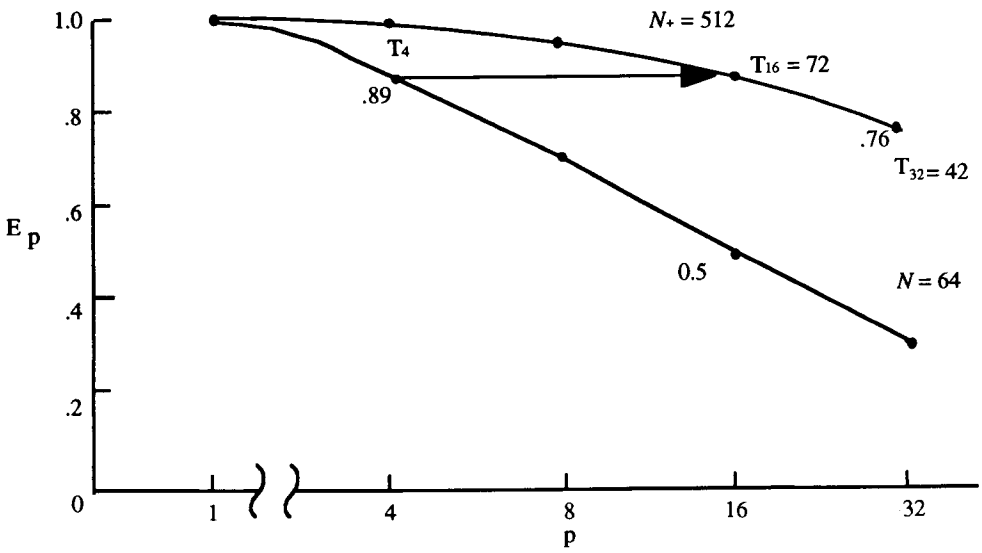
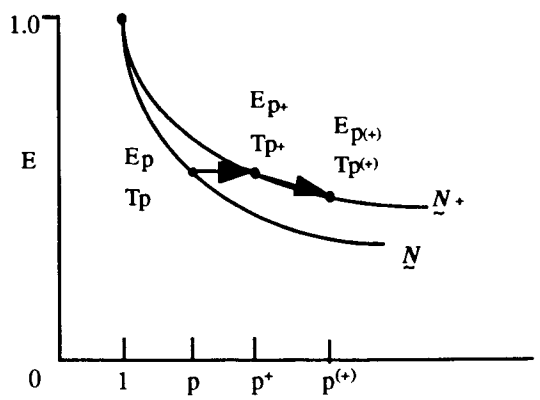


Figure 2. Scalability of the example system



Ep = Ep+
Tp < Tp+
} Constant – Efficiency scaling

Tp = Tp(+)
Ep > Ep(+)
} Constant – Time scaling

Figure 3. Efficiency versus execution time

4. CONCLUSION

Well scalable computer systems are characterized by low growth isoefficiency functions. Such systems can combine near-constant efficiency with a moderate growth of the execution time for larger workloads, or near-constant execution time with a moderate loss of efficiency. Equation (4) determines the growth rate of the workload W(p) needed to maintain E_p = constant. The growth rate of W(p) has to match the growth rate of the overhead $T_0(p, W)$. Higher growth rates of the overhead $T_0(p, W)$ correspond to less scalable systems.

The presented results are based on several simplifying assumptions listed in Section 2. One of them states that we consider a single algorithm or a family of algorithms with similar data structures, and patterns of computation and communication. In reality we need to broaden this assumption and include a wider range of applications and algorithms. A broad definition of a scalable computer architecture could be:

A computer architecture capable of maintaining good performance (high efficiency and near-constant execution time) for a broad range of practical applications that vary in program and data structures, by adding more processors as the problems sizes increase, is said to be scalable.

In this paper we have attempted to clarify some basic notions about scalable systems. In real-life complex applications the total overhead function T_0 cannot be expressed analytically. We can only analyze empirical results obtained from benchmarks and production runs. This practical difficulty does not invalidate our conclusions.

We recognize that there are many open issues related to systems scalability that require long term research and experimentation. Some of them are:

(i) scalability of heterogeneous computing network systems,
(ii) shared systems (as opposed to single user systems),
(iii) systems processing very diverse workloads,
(iv) impact of problem scope as scalability.

We also recognize that there is a trend in parallel systems benchmarking away from measuring efficiency of different systems that are compared. See, for example, Hockney and Berry [1994]. They present several reasons for not using speedups and efficiency in parallel benchmarks. In this paper we use efficiency to study scalability of an individual parallel system and the defined value of efficiency is meaningful as long as T_1 can be adequately estimated.

REFERENCES

1. D.H. Bailey, E. Barszcz, L. Dagum and H. D. Simon, RNR Technical Report RNR-94-006, March 21, 1994.
2. A. Gupta, and V. Kumar, Performance Properties of Large Scale Parallel Systems, Journal of Parallel and Distributed Computing, 19, 234-244, 1993.
3. H. S. Morse, Practical Parallel Computing, AP Professional, 1994.
4. J. P. Singh, J. L. Hennessy and A. Gupta, Scaling Parallel Programs for Multiprocessors: Methodology and Examples, IEEE Computer, 42-50, July 1993.
5. D. J. Kuck, What Do Users of Parallel Computer Systems Really Need?, International Journal of Parallel Programming, 22, 1, 99-127, 1994.
6. B. Buzbee, Examples of Scalable Performance, in Software for Parallel Computation, (ed. J. S. Kowalik and L. Grandinetti), NATO ASI Series, Series F: Computer and Systems Sciences, Vol. 106, 3-33, 1993.
7. A. Y. Grama, A. Gupta, and V. Kumar, Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures, IEEE Parallel & Distributed Technology, 12-21, August 1993.
8. R. Hockney and M. Berry, Public International Benchmarks for Parallel Computers, Scientific Programming, 3, 2, 101-146, 1994.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Architectural Features Affecting Total Scalability of Parallel Systems*

J. S. Sobolewski

Computer and Information Resources and Technology/High Performance Computing Education and Research Center, University of New Mexico, 2701 Campus Blvd. NE, Albuquerque, New Mexico 87131, USA.

Abstract: Traditional scalability studies of (algorithm, architecture) pairs have concentrated on effective algorithms to implement application codes running on particular parallel computer system architectures. This paper concentrates on the architectural features of system hardware and software which, if implemented, will result in systems that are *totally scalable* in that they will have near ideal speedup for a wider range of algorithms and applications. It describes the major types of current systems, technology trends and the scalability of desirable architectural features for parallel computing systems.

Key words: high performance computers, parallel computers, computer architecture, scalability.

1. INTRODUCTION

Modern business, science and engineering problems require increasing use of high performance computers to solve larger problems within acceptable time limits or to solve fixed size problems in shorter times. Because of the performance limits of single processors, high performance computer systems must use multiple processors in parallel to achieve the needed performance levels. In the ideal case, an application running on a parallel computer system with p processors will run p times faster for a given problem size (linear speedup) or will execute a problem p times larger in the same time as a single processor system.

The above ideal case is not achievable in practice for a wide range of real problems and architectures because:

a. The algorithms used to implement real world applications are not completely parallelizable and invariably include some sequential code that must run on one or some subset of available processors.

---

*Research sponsored in part by the Phillips Laboratory, Air Force Material Command, USAF, under cooperative agreement number F29601-93-2-0001. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Phillips Laboratory or the U.S. Government.

b. The parallelizable portion of the application may not be evenly distributed among all available processors.
c. Communication, input/output and system overhead for process synchronization and sharing of partial results during execution can become significant and degrade performance with a large number of processors.

If we assume that the worker processors of a parallel system cannot compute, communicate or perform input/output simultaneously, then T_p, the total time to execute a parallel application can be written as:

$$T_p = t_p + t_s + t_o \tag{1}$$

where t_p is the time needed to execute the parallel portion of the application, t_s is the time to execute the sequential portion and t_o is the overhead time for process synchronization, interprocessor communication and input/output. If we further assume that a fraction A of the application ($0 < A << 1$) is not parallelizable (i.e., it must run serially on a single processor) while the remaining (1-A) fraction can be uniformly distributed among p processors, then

$$T_p = (1\text{-}A)\ \frac{T_s}{p} + AT_s + t_o \tag{2}$$

where T_s is the total serial execution time of the application on a single processor. The speed up, S_p, can then be written as:

$$S_p = T_s/T_p = \frac{1}{A + (1\text{-}A)/p + t_o/T_s} \tag{3}$$

As p, the number of worker processors, gets large, the speedup is limited by A and the t_o/T_s ratio, both of which should be as small as possible to achieve large speedups.

Even if an application is 99.9% parallelizable (A=0.001) and is uniformly distributed among all processors, then the maximum achievable speedup is limited to 1,000 with an infinite number of processors even if the t_o/T_s term is zero. With 1,000 processors, the speedup is reduced to only 500 under the same conditions.

The t_o/T_s term represents the communication to computation ratio and is a function of the algorithm used and architectural characteristics of the parallel system. It can be minimized by trying to localize data references, using coarse synchronization and hiding the effects of communication by overlapping it with computation if the system allows it. The effect of the communication to computation ratio can be especially important if it grows with p since in such cases the speedup can start to decrease as more processors are used. If this ratio is fixed at 0.001 for the above example, the speedup is reduced from 1,000 to 500 with an infinite number of processors and 333 with 1,000 processors.

2. FEATURES AFFECTING TOTAL ARCHITECTURAL SCALABILITY

The simple analysis and example in the previous section illustrate the need for both algorithms that can be effectively parallelized and architectures that limit the effects of communication and other sources of overhead. Since algorithms can be sensitive to machine architectures [1], scalability analysis must be done on (algorithm, architecture) pairs. The simplest definition of a scalable (algorithm, architecture) pair is that the performance (speedup) of the algorithm increases (almost) linearly with respect to the number of processors. Such studies determine the degree of matching between an algorithm/application and a computer architecture. They may be performed under different constraints on the problem size, the machine size in terms of the number processors or both.

A good parallel computer architecture should be effective for execution of a large class of algorithms/applications. In the ideal case, a scalable parallel computer architecture is one that provides linear (or near linear) speedup with an increasing number of processors for a wide range of parallel algorithms and workloads. Architectures with this property are *totally scalable.*

Total scalability, as defined above, is not achievable today. It requires scalability of many individual hardware and software features some of which we are just beginning to understand. Consequently, most of the so called "scalable" systems available today are only *partially scalable* using the above definition. Factors influencing scalability of parallel computer architectures are shown in Figure 1 and are discussed further in Section 6. Systems possessing a larger number of these desirable features are more likely to provide linear or near linear speedup for a greater range of algorithms/applications than systems with a smaller number of these features.

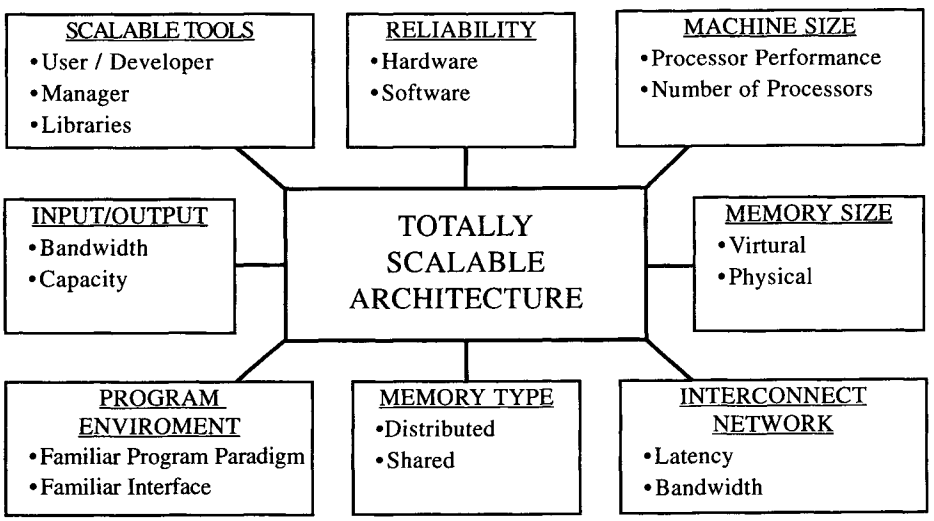


Figure 1. Hardware and system software features affecting scalability of parallel systems.

For parallel systems to be effective for the solution of business, scientific or engineering problems, they must either be able to solve important problems that cannot be readily solved by other means or be able to solve current or new problems better, faster, cheaper and more reliably than using existing systems. The latter raises a number of issues related to cost, reliability and programmability for users, developers and system managers. These are shown in Table I and are discussed further in Section 6.

Table 1
Important features needed by users, developers and system administrators

Feature	Users	Developers	Managers
Familiar user interface	X	X	X
Familiar programming paradigm	X	X	
Commercially supported applications	X		X
Standards		X	X
Scalable libraries	X	X	
Development tools		X	
Management tools			X
Total System cost			X

The sections that follow will describe the major types of parallel systems, technology trends, current limitations and scalability of the desirable features.

3. SCALABILITY OF CURRENT PARALLEL SYSTEMS

Current systems achieve high performance levels by using a continuum of small numbers of very powerful processors to very large numbers of low performance processors. For the sake of simplicity, we can divide them into coarse, medium and fine grain systems where the term grain refers to the "power" of individual processors in a parallel system, with coarse and fine grain systems having the highest and lowest single processor performance respectively. Such systems also require large amounts of memory which is usually:

a. Distributed, consisting of a collection of disjoint memories each being a separate address space or,
b. Shared, consisting of a single and very large address space with either uniform memory access (UMA) times or non-uniform memory access (NUMA) times.

Table 2 summarizes the main architectural features of coarse, medium and high grain systems while Figure 2 illustrates their growth potential in terms of single processor performance (vertical arrows) and number of processors per system (horizontal arrows). The combination of these features leads to the following observations:

a. While problems associated with building powerful processors and large shared memories make coarse grain systems relatively expensive, their low interprocessor communication overhead, high I/O bandwidth and mature software make these systems efficient performers for general purpose applications.
b. Advances in processor chip technologies imply that medium grain systems have today's highest total peak performance ratings. Their software is getting more nature and such systems can be very cost effective for applications that can explore data locality or where the dominant communication pattern is to the nearest neighbor. They show great promise in becoming the high performance workhorses in the late 1990's.
c. Relatively immature software and poor processor and interprocessor communication performance for arbitrary communication patterns does not make fine grain systems suitable for general purpose applications. However, they can be very effective for special applications that are highly parallelizable and do not require interprocessor communication, such as some image or signal processing applications.

Table 2
Typical features of various parallel systems.

Feature	Coarse Grain Systems	Medium Grain Systems	High Grain Systems
Approximate number of processors	1-32	32-2048	>2048
Relative single processor performance	High	Medium	Low
Typical memory architecture	Shared	Distributed	Distributed
Memory bandwidth	High	Medium	Low
Total peak theoretical performance	Low-Medium	Medium-High	Medium-High
Interprocessor communication bandwidth	High	Medium	Low
Interprocessor communication latency	Low	Medium	High
Software maturity	High	Medium	Low
Cost	High	Medium	Low
Growth potential - per processor performance	Low	High	High
Growth potential - number of processor per system	Low	Medium	High
Reliability	High	Medium	Medium

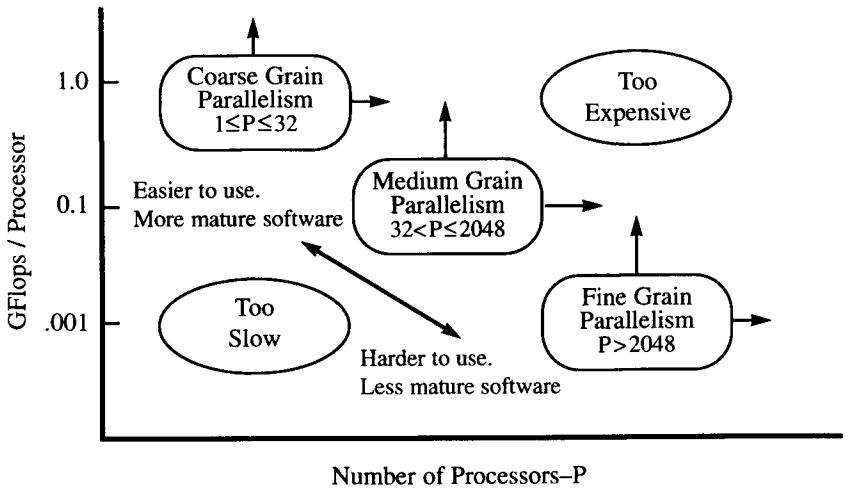


Figure 2. Trends in growth potential and ease of use. A longer vertical arrow implies greater growth potential in single processor performance while a longer horizontal arrow implies greater growth potential in the number of processors per system.

4. TECHNOLOGY TRENDS AND IMPLICATIONS FOR MEDIUM GRAIN SYSTEMS

In recent years, there have been dramatic advances in hardware and software technologies that have the potential for greatly improving the cost effectiveness of medium grain parallel systems. These advances and their implications are:

a. Single processor performance of 200-300 Mflops is commercially available today and is doubling every 15-18 months. This improves performance by reducing the number of processors for a given workload which, in turn, can reduce the communication to computation ratio.
b. The above performance is achieved using superscalar chips without resorting to vector pipes. This facilitates programming and enhances the portability of codes.
c. Memory sizes for superscalar chips are increasing. This reduces the number of processors required for memory bound applications and can result in a reduction of the communication to computation ratio.
d. The local disk space per node is increasing, facilitating the development of data intensive applications not previously possible.
e. Interprocessor communication networks are becoming denser with higher bandwidth and lower latencies that are constant. This decreases the communication to computation ratio, facilitates programming and synchronization and allows a wider range of applications to run effectively.
f. Node processors are beginning to support a *full* UNIX operating system with extensions to support the parallel environment and interprocessor communication. This provides users with a familiar interface, facilitates the porting of codes and facilitates parallelization and tuning of ported codes.

g. Compilers, parallel development and tuning tools are getting more mature. This results in faster, more efficient codes and reduces development and tuning time of codes.
h. The use of standard (widely available) processor chip sets is becoming prelevant as opposed to custom made chips. This reduces costs due to large volumes, enhances availability of commercially supported applications and facilitates portability of codes.
i. Support for heterogeneous configuration of processor nodes and more flexible partitioning of the system are becoming available. This helps optimize the configuration for given workloads and makes more efficient use of available resources.

Items a, c and d above can reduce the number of processors for a given workload. It should be noted that it is easy to scale down an algorithm from p to q processors, where p > q, with little loss of efficiency (efficiency is defined as speedup divided by the number of processors used) but some loss in speedup when the distribution of parallelizable code on q processors is more uneven than on p processors. The converse does not hold. It is *not easy* to convert an algorithm designed for a small number of processors into a faster algorithm that makes use of a larger number of processors. This, in fact, is an important challenge in parallel computation.

5. CURRENT LIMITATIONS AND OPPORTUNITIES

While the trends in hardware and software technologies bode well for medium grain parallel systems, it is important to understand some important current limitations of such systems. These limitations offer opportunities for further research and development.

5.1 Disparity between processor speed and other desirable features.

The doubling of single processor performance every 15-18 months is leading to a growing disparity between processor performance and other desirable architectural features required for scalable and balanced systems, including memory speed, interprocessor communication network performance in terms of bandwidth and latency, input/output in terms of bandwidth and archival storage capacity as well as local and wide area network bandwidths. Off the shelf main memory performance has been improving at only 10-15% per year [2] and it is only the availability of improved and larger cache memories that have helped ensure current systems are balanced. Research to improve main memory, communication network, and input/output performance is needed to prevent future system bottlenecks.

5.2 Lack of standards.

The cost to rewrite and tune major application codes for a one of a kind machine is sufficiently large to make this a major barrier. The trend towards standard processor chip sets running an extended UNIX operating system is a welcome step that will help portability of codes from desktops to parallel systems but much work is still required to establish needed standards such as portable libraries for message passing.

5.3 Lack of key high performance instructions.

The processors used for current medium grain parallel systems are the same as those used in workstations which are designed for optimum performance using the standard workstation benchmarking suites, including the SPEC floating point and integer benchmarks. Some of the newer processors include multiply and add in a single instruction to improve their performance for linear algebra codes. However, they lack instructions such as gather/scatter and n-way striding which are important for sparse matrix operations. Incorporating these into the native instruction set would significantly improve the performance of current processors for many scientific applications. Perhaps the best way to accomplish this is to expand the current benchmark suites for workstations to include performance evaluation for sparse matrix operations in order to induce chip designers to incorporate such instructions into their future products.

5.4 Lack of scalable libraries.

As described in Section 4, it is relatively easy to scale parallel algorithms down to a smaller number of processors but the converse is not true. Parallel libraries are needed that minimize the amount of serial code, scale to as many processors as possible below the point of diminishing returns, can be conveniently scaled down to smaller configurations while distributing parallel code as evenly as possible among processors, and minimize the communication to computation ratio.

5.5 The architectural scalability problem.

Many features of a parallel system must scale *in the right way* if a system is to be *totally scalable* and provide near linear speedup for a wide range of algorithms/applications. These are shown in Figure 1 and include tools and utilities for users, developers and managers for system administration, monitoring and operation. The emphasis that features should scale *in the right way* refers to the fact that, as the number of processors in a system in increased, some features should scale up (e.g., total peak theoretical processing speed), while others should stay constant or even decrease. It is highly desirable, for example, that the time to reboot an entire system does not increase in direct proportion to the number of processors but remains constant.

6. SCALABILITY OF MEDIUM GRAIN SYSTEM FEATURES

Most of today's medium grain systems are only *partially scalable* in that the various hardware and software features do not all scale in the right way to provide near linear speedup for a wide range of algorithms used to implement real world applications.

6.1 Processing speed.

In general, the peak theoretical performance scales linearly with the addition of more processors but the speedup for real workloads does not scale so well. Because of the relatively small number of processors and very low communication to computation ratios achievable for shared memories, coarse grain systems tend to have the most linear speedup and highest efficiencies over a broader range of scientific and engineering workloads. Medium grain machines are much less efficient while fine grained machines are least efficient for general purpose workloads. Note, however, that the excellent performance of today's

processor chip sets, together with their mass production, makes medium grain systems more cost effective than coarse grain systems for an increasing number of applications.

6.2 Memory size and speed.

Total memory tends to scale linearly with the number of processors. The newer medium grain systems can support up to 2 GB of local memory per processor which improves performance for memory bound problems by reducing the number of processors needed for a given workload. Larger and more effective cache memories have allowed effective memory access times to keep up with the rapid improvements in single processor performance. Increasing the memory bus width has also allowed main to cache memory bandwidths in excess of 2 GB/sec/processor using currently available systems. Nevertheless, speed improvements of only 10%/year for commodity memory chips are a cause of concern since they can result in future imbalances between processor and memory speeds as described in Section 5.1.

6.3 Interconnect network bandwidth and latency.

Shared memory systems with near uniform access times provide very high bandwidths and very low latencies for interprocess communication. However, such shared memory interconnect networks do not scale well since it is difficult to build large and fast memories that are shared by more than 16 to 32 processors. Medium grain systems with distributed memories must use a variety of interconnect networks to provide effective interprocess communication [3]. In general, such networks did not scale well for older systems in terms of both total bisectional bandwidth and latency. For some of the newer systems, the bisectional bandwidth scales linearly and the latency is almost constant. Systems with a global shared memory but non-uniform memory access times (NUMA) are beginning to appear. They have high bandwidths and relatively low latencies to reduce the interprocessor communication overhead but the scalability of this type of interconnect network is not yet proven in practice for systems having hundreds of processors. Software to support latency hiding by prefetching information whenever feasible can reduce interprocessor communication overhead even further.

6.4 Input/output bandwidth and storage capacity.

For medium grain systems, the I/O bandwidth does not scale well [4]. While peak processing performance has improved by a factor of 4-5 over the past three years, I/O bandwidth to mass storage has improved by only a factor of two or less. The growing disparity between processor speed and I/O bandwidth imply that the effectiveness of teraflop parallel computers will be severely limited unless I/O support and performance are significantly improved. By the end of 1994, some installed systems will have 50-80 GB of distributed memory and checkpointing a large job on such a system in 100 seconds or less will require a bandwidth of 4-6 Gbps, not an easy task to accomplish.

The mass storage requirements for the next generation of applications at the Lawrence Livermore National Laboratory, for example, will generate 10-80 GB of data/hour/run and will require up to 10,000 GB/run for long term storage [5]. Much work is needed to help solve such storage problems including research in scalability of various I/O architectures, software to provide users with more effective control over I/O and determination of I/O communication patterns over the interconnect network between compute and I/O nodes since

little is known about the characteristics of such traffic for conventional scientific and engineering workloads. Current trends are to:

a. Use internal parallel I/O systems directly connected to the high reliability, high bandwidth interconnect network since it provides scalable bandwidth.
b. Use parallel striping or file debustering. In disk striping, *parts of each block* are distributed among the disks. In file debustering, *different blocks* of a file are stored on distinct disks and read or written in parallel.
c. Provide users, who know what data will be used when and on what processor, with more control over how and where the data is to be distributed across the I/O devices. This could be done by means of appropriate compiler directives.
d. Integrate parallel file systems with tera or petabyte mass storage systems.

6.5 Scalable system software.

To take advantage of scalable hardware, all operating system and application software must also be scalable. Current programming practice for most parallel system is based on the *data parallel model* of computation which scales well. Large *functional* parallelism, where hundreds of processors are running functionally distinct parts of a job, is relatively rare today. Future architectures must effectively support environments that facilitate both functional and data parallelism in a massive way. Finally, there is a perception among some in the parallel system user community that increased difficulty of programming is a necessary price to be paid for the benefit of achieving high performance. The challenge in scalable system software is to show that this need not be so. Efforts currently under way to help resolve these issues include:

a. Design of scalable libraries that provide a corresponding performance improvement as the number of processors increases for a given problem size in a way that is transparent to the user [6]. It should be noted that the range of scalability is a function of both the problem size and the number of processors.
b. Development of compilers that allow programmers to specify data partitioning and automatic generation of the message passing code for that partitioning. Without such tools, this must be done by hand coding the necessary message passing commands.

Other problems that need to be addressed include the extension of virtual memory support on individual nodes to provide a *global virtual memory* for entire systems [7] and the linking of separately compiled programs. Currently, there does not seem to be any standard for communicating large partitioned data structures between compiled modules. Finally, there is a need to develop *scalable and more usable* tools (e.g., for code development, debugging and tuning) for parallel systems. Tools that might not be needed for a system with a few processors can become extremely important to fully exploit systems with hundreds of processors. The information such tools provide must be displayed in a *way that is intuitive to the target user.* Displaying communication patterns on a job running on several hundred nodes using conventional GUI interfaces may not meet that criterion.

6.6 Reliability and fault tolerance.

Business enterprises require that hardware and software meet industrial robustness standards. Incorporating reliability and fault tolerance into parallel systems is mandatory to ensure downtime does not scale with the number of processors in the system. Single points of hardware or software failure need to be avoided wherever feasible and scalable tools are needed to monitor all resources, detect faults and take corrective action. Hardware and software mechanisms for "graceful" degradation rather than a hard crash need to be developed and failed system components should be fixed and brought on line without having to bring the entire system down. Finally, more effective checkpoint and restart packages need to developed for parallel systems to allow restart of jobs affected by a hardware or software fault.

6.7 System, resource and job management.

Growth in the number of users, applications and processing nodes in a system require the development of scalable and effective tools for allocating system resources such as processors, memory and storage hierarchies. As the number of system resources grow, their effective management becomes even more critical, as is the need for a scalable repository of configuration data.

7. CONCLUSIONS

Architectural limitations of traditional coarse grain parallel systems using very high performance vector processors and the lack of generality for fine grain systems imply that, for the rest of this decade, most computational performance increases will come from medium grain systems. The performance increase will be due to both an increase in the performance of individual processors, which is doubling every 15-18 months, and from increased parallelism. The trend is clearly away from custom processors to commodity processors used in high performance workstations. The use of commodity (and therefore high volume) processor chip sets will help drive down the cost of parallel systems, help the portability of new applications from the desktop to the parallel system and encourage more users to use both kinds of systems by providing the same familiar user interface on each. This, together with a better understanding of the hardware and software features needed to make parallel systems more effective and scalable for a broader range of real world applications, will result in new generations of parallel systems that will find broad acceptance in business, science and engineering. They will be able to meet more effectively the criteria needed by industry - ease of use, ease of management, reliability, broad availability of commercially supported software and greater cost effectiveness. Consequently, they will be increasingly seen as effective production tools rather than merely research tools.

REFERENCES

1. A. Gupta and V. Kumar, The Scalability of FFT on Parallel Computers, IEEE Transactions on Parallel and Distributed Systems, 922-932, August 1993.

2. National Science Foundation Workshop on High Performance Memory Systems - Final Report, University of virginia Computer Science Report CS , 93-35, June 1993.

3. G.S. Almasi and A. Gottlieb, Highly Parallel Computing, Benjamin/Cummings, Redwood City, 377-409, 1994.

4. J. M. del Rosario and A. N. Choudhary, High-Performance I/O for Massively parallel Computers - Problems and Prospects, IEEE Computer, 59-68, March 1994.

5. Personal Communication, National Storage Laboratory, Lawrence Livermore National Laboratory, Livermore, California.

6. L. M. Ni, H. Xu and E. T. Kalms, Issues in Scalable Library Design for Massively Parallel Computers, Tech. Report, MSU - CPS - ACS - 82, Department of Computer Science, Michigan State University, March 1993.

7. H. D. Siegel et al., Report of the Purdue Workshop on Grand Challenges in Computer Architecture for the Support of High Performance Computing, Journal of Parallel and Distributed Computing, No. 16, 199-211, 1992.

II. HARDWARE SYSTEMS

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Trends in High-Performance Computer Architecture

Jean-Loup Baer[a*]

[a]Department of Computer Science and Engineering, University of Washington, Seattle, Wa, 98195, U.S.A.

Technological advances have led to microprocessors running at 300 MHz, to memory capacities of hundreds of megabytes per processing element, and to parallel processors with a few hundred nodes. In this paper, we review advances in processor design, techniques to reduce memory latency, and means to interconnect powerful nodes. In the case of processors, we emphasize the implementation of techniques such as speculative execution based on branch prediction and out-of-order execution. We describe alternate ways to handle vector data. While memory capacity is increasing, so has, relatively speaking, its latency. We describe how add-ons to caches and cache hierarchies can help reduce the memory latency. In the case of shared-memory multiprocessors, we show how relaxing sequentiality constraints is one way to reduce latency. In the case of interconnects, there is no definite trend for topology at this time. It appears that the real problem is latency, not bandwidth.

1. INTRODUCTION

Over the last 15 years, we have witnessed radical changes in high-performance computing. Improvements have come from two directions: faster and more memory intensive single processors on one hand, and parallel processing on the other. Technological advances in VLSI have led to microprocessors operating at 300 MHz with nearly 10 million transistors on chip. Parallel systems with several hundreds of full-fledged processors and others with close to 100,000 serial processors have been operational for several years. In this introductory section, we consider the improvements in single processor performance and in parallel systems. We then dwell in more detail on processor design (Section 2), memory hierarchies (Section 3), and interconnection structures (Section 4). Finally, in our concluding section, we will describe our views on what is needed for successful high-performance systems of the future.

Single processor performance

Microprocessors of the next generation will have clock rates around 300 MHz. We should

*This work was suported in part by NSF Grant CCR-94-01689

see soon processors with a peak rate of 500 MFlops. Computational units will be superscalar with about 10 functional units (integer, floating-point, branch, load-store), some of them with deep pipelines, and the capability to issue (out-of-order) 4 to 8 instructions per cycle. Speculative execution, i.e., the ability for the processor to continue executing the predicted path of a branch before the branch is resolved, will be mandatory to efficiently use these resources. This requirement translates into sophisticated branch prediction and recovery mechanisms. Significant memory resources – register files, instruction and data caches – will be needed on-chip to sustain the instruction issue rate and the data hungry functional units. These attributes look strikingly similar, and in fact slightly more comprehensive, than those of a "supercomputer" (e.g., Cray Y-MP, NEC SX) minus the vector units.

Memory densities continue to increase so that the capacity of SRAM and DRAM chips quadruple every three years. 16Mbit SRAMs and 64Mbit DRAMS are available now. However, it is a well-known fact that decreases in memory latencies are progressing at a slower rate than increases in processor speeds. Thus, memory hierarchies with as many as 4 levels (not counting disks and archival storage) are starting to appear. The first two levels are caches on-chip. The next level is an external cache of megabyte capacity and access time an order of magnitude slower than a processor cycle. Main memory capacity can be over 100 Megabytes with an access time a few multiples of that of the external cache although synchronous SRAM and paged DRAMs operating in sequential mode have a latency approaching that of the external caches.

The capacity of external storage devices has also progressed significantly and it is predicted that density will increase by 50% per year for the next four years. Personal computers come with hard disks of several hundreds Megabytes. File servers are delivered with tens of Gigabytes of storage. However, there is no real decrease in latency or increase in bandwidth. RAID's (Redundant Arrays of Inexpensive Disks) and caching, at all levels of the memory hierarchy, might help close the secondary memory gap.

Parallel Systems

The increase in performance of single processor systems has certainly be dramatic. Nonetheless, a single processor has neither the computing power nor the memory capacity to solve problems such as those defined as Grand Challenges. Similarly, a single processor cannot manage the data requirements (security, real-time deadlines) of large commercial installations or of multimedia applications. Parallel systems, whether they are of the "supercomputer" variety or whether they are ensembles of high-performance processing elements (processor + memory) are needed.

One of the tenets of parallel processing is that it should be used not only to solve problems faster but also to solve larger problems. Looking at the latest entry in the market of large vector computers, the NEC SX-4, we see not only very large peak rates for the floating-point pipeline units but also a tremendous amount of main storage when the system is fully configured. To wit, the SX-4 can have up to 512 tightly coupled processors addressing a physical memory of up to 128 Gigabytes. However, what really differentiates a single processor SX-4 from a conventional high-performance micro is not so much the speed or the amount of memory but the presence of vector units, the possibility of tight-coupling

to other processors, and the cost of the system.

Since there is such a difference in cost between a vector processor and a micro, the natural idea is to link together several (many?) micros to form a (massively) parallel processing system. There are several obstacles to a successful parallel processor built upon these premises. Some of them are technical, for example (we will uncover more in the rest of this chapter): choosing the right interconnection structure so that the system can be easily expanded; building a software architecture (operating system, applications) that can scale with the system – this entails the selection of the right granularity for decomposing the applications, the right primitives for synchronization etc. –. Some of them are economical: since cost is an issue, the parallel processor should also service the common single users, unlike a vector computer that is known to be dedicated to a set of applications; since microprocessor technology is moving so fast, the individual processors of a parallel system should be easily replaceable by the next generation of the same family.

This set of constraints and the associated costs dampen the enthusiasm for Massively Parallel Processors with thousands of nodes. It is symptomatic though that Cray Research, Convex, and IBM, all companies that have manufactured vector processors of one form or another, are offering or are planning to offer products based on the interconnection of commodity high-performance microprocessors.

What performance is needed?
A few years ago, the goal for personal computing was the GigaPC: 1 GIPS (10^9 instructions per second), 1 GByte of main memory, and a few GBytes of disk. This was deemed sufficient for applications such as real-time speech understanding, "expert" applications in business, and maybe also for Computer Aided Design in many fields. The next, or maybe subsequent, generation of workstations will have these characteristics. Of course, at that time, the user will ask for more, for example because of the computing, communication, and storage requirements of multimedia applications.

In the realm of large parallel systems, we are still far from having systems with the power to solve some of the Grand Challenges. There is a need for computational power (Teraflops), for software to take advantage of parallelism without too much overhead (recall Amdahl's law), and for the ability to manage enormously large scientific and textual databases. The cost of such systems looms as a formidable short-term deterrent.

2. PROCESSOR DESIGN

A program execution speed is dictated by three components: number of instructions, cycle time, and CPI, the average number of cycles per instructions. The current trend, unlikely to be reversed, is to have Reduced Instruction Set Computers (RISC) rather than Complex Instruction Set Computers (CISC). Compilers generate more instructions for RISC's but the simplicity of the instruction set allows more efficient implementations thus reducing the CPI. Choosing the cycle time does not depend uniquely on advances in technology. The choice is also influenced by the amount of computation done at each stage of the pipeline. The compromises are between fast cycle times and deep pipelines

versus fewer stages in the pipe but more computation being performed in each stage.

The RISC design philosophy is not new. It dates back to the Control Data 6600 [1]. Similarly, pipelining can be traced back to the IBM Stretch and its commercial successor the IBM 360/91 [2]. These machines, that are close to be 30 years old, had other features that are often touted as "advanced" nowadays: multiple functional units, out-of-order issue of instructions, (limited) speculative execution, renaming registers etc. The difference is that all these features are now included on a single chip.

A fast clock is the technological foundation that leads to low cycle time. The job of the designer is to take advantage of the large number of transistors available on a single chip to incorporate the functional requirements for a high-performance processor. Stated differently, the goal of the designer is to attain a CPI as small as possible. A secondary goal is to have an architecture that is expandable. These goals can be translated into the following requirements:

1. *Speed*: The goal is to keep the instruction stream flowing in spite of control hazards (with branch prediction), data dependencies (with out-of-order execution), and instructions whose executions takes multiple cycles (with specialized units). Furthermore, fine grain parallelism within the instruction stream should be exploited (with superscalar CPU's).

2. *Handling of vector data*: Scientific applications generate a large amount of ordered accesses to large data sets. Vector processors, with vector instructions and registers, are well suited to that task. Since neither of these specialized instructions or registers are included in the instruction set architecture of high-performance microprocessors, means of prefetching the data in caches will be needed. Prefetching can be software-based with special instructions or be assisted by specialized hardware units.

3. *Reduction of the memory latency*: An extensive memory hierarchy, starting with an on-chip hierarchy of caches, is only a partial solution to this daunting problem.

4. *Support for operating system and expansion to multiprocessing*: Access to the lower levels of the memory hierarchy is faster if it is done with virtual addresses or if the translation between virtual and physical addresses can be done very fast. In the latter case, the TLB (or MMU) design is closely related to the determination of the critical path to achieve minimal cycle time. Adequate interface to the external cache must be included for input-output and extension to multiprocessing

All modern processors are pipelined. With perfect pipelining, discarding the start-up effects, and a single functional unit the CPI would be 1. With instruction level parallelism, i.e., multiple functional units and issue of several instructions in the same cycle, the CPI could be less than 1. However the hazards mentioned above and the effects of cache misses will prevent such ideal conditions. We describe now how one can mitigate the effects of branches and data dependencies so that the CPI is not increased unduly.

Branch prediction

Breaks in control flow – conditional and unconditional branches, subroutine calls and re-

turns – occur about once every five or six instructions in integer programs and slightly less frequently in floating-point ones. Conditional branches are the most common, about 80% of the breaks. In scientific programs, backward branches (loops) dominate while in integer programs there is about the same number of forward and backward branches. Backward branches are almost always taken while forward conditionals have a 50-50 chance of being taken but with an almost bimodal distribution of almost always taken or almost always not taken.

To see how branches impact the CPI, assume a 5-stage pipeline in a single functional-unit CPU where, except for branches, pipelining is perfect. Assume further that it takes 3 stages to recognize that an instruction is a conditional branch and to resolve the condition. Upon recognition of a branch, the pipe is frozen, i.e., no further instruction is issued until the branch is resolved. With a conditional branch frequency of one out of six instructions, the CPI contribution of branch handling is $1/6 \times 3 = 0.5$, i.e., a third of the execution time will be spent waiting for branches. If now we have longer pipelines and a superscalar CPU, the relative contribution of branch handling under this frozen pipe policy will become much larger. The need for some other branch handling policy is evident.

The reduction of the CPI penalty caused by branches requires knowledge of one or more of the following[3]: (i) is the instruction a branch, (ii) is the branch taken or not taken, (iii) if taken what is the target address, and (iv) what is the instruction at the target address.

A *Branch Table Buffer* (BTB) is a hardware structure that can take care of these requirements, albeit at a serious cost (real estate on the chip) if done in a complete way. A BTB is organized as an instruction cache with each entry, set generally when the corresponding branch is taken for the first time, consisting of the branch instruction address, a field used for prediction, and the target address; even the instruction at the target address could be included but the price to pay, an extra 64 bit/entry for a single functional unit, or several 64-bit entries for a superscalar architecture, is much too costly for current BTB's. The BTB is probed at every cycle. If the current value of the Program Counter (PC) corresponds to an entry, i.e., the instruction is a branch, then the prediction field indicates whether the branch should be taken (i.e., the PC is set to the target address), or not taken, in which case the execution proceeds sequentially. If the prediction was incorrect, a condition known when the branch is resolved, then the pipeline is flushed and the path of execution resumed on the correct path. Also, the prediction field is updated. This scheme does not present too many difficulties in a single unit CPU.

In a superscalar scheme, a BTB should be probed for all instructions issued in the same cycle. This means searching the BTB in parallel, which is possible only if the number of issues is small (e.g., two in the Intel Pentium that has a dual-ported BTB). If the issue rate is larger, then one can wait till the decode stage and allow only one branch to be issued in a given cycle. This results in a penalty of one cycle if the branch is (correctly) predicted taken. Instead, several microprocessors (e.g., the MIPS R10000) include in their design a predecode unit that partially decodes instructions when they are loaded in the instruction cache. When instructions are fetched from the cache, the predecoding directs the instructions to their assigned functional unit. At the same time as an instruction is

directed towards the branch unit, the BTB can be probed.

The range of prediction mechanisms varies from very simple static schemes to elaborate dynamic implementations. The rationale for static schemes is provided by the statistics given above. For example, in the HP PA-RISC, backward branches are predicted taken and forward branches are predicted not taken. This method can be accompanied by a compiler optimization, known as filling the branch delay slot, to insert an instruction following the branch so that delays will be minimized in case of misprediction. Unfortunately this method does not carry over to superscalar architectures since, in general, only one delay slot can be filled successfully by the compiler. The basis for most dynamic schemes is a 2-bit (4 states) up/down counter without overflow. The counter can be the prediction field in the BTB (e.g., Intel Pentium, IBM Power PC), or there can be a Branch Prediction Table (BPT) of counters that will be accessed by the lower bits of the PC (or some hashing function), or the counter can be associated with a cache line in the instruction cache (DEC Alpha, MIPS R8000) and a target address can also be included in the cache (UltraSparc). The prediction accuracy can reach 90%. Various proposals have been made to correlate branch prediction with the recent history of the branch outcomes since often consecutive branches are not independent. In these so-called two-level schemes [4] the first level is a k-bit shift register (or a table of shift registers) which records the outcome of the last k branches and the second level is a table (BPT) of 2^k 2-bit counters which, when indexed by the shift register, yields the prediction for the current branch. This whole mechanism can be coupled with a BTB for finding the target address of a predicted branch. Simulations have shown prediction accuracies of over 95%, and up to 98% for scientific programs, but the price to pay (logic for updating that does not slow down the pipeline, BTB, and BPT) is significant.

Note that in some cases, branch prediction can be totally accurate if the condition can be computed and stored several cycles ahead. The IBM Power PC, with condition code setting, takes advantage of this feature for a "zero" execution time branch (but condition codes have disadvantages also since they are part of the processor state).

With branch prediction, parts of the program are executed speculatively. In some processors like the MIPS 10000 and IBM PowerPC 620 speculation can be as much as 4 branches deep. What happens if the speculation was wrong? First, any state information that was stored on the incorrect speculative path has to be removed and the correct state has to be restored. This is relatively easy when registers or memory are not modified until branches are resolved. However, this requires in-order execution and, even so, prevents a load to be issued at the same time as a branch since the loaded register might be modified in the same pipeline stage as the one resolving the branch on which it depends. A solution is to save important state information in "shadow" registers and to keep track of which registers are current. This is facilitated by register renaming, a technique pioneered in the floating-point unit of the IBM 360/91, that allows more hardware registers than included in the instruction set. Second, the instructions have to restart from the point where the first mispredicted branch occurred. The pipeline has to be flushed, the prediction tables have to be corrected (in general they are speculatively updated), and the correct instructions have to be fetched. To speed-up this last action, the MIPS 10000 has a resume buffer

that stores the sequential instructions that were bypassed when a branch was predicted as taken.

Out-of-order execution

In early supercomputers with multiple functional units, such as the CDC 6600 and IBM 360/91, instructions were issued in order and could terminate out-of-order. The data dependencies were checked through either a *scoreboard* [1], i.e., a map of registers and functional units determining whether an instruction could proceed to its execute stage, or were queued in *reservation stations* [2] attached to functional units and searched associatively for readiness of execution. These two concepts are, or will be, present in several microprocessors of this and the next generation (HP PA-8000, Power PC 620, MIPS 10000, SPARC Thunder) in various forms. For example, there can be a large window of entries corresponding to decoded instructions which is filled from the instruction cache at every cycle (unless the window is full). Each entry contains enough information for dispatching to the right functional unit, to ascertain when it is ready to do so, and to identify the location of the result. In some sense, the window is a global reservation station with scoreboarding information. Another design is to have the reservation stations be buffers (not FIFO queues) associated with each functional unit and to have busy bits associated with registers indicating their availability. In both cases, priority is given to non-speculative instructions.

It should be clear from the above that implementing deep speculation with high prediction accuracy and out-of-order execution is difficult and hardware-intensive. Moreover, the amount of decision making that has to be done in a single cycle, e.g., finding an instruction ready to be executed in a large window, is not trivial and might slow down the cycle time. Similarly, repairing a wrong prediction can be quite costly. Another design philosophy, followed in the DEC Alpha, is to keep the design simple and reduce the cycle time to a minimum. For example, the Alpha 21164 which has currently the fastest clock among all microprocessors, is a superscalar design where four instructions can be issued at a time but only those in the same cache line, keeps in-order execution, and has a rather simple prediction table, a 2-bit counter associated with each instruction cache line.

Handling vector data

None of the current microprocessors have special facilities to handle vector data although some are designed specifically with floating-point intensive applications in mind (e.g., the MIPS R8000) or provide graphics instructions and data types for multimedia applications (UltraSparc). There are several reasons for the lack of vector instructions and supporting hardware. First, vector architectures take a lot of real estate on the chip – in addition to the vector computation units, they use vector registers that are as expensive, bit-wise, as a cache. Second, including vector instructions in a standard RISC instruction set architecture after the fact is a daunting task. A third reason would be the difficulty of interfacing the vector architecture with the superscalar features mentioned above. And finally, a single processor vector computer by itself is not that interesting. Several of them have to be tightly-coupled together and they require superlative memory bandwidth.

Consider the computing power and register storage in vector units of two of the entries in the supercomputer arena circa 1990. The CRAY Y-MP can have 2 vector-vector and

2 vector-scalar units per processor and the NEC SX-3 has also 4 computing units (these numbers have doubled in the latest generation). The amount of register storage for vector registers is $8 \times 64 \times 8 = 4$KB for the CRAY and twenty times more for the SX-3. Of course there are also scalar registers and, in the case of the SX-3, a 64 Kbyte cache. In the CRAY 16 processors can access up to 2 Gigabytes of memory through a cross-bar network (which in itself is a very expensive item!). The NEC SX-4 has been announced as expandable up to 512 nodes but the price is ... unavailable at this time.

An alternative is to have a vector co-processor coupled with a standard microprocessor. Some of the drawbacks mentioned above disappear: the feature is optional and the cost is not as great. However, some remain, namely modifications to the instruction set; logic "glue" to interface with the processor with the need for the logic to be easily changeable when a new, faster, processor is released; and redesign of the memory interface to support streams of data to and from the co-processor. This vector unit approach exists in the TMC CM-5. Assumedly, there were many implementation difficulties.

At this point, it might be useful to look at Amdahl's law applied to vector processors [5]. Let f be the vectorization ratio of the application, that is the fraction of the scalar running time where vectorization could be applied, and let r be the vector/scalar speed ratio, i.e., the ratio of execution time to process one element of a vector in the scalar unit over the time to do the same computation in a vector unit. Then the speedup P is:

$$P = \frac{r}{(1-f) \times r + f} \tag{1}$$

This is shown in graphical form in Figure 1. As can be seen even with half of the computation vectorizable and fast vector units, the speedup does not reach 2. Only when vectorization reaches 90% or more, do we see a significant speedup. Of course, there are applications where this is possible but this restricts vector supercomputers to very specific tasks.

Vector units speedup computations because they necessitate only one instruction per (sub)vector operation and they fetch chunks of data at a time. This latter feature can be emulated in a standard processor by *prefetching* data in the cache. We have proposed several variations of a hardware-based prefetching scheme in [6]. The essential hardware component is a support unit for a conventional data cache whose design is based on the prediction of the instruction execution stream and associated operand references in load instructions. A reference prediction table (RPT), organized as an instruction cache, records the referencing patterns. Each entry, indexed by the corresponding load instruction address, includes the effective address of data access and the stride information that is computed dynamically. A 2-bit state automaton indicates whether prefetching should be initiated or not. In the most efficient scheme, the RPT is accessed ahead of the regular program counter by a lookahead program counter (LA-PC). The LA-PC is incremented and maintained in the same fashion as the PC with the help of a dynamic branch prediction mechanism. We found that such a scheme is quite beneficial for scientific programs, mostly when the latency between the cache where prefetching occurs and the next level of the memory hierarchy is small.

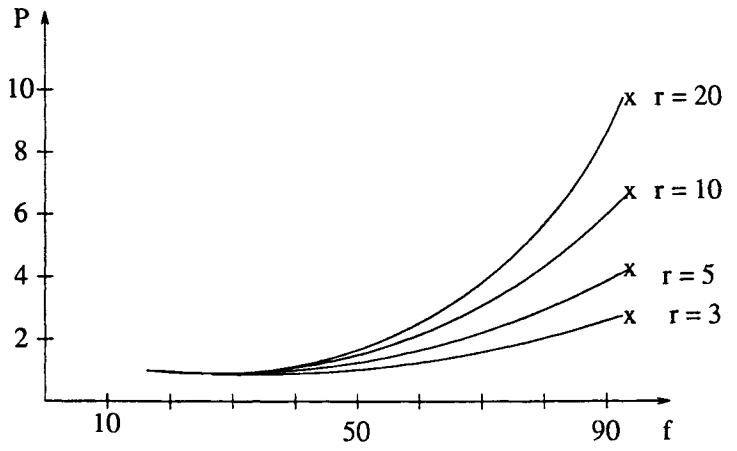


Figure 1. Speedup P in a vector processor in function of the vectorization factor f for given vector/scalar speeds r.

In this section, we have reviewed recent developments in processor design. Although conceptually, there might not be that many new advances, putting all these features at work together without sacrificing speed is an implementation tour de force.

3. MEMORY HIERARCHY

The gap between processor and memory speeds, the instruction level parallelism found in high-performance microprocessors, the large working sets required to solve many scientific applications, and the concurrent distribution of data to many processors put enormous demands on the memory system both in terms of latency and bandwidth. In this section, we concentrate on techniques to tolerate memory latency.

The introduction of caches, in the late sixties, was motivated by the goal to balance processor speed and memory access time. Caches have been extremely successful since they have provided significant performance enhancements while remaining completely transparent to the software. With larger caches though, the cache access time has in turn become disproportionate with the processor speed. Today, the cost of memory access is significant enough so that multi-level cache hierarchies and hardware assists (e.g., write buffers, victim caches) are common. In shared-memory multiprocessors, cache coherence protocols must be present since several caches can contain their own copy of a particular memory location. Thus, while until recently the introduction of a cache had been an organizational matter, it is has become now an architectural decision; caching is not always oblivious. A system approach must be taken for the management of the cache hierarchy either at the instruction level (e.g., flush, fence, prefetch, and poststore instructions), or

at the algorithm/compiler level (e.g., tiling, prefetching), or at the programming model level (relaxed consistency).

In the remainder of this section, we consider some of these issues first in the context of single processors, and then by looking at shared-memory multiprocessors.

Add-ons to single processor caches

We first review features designed to decrease write latency and read latency, and to increase the "associativity" of a cache. We then briefly consider multilevel caching.

1. Increasing write hits; decreasing write latency.
The primary goal for cache enhancements with respect to writes is to insure that write traffic to the next level of the memory hierarchy will not slow down the CPU. The *write-back* policy is superior in that regard (fewer writes) but *write-through* has advantages in terms of reliability and ease of implementation. High-performance systems use a write-back cache at the interface cache-main memory. At the lower level of the cache hierarchy, on-chip write-through (e.g., Alpha 21064) or write-back (e.g., Pentium) caches are both possible (or even the choice of a policy is possible).

On a write miss, we can choose between several policies that affect latency. A cache line can be allocated or the cache can be by-passed. If a cache line is allocated, the corresponding line in memory can be fetched or ignored. The *write validate* policy that allocates a new line but does not fetch it from memory, relying on valid bits per entity being written rather than a single valid bit for the whole line, has been shown to perform best [7].

In general caches have *write buffers* associated with them. These buffers store, in FIFO order, data that has to be written back to memory. With a write buffer, checks must be made on read misses to see whether the requested line is in the write buffer. A coalescing write buffer is one in which writes to an address already in the write buffer are combined within the buffer. Note that there is a tension between writing entries in the coalescing buffer to memory at a fast rate (little coalescing takes place) and allowing much coalescing to happen by keeping the buffer as full as possible (but then the processor will stall on stores with full buffers and no coalescing). To overcome this dilemma, coalescing write buffers can be extended to become *write caches*, i.e., small caches with a write miss allocate, write-back, no allocate on read miss strategy.

2. Decreasing read latency. Lock-up free caches

In most current microprocessors, the CPU stalls on a cache read miss. Although the concept of *lock-up free* caches, i.e., caches that allow several outstanding read miss requests (write latency is taken care of with write buffers), was proposed as early as 1981 [8], they have not been widely used because of their implementation complexity. Restricted forms of non-blocking loads have existed for some time, such as the "hit under miss" policy in the HP PA7100 whereby one outstanding read miss is allowed. The most advanced current implementation of lock-up free caches is found in the DEC Alpha 21164 where up to 6 read misses can be outstanding between the first level and second level caches that are both on-chip.

To make a regular cache lock-up free, a set of MSHR's (Miss Status Holding Register) is added, with one such MSHR per outstanding read miss that will be allowed. Each MSHR must hold a valid (busy) bit (if all MSHR's are busy, the CPU will stall), the address of the requested cache block, and a comparator so that further misses to the same block do not acquire another MSHR. The cache index can also be kept although it can be recomputed when the data comes back from the next level in the memory hierarchy. This is sufficient if the loaded data passes through the cache first as, e.g., in the prefetching scheme described previously. If the request is to be forwarded directly to the CPU, then for each possible word in a cache line the MSHR must contain a valid bit (in case of writes), the destination address (a register), as well as the type of load. The MSHR's can be extended to handle loads of different bytes of the same word or multiple loads of the same word to different registers.

There is a direct interaction between code scheduling and the added value of lock-up free caches. Loads should be scheduled as far away as the latency to the next level in the memory hierarchy, although in practice it has been found that there is little incentive to go beyond 10 instructions. Methods such as trace scheduling, loop unrolling, and/or enhancements to traditional list schedulers are to be used to take full advantage of lock-up free caches.

3. Increasing the on-chip cache "associativity"

Direct-mapped caches have the advantages of simplicity of design and have slightly faster access times. They have been the usual organization for small on-chip caches but the trend now is more towards set-associative caches. Set-associative caches are to be preferred if their access does not slow down the cycle time. The two main reasons are: (1) the number of conflict misses is significantly reduced when passing from a direct-mapped cache to a, say two-way, set-associative one of same capacity, and (2) fast address translation requires that the cache capacity be limited to the page size multiplied by the set-associativity thus limiting direct-mapped caches to the capacity of a single page. With very fast cycle times (DEC Alpha) we have direct-mapped (8KB) first level caches. With slower cycle times, we have –at the extreme of the spectrum – large set-associativity (32KB, 8-way set-associative in the IBM Power PC). It is worth noticing that the 3 stages devoted to cache access in the Alpha take less time than the 2 cycles needed in the Power PC.

There have been several ideas proposed for retaining direct-mapped caches while reducing conflict misses. For example, the cache can be split in two. The basic idea is as follows (for more detail, see [9]). The first half is accessed using the standard hashing function. In case of a hit we are done (1 cycle). In case of a miss, we rehash and access the second half (one more cycle). If this second access results in a hit, the tow locations are swapped (1 or 2 cycles) and the data is forwarded to the CPU. In case of misses in both halves, data has to be fetched from memory. A related idea, implemented in the off-chip cache of the MIPS R10000, is to use a set-associative cache and to access first the most recently used line in a set. This information is kept in a separate table. Yet another promising design, for small caches, is to place a small fully associative buffer "behind" the cache and "before" the next level in the memory hierarchy. This *victim cache* [10] keeps the lines most recently evicted from the regular cache. On a miss to the regular cache, the victim

cache is probed and if it contains the missing item, that item is swapped with the line that is to be replaced.

For large caches, the operating system can help in reducing conflicts by using *page coloring* when mapping a virtual page to a physical frame. The system tries to select a frame from the available pool such that the bits used to index the cache match those of the virtual address of the faulting page. When it is successful, cache conflicts between sequential virtual pages in the same process are avoided. Inter-address space conflicts can also be avoided by further hashing on process identification (PID) bits.

Multilevel Cache Hierarchies

With one exception, the HP PA-RISC, all current high performance processors have an on-chip, or first level L1, cache. It is backed-up by an off-chip L2 cache. In general L1 is in fact two caches, an instruction cache and a data cache while L2 is unified. With increasing "real estate" available on-chip, the hierarchy can be extended to L1 and L2 on-chip, see, e.g., the DEC Alpha 21164, and L3 off-chip.

An interesting question is whether the contents of L1 should be a subset of those of L2, if the latter is much larger, and so on in the hierarchy if there are more than two levels. If the L1's are write-back, the subset property means that there is room in L2 to write-back the contents of dirty blocks in L1 without replacing an L2 block that covers an L1 block. This *multilevel inclusion property* (MLI) becomes particularly important in multiprocessors where L2 could be the parent of several L1 children. The main reason is that L2 will shield the L1's from interferences due to I/O and cache coherence. Imposing MLI requires first a replacement algorithm that takes into account the number of valid children of a given parent. When a parent has no valid children, then it can be replaced. Second, MLI restricts the design of L2. In the case of set-associative caches, still under the replacement algorithm described above, the general result is that the degree of associativity of the parent cache must be at least as large as the product of the number of its children, their set associativity, and the ratio of block sizes [11]. This has serious implications since the degree of associativity might become unduly large. Instead, one can use *partial inclusion*, whereby when a line in L2 is to be replaced and it is known (via a state bit/line) that there are corresponding lines in some of L2's children, the latter are notified (e.g., by broadcast in a bus-based system) and their corresponding lines are invalidated.

Multiprocessors. Relaxed models of memory consistency

A number of features that are present in single processors must be used with caution in shared memory multiprocessors. The reason is that they destroy sequential consistency, the programming model that is assumed by most programmers. Under sequential consistency, the execution of a program appears as some interleaving of the parallel processes. In particular, each shared memory access must appear as being visible to the whole system before the next shared access is started and these accesses, for a given processor, are executed in program order. Features such as write buffers, out-of-order execution, lock-up free caches clearly do not follow these rules. To see how important this is consider the program of Figure 2 where sequential consistency can be destroyed if write buffers are used.

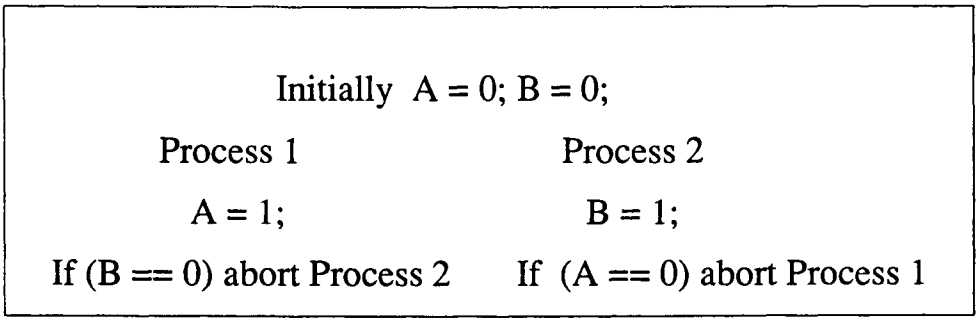


Figure 2. In this program, under sequential consistency Processes 1 and 2 cannot both be aborted. If write buffers are present and reads can bypass writes, then both processes 1 and 2 could be aborted.

The disadvantage of sequential consistency is that memory latency is totally exposed at each shared memory access. Various weak [12], or relaxed models, of memory consistency [13] have been proposed. For example, *released consistency* [14] which requires properly "labelled" programs distinguishes between access to synchronization variables through Acquire and Release primitives that have to be accessed in restricted order (almost as restricted as sequential consistency), and ordinary accesses whose only access restriction is through data dependencies. It has been shown in several studies that release consistency provided important performance benefits relative to sequential consistency while not restricting unduly the programming model.

In this section, we have reviewed a sample of techniques used to tolerate memory latency. Our bias has been towards hardware assists but we need to recognize the ever larger importance that algorithm design, e.g., tiling of matrices in linear algebra, compiler optimizations, e.g., loop exchange to increase data locality and data placement in multiprocessor systems, and operating system decisions, e.g., migration of data pages, have on the performance of the memory hierarchy of high-performance single and multiprocessor systems.

4. COMMUNICATION AND SYNCHRONIZATION

Single processor power and memory capacity are still increasing at an amazing rate. It is clear nonetheless that many scientific problems require several orders of magnitude more processing power and much more database massaging capability than is available on the current fastest uniprocessor supercomputer. Harnessing the power of several computers so that they can cooperate to solve large and compute intensive programs, i.e., parallel processing, has been present for some time now. Parallel processing requires the physical implementation of a switch, or interconnection network, to link physically the various processing elements (processors and memory modules), a means for logical communication

between processes running on the various processors, and synchronization primitives so that the processes can coordinate their actions.

Most often, the vector supercomputers provide a switch with maximum parallelism, namely a crossbar. The drawback is that such a switch has an $O(n^2)$ expansion cost when connecting n processors to n memory modules. Moreover, the maximum parallelism feature does not mean that care must not be taken to avoid memory module contention which can arise in one of two ways: several processors addressing the same module concurrently, or addressing vectors with strides such that the same modules are accessed before they can deliver the previous element. In general in these systems, *hot spot* contention, i.e., addressing of the same memory location by several processors is avoided by using special registers for synchronization purposes.

At the other end of the spectrum, the simplest interconnection structure is a single shared-bus. The first successful commercial multiprocessors, the "multis" – a term coined by Gordon Bell – such as Sequent Balance and Symmetry and Encore computers, were built using this shared-bus paradigm. The limitation for expanding these systems is of course the contention for the shared-bus. This contention is reduced by the use of private caches, or cache hierarchies, associated with each processor. The caches are kept coherent via snoopy protocols that do not intrude on processor performance. Nonetheless, as processors become more powerful, they place a heavier load on the bus and the larger systems are limited to 32 processors. This is already quite a feat when processors are of the 150 MHz MIPS R4000 or R8000 class, as in the SGI Challenge multiprocessor. In the SGI system, the shared-bus [15] is wide (256 lines for data, 40 for address), allows split read transactions, permits different transactions to use the data and address lines simultaneously, and can deliver a 1.2 Gigabytes per second sustained transfer rate. The implementation of efficient synchronization in shared-bus systems requires optimizations of the test-and-set primitive, such as the test being done in cache (test-and-test-and-set), or the competition for the bus at synchronization time being lessened (back-off retries à la Ethernet), or better yet the use of queuing locks requiring a single atomic exchange operation[16].

Neither the cross-bar, because of its cost, nor the shared-bus, because of contention, are the right switches for systems that are meant to be expanded to several hundred processors. In this case, the choices for interconnect are hierarchical structures or multistage interconnection networks or topological structures such as meshes and tori.

The basis of a hierarchical structure is a cluster, e.g., a "multi" as defined above or a ring of processors like in the KSR machines. The idea is that a bus – or ring – based structure is best when the number of processors is small. Rather than saturating the bus with more processors, or adding busses (a palliative at best), clusters will be connected together in some hierarchical form. For example in the KSR machines, rings of 32 processors are connected through another ring that can link 32 basic rings for a total of over 1,000 processors. In the Stanford DASH experimental multiprocessor [17], clusters are multis of 4 processors and clusters are connected via two 2D meshes, one for requests and one for replies. The challenge in these systems is to place, or migrate, data in the clusters so that they are close to the processors accessing them. Otherwise the intra-cluster bus or ring contention will be exacerbated by the inter-cluster requests with the additional

drawback that the inter-cluster references might take an order of magnitude longer than the intra-cluster ones. Synchronization is clearly an expansive operation and care must be taken to choose the right task granularity.

Years of research on multistage interconnection networks (MIN) have led to many variations on the Omega network and the perfect shuffle. Yet, few systems have been built using this topology (e.g., BBN Butterfly, Ultra computer at NYU, Cedar at Illinois). MIN's expand in an $O(nlogn)$ fashion, thus are less costly than cross-bars, and reduce contention considerably compared to shared-busses. Moreover, fault-tolerance can be embedded in the MIN in a relatively cheap manner (e.g., with one extra stage). However, MIN's induce longer latencies than a cross-bar, are prone to the presence of hot-spots, and do not allow easy broadcast as does the shared-bus. IBM's choice of a MIN for the SP series seems to indicate that the reliability arguments won over the potential drawbacks.

Instead of a "direct" network, like a MIN, many systems have been built on indirect networks such as hypercubes, or less rich topological structures such as mesh and tori. The main advantage of an hypercube-based network is that the degree of the interconnection graph is large (i.e., many neighbors) and its diameter is small (no node is too far). The problem for hypercubes is that the degree of a node cannot be too large (wire and pin constraints). The concept of clusters can again be used here with nodes on the hypercubes being replaced by, say, a ring and nodes sharing in a multiplexed mode a slot to communicate with other ring-nodes on the hypercube.

Recently though, the trend has been towards 2D or 3D meshes and tori with possibly co-processors handling the routing. Since processors that are not senders or recipients of a message are not disturbed, the diameter constraints are no longer as important as one would have expected and advances in routing (worm-hole, virtual channels etc.) reduce contention. Another approach, exemplified by the data network of the Connection Machine CM-5, is to use a fat-tree [18] where the bandwidth is higher near the root of the tree (processors are at the leaves). In fact, the bandwidths between different levels of the tree can be adjusted according to engineering constraints. Moreover this system has different networks for data messages and for control, i.e., synchronization.

At this point, we could embark on a discussion of the impact of the programming model, shared-memory vs. message passing, on the choice of interconnection structure. A number of recent studies have shown that the differences in performance for programs "well-written" for the two models are not that great. This stems from advances in minimizing the latency overhead in preparing packets to be sent on the interconnect in the case of message-passing systems and from releasing the sequential consistency models in the case of shared-memory as indicated in the previous section.

Finally, it seems clear that in years to come, the problem will not be one of bandwidth (e.g., ATM switches can provide sufficient bandwidth) but one of latency. Thus the choice of topology might not be that important as long as systems are expandable. Minimizing latency can be achieved by more efficient software, reducing the number of protocol layers to the network interface, and by linking the network more directly to the processing node by considering the network to be a resource that can be directly accessed like main memory.

5. CONCLUSION

Experts in VLSI technology don't forecast any slow-down in processor speed improvements and in the increase in the number of transistors that can be put on a chip. Thus, we will continue to have faster processors with added functionality. Multiple issue processors with multiple deeply pipelined functional units, speculative and out-of-order execution, on-chip cache hierarchies with lock-up free caches, on-chip TLB's supporting physically and virtually indexed caches will be present and there will be still room to spare for hardware assists for either specific functionality (graphics, multimedia, emulation of other instruction set architectures) or assists to the memory hierarchy (write caches, victim caches, prefetching units)

All these advances in processor speed and functionality will be for naught if the memory access times cannot be reduced. Multiple level hierarchies will help. Write latency will be hidden via write caches and the like. Synchronous RAMS or equivalent devices will allow fast sequential access. Nonetheless, memory latency is "the" problem to be solved, either by hardware or by software or by a combination of both, if we want to continue to see the performance of single processor systems remain on the same growth curve.

The short-term future of single processor systems is bright. The same cannot be said of conventional supercomputing and of massive parallel processing. Granted, considerable progress has occurred in the last decade but economical and technical factors have contributed to a noticeable dip in performance improvements. Monolithic supercomputing systems are not cost-effective and are being replaced by less expensive ensembles of processing elements. There still exists many problems where a vector supercomputer achieves best results but the gap is narrowing and the cost/performance ratio is favoring the microprocessor based systems. In the case of massive parallelism, the problem is not so much one of hardware but one of software and algorithm design: how does one scale a problem so that the communication/computation ratio does not degrade performance. Thus one should not confuse expansion, i.e., the capability of adding more processing elements, with scalability which includes not only the hardware but also the operating system and the algorithm.

A possible avenue towards larger modular and scalable systems is through the use of clusters. Each cluster could be a "mini", i.e., a shared-bus or shared-ring system, easy to design and to program. Clusters could be connected via switches of high bandwidth (and hopefully low latency) such as ATM's that would allow reconfigurability according to the topology of the problem, and decomposition if the machine is used for several applications at once. Inter-cluster communication should be under tight control of the software that would define task granularity, data placement, coherence and migration, and means of scheduling, load balancing, and synchronization. These software requirements are a real challenge but the definite trend in high-performance computing, whether it is at the single processor or at the massively parallel level, is that reliance on software is more and more crucial.

REFERENCES

1. J.E.Thornton. *Design of a computer. The Control Data 6600.* Scott, Foresman and Co., Glenview, Ill, 1970.
2. R.M. Tomasulo. Machine philosophy and instruction handling. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.
3. C.Perleberg and A.Smith. Branch target buffer design and optimization. *IEEE Trans. on Computers*, C-42(4):396–412, April 1993.
4. T.-H. Yeh and Y.Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. of 19th Int. Symp. on Computer Architecture*, pages 124–134, 1992.
5. D.W. Rain D.H. Gibson and H.F.Walsh. Engineering and scientific processing in the ibm 3090. *IBM Systems Journal*, 25(1):36–50, January 1986.
6. T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high-performance processors. to be published in IEEE-TC, 1995.
7. N. Jouppi. Cache write policies and performance. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 191–201, 1993.
8. D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. of 8th Int. Symp. on Computer Architecture*, pages 81–87, 1981.
9. A. Agarwal and S. Pudar. Column-associative caches: a techniqque to reduce the miss rate of direct-mapped caches. In *Proc. of 20th Int. Symp. on Computer Architecture*, pages 179–190, 1993.
10. N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Int. Symp. on Computer Architecture*, pages 364–373, 1990.
11. J-L. Baer and W-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. of 15th Int. Symp. on Computer Architecture*, pages 73–80, 1988.
12. M.Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Symposium on Computer Architecture*, pages 434–442, June 1986.
13. S. Adve and M. Hill. Weak ordering - a new definition. In *Proc. of 17th Int. Symp. on Computer Architecture*, pages 2–14, 1990.
14. K. Gharachorloo, D. Lenoski, Laudon J, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of 17th Int. Symp. on Computer Architecture*, pages 15–26, 1990.
15. M.Galles and E.Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Proceedings of the 27th HICSS*, pages I–134–143, 1994.
16. G.Graunke and S.Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, June 1990.
17. Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: logic overhead and performance. *IEEE TPDS*, 4(1):28–40, January 1993.
18. C.Leiserson, Z.Abuhamdeh, D.Douglas, C.Feynman, M.Ganmukhi, J.Hill, D.Hillis, B.Kuszmaul, M.StPierre, D.Wells, M.Wong, S-W.Yang, and R.Zak. The network architecture of the Connection Machine CM-5. to be published in JPDC, 1995.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

The CONVEX Exemplar SPP1000 and SPP1200 - New Scalable Parallel Systems with a Virtual Shared Memory Architecture

Frank P. E. Baetke
CONVEX Computer Corp. EAME
Stefan-George-Ring 6, D - 81929 Munich, Germany
E-mail: baetke@convex.com

Convex has introduced the *Exemplar* Series, an advanced technology Scalable Parallel Processor (SPP) which is based on a novel virtual shared memory architecture combined with the PA-RISC CPU technology developed by Hewlett-Packard. The discussion below overviews the architectural principles and its insertion into the Exemplar Series SPP product line which complements Convex C-Series of Vector/Parallel Systems.

1. INTRODUCTION

The Exemplar series is a family of computer systems that provide multi-purpose supercomputing-class resources for a wide range of applications. The system delivers performance through a combination of high-performance RISC processors, a unique hybrid memory system, and a high-performance I/O subsystem. At the same time, extensive use of off-the-shelf components, such as the RISC CPU, permits price/performance that tracks that of workstations and workstation servers.

The Exemplar design is based on the observation that truly successful MPP products cannot live on hardware alone. They must also offer a rich software development environment as well as programming models which reflect not only highly sophisticated programming styles but also enable codes written in a straightforward way to be compiled with a high chance of parallel speedups. It is vitally important that this environment be evolutionary, not revolutionary, in nature. Users must be able to gracefully migrate to MPP products without abandoning their existing software investment.

The Exemplar systems provide performance and functionality in a variety of environments that may exist simultaneously on any system:

- multiprocessing throughput — running many jobs simultaneously, taking advantage of the convenience and cost-effectiveness of an accessible, shared resource;
- improving time-to-solution — increasing the performance of individual applications through the distribution of portions of the application across multiple processors (parallelization);
- data management services — providing access to large amounts of secondary or tertiary storage systems as a shared resource.

The resulting system overcomes the classic problem of application availability that has previously plagued parallel systems. The availability of the HP-UX ABI combined with a sophisticated compiler technology allows to execute thousands of popular application programs cost-effectively, without extensive investment in application porting or conversion. The result is a spectrum of application availability: traditional HP-UX applications, moderately parallel shared memory applications, and highly optimized parallel applications.

The Exemplar SPP systems combine architectural features of scalable parallel computer systems with the programming model and operational characteristics of a desktop system.

2. SPP1000 and SPP1200 ARCHITECTURE AND IMPLEMENTATION

In the simplest terms, the SPP systems consists of a series of computational nodes or CPUs tied together over a very high performance multi-dimensional interconnect. Physically this network is realized by crossbars in one direction and a multitude of parallel rings in the other direction, see Fig. 2. This hybrid interconnected scheme provides full cache coherence and is termed "Coherent Toroidal Interconnect" or CTI. The structures along the crossbar direction consist of up to 8 CPUs connected to a local memory and a high performance I/O sub-system. Because these structures are much more complex and offer between one and two magnitudes more performance than "classical MPP nodes" of other vendors, the synonym "hypernode" is introduced for such a structure. Each hypernode supports up to 1.9 GFLOPS in performance, 2 Gigabytes of physical memory, and hundreds of Gigabytes of high performance disk.

The interconnect and supporting logic utilize state-of-the-art gallium arsenide technology to enable the system to behave as if all processors in all hypernodes shared a single global memory system.
The system designated SPP1000 was the first generation of the CONVEX *Exemplar* Series. It consists of up to 16 computational hypernodes based on HP's 7100 processor connected via a very high-speed coherent interconnect, termed CTI. The SPP1200, based on HP's 7200 processor, includes additional architectural improvements. With 8 CPUs per hypernode, both systems can be configured with up to 128 CPUs. The systems also supports up to 32 Gigabytes of system memory, and approximately one Terabyte of very high performance on-line storage. SPP1000 has a peak performance of 25 GFLOPS and has been delivered to customers since April 1994, SPP1200 reaches a peak performance of 30 GFLOPS.

It is important to understand that the hypernodes form a physical structure rather than a logical one. The logical view of SPP1000 and SPP1200 is completely independent of the hypernodes and will be determined mainly by the subcomplexes described in chapter 3.4.

2.1. Memory Hierarchy

The SPP systems support a two-level hierarchy of memory, each level of which is optimized for a particular class of data sharing. The first level consists of a traditional memory crossbar interconnect that supports up to eight processors sharing a common physical memory, i.e. a *hypernode*. The hypernode is a symmetric multiprocessor (SMP) with its own

CPUs, memory and I/O subsystem. The use of a crossbar in the first-level memory interconnect allows processors and the I/O subsystem simultaneous, non-blocked access to memory, which permits scalability not found in bus-based systems.

The second level of the memory subsystem is a high-performance, four-way interleaved ring connecting the hypernodes. This toroidal structure, called CTI (Coherent Toroidal Interconnect), provides multiple rings for additional bandwidth and fault-resilience. The system then becomes multiple hypernodes connected by the CTI, as shown in Figure 1.

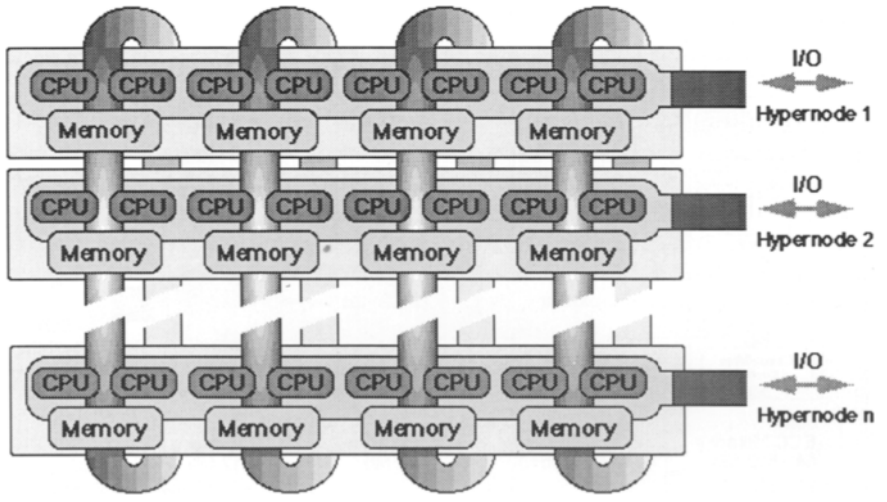


Figure 1. Architecture of the SPP1000 and SPP1200 systems.

This hierarchical memory subsystem provides several benefits:

1) The low-latency shared memory of a hypernode can effectively support fine-grained parallelism within applications, thus improving performance. This is often accomplished by simply recompiling the program with the automatic parallelizing compilers.

2) The two levels of memory latency will likely be the model of the future: as semiconductors become more dense, multiple CPUs will likely be placed on a single die. Thus, the first level of the hierarchy will be multiple CPUs sharing a common memory; the second level of the memory hierarchy will be "off-chip" (possibly to a nearby multiprocessor chip). Additionally, as multiprocessor workstations are introduced, clusters of these workstations will follow the same model.

3) This system organization is a superset of a cluster of workstations, traditional experimental MPP systems, and SMP systems. Processors within a hypernode are tightly coupled to support fine-grained parallelism. Hypernodes implement coarser-grained parallelism with communication through shared memory and/or explicit message passing mechanisms.

2.2. Hypernode Structure and Components

Each hypernode contains its own 5 by 5 memory crossbar, very similar to traditional supercomputer memory subsystems. The crossbar provides high bandwidth, low latency, non-blocking access from CPUs and I/O channels to the hypernode-local memory. The crossbar is implemented in high density gallium arsenide (GaAs) gate arrays. Proven in earlier Convex systems, the use of GaAs permits high performance while keeping the power and cooling requirements to a minimum.

The use of a crossbar prevents the performance drop-off associated with systems that employ a system-wide bus to handle CPU and I/O traffic. Figure 2 illustrates the physical components of an SPP1000 hypernode.

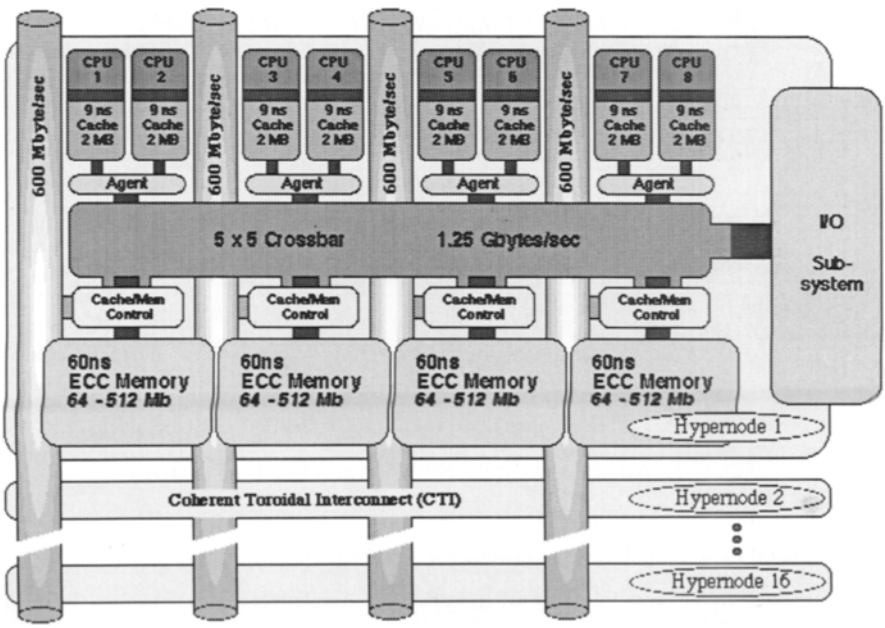


Figure 2. An SPP1000 hypernode.

Note that the Convex SPP1000 and SPP1200 provide hardware support for global shared memory access, whereas a system without this feature can only emulate shared memory by moving pages from node to node under software control. On the SPP systems, cache lines may be automatically copied and encached between hypernodes without software intervention, resulting in lower overhead to programs using shared memory.

2.3. Coherent Toroidal Interconnect (CTI)

The low-latency interconnect (CTI) connects multiple hypernodes. CTI is derived from the IEEE standard 1596-1992, SCI (Scalable Coherency Interface). This interconnect

combines high-bandwidth with low-latency to provide system-wide coherent access to shared memory. Explicit message passing (EMP) applications may also use the CTI for inter-hypernode communication.

The CTI is composed of four uni-directional 18-wire rings attached to each hypernode. Four rings are used to provide higher interconnection bandwidth, lower inter-hypernode latency, and redundancy in case of ring failure.

Sequential memory references to global memory (by linearly ascending physical address) are interleaved across the four rings, using the ring in the same functional unit as the target memory. Since the memories are interleaved on a 64 byte basis, the four CTI interconnects are interleaved on this basis as well. This interleaving balances the traffic across all four rings. Global memory references from a CPU to the global memory on the same hypernode use the hypernode's crossbar, providing data access at full memory bandwidth.

To minimize memory request latencies across the CTI, each hypernode contains a cache of memory references made over the interconnect to other hypernodes. This is referred to as the CTIcache. Any data that has been moved into a CPU cache on the same hypernode, and is still resident in the CPU cache, is guaranteed to also be encached in the CTIcache. Consequently the CTIcache directory information can be used to locate any global data that is currently encached by the hypernode. The CTIcache is physically indexed and tagged with the global physical address.

The SPP1000 and SPP1200 systems guarantee cache coherence between multiple hypernodes; two or more hypernodes that map the same global address will get a consistent view. This is done by maintaining a linked sharing list that contains a list of all the hypernodes sharing each cache line, or the hypernode that exclusively owns the cache line. The system keeps a record of which CPUs have encached each line in the CTIcache, so that interconnect coherency requests can be forwarded to the appropriate CPUs. Please note that multiple coherence domains can be defined by the subcomplex manager described in chapter 3.4.

2.4. I/O Subsystem

The I/O subsystem is highly scalable and may be distributed across the hypernodes. Each hypernode supports a 250-megabyte-per-second non-blocking I/O port to memory. Each intelligent I/O port is capable of Direct Memory Access (DMA) transfers directly to and from any of the distributed physical memory units in the system. This eliminates CPU involvement in data transfers, reserving them for user work. It also streamlines data transfers for such things as large disk blocks and high-speed network connections.

The I/O subsystem is physically distributed across any or all the hypernodes in the system, as shown in Figure 3. Over the interconnect and the crossbar, peripherals have access to any memory unit in the system. Similarly, any processor can access any filesystem mounted on any hypernode. The CTI (the system interconnect) and non-blocking crossbars are used to route data and control packets to the proper peripheral, wherever it may be within the system. Across all hypernodes in a fully configured system, the user has available an aggregate of 4 gigabytes per second of DMA I/O bandwidth to system memory.

The SPP1000 and SPP1200 peripherals are supported through industry standard IEEE 1496 SBUS host adapters. Peripherals supported include standard SCSI-2 fast and wide disks, DAT and 3490 tape drives.

The available disk subsystem provides reliable, cost-effective mass storage through arrays of standard 3.5-inch disks. A fully configured system can support more than a Terabyte of scalable disk storage for sites that require a large amount of storage.

Disks in the subsystem may be striped to improve data transfer performance. Striping is performed in the striped disk driver, and is thus transparent to the user.

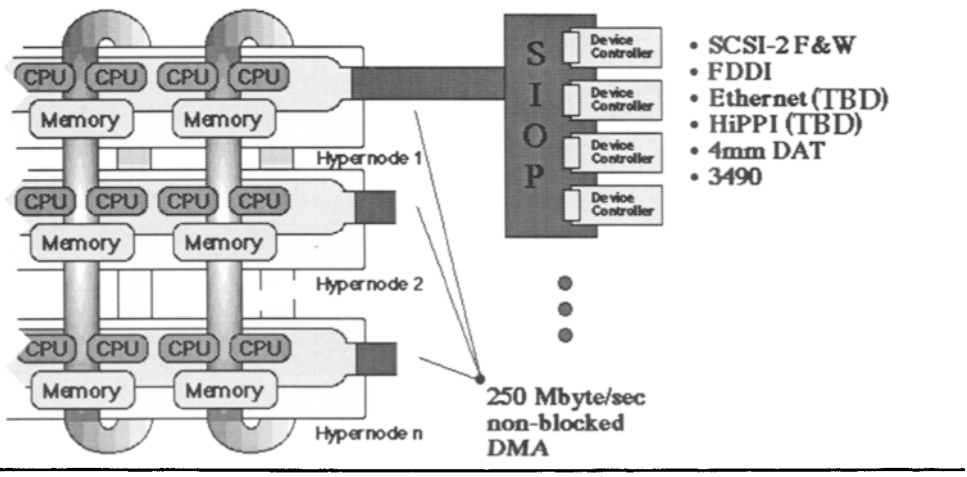


Figure 3. SPP1000 and SPP1200 I/O Subsystems

2.5. Packaging

The SPP1xxx family consists of two packaging options: the SPP1xxx/CD and the SPP1xxx/XA. Both are air-cooled and stand-alone (i.e., do not require a front-end) systems. As shown in Figure 4, the systems provide PA-RISC processing power across a broad range of price and performance.

The /XA ("eXtended Architecture") system offers from eight to 128 CPUs, 32 GBytes of memory and 4 GBytes/second of I/O bandwidth. The /XA towers support a user-programmable meaningful optical display subsystem (MODS). As the system grows, additional towers are "clicked" together, with the Coherent Toroidal Interconnect (CTI) and service cabling routed through cable channels between the towers. The SPP1000/XA and the SPP1200/XA do not require a raised floor environment.

The SPP1000/CD ("compact packaging") is the entry-level member of the SPP1xxx family. The /CD system offers from two to sixteen CPUs, up to four GBytes of memory and 500 MBytes/second of I/O bandwidth. The system is composed of one or more desk-high (36 inches or 92 centimeters) cabinets. The first eight CPUs fit in a single cabinet; if more than eight CPUs are configured, then an additional cabinet may be added.

Should it be required, the /CD system is fully upgradable to the /XA system. Thus, if computing needs grow and more than sixteen CPUs, additional memory, or I/O is required, the internal chassis that are resident in a /CD cabinet may easily be moved to an /XA cabinet.

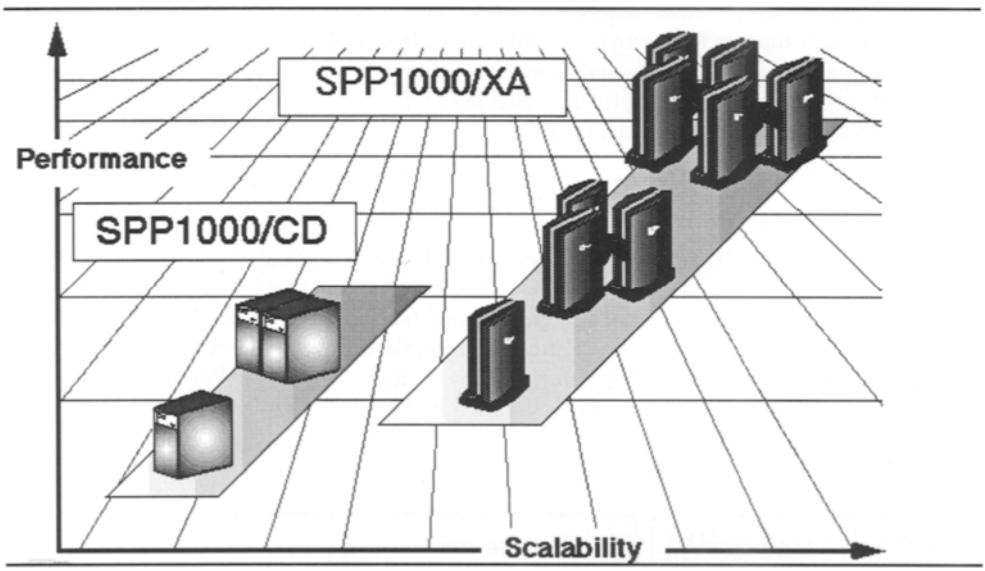


Figure 4. The SPP1000 product family.

3. SYSTEM SOFTWARE

The architecture of SPP systems requires a fresh approach to the implementation of UNIX. For this reason, Convex selected an operating system based on a distributed microkernel approach. The microkernel provides basic kernel functionality such as virtual memory and scheduling of processors. The rest of the system is supported as a set of applications running in user space. These servers provide the standard functions of a full-featured operating system, such as file system management, device management and network services. The microkernel and the servers communicate through a low-level message passing mechanism.

This approach has multiple advantages:

- *Scalability.* The distributed operating system allows resources to scale with system requirements, eliminating the operating system as a bottleneck to performance.
- *Modularity.* By splitting operating system functions up into distinct modules, the operating system is much more maintainable. In addition, new functionality or personalities may be easily added.

• *Performance.* Employing multiple servers allows additional performance. For example, running a file server on each hypernode allows I/O operations to take place in parallel across the system.

Convex has added features to the UNIX operating system to satisfy the needs of data centers and enterprise systems. These include job scheduling, checkpoint/restart, a high performance file system with support for terabyte-size files and file systems, and a sub-complex manager for efficient utilization of the system resources. Other extensions to UNIX include advanced system administration and performance evaluation facilities. Many of these services are implemented in SPP-UX or planned for future releases.

3.1. SPP-UX Operating System

The Exemplar SPP1xxx operating system, SPP-UX, is a scalable, microkernel-based operating system designed specifically to provide high performance to a wide variety of applications in an SPP environment. Applications may be single-threaded HP-UX binary executables, moderately parallel applications compatible with the Convex C Series, or highly parallel applications. Applications may employ shared memory and/or message-passing parallelization to increase performance. Figure 5 illustrates the application support layers of the SPP-UX.

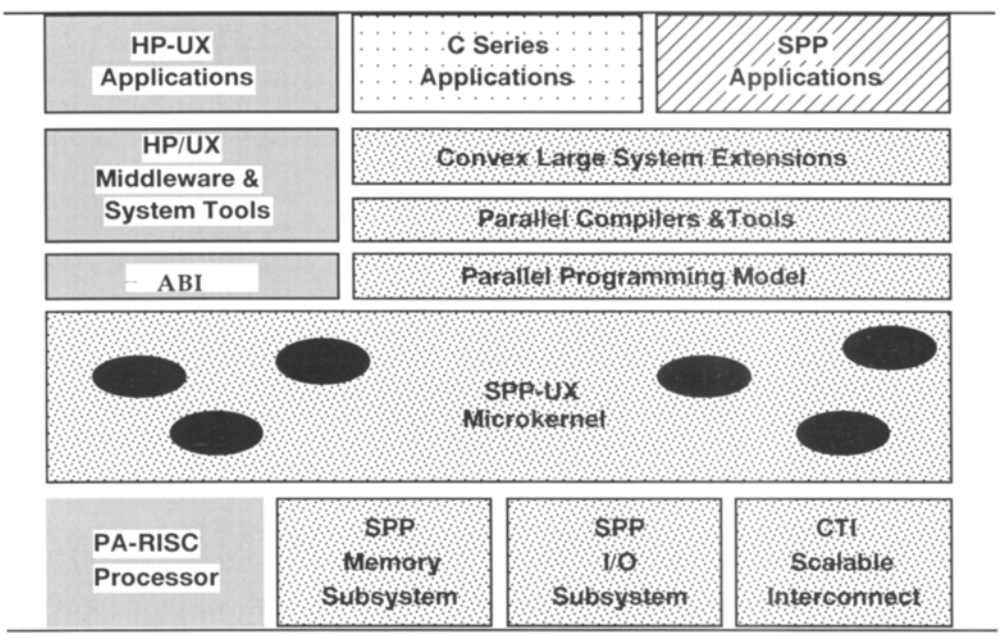


Figure 5. The different layers of the Exemplar SPP1xxx operating system.

The SPP-UX runs a copy of the microkernel on each hypernode (see Chapter 2.2 for more information on hypernodes). In addition, there are several servers that provide traditional UNIX abstractions:

- *PM Server.* Provides process management functions. There is one PM Server per system. The PM Server is always on the hypernode that contains the root file system.
- *SC Server.* Sub-complex server. There is one SC Server per system. It resides on the hypernode that contains the PM Server.
- *FS Server.* File System server. There is one FS Server on each hypernode that contains disks.
- *NS Server.* Network services server. There is one NS server on each hypernode that contains network devices such as FDDI.

Figure 6 illustrates how the microkernel and servers run on an Exemplar multi-hypernode system.

The microkernel and servers provide system services to user-level programs through system calls. System calls are handled by a section of code in the process' virtual address space. System calls are handled as follows:

1) The application traps into the microkernel
2) The microkernel sets up the applications stack to continue
3) The system call is either handled in the process' address space, or sent (via message passing) to one of the servers.

Each server handles system calls that pertain to its area of functionality.

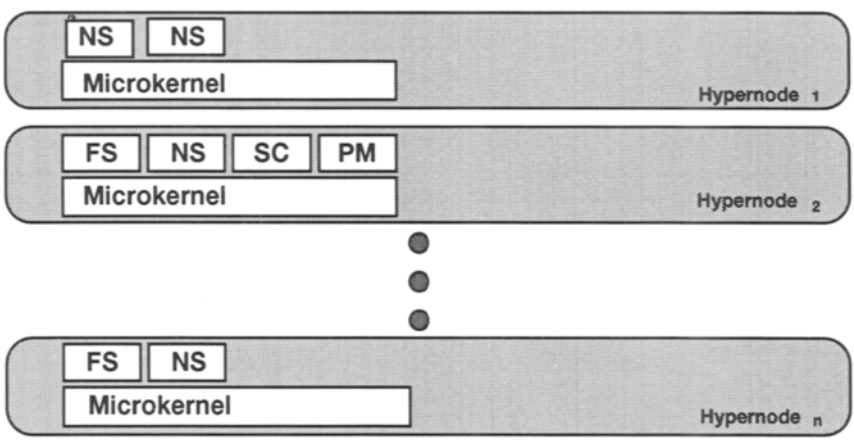


Figure 6. Microkernel and server distribution on the SPP1xxx systems.

3.2. Functionality of the SPP/UX Microkernel

The SPP-UX microkernel resides on each multi-CPU hypernode, and provides low-level, machine dependent functions. These functions include physical memory management, virtual memory management, processor management and scheduling. The main microkernel abstractions are the following:

thread	the basic execution entity for the machine
task	the basic resource container for the SPP-UX kernel. Tasks hold threads' address space and ports.
port	A unidirectional communication channel between a client and a server.
processor	The execution unit of the machine, the 7100 or 7200 CPU.
processor set	A pool of zero or more processors that provide the basis for scheduling threads. A processor set is most commonly all the processors on a hypernode that belong to a subcomplex.
host	The set of resources associated with a single invocation of the SPP-UX kernel. This includes physical memory, devices, processors and time.

The SPP-UX microkernel manages the partitioning of physical memory into CTIcache, node private memory, and global memory. Changes to this partitioning take place through system calls from system management utilities and SPP-UX servers.

The microkernel is also responsible for the management of hardware address translation tables called Block Descriptor Tables (BDTs). The BDTs allow the hardware mapping of each processor's 4 GByte physical address space into the larger 32 GByte system physical memory, note that SPP1xxx uses 36 bit addresses for physical and 64 bit addresses for virtual addressing. The 64 bit addresses also provide compatibility with future systems employing much larger address spaces.

3.3. Processor Management and Scheduling

The microkernel manages processors in *processor sets*. A processor set is zero or more processors on a single hypernode that belong to a specific subcomplex. The microkernel supports privileged operations for adding or removing processors from a processor set. These operations are typically accessed by servers or other microkernel functions.

There are two levels of scheduling in the SPP-UX: the scheduling of processes and the scheduling of threads. Processes are scheduled by the Process Manager Server (see the next section on Process Management). The microkernel on each hypernode is responsible for scheduling threads. Figure 7 illustrates process and thread scheduling. Referring to figure 7, note that:

- A process is confined to a single subcomplex and is "spread across" one or more tasks. (In this example the process consists of three tasks.)
- A given process has at most one task per processor set.
- Normally a subcomplex has at most one processor set per hypernode.

- Tasks do not span hypernodes nor do they span processor sets.

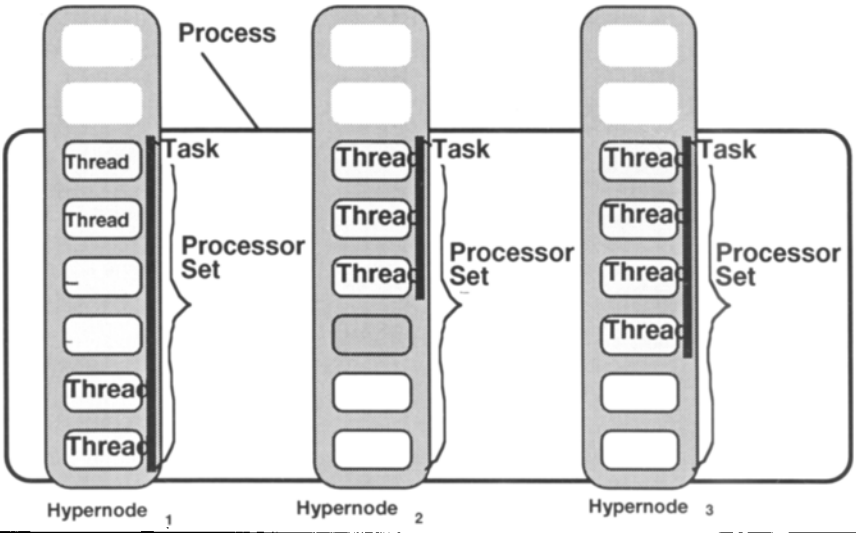


Figure 7. Process and Thread Scheduling.

The microkernel preemptively schedules threads on a processor set. It is preemptive in that the scheduler will preempt a user or server thread when a hardware exception or interrupt occurs on the executing thread. A user thread may also be preempted if a server thread or a higher-priority user thread needs to run. Any given thread has associated with it a scheduling policy to use, and various parameters that influence that policy. Threads from different tasks compete for CPU resources within a processor set much like processes compete for CPU resources in classic UNIX.

If a thread is preempted or voluntarily gives up its processor — either by an I/O request or by a system call — the microkernel scheduler will attempt to place the thread back onto the same processor when it's chosen to run again. This placement, called processor affinity, will mean that the thread begins re-executing with a "warm" cache and TLB state. This increases performance by reducing cache faults after a context switch.

The process model on the Exemplar systems allows a process to span hypernodes. A process is composed of one or more tasks. Each task is on a different hypernode and has one or more threads. Threads are scheduled by the respective microkernel running on the hypernode, as described in the sections under the description of the microkernel.

Each process has a scheduling priority that it inherits from its parent, and each thread within the process also has a scheduling priority that it obtains from the process's scheduling priority. A process may change its scheduling priority and the scheduling priority of all the threads within the process.

A process creates threads through a system call. The system call may specify the hypernode on which the thread is to be created, or allow the system to choose. If the system chooses then

threads are created round-robin across all hypernodes in the subcomplex on which the process is executing.

3.4. Sub-Complex Management

A SPP1000 or SPP1200 system may potentially consist of a relatively large number of processors and a large quantity of memory. A single system may be used for a variety of different applications, each having different resource requirements, including different levels of application parallelism.

To efficiently control and account for these resources, the SPP-UX offers a new and unsophisticated approach somehow similar to the "Virtual Machine" model found in classical operating system - the so called *sub-complexes.*

A sub-complex is the primary vehicle through which a system administrator can control the allocation of processor and memory resources. Every user process will run on a sub-complex, and will use only the processors and global memory allocated to that sub-complex. The system administrator will define sub-complexes to meet the needs of specific applications or groups of users, with consideration of the overall priorities for the use of the machine. Processors from one or more hypernodes may be assigned to a sub-complex, and global shared memory may be allocated on the hypernodes that the sub-complex is using. Processes and/or threads are timeshared on each sub-complex as if it were its own multiprocessor, shared-memory system.

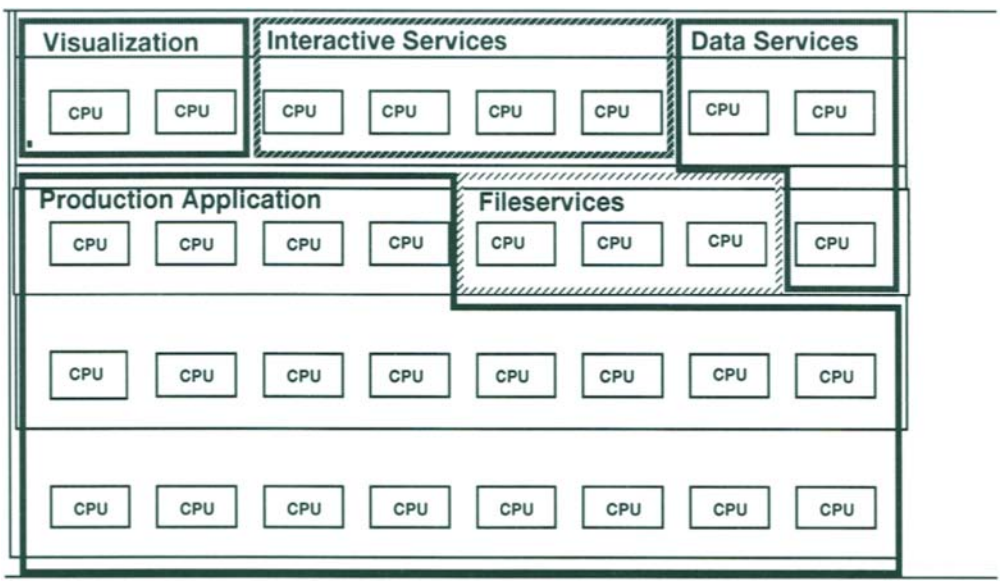


Figure 8. An Exemplar system with five sub-complexes.

Each processor in the system may be assigned to at most one sub-complex. Since there are up to 8 processors per hypernode, it is possible for processors on the same hypernode to be

divided among different sub-complexes. Figure 8 illustrates a sample 32-processor system that has been configured with five sub-complexes.

Additional resources that are configurable are the hypernode's server set and the hypernode's CTIcache size. Each hypernode has exactly one server set and one CTIcache, regardless of the number of sub-complexes that may be using the hypernode.

The server set for the hypernode handles all operating system activity without regard to sub-complex boundaries. A processor may be assigned to both the hypernode server set and a sub-complex, i.e., assignment to one does not preclude assignment to the other.

Sub-complexes are used to control the allocation of processor and memory resources on the system. The sub-complexes are defined and managed by the system administrator through a GUI-based sub-complex manager. It allows the system administrator to graphically view the sub-complexes, change them, and experiment with new sub-complex configurations on-line.

Sub-complex attributes include the name of the sub-complex, access control permissions, and scheduling policies (which allows "timeshare" or "fixed priority" scheduling on this sub-complex).

The following resources may be allocated to a sub-complex:

• *processors* — any number of processors from any number of hypernodes may be allocated to the sub-complex.

• *global memory* — global memory may be allocated on any hypernode having at least one processor assigned to the sub-complex. Global memory is allocated in 16 MByte increments.

• *CTIcache* — for each hypernode being used by the sub-complex, the user may request that the CTIcache be made a certain size. The CTIcache is allocated in powers of two times 16 MByte increments (for example, 16 MByte, 32 MByte, 64 MByte, and so on).

Sub-complexes are able to be redefined while processes are executing on the sub-complex; however there are some restrictions to reconfigurations. The following reconfiguration actions are allowed on a busy sub-complex:

Processors may be added to or removed from a sub-complex at any time (with some restrictions involving global memory). Note that code generated by compilers will still run in parallel, since the number of processors to use is determined dynamically at run-time.

All processors may be removed from a busy sub-complex, but doing so will suspend all user processes on that sub-complex until processors are once again allocated to the sub-complex.

The CTIcache may be reconfigured at any time.

Scheduling policies may be enabled or disabled on a sub-complex at any time.

3.5. File Services

The file subsystem is one of the major subsystems of the operating system. It controls access to files as well as devices and provides a hierarchical organization for files and directories, buffering of data and control of file system resources. This section describes the structure of files in SPP-UX, how they are arranged on the disk, and the routines used to access, manipulate, and protect files.

The SPP-UX file subsystem has a standard UNIX hierarchical directory structure. Each hypernode in an Exemplar system that has a file system attached to it runs a copy of the file server code. This allows applications to communicate with each file server independently and in parallel.

Each hypernode may have disks attached to it. The file server on the hypernode will provide access to its devices and will provide file services for the file systems that are on these devices. In addition, each file server maintains its own buffer cache. This allows the total size of the system's buffer cache to scale with the addition of more hypernodes. As each device is mounted on a mount point the file server that provides the services is registered with the in-memory mount table, and the semantics of a single UNIX file name space are provided transparently.

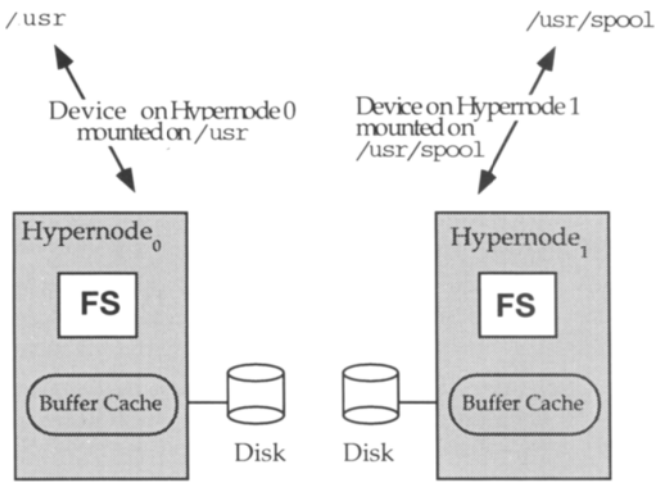


Figure 9. The relation between hypernodes, file servers and file systems.

A file is the basic unit of the file subsystem and is regarded as a named object in the operating system. There are six basic file types.

3.6. The File System Buffer Cache

If every implied disk transfer really had to be done, the CPU would spend most of its time waiting for I/O to complete. Because of this, the file server maintains a pool of internal data buffers, called the buffer cache, to minimize the frequency of disk accesses. Unless otherwise specified, all file I/O operations use the buffer cache as an intermediate storage area between user address space and the device itself.

The task of the buffer cache is to act as a cache of recently used disk blocks. When writing a file, data is written to the buffer cache instead of disk so that the data will be readily available in case the system needs to reference those blocks again. Periodically, the operating system

issues the *sync* system call to flush the contents of the buffer cache to the disk. When reading a file, the file server first attempts to read from the buffer cache. If the requested data blocks are not there, the server reads the data from disk and caches it in the buffer cache.

The buffer cache is a region of memory within the file server's address space. With each hypernode possibly being a file server, this provides a scalable size of buffer cache to match the larger system configuration. It also provides for distinct parallel paths through the file system code and accesses to data on disk.

The memory that the file server is using to provide the buffer cache is pageable. This is done to provide dynamic usage of physical memory. It also greatly simplifies the support for variable block size file systems and optimal memory usage. This form of dynamic buffer cache also eliminates the relationship between the virtual memory management and the file server. The microkernel implements virtual memory and is responsible for keeping the most useful physical pages of memory filled with the most popular pages. The maximum block size of a filesystem block sets the buffer cache element size.

In SPP-UX, a read-ahead mechanism has been introduced in addition which increases the number of blocks of read-aheads for processes performing sequential reads. This enhances the performance of applications which sequentially read files.

The read-ahead algorithm utilizes a heuristic approach to obtaining disk blocks of a file that is being read by an application. After a disk block is read, if the application reads a second block in a sequential fashion then a read-ahead will be issued for 2 more blocks. If the application again reads sequentially, then the server issues read-aheads for 4 blocks. If the application again reads sequentially, then the server issues read-aheads for 8 blocks. This algorithm is called a progressive read-ahead. It will attempt to expand the size of the read-ahead region based on the continued sequential behavior of the application.

3.7. HP-UX Application Binary Interface (ABI)

The Hewlett-Packard implementation of UNIX is HP-UX. HP-UX is an open, widely used operating system that supports thousands of applications. In addition, many sophisticated system management tools, visualization products and performance tuning and monitoring tools exist that run under the HP-UX operating system.

The system call layer of the SPP-UX operating system supports compatibility with HP-UX at the ABI (Application Binary Interface) level. This provides important functionality in several areas:

• *Application Availability.* Thousands of applications run under the HP-UX operating environment. For the most part, these applications will execute unmodified on the Exemplar system, with no need for recompilation or relinking.

• *Ease of Use*. Where Convex extensions are made, most often related to parallel aspects of the system, a strong emphasis is placed on ease of use and providing no surprises to a user versed in HP-UX. For example, utilities created to support the stripe driver interface use the standard HP-UX utilities and the usual interface to SAM.

• *User Familiarity.* A person familiar with HP-UX will find virtually no operational differences at the user level from the HP 700 Series and a minor number of system extensions.

At the systems administration level, the system administrator will find the expected SAM utility (System Administration and Management), find system files in their normal positions, and in general find the system intuitively easy to use.

• *HP-UX "middleware,"* such as Motif, X, system management tools, and libraries will execute on the Exemplar. This permits conformance with the emerging COSE (Common Open Software Environment) standards which have been proposed by HP, IBM, SunSoft and others.

• *Application time-to-port.* Often an application consists of hundreds, thousands, or even millions, of lines of code. Using the binary compatibility of the Exemplar, it is possible to directly move the application binaries (usually called ".o files") to the Exemplar system, and replace the critical binaries with parallel, tuned versions. This allows the developer to produce a highly tuned version of the code without recompilation of the entire application. This reduces testing and validation of the optimized application.

ABI CapabilitiesSeveral areas of SPP-UX present a strictly conforming HP-UX personality. These areas are:

System Calls
Shared Libraries
File System Behavior
Signal Behavior
Memory Behavior
System Files
Utilities

Each of these functional areas are nearly identical to HP-UX. This provides the user and developer with the advantages as outlined above.

4. PROGRAMMING ENVIRONMENT AND TOOLS

The goal of the SPP1xxx programming environment is application availability. The development tools that Convex provides are designed to automatically parallelize to some extend even legacy codes (the so-called "dusty deck" FORTRAN programs in particular). Yet the toolset also empowers the programmer to take explicit control of the machine to exploit the full benefit of its architecture.

To facilitate the porting of existing applications, and the development of new ones, the system provides the familiar shared memory and explicit message passing development paradigms. The result is a single programming model that supports two programming styles that may be combined if desired. For Convex' implementation of a virtual shared memory environment the term GSM (Global Shared Memory) is used which reflects the fact that part of a shared area can be physical (inside a hypernode) and part can span multiple hypernodes within a subcomplex.

The programming model for SPP systems provides a full range of support for the underlying architecture. Fully automatic parallelization, combined with the hardware's ability to dynamically distribute data, provides immediate benefit to existing C and FORTRAN shared

memory programs. PVM, PARMACS and MPI message passing is provided for existing programs written in that style. Programmers can utilize features supporting explicit parallel thread management, explicit synchronization, and explicit data distribution.

4.1. Global Shared Memory Parallelism

Shared memory (GSM) parallelism embodies the characteristics of conventional serial programming, enhanced to provide parallel execution. A programmer wishing to use GSM parallelization to increase the performance of an application has only to recompile the program with one of the automatic parallelizing compilers. The GSM style is best suited for porting existing serial or shared memory parallel programs and for developing new programs in a shared memory style.

When parallelization options are invoked, the SPP1xxx compilers will automatically parallelize data-independent loops, Fortran 90 array expressions and data-independent loops containing calls to subprograms. Parallelization is achieved by dividing the work of the loops among threads of execution, one per processor in the subcomplex. Code generation is such that a process automatically adapts itself to the number of processors available to the user at runtime — the number does not have to be supplied to the compiler.

When a program compiled for GSM parallelization begins execution, there is one thread of execution per available processor.The number of processors may be specified by the user or assigned by the run-time libraries at the time the program is invoked. All threads are idle except for thread 0, which executes alone until it encounters a parallel construct. It then activates, through procedure calls to the compiler's run-time libraries, all the idle threads in the process. If the threads are not currently executing on a processor, the microkernel will schedule them for execution. At the end of the parallel construct the threads synchronize at a barrier, and all threads except thread zero go idle.

4.2. Explicit Message Passing Parallelism

An explicit message passing (EMP) program is typically many single-threaded processes, each executing on a processor of the subcomplex. The processes coordinate their mutual operations and share data values through explicit message passing. The user specifies the communication of data values between the processes using a message passing library (e.g., PVM, PARMACS, MPI). Within a subcomplex, message passing is done through shared memory mapped among the executing processes.Across subcomplexes and to other machines, PVM uses conventional UNIX sockets. Specific subcomplexes on the same SPP1xxx system or on another system may be specified as destinations for message passing communication.

EMP programs are inherently parallel, and unless explicitly coordinated by message-waiting, all processes execute independently. In a conventionally coded message passing program, all variables are private to each process. Regardless of whether variables have been declared to be in any of the four memory classes, no process can access the variables of any other process. Synchronization among the processes occurs explicitly through message passing.

Hybrid programs are also supported so that message passing can be performed among several multi-threaded, shared memory programs. This permits programs that have been designed for EMP, and written to use a defacto standard EMP library like PVM, to be immediately moved automatically to SPP systems. The individual EMP processes may themselves be parallelized using the GSM model, resulting in extremely high performance EMP applications.

4.3. Compiler Technology

The SPP1xxx system supports popular programming languages and extensions, including FORTRAN 77, Fortran 90, C, and C++. The compilers allow the user to migrate applications from other platforms and develop new applications to take maximum advantage of the SPP1xxx architecture. The goal of the Convex compilers is to increase the overall efficiency of the system, while maintaining high productivity and application portability.

The compilers provide a hierarchy of automatic optimizations, ranging from instruction-level optimization through coarse-grained parallelization through tasks or loops. The lowest level of optimization is at the code generation level. Here, as much scalar performance as possible is extracted using advanced RISC optimization techniques, including instruction scheduling, software pipelining, and tiled global register allocation.

Data locality provides additional single CPU performance by performing loop blocking to improve cache performance, as well as classic machine-independent optimizations. The next level of performance is achieved by compiling for automatic parallelization, which enables loop-level parallelization on data independent loops. ypically, this form of parallelism results in as many threads of parallelism for a particular process as are allowed in the subcomplex.

The compilers also support directives/pragmas which allow the user to explicitly control parallelization and data distribution. For example, a directive may be used to indicate that a loop should be run with one thread per node allocated. This is used to parallelize an outer loop in which each iteration of the loop could spawn additional threads for an inner loop, providing multiple levels of parallelization.

The Convex SPP1xxx FORTRAN compiler adheres to the American National Standard programming language FORTRAN 77, X3.9-1978, ISO 1539-1980(E). The default language interpretations are FORTRAN 77 with default Convex extensions and certain features of the International FORTRAN Standard, ISO/IEC 1539.1991, which is identical to the ANSI Fortran 90 programming language, ANSI X3.198-1992.

The Convex SPP1xxx C compiler supports full ANSI C as well as the Kernighan and Ritchie derivative of the language. The compiler performs extensive loop-level, scalar, and machine specific optimizations, including global register allocation and instruction scheduling.

Both compilers perform extensive loop level and scalar optimizations. They perform machine-specific optimizations including global register allocation and instruction scheduling The capabilities of the highly advanced Application Compiler (APC) can not be discussed here for details see the related reference manuals.

Scalable parallel processing and high-performance parallel systems today present a challenge to code development. Convex has met this challenge by providing sophisticated software tools to ease the development and porting of parallel applications. CXtools is a complete and

powerful set of tools that aid programmers in debugging, testing, and optimizing highly parallel applications.

This high-productivity tool kit includes three visually oriented products: CXdb, the parallel debugger; CXpa, the parallel performance analyzer; and CXtrace, the parallel threads trace analyzer. Each of these products is geared toward developing and tuning applications that run on scalable parallel systems and vector/parallel systems.

4.4. The SPP-UX Debugger - CXdb

CXdb is a full featured debugger with an X11-based graphical user interface. CXdb has numerous features that support debugging at the source code or at the machine code levels. It is capable of debugging multi-threaded applications and optimized code.

CXdb enables debugging of applications compiled at any optimization level. For example, users can use CXdb to debug code that has undergone loop level optimizations like distribution, interchange, and induction variable analysis. One of the first requirements for debugging a parallelized application is to have a debugger that is cognizant of multiple threads. CXdb maps multiple threads within a parallel application on a single source or disassembly window. Activity within either of these windows is indicated by highlighting the active expressions or statements within the source code. An example of this is shown in Figure 10.

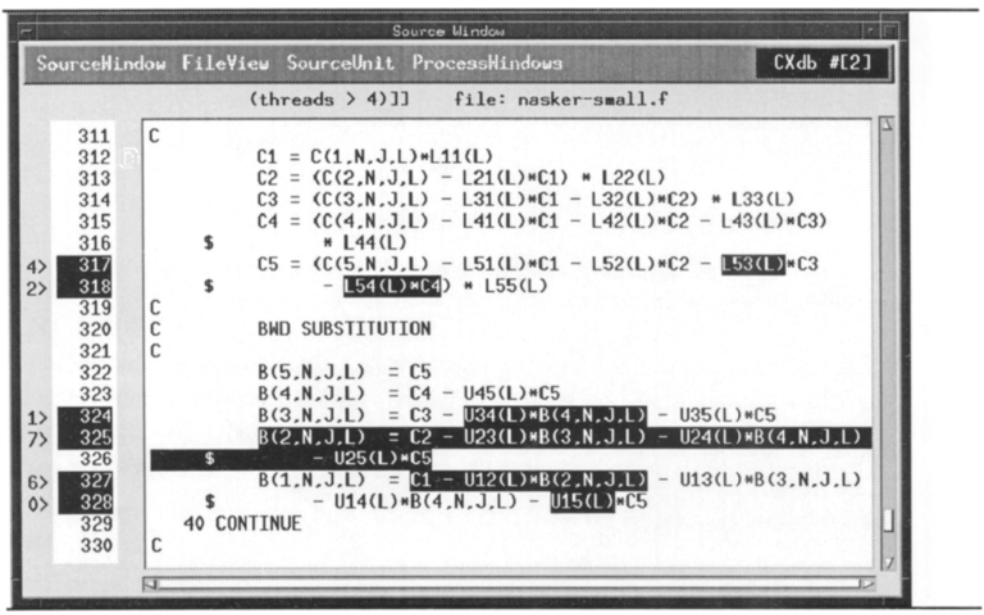


Figure 10. A sample debugging session of parallel code.

Another requirement for debugging a parallelized application is to be able to individually control the threads within an application. CXdb allows complete control over the threads in an application. You can select which thread(s) are to be associated with any of the windows.

You can set breakpoints within an individual thread, print values of variables with specific threads, and selectively enable different threads.

To help developers cope with a large number of threads, CXdb provides mechanisms to view thread activity (e.g., number of threads) across an entire application. Facilities are provided to navigate through a volume of threads, displaying source code associated with different areas of activity, and repositioning the source code window from one area of activity to the next. This allows parallel application developers to quickly examine variables and states when the application reaches an exception or a breakpoint.

4.5. SPP-UX Profiler CXpa

CXpa is an interactive profiler with a graphical user interface that provides detailed profiles of optimized loop nests. Profiling information is provided at the routine, loop, parallel region, and basic block levels to help pinpoint and diagnose performance problems.

Interactive profiling allows a user to stop during the execution of an application and analyze the profile data without the need to wait until the application has completed. This is an important capability in profiling long-running applications. CXpa is capable of profiling optimized code at the routine, loop, or parallel region level. CXpa is tightly integrated with the Convex compilers, allowing it to display profiling information on the loop nests generated by the compiler.

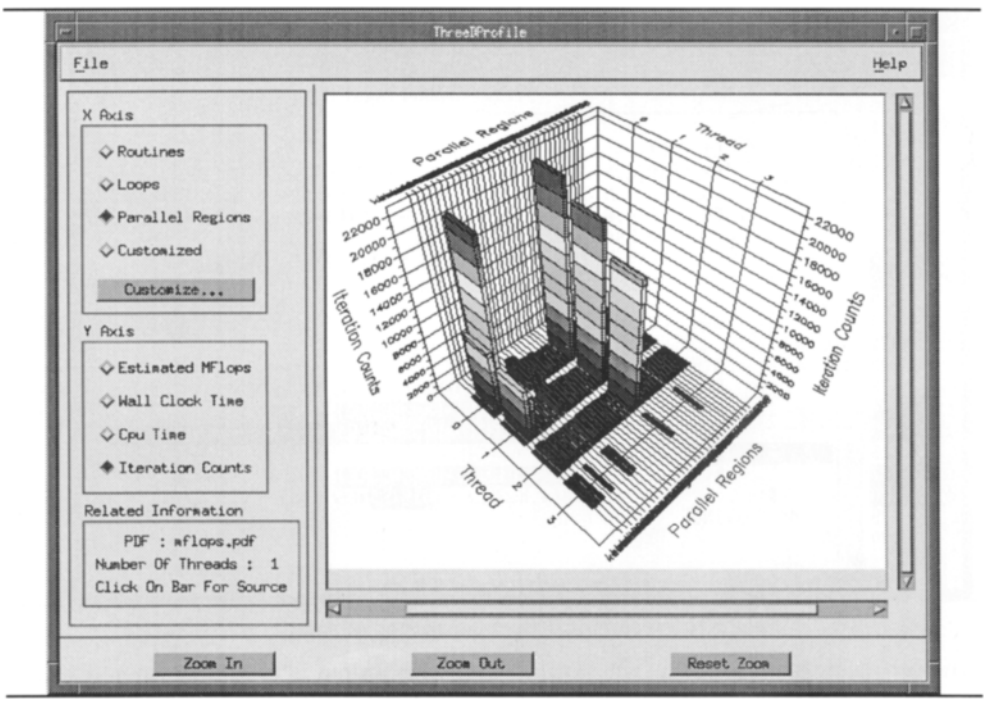


Figure 11. Sample output from the CXpa performance analyzer.

Identifying specific regions of source code responsible for longer memory latencies is important in improving application performance on non-uniform memory access systems. CXpa can provide this information by performance monitors in the hardware; this supplies in-depth program execution information without intrusive library calls. These performance monitors allow users to determine which routines or loops in their programs cause significant numbers of cache misses. Users are able to selectively monitor private memory, shared memory accesses, interconnect cache (CTIcache) accesses, or any combination of these metrics.

Besides providing a number of profiling metrics, CXpa provides two- and three-dimensional bar charts to easily identify "hot" routines or loops. Two dimensional bar charts display profiling data (wallclock time, cache misses, memory latency, etc.) aggregated across all threads in the application. Three dimensional bar charts display profiling on a per-thread (and on a loop or routine) basis. This allows users to determine how evenly their application is distributed across multiple threads as shown in Figure 11.

4.6. SPP-UX Trace Analyzer - CXtrace

CXtrace is a trace-based tool that allows a developer to instrument an application that has been parallelized. CXtrace allows the developer to determine the order of events, and the latencies between events for these types of applications. When the application is run, selected time-stamped events are emitted to a log file. A visualizer is then used to graphically depict events in the program. An example of an 8-way parallel program is shown in Figure 12.

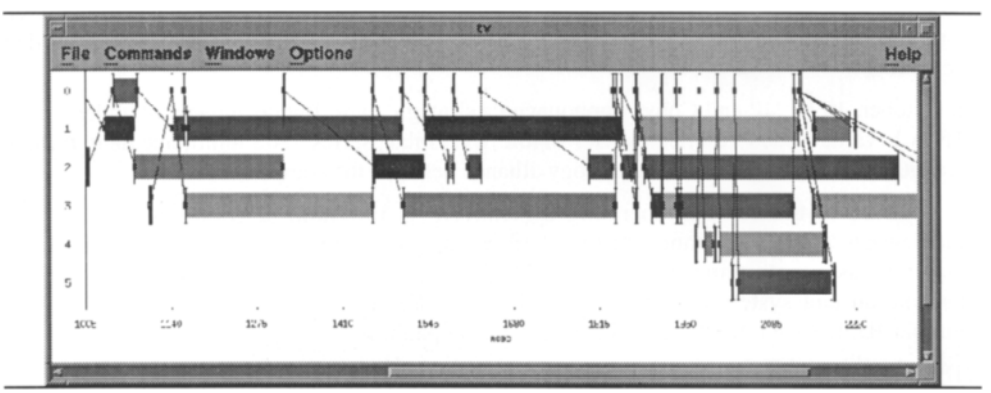


Figure 12. Output from CXtrace.

This figure illustrates the computational and message passing components of an integer sorting algorithm. Activity from different processors is displayed on separate timelines. Message passing *send* and *receives* between processors are indicated by the lines linking different timelines. The display also points out the latencies inherent in the message passing portions of the application. Disproportionate latencies or deviations in anticipated communication patterns or event orderings indicating potential performance problems become evident in such displays.

5. THE CONVEX AND HEWLETT-PACKARD ALLIANCE

Convex Computer Corporation is a leading worldwide supplier of high-performance computing technologies. Founded in 1982, the company provides supercomputing power to an increasing number of customers in a broad range of industries and research organizations. The company's targeted market segments are discrete manufacturing, which includes automotive, aerospace and construction; government/defense; chemistry; petroleum, university/research, environmental, and financial and commercial.

With the increasing complexity of technology today, and the accelerated cycle time for new products, more and more companies are forming partnerships to give customers faster access to new technologies. Companies can share expertise to reduce both the time to market for new products, and reduce their cost to customers. In March of 1992, Convex and Hewlett-Packard Company formed an extensive business and technology alliance that includes:

- Convex's adoption based on HP's 7200, 8000 and 9000 CPU's of HP's PA-RISC technologies for the SPP1000, SPP1200 and future Exemplar computers;
- An exchange of core technologies between the two companies, including HP's RISC compiler technology and operating system, and Convex's supercomputer compiler technology;
- A five percent interest in Convex by HP.

The first of Convex and HP's business and technology agreements included HP's nearly $18 million investment in Convex. As part of this agreement, Convex adopted HP's Precision Architecture-RISC (PA-RISC) technologies for use in its SPP systems. Both companies also agreed to exchange core technologies, with Convex providing leading supercomputer compiler technology and HP providing leading RISC microprocessor and compiler technology.

In October, 1994, HP and Convex announced a strategic, world-wide agreement under which HP is a value-added reseller (VAR) of the Exemplar Series. The announcement further strengthened the business and technology alliance between the companies.

In addition to the HP-UX operating system, HP is providing a comprehensive set of middleware, a suite of standards-based libraries, commands, and utilities that define a complete user environment. These include Motif, X Window System, and software distribution and system administration utilities. Middleware provides the customer with familiar HP-UX user interfaces and application compatibility.

HP and Convex are founding members of the Precision RISC Organization (PRO), an association of industry-leading companies that are delivering PA-RISC benefits to customers worldwide. Both HP and Convex are also committed to the adoption of the PRO ABI (Application Binary Interface) standard. PRO API (Application Program Interface) and ABI standards are expected to be approved soon. These interfaces will let users easily move software applications among different PA-RISC platforms.

The complementary Convex and HP technologies will provide fully compatible UNIX-based RISC products extending from the desktop to high-performance supercomputing.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Hardware Performance of the VPP500 Parallel Supercomputer

Akira Nodomi, Masayuki Ikeda, Moriyuki Takamura, [a] and Kenichi Miura [b]

[a] Supercomputer Development Department, Fujitsu Limited
1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, Japan
{*nodomi, ikeda, takamura*}*@ayame.mfd.cs.fujitsu.co.jp*

[b] SuperComputer Group, Fujitsu America, Inc.
3055 Orchard Drive, San Jose CA 95134, U.S.A.
kenm@fai.fujitsu.com

This paper describes the performance of the Fujitsu VPP500, at both the hardware and application levels. The VPP500 is a distributed memory parallel supercomputer that is based on high performance vector processing elements interconnected by a crossbar network. First, we measured the performance of basic aspects of the VPP500 and confirmed its vector performance and its high data transfer performance among processing elements.

The replicated functional units in each of the vector pipelines give the VPP500 its high vector performance. The interprocessor communication hardware achieves bandwidth that approaches the peak rate (400 Mbytes/s) for representative transfer patterns and provides good throughput even for relatively small-sized data. We also measured the performance of the VPP500 using the NAS Parallel Benchmark Suite.

1. Introduction

The performance of any parallel computer centers on the computational power of the individual processors and the capability of the interprocessor communication hardware.

We focus on the above two points and present and evaluate measurements taken on the VPP500.

In the next section, we begin with a brief overview of the VPP500. Section 3 gives an outline of the performance measurements and methodology. Then we describe PE vector and inter-PE data transfer performance in Section 4 and Section 5. Finally, we describe the system performance of VPP500, with the NAS Parallel Benchmark as an example. We summarize the results of these evaluations and conclude the paper in Section 7.

A detailed description of the VPP500 system architecture is discussed in [9].

2. Overview of the VPP500

The VPP500 is Fujitsu's newest parallel supercomputer [6]. The most characteristic point of the VPP500 is the vector-parallel aspect of its architecture -- the processing elements (PEs) are equipped with high performance vector processing units and their own local memories. In contrast, other parallel supercomputers with high-speed vector

processing capabilities are based on a shared memory architecture, and therefore have limited scalability (e.g., [8,10]).

The following summarizes the basic characteristics of the VPP500:

- **parallel computer**

  The VPP500 scalably supports from 4 to 222 PEs. The system can be configured and upgraded with any number of PEs to match computational requirements.

- **vector processing**

  The PEs are based on a high performance 1.6 GFLOPS vector unit. The total size of the vector register is 128 Kbytes. These allow software assets to execute at high speed on a single PE by simply recompiling without the need to parallelize codes.

- **distributed memory**

  Main memory is distributed among each of the PEs. As a result, performance degradation observed on shared memory multiprocessors due to interprocessor memory conflicts is eliminated.

  Each PE comes with up to 1 Gbyte of main memory, for maximum of 222 Gbytes in a full configuration. The system can thus handle very large scientific problems because of its memory capacity.

- **crossbar network**

  Due to its advantageous characteristics, we selected the crossbar as the topology for the interconnection network. By nature communication between any two PEs are all homogeneously supported by the crossbar network. This greatly simplifies optimization of data transfer scheduling and dynamic allocation of physical PEs for parallel jobs.

Figure 1 shows the system organization of the VPP500 and its PE. Table 1 shows the basic specifications.

3. Outline of Performance Evaluation

The purpose of this evaluation of hardware performance is to characterize

1. PE vector performance and
2. inter-PE data transfer performance,

and to summarize the strengths and weaknesses of the vector and data transfer architectures of the VPP500.

Each PE has a hardware sub-μsec resolution timer. We used this timer to measure execution times, and then calculated the metrics described in following sections. Measurements were taken by noting values of the timer at the start and end of each measurement and taking the difference of the two.

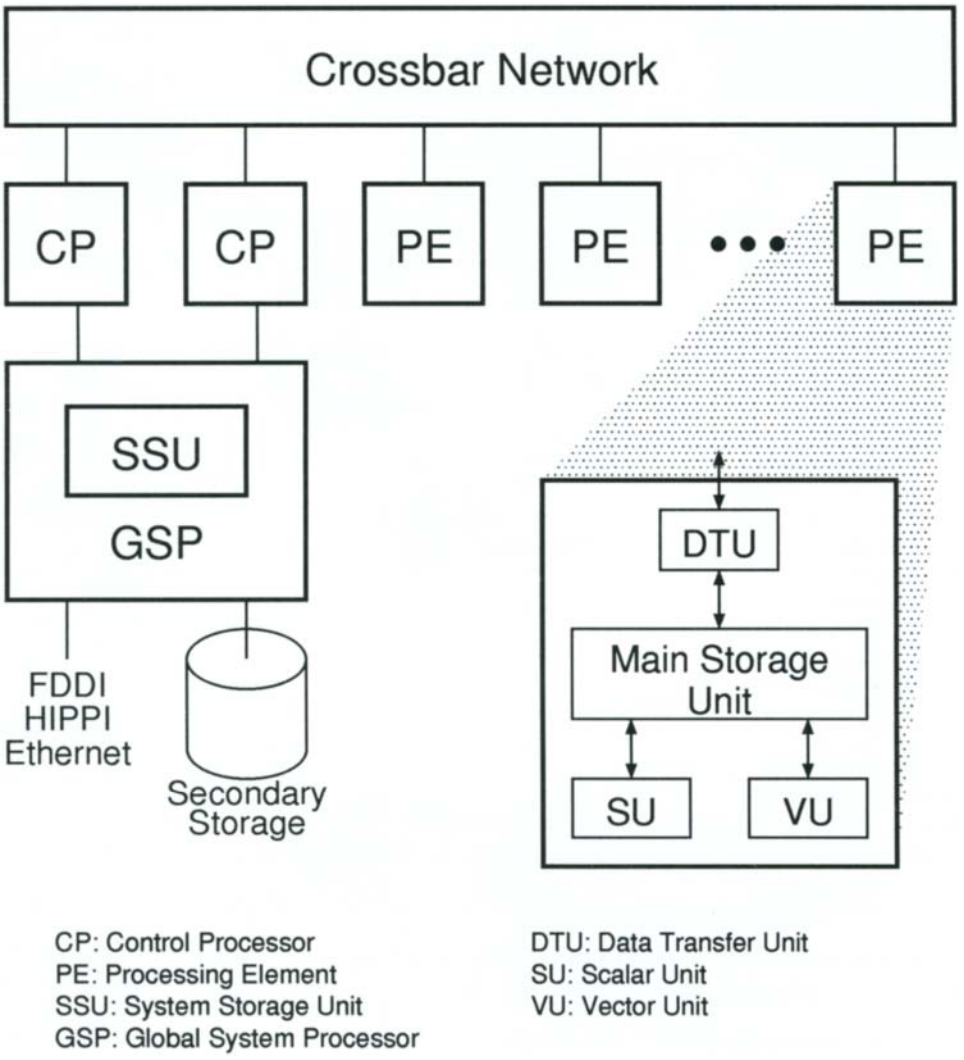


Figure 1. System Organization of the VPP500

Table 1
VPP500 Specifications

System			
number of PEs	4 - 222	performance range	6.4 - 355 GFLOPS
machine cycle time	10 ns	main memory	1 - 222 Gbytes
network topology	crossbar		
Processing Element (PE)			
Scalar unit			
instruction form	LIW ††	max. degree of parallelism	3
instruction size	8 bytes		
peak integer performance	300 MOPS §	peak F.P. performance	200 MFLOPS
instruction cache size	32 Kbytes	data cache size	32 Kbytes
Vector unit			
arithmetic pipelines	add×1 †	load/store pipelines	load×1 †
	multiply×1 †		store×1 †
	divide×1 ‡	mask pipeline	mask×1 †
vector register size	128 Kbytes		
peak performance	1.6 GFLOPS		
Main storage unit			
maximum capacity	1 Gbytes	interleave	32 ways
memory cycle time	20 ns		
Data transfer unit			
transfer rate send *	400 Mbytes/s	transfer rate receive *	400 Mbytes/s

††: Long Instruction Word
†: 8 results per cycle
§: Million Operations Per Second
‡: 8/7 results per cycle
* : send and receive can operate concurrently

3.1. Metrics

3.1.1. Metrics for PE Vector Performance

There are two metrics that characterize the performance of vector processing [1]:

- **throughput**: the number of elements a vector operation processes in unit time.
- **half performance length** ($n_{1/2}$): the vector length which achieves half of observed peak performance.

We considered the above two metrics and measured the execution time of each vector instruction using the hardware sub-μsec resolution timer. Measurements were taken for combinations of the parameters shown below:

1. vector arithmetic parameters
   - operation: addition, subtract, multiplication, divide, find, sum
   - precision of elements: single precision (4 bytes), double precision (8 bytes)
   - type of operands: scalar operand, vector operand

2. vector load/store parameters

   - operation: load, store
   - precision of elements: single precision (4 bytes), double precision (8 bytes)
   - access pattern: contiguous, stride, indirect

These operations were not chained but run individually so that the influence of start-up times was significant in general.

To investigate the relation between vector arithmetic operations and vector load/store operations, we selected some vectorizable loops from [5] and ran them on a VPP500 PE. These loops are coded to reflect a given ratio defined in the below equation:

$$\begin{aligned} ratio &= \frac{number\ of\ operations}{number\ of\ operands} \\ &= \frac{number\ of\ vector\ arithmetic\ operations}{number\ of\ vector\ load/store\ operations}. \end{aligned}$$

Code for loops with ratios less than one are:

```
C  ONE OPERATION - THREE OPERANDS  RATIO = 1/3
      DO 41020 I=1, N
       A(I)=B(I)+C(I)
41020 CONTINUE

C  TWO OPERATIONS - FOUR OPERANDS  RATIO = 1/2
      DO 41022 I=1, N
       A(I)=B(I)*C(I)+D(I)
41022 CONTINUE

C  THREE OPERATIONS - FIVE OPERANDS  RATIO = 3/5
      DO 41023 I=1, N
       A(I)=B(I)*C(I)+D(I)*E(I)
41023 CONTINUE

C  TWO OPERATIONS - THREE OPERANDS  RATIO = 2/3
      DO 41024 I=1, N
       A(I)=C0*B(I)+C(I)
41024 CONTINUE
```

Code for loops with ratios greater than one are:

```
C  THREE OPERATIONS - TWO OPERANDS  RATIO = 3/2
      DO 41031 I=1, N
       Y(I)=C0+X(I)*(C1+X(I))
41031 CONTINUE

C  FIVE OPERATIONS - TWO OPERANDS  RATIO = 5/2
      DO 41032 I=1, N
```

```
      Y(I)=C0+X(I)*(C1+X(I)*(C2+X(I) ))
41032 CONTINUE

C  SEVEN OPERATIONS - TWO OPERANDS  RATIO = 7/2
      DO 41033 I=1, N
       Y(I)=C0+X(I)*(C1+X(I)*(C2+X(I)
     *             *(C3+X(I)         )))
41033 CONTINUE
       .
       .
       .
C  NINETEEN OPERATIONS - TWO OPERANDS  RATIO = 19/2
      DO 41039 I=1, N
       Y(I)=C0+X(I)*(C1+X(I)*(C2+X(I)
     *             *(C3+X(I)*(C4+X(I)
     *             *(C5+X(I)*(C6+X(I)
     *             *(C7+X(I)*(C8+X(I)
     *             *(C9+X(I)         )))))))))
41039       CONTINUE
```

The aim of these loops is to show the correlation among vector length, the operation to operand ratio, and execution time for vector operation streams that include raw vector additions, vector multiplications, and vector load/store operations. We show these results in Section 4.2.

3.1.2. Metrics for Inter-PE Data Transfer Performance

The data transfer unit (DTU) in the PE handles interprocessor communication, which frees the scalar unit for computation. The DTU handles both send and receive operations concurrently.

There are four important metrics to evaluate inter-PE data transfer [3,2]:

- **transfer start-up time**: the time between the issue of a data transfer command by the scalar unit and the arrival of the first datum.
- **transfer throughput**: the amount of data which the DTU transfers in unit time.
- **half performance length** ($n_{1/2}$): the amount of data for which half of observed peak performance is achieved.

The communication patterns measured are shown in Figure 2.

(1) data transfer between a local PE and a remote PE: Only one data transfer takes place at a time in this pattern.

(2) simultaneous 1-to-1 data transfer among multiple ($N, N \geq 2$) PEs: PE_i transfers data to PE_{i+1}, and PE_{N-1} sends to PE_0. All PEs transfer data simultaneously. The total number of data transfers is N. Note that the VPP500's crossbar network allows any pair of logical PEs to be treated as physical neighbors (e.g., PE_i and PE_{i+1}).

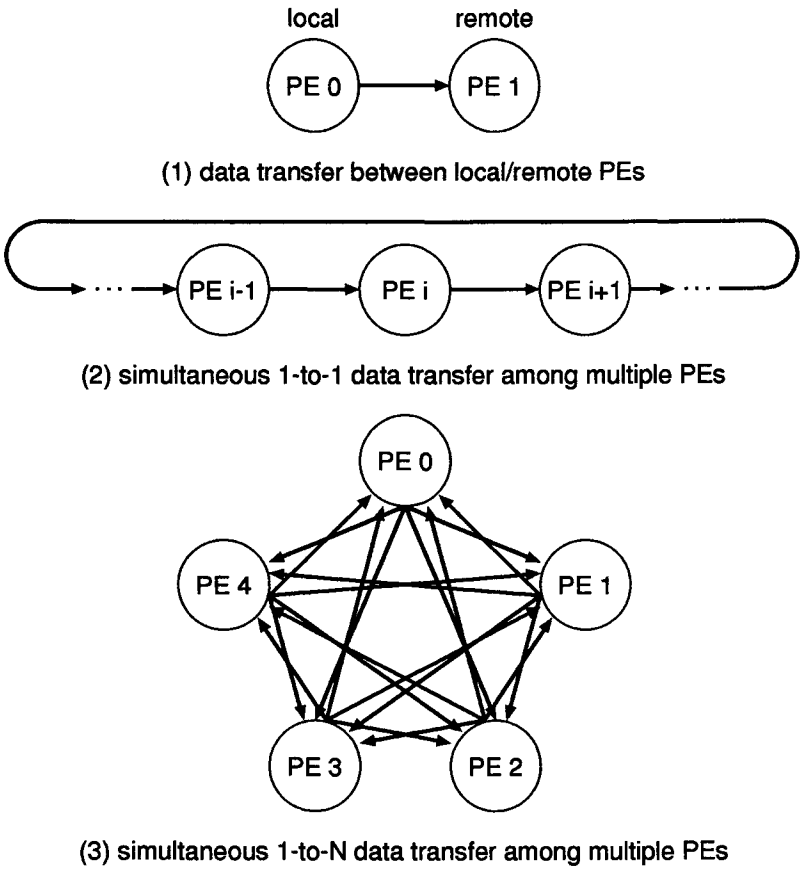


Figure 2. Communication Patterns of Data Transfer

(3) simultaneous 1-to-N data transfer among multiple ($N, N \geq 2$) PEs: PE_i transfers data to all PEs, including itself, and all PEs transfer data simultaneously. The total number of data transfers is N^2.

In addition to the communication pattern, we considered the different transfer modes of DTU:

- memory access pattern: contiguous, stride, indirect, sub-array
- width of elements: 4 bytes, 8 bytes
- type of data transfer: read, write

Data transfer performance was measured for combinations of the above mentioned communication patterns and transfer modes.

Table 2
Throughput of Vector Operations

vector operations	throughput		vector operations	throughput	
	VL=8	VL=2048		VL=8	VL=2048
add/subtract	0.320	7.288	find max/min	0.267	7.186
multiply	0.320	7.288	summation reduction	0.157	6.715
divide	0.190	1.121			
load	0.205	6.966	store	0.222	7.038

unit: elements/cycle

4. PE Vector Performance

4.1. Performance of Individual Vector Operation

First we evaluated the throughput of individual vector operations. Table 2 shows results of typical vector operations.

Vector lengths (VLs) of 8 and 2048 elements were chosen. There are two reasons for these lengths. One is that all pipelines of the vector unit — the arithmetic pipelines, load/store pipelines, and mask pipeline — produce eight results per cycle (see Table 1), which results in no difference between the execution times of one element and eight elements. The other reason is that the length of the vector registers is reconfigurable up to a maximum of 2048.

Table 2 shows that for sufficiently long vector lengths (VL=2048), most operations achieve a throughput of $7.0 \sim 7.3$ elements per cycle, a degradation of only 10% compared to peak performance. We can regard these results as achieving high performance even though these measurements were made for only individual vector operations.

Other vector operations show poor performance. The reasons are that:

- for divide: the issue frequency for the divide pipeline is 7 cycles, so that the maximum throughput drops to 1/7 of the add and multiply arithmetic pipelines, and
- for sum: the operation returns a result to the scalar unit, so the execution time is longer than other operations.

4.2. Performance of Vector Loops

Figures 3, 4, and 5 show the execution results of the vector loops discussed in Section 3.1.1. Figures 3 and 4 are graphs of MFLOPS vs. vector length. Figure 5 is a graph of MFLOPS vs. the *ratio* of the number of operations to the number of operands.

For ratios less than one, i.e., the number of arithmetic operations is less than the number of load/store operations, performance is lower than one-half of peak (See Figure 3). This is caused mainly by the lack of a second load pipeline. The effect can be minimized by utilizing the large vector register capacity to hold reusable data and by chaining vector arithmetic operations.

As shown in Figure 4, $n_{1/2}$ is about 170 for the ratio 3/2, between 100 and 130 for ratios between 5/2 and 15/2, and less than 100 for ratios greater than 17/2. As shown in Figure 5, $r_{1/2}$, which we define as the ratio which achieves half of the observed peak performance, is less than one for lengths greater than 512 and less than three for lengths

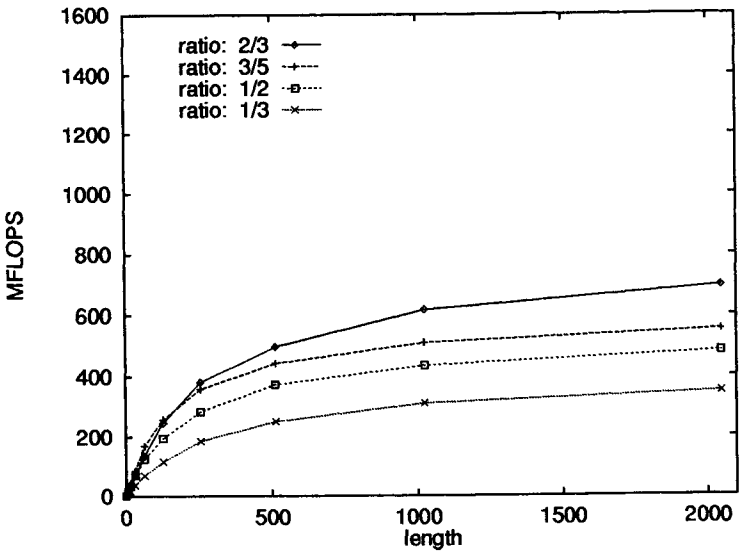


Figure 3. Results of DO loops 41020 - 41024 (ratio<1)

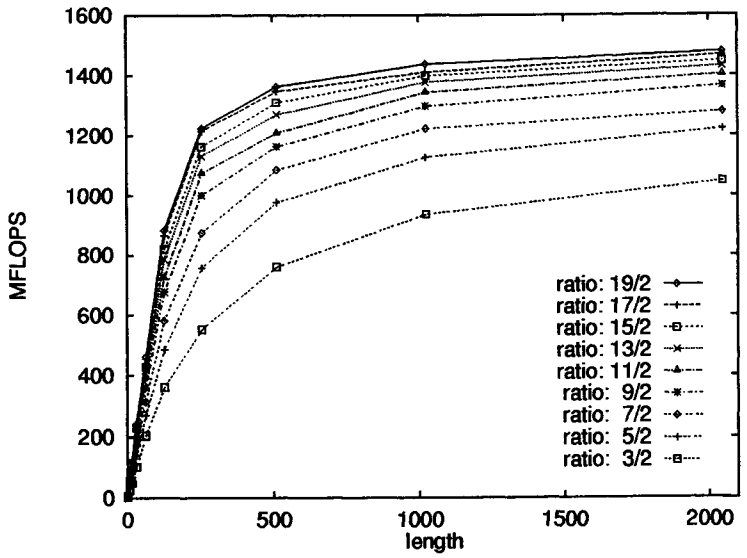


Figure 4. Results of DO loops 41031 - 41039 (ratio>1)

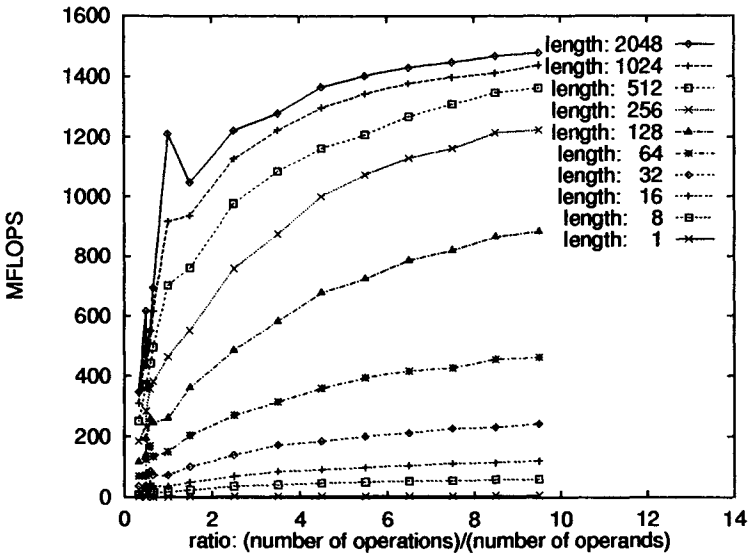


Figure 5. Results of DO loops 41020 - 41039

greater than 32. This shows that chaining just a few vector operations achieves half of the observed peak performance as well as moderate values of $n_{1/2}$.

4.3. Evaluation

In Section 4.1, we verified the high potential for vector performance on the VPP500. In fact, all pipelines in vector unit are eight elements wide and each vector unit has 128 Kbytes vector registers (see Table 1), and this presents a great potential for high vector performance.

To fully utilize this potential, it is necessary to chain vector operations and to use long vectors. We confirmed this effect in Section 4.2. The disadvantage that vector unit has only one load pipeline can be reduced by these methods.

5. Inter-PE Data Transfer Performance

In this section, we summarize the results of the measurements for each of the three communication patterns mentioned in Section 3.1.2. We used two PEs for data transfer between local and remote PEs (Section 5.1), and 23 PEs for simultaneous 1-to-1 and 1-to-N data transfer among multiple PEs (Sections 5.2 and 5.3).

5.1. Data Transfer between Local/Remote PEs

For data transfer between local and remote PEs, we did not find any meaningful difference between read and write. Hence we limit our discussion here to write transfers. We measured transfer performance for the combinations of local and remote modes shown below.

local PE	remote PE	
contiguous	contiguous	◇
contiguous	8-byte stride	
contiguous	256-byte stride	
8-byte stride	contiguous	
8-byte stride	8-byte stride	
8-byte stride	256-byte stride	
256-byte stride	contiguous	
256-byte stride	8-byte stride	
256-byte stride	256-byte stride	◇
indirect	indirect	◇

In this case, indirect lists used for the indirect mode indicate consecutive addresses, so that accesses to memory do not conflict with each other. Because we found that some modes behave similarly, the results shown in this paper include only the three modes marked "◇" in the above table.

Figure 6 and Figure 7 show the results for four-byte elements and eight-byte elements for data transfer between local and remote PEs.

There are two patterns in Figure 6 while Figure 7 has only one pattern. For 256-byte stride access mode, all accesses to memory conflict with each other. The memory bandwidth between the DTU and the main storage unit, therefore, degrades to one request per two cycles because the memory cycle is two cycles (see Table 1). For indirect access mode, memory bandwidth degrades to one requests per two cycles because the DTU can only generate four requests for data per eight cycles in indirect access mode. These are identical to the throughput of the crossbar for eight-byte data. This is because the memory bandwidth, which is eight bytes per two cycles, is equal to the bandwidth of the crossbar, which is four bytes per cycle. We cannot fully utilize the throughput of the crossbar if we use four-byte data because the memory bandwidth drops to four bytes per two cycles.

5.2. Simultaneous 1-to-1 Data Transfer among Multiple PEs

The simultaneous 1-to-1 data transfer among multiple PEs also shows little difference between read and write accesses so we show again only the results of write accesses. Figure 8 and Figure 9 show the results of four-byte elements and eight-byte elements for simultaneous 1-to-1 data transfer among multiple PEs.

The reason for the difference between Figure 8 and Figure 9 is similar to that between Figure 6 and Figure 7.

The throughputs of 256-byte stride and indirect access mode of Figure 8 are lower than those of Figure 6, because each DTU has to send and receive data for this communication pattern.

For simultaneous 1-to-1 data transfer among multiple PEs, the throughput of the indirect access mode is higher than that of the 256-byte stride access mode because fetch requests and store requests from the DTU to memory can be overlapped partially on indirect access mode.

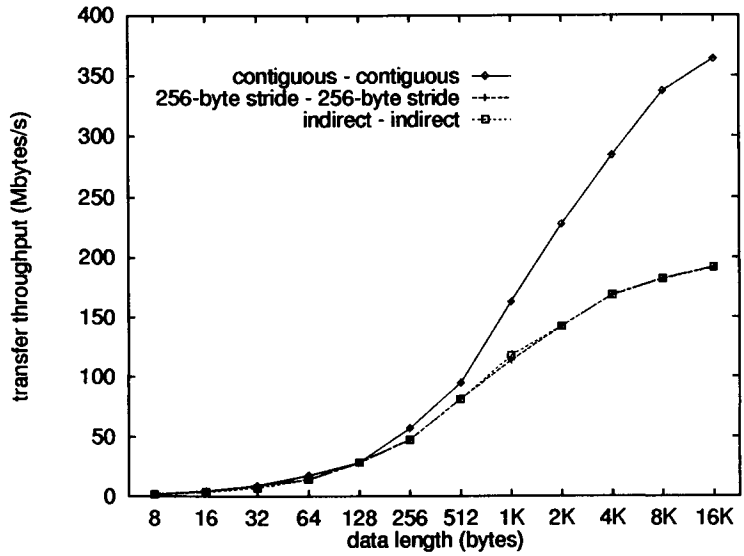


Figure 6. Data Transfer between Local/Remote PEs (four-byte data, write access)

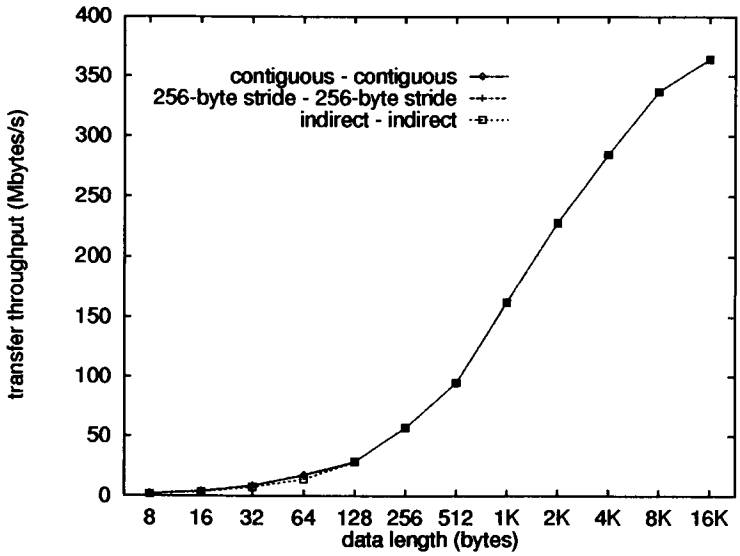


Figure 7. Data Transfer between Local/Remote PEs (eight-byte data, write access)

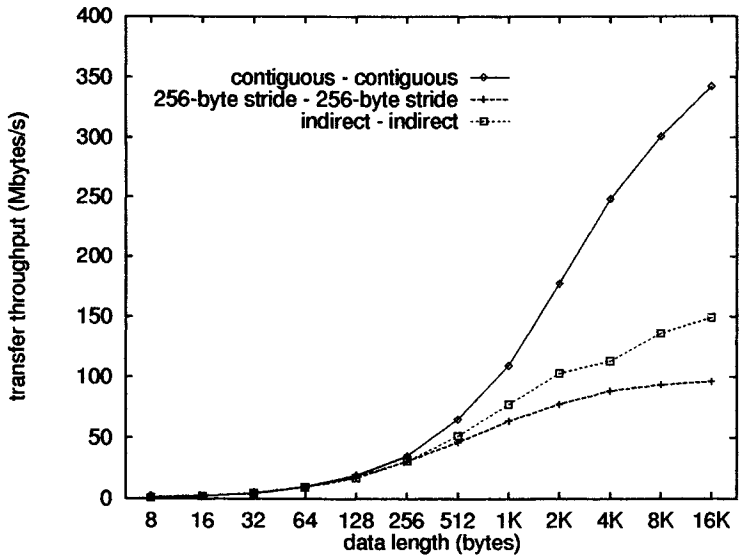


Figure 8. Simultaneous 1-to-1 Data Transfer among Multiple PEs (four-byte data, write access)

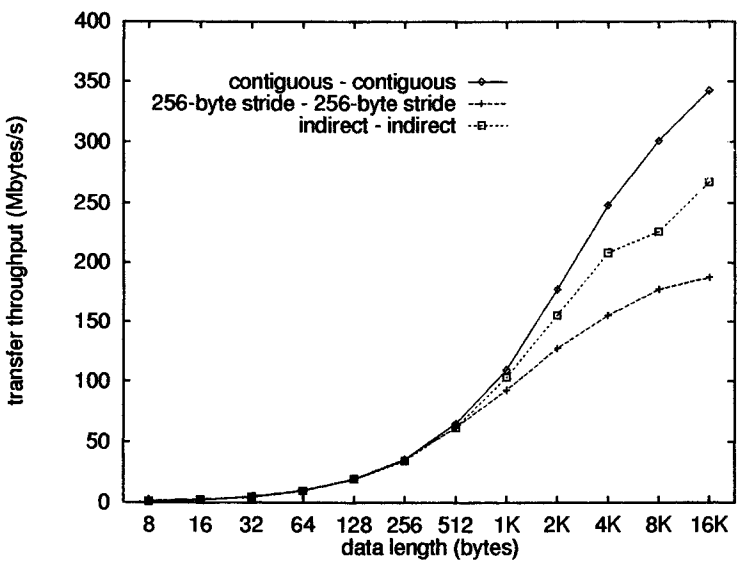


Figure 9. Simultaneous 1-to-1 Data Transfer among Multiple PEs (eight-byte data, write access)

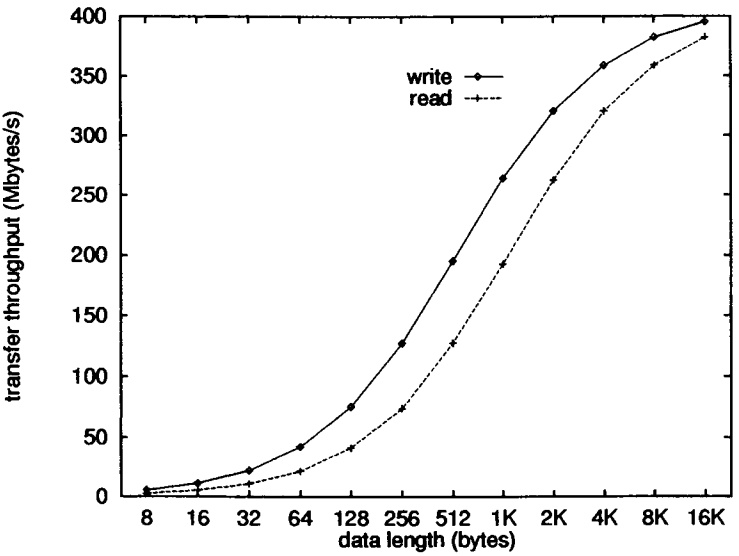


Figure 10. Simultaneous 1-to-N Data Transfer among Multiple PEs (eight-byte data, contiguous–contiguous mode)

5.3. Simultaneous 1-to-N Data Transfer among Multiple PEs

For simultaneous 1-to-N data transfer among multiple PEs, there is an obvious difference between read access and write access. Figure 10 shows the results of read and write access for simultaneous 1-to-N data transfer among multiple PEs.

The DTU reads or writes data as follows:

- read access (PE_i reads data from PE_j):
  1. send packet from PE_i to PE_j to request data
  2. send packet with data from PE_j to PE_i
- write access (PE_i writes data to PE_j):
  1. send packet with data from PE_i to PE_j

This means that two packets are needed for each read access while only one packet is needed for each write access. The throughput degrades because many packets are generated for this communication pattern.

We find that the $n_{1/2}$ values of Figure 10 is much better than that of Figure 9. The former is about 500 bytes while the latter is about two Kbytes. There are many contiguous data transfers on simultaneous 1-to-N data transfer among multiple PEs, so the observed transfer start-up time per packet is very small. As a result the total transfer throughput becomes higher.

5.4. Evaluation

Table 3 shows a summary of the inter-PE data transfer performance of the VPP500.

Table 3
Data Transfer Performance

transfer configuration	transfer mode	read/ write	start-up time per packet (μsec)	throughput † (Mbytes/s)	$n_{1/2}$ (bytes)
data transfer between local/remote PE	contiguous/ contiguous	read	4.28	363	1.3K
		write	3.60	364	1.2K
	256-byte stride/ 256-byte stride	read	4.28	363	1.3K
		write	3.60	364	1.2K
	indirect–indirect	read	4.42	361	1.3K
		write	4.19	364	1.2K
simultaneous 1-to-1 data transfer among multiple PEs	contiguous/ contiguous	read	8.25	331	2.5K
		write	6.60	342	2.0K
	256-byte stride/ 256-byte stride	read	8.25	184	1.6K
		write	6.60	187	1.1K
	indirect–indirect	read	8.25	275	2.5K
		write	6.60	268	2.0K
simultaneous 1-to-N data transfer among multiple PEs (N = 23 PEs)	contiguous/ contiguous	read	2.94	383	1.0K
		write	1.40	395	500
	256-byte stride/ 256-byte stride	read	2.97	103	250
		write	1.40	198	200
	indirect–indirect	read	2.88	140	150
		write	1.44	285	250

†: transfer data = 16 Kbytes

In Figures 6 - 10 and Table 3 we find that the data transfer throughput comes very close to the 400 Mbytes/s peak using the simple modes such as contiguous access.

We also find that the $n_{1/2}$ of simultaneous 1-to-N data transfer among multiple PEs is very small. This means that the VPP500 can provide good throughput even when sending small amounts of data.

Furthermore, the DTU of the VPP500 supports various transfer modes (see Section 3.1.2). This high versatility simplifies software and frees the scalar unit from complex processing.

In addition, since the crossbar does not suffer from network contention, the number of PEs has little influence on total transfer throughput.

6. NAS Parallel Benchmark Results

As for the performance of the VPP500 at the application level, we have reported the results of a linear equation solver in [7] . Fujitsu America, Inc., Fujitsu Systems Business of America, and Fujitsu Limited jointly conducted benchmark tests using the NAS Parallel Benchmark Suite [11]. Researchers at NASA Ames Research Center prepared this

Table 4
NAS Parallel Benchmark Results

Name	Class A			Class B		
	PEs	Elapsed Time (sec.)	Ratio to Y-MP (1cpu)	PEs	Elapsed Time (sec.)	Ratio to C-90 (1cpu)
EP	1	44.25	2.85	1	176.64	1.05
	4	11.24	11.21	4	44.52	4.16
	8	5.67	22.22	8	22.36	8.29
	16	2.87	43.97	16	11.26	16.45
	32	1.46	86.44	32	5.68	32.62
	64	0.75	167.92	64	2.88	64.33
MG	4	1.58	14.06	4	7.53	5.02
	8	0.86	25.84	8	4.07	9.28
	16	0.49	45.35	16	2.35	16.07
	32	0.33	67.33	32	1.56	24.21
CG	1	5.68	2.10			
	2	3.06	3.90	2	104.51	1.18
	4	1.72	6.93	4	55.40	2.22
	8	1.04	11.46	8	31.80	3.86
	16	0.80	14.90	15	20.85	5.89
				30	15.21	8.08
FT	4	2.93	9.82			
	8	1.45	19.81			
	16	0.75	38.51	16	7.95	16.03
	32	0.40	72.47	32	4.07	31.33
	64	0.24	121.91	64	2.18	58.54
IS	1	2.19	5.24			
	2	1.57	7.28			
	4	1.10	10.44	4	3.70	3.49
	8	0.92	12.50	8	3.03	4.26
LU	1	146.89	2.27	1	591.05	1.10
SP	1	176.75	2.67	1	664.76	1.07
	2	108.85	4.33	2	417.78	1.71
	4	57.24	8.24	4	228.37	3.12
	8	29.87	15.79	8	120.05	5.94
	16	20.99	22.47	17	53.12	13.42
				34	39.01	18.28
BT	2	75.17	10.54			
	4	39.14	20.25			
	8	19.82	39.98			
	16	9.99	79.32	17	37.26	33.85
	32	5.09	155.68	34	18.82	67.02
	64	2.66	297.89	51	12.61	100.03

benchmark suite in order to compare the performance of various parallel platforms.

The main emphasis of this benchmark suite is to test the numerical algorithms frequently encountered in the computational fluid dynamics applications in use at NASA Ames REsearch Center. The benchmark suite consists of eight programs: Embarrassingly Parallel (EP), Multigrid (MG), Conjugate Gradient (CG), Fourier Transform-based PDE Solver (FT), Integer Sort (IS), Lower-Upper Triangular SSOR Solver (LU), Scalar Pentadiagonal Solver (SP) and Block Tridiagonal Solver (BT). Two different problem sizes are specified for each benchmark, namely, Class A (small size) and Class B (large size). Two of the programs, EP and IS, were originally implemented with pure scalar algorithms, so that vectorized algorithms had to be developed for these programs prior to parallelization.

Table 4 summarizes the most recent official results of the NAS Parallel Benchmarks for VPP500 [11]. It should be noted that the most data communication intensive programs for any distributed memory parallel processing system are FT, SP, and BT, all of which involve the transposition of three-dimensional arrays. Here, the overall system performance is largely determined by how efficiently these arrays can be transposed through the interconnection network. The crossbar interconnection network of the VPP500 has proven to be very effective in these three benchmark programs, as is clear in Table 4. This is a on-going effort, and the updated performance results will continue to be reported to NASA Ames Research Center.

7. Conclusion

We have measured and evaluated the vector and data transfer performance of the VPP500. We have also conducted extensive benchmark tests using the NAS Parallel Benchmark Suite. The results are summarized below.

With regard to the vector performance of the VPP500, all pipelines in vector unit are eight elements wide, and each vector unit has 128 Kbytes vector register. This presents a great potential for high vector performance.

To make full use of this potential, it is necessary to chain vector operations and to use the large capacity of the vector registers efficiently. The drawback of having only one load pipeline can be reduced by chaining vector arithmetic operations and using long vectors.

With regard to the data transfer performance, the transfer throughput comes very close to the peak rate of 400 Mbytes/s for simple modes such as contiguous access. The VPP500 can also provide good throughput even for relatively small-sized data.

The Data Transfer Unit of the VPP500 supports various data transfer modes. Since the VPP500 has a crossbar as its interconnection network, the number of PEs has little influence on total transfer throughput.

Finally, with regard to the performance at the application level, the VPP500 can be regarded as one of the top-performing parallel platforms for the NAS Parallel Benchmarks, as indicated in both Table 4 and in [11]. The key architectural features of the VPP500 which contribute to the high performance results are: the vector processing capability and large-sized memory of each PE, and the flexible and high throughput crossbar network.

REFERENCES

1. S. Fernbach, "Supercomputer", Elsevier Science Publishers B. V. (North-Holland), 1986.
2. R. Hockney, "Performance Parameters and Benchmarking of Supercomputers", *Parallel Computing*, vol. 17, pp. 1111–1130, Dec. 1991.
3. R. W. Hockney, and C. R. Jesshope, "Parallel Computers 2", Adam Hilger Ltd, Bristol., 1988.
4. H. Iwashita, S. Okada, and H. Nagakura "VPP FORTRAN and Parallel Programming on the VPP500 Supercomputer", to appear in *Proceedings for Poster Sessions*, International Symposium on Parallel Architecture, Algorithms, and Network (ISPAN'94), Kanazawa, Japan, Dec. 1994.
5. J.M. Levesque and J.W. Williamson, "A Guidebook to FORTRAN on Supercomputers", Academic Press, Inc., 1989.
6. K. Miura, M. Takamura, Y. Sakamoto, and S. Okada, "Overview of the Fujitsu VPP500 Supercomputer", Digest of Papers, COMPCON Spring 93, pp. 128-130, San Francisco, Feb. 1993.
7. M. Nakanishi, H. Ina, and K. Miura, "A High Performance Linear Equation Solver on the VPP500 Parallel Supercomputer", *Proceedings of Supercomputing '94*, pp. 803-810, Washington D.C., Nov. 1994.
8. O. Serlin, *The Serlin Report On Parallel Processing*, No.54, pp.8–13, Nov. 1991.
9. T. Utsumi, M. Ikeda, and M. Takamura, "Architecture of the VPP500 Parallel Supercomputer", *Proceedings of Supercomputing '94*, pp.478-487, Washington D.C., Nov. 1994.
10. T. Watanabe, H. Matsumoto, and P. D. Tannenbaum, "Hardware Technology and Architecture of the NEC SX-3/SX-X Supercomputer System", *Proceedings of Supercomputing '89*, pp. 842–846, Nov. 1989.
11. D. Bailey, E. Barszcz, L. Dagum and H. Simon, "NAS PARALLEL BENCHMARK RESULTS 10-94", NAS Technical Report NAS-94-001, Oct. 1994.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

The balance of power
A brief history of Cray Research hardware architectures

Lester T. Davis

Cray Research, Inc.
1168 Industrial Boulevard, Chippewa Falls, Wisconsin, USA, 54729

When Seymour Cray and a group of associates from Control Data Corporation founded Cray Research in 1972, few people were familiar with the term "supercomputer." But that soon changed. Supercomputers and Cray Research systems became synonymous with the introduction of the new company's first product, the CRAY-1 computer system.

The CRAY-1 system, introduced in 1976, shattered all previous performance records for scientific computing. Its technical and commercial success was due to three major advancements: a fast, 12.5 nsec, processor clock, which improved scalar processing over the CDC 7600 computer by a factor of two; successful vector implementation, which could improve performance over scalar processing by a factor of five to ten; and a large, 1 Mword, memory, which eliminated the need for the high-level language programmer to explicitly allocate arrays in "Large Core Memory." In the most general sense, Cray Research's hardware philosophy was and remains a practical one of preserving balance among the three primary system functions: processing speed, memory size and speed, and I/O bandwidth. Maintaining this balance ensures that no one function becomes a bottleneck that compromises system throughput. And each of these functions has evolved through technical innovations pioneered at Cray Research.

Today an additional factor has come to play an increasingly large role, namely, system cost. Today a successful supercomputer has to be able to deliver high performance with markedly improved price/performance compared to previous generations.

1. THE CRAY-1 SYSTEMS

At Control Data, Seymour Cray had led several successful design teams, which had built the company's most powerful computer systems, including the CDC 6600 and CDC 7600 systems. Nonetheless, these systems were conventional in many aspects of their architecture. The CRAY-1 system was unconventional in as many ways and laid the groundwork for a long line of successful vector supercomputers.

1.1. Vectorization

The CRAY-1 system incorporated vector registers in addition to scalar registers in the CPU, delivering unprecedented speedups compared to other computer systems of the time. Previous systems, with few exceptions, operated exclusively in a scalar mode. One number was loaded into a register, another number was loaded into another register, then the two numbers were operated upon, added, for example, and the sum was stored in a third register and then put into memory. The above scalar sequence required four instructions from a CRAY-1 system operating in scalar mode.

Vectorization, on the other hand, involves performing operations on vectors—lists of numbers—with a single instruction. As a result, four instructions could, for example, add 20 numbers to another 20 numbers to produce 20 sums and store them in 20 memory locations. In 1978 we introduced the CFT Fortran compiler, the first automatically vectorizing language compiler. With CFT, vectorization became accessible to a much larger population of programmers, because it now required less technical expertise to program the CRAY-1 system, although vectorization still required some additional programming work. Moreover, not all applications lend themselves to vector processing, but highly vectorizable applications would run a factor of 50 to 100 times faster on the combined technologies of the CRAY-1 system than on exclusively scalar systems. For this reason, the CRAY-1 computer system became the first commercially successful vector computer, despite its multimillion dollar price tag.

There had been previous attempts at vectorization. Our approach was unique, however, in that we designed a memory-to-register vector system. We had vector registers—very high-speed registers that acted as a form of intermediate storage—connecting on one side to memory and on the other to functional units. The system loads the vector registers, and the functional units operate out of them at a very high speed. Other attempts at vector architectures ended up going from memory to the functional units and then back to memory without any sort of intermediate storage. Later, proponents of microprocessors for high-speed computing would point out the advantages of cache memory. Both vector registers and cache are forms of intermediate storage. Our systems continue to deliver high performance from intermediate storage of operands.

Along with "supercomputer," "parallel processing" was a term rarely heard during the time of the CRAY-1. This term would become widely used more than a decade later, in the context of massively parallel processing, or MPP. In a dependence sense, vectorization is logically equivalent to single-instruction, multiple-data, or SIMD, parallelism. In both cases a single instruction acts on multiple data to produce multiple results. In a single-pipe, vector implementation, hardware streams data through the system continuously. In the SIMD parallel-processing implementation, the multiple data are handled simultaneously by duplicating functional units and data paths. In multi-pipe vector implementations, the multiple data are handled both by streaming and by simultaneous execution. Additional parallelism in Cray Research processors results from having multiple, independent functional units. For example, in a SAXPY operation [y(i) = y(i) + a*x(i)], it is routine using compiler-generated code for the CRAY C90 two-pipe vector computer to sustain ten simultaneously busy functional units by issuing

five vector instructions. Although Cray Research was not an early adopter of the microprocessor-based MPP approach, vectorization taught us the advantages of processing data in parallel. And despite recent inroads made by MPP systems, including our own, into the scientific computing arena, the majority of large-scale scientific computing projects underway around the world today are made practical by vector processing on Cray Research systems.

1.2. RISC strategy

Another term used widely today is "RISC," an acronym for reduced instruction set computing. Although we did not use this term in the 1970s, we were in effect designing a RISC architecture with the CRAY-1 system, and our subsequent systems have retained this feature. RISC architectures minimize the complexity of individual instructions. The instructions don't do much, but are very efficient at what they do, which places some burden on the programmer to direct the more complex operations. The use of a simple instruction set actually predates the CRAY-1 system. Seymour Cray's systems at Control Data also implemented this strategy. One of Seymour's great skills is his ability to write simple, clean, and very efficient instruction sets. Other high-speed systems of the early 1970s tended to have more complex instructions, which made the programmer's job easier but often compromised performance. At Cray Research, we were more interested in speed than in ease of assembly programming.

1.3. Pipelining

Two more architectural features worth mentioning that contributed to the success of the CRAY-1 system and remain part of the Cray Research architecture are pipelining and chaining. Pipelining is a natural complement to vectorization, allowing the CPU to produce a result every clock period. A floating multiply, for example, takes several clock periods. Without pipelining, you would submit an operand and have to wait through several clock periods for the result. In a sense you would be losing some of the advantages of the vector capability. The fully segmented functional units in a Cray Research CPU enable a pair of elements from two vectors being operated upon to enter a given segment of the functional unit during the same clock period in which the previous pair of elements moves into the next segment. Once all segments in a functional unit are occupied, the unit will produce a result each clock period.

1.4. Chaining

Chaining built on this concept, applying it to the level of the functional units themselves. Because the functional units in the CPU can function independently, or in parallel, an add operation can proceed simultaneously with a multiply, for example. As a result, output from an operation upon the contents of two vector registers can be input directly into another functional unit along with the contents of another vector register. The result of the first operation need not be stored first in memory as an intermediate step, then retrieved to a functional unit. This chaining of functional units, without an intermediate storage step, further accelerates processing.

1.5. Memory

The CRAY-1 system also incorporated semiconductor memory instead of the magnetic core memories commonly used at the time, as in the CDC 6600 and

7600 systems. This hardly seems worth mentioning today, but at the time the choice of technologies had a significant impact. Although it did not deliver significant performance gains, SRAM semiconductor technology consumed less power than core memory, cost less, and enabled us to reduce the physical space allotted to memory, which in turn enabled us to pack more memory into a given space. A later variation of the system, the CRAY-1M system, used metal oxide silicon (MOS) memory in place of bipolar semiconductor memory. This enhancement allowed a larger directly addressable memory, improved system reliability, and allowed us to offer performance equal to the CRAY-1S system (see below) at about half the price.

The CRAY-1 system was instantly recognizable because of its cylindrical shape and celebrated "bench." The bench surrounding the machine housed the power supplies, and the cylindrical shape actually was integral to the performance of the system. Concentrating the wiring near the inside of the cylinder minimized wire lengths which reduced signal propagation times.

1.6. I/O

Even with the CRAY-1 system we were very conscious of the need for balance and made sure that I/O capacity was appropriately matched to the CPU and memory capabilities. Eventually a separate I/O subsystem, to drive peripheral equipment and communicate with the CPU, was added, along with three more Mwords of memory, to make the CRAY-1S system.

2.0. THE CRAY X-MP COMPUTER SYSTEMS

Even though vectorization can be considered an early form of parallel processing, the CRAY-1 computer system differed significantly from what we today consider parallel computers: it had only a single CPU. Parallel processing, as the term is used today, entered the Cray Research product line with the 1983 introduction of the CRAY X-MP system. This system retained vectorization and the other innovations that characterized the CRAY-1 system and extended parallel processing to multiple CPUs.

2.1. Parallel processing

The CRAY X-MP system was introduced with two CPUs; later four-processor models became available. Even with this modest number of processors, the CRAY X-MP system presented us with the challenge that all architects of parallel systems must face: providing efficient interprocessor communication and synchronization. We met this challenge through the use of shared registers. The CPUs in the system had access to a common set of registers that did not move data back and forth but were dedicated to interprocessor communication and synchronization. Shared registers became a standard feature of the Cray Research architectures that descended from the CRAY X-MP system.

In addressing the issues surrounding parallel processing we developed software technologies we call macrotasking, microtasking, and Autotasking. We have continued to refine these technologies, and they remain available on our parallel-

vector systems. To macrotask a job across multiple processors, the programmer uses Fortran-callable subroutines to define and synchronize tasks. The system then launches subjobs that compete for CPUs. Microtasking, instead of specifying portions of a program to be handled by different CPUs, looks for idle CPUs on which to execute parts of the program. Microtasking is a dynamic function that operates at run time and is used primarily with Fortran DO-loop-level routines. In microtasking the user manually identifies parallel loops and inserts directives that specify which variables are private and which are shared. As the program runs, each iteration of a parallel loop is executed by the next available CPU. Autotasking involves essentially the same steps as microtasking, but it operates automatically, with no user intervention required. It needs several passes through a program to analyze the program and divides up the work as it sees fit.

2.2. Compressed index and gather/scatter

Compressed index and gather/scatter functions also were introduced with the CRAY X-MP system. These functions are useful in many applications, such as structures, computational fluid dynamics, particle-in-cell, molecular dynamics, quantum chemistry, and many others, whenever irregular data structures or access patterns are required. Many engineering problems of practical importance are not well described by regular meshes; complicated geometries often are most easily described by irregular meshes or finite elements, both of which lead to gather/scatter operations. Compressed index provides a means of creating an index of locations of a subset of the data. The gather function then loads only the desired subset of the data into adjacent elements of a vector register, which are then fed continuously into a functional unit. The scatter function writes adjacent elements of a vector register into an arbitrary set of memory locations. Operating only on the desired subset of the data conserves system resources.

2.3. Solid-state storage

Concurrent with the CRAY X-MP introduction, we announced a peripheral device, the SSD solid-state storage device. Often referred to by users as a solid-state disk, the SSD functioned as a very large directly addressable memory. Its advantage over disk storage was its very high data-transfer speeds. At the time, transfer rates from disk storage typically were about 4 million bytes per second, compared to the SSD's transfer rates of up to 1250 million bytes per second. Unlike some other high-performance systems, Cray Research systems have always relied on real memory; we have not offered demand page virtual memory options. Real memory ensures the best possible memory performance but limits memory size. The SSD was our way of overcoming the size constraint, although using it required some intermediary programming steps. The use of a physically separate IOS carried over to the CRAY X-MP system from the CRAY-1 system.

3.0. THE CRAY-2 COMPUTER SYSTEMS

The CRAY-2 development project proceeded in parallel with the CRAY X-MP project. In 1984 we introduced the four-processor CRAY-2 system. This system pushed several technologies very aggressively. Its 4-nsec clock was more than twice as fast as the CRAY X-MP clock.

3.1. Memory

The CRAY-2 system offered the largest central memory available on any computer, originally 256 Mwords with 512 Mwords available in later models, implemented in DRAM technology. This technology allowed such a dramatic increase in memory size that even though it performed slower than SRAM technology, it was the system's greatest advantage. The large memory made an external storage device, such as the SSD, unnecessary.

3.2. Immersion cooling

The most visible innovation in the CRAY-2 system was the circulating liquid coolant visible through the cabinet's transparent panels. The system's circuitry was immersed in an inert bath to keep it at a safe operating temperature. This technology was required to cool the chips that made possible the 4-nsec clock and was used also in the CRAY T90 series, announced early in 1995.

3.3. I/O

The architecture of this system eliminated the need for not only an SSD but also a separate IOS. The CRAY-2 system included an internal background processor that acted in place of an IOS. This alternative approach to I/O was a major distinction between the CRAY-2 and the CRAY X-MP systems.

3.4. The UNICOS operating system

The CRAY-2 system brought with it an important software development as well, one that has remained with the product line. The CRAY-2 system was the first Cray Research system delivered with the UNICOS operating system. UNICOS is based on the UNIX System V operating system, enhanced for the supercomputing environment but fully compliant with the UNIX standard. The UNIX operating system since has become the de facto standard operating system for high-performance computing. However, in the mid-1980s it was not an obvious choice. In fact, many customers, especially outside the United States, voiced concerns about the decision. Prior to adopting UNICOS, we offered the COS operating system, which was proprietary and which had to be rewritten virtually from the ground up with each new hardware architecture we brought to market. Our software leadership decided that we would be better off with a flexible system that could be ported more easily across architectures, and UNIX, with its small kernel, supported this objective. The move to UNIX also foreshadowed the industry's adoption of software standards generally. Standardization allows customers to move applications from one hardware platform to another whether or not the platforms are from the same vendor. It also reduces the amount of training customers must endure to learn the idiosyncracies of multiple systems.

4.0. Approaches to parallelism

Seymour Cray was the chief architect of the CRAY-2 system, and it formed the starting point for the designs of the CRAY-3 and CRAY-4 systems offered by Cray Computer Corporation, a company founded by Seymour in 1989. The subsequent Cray Research product line descends more directly from the CRAY X-MP system. As the Cray Research architecture has evolved, doubling the maximum number of processors with each new generation, we have continued to address the many

issues surrounding parallel processing. During the 1980s, MPP architectures captured the attention of many academic and government researchers. The U.S. federal government at this time helped fund the development of various types of MPP systems, and many new developers popped up. The MPP architectures that proliferated during this period took advantage of the rapid advances that had been made in microprocessor technology, namely the rising performance and falling costs of the chips.

The basic concept of MPP involves linking many microprocessors, up to hundreds or thousands, within a system and running them in parallel. Two fundamental shifts that occur in going from a typical vector multiprocessor to an MPP are distributed memory and a large number of processors. The most common way to connect hundreds of processors to memory is through a distributed memory approach. In this approach each processor is physically packaged "close" to its own (local) memory. The memory associated with other (remote) processors is "far" in terms of long latency and low bandwidth. This is in contrast to Cray Research vector multiprocessors, in which some small number of processors are uniformly connected to many shared memory banks. Unfortunately, in the distributed memory approach, no practical software exists that can automatically and appropriately map portions of an array to the processor that needs the data. This process is done manually by the user in terms of data distribution directives or language extensions. And even then some applications require that data be dynamically redistributed in different phases of the computation.

The second shift in parallel processing, that of going from a few to many hundreds of processors, makes MPP architectures extremely dependent on efficient interprocessor communication and synchronization. It also limits their applicability, because some application codes may not lend themselves to making such high levels of parallelism accessible. These two facets of programming with MPPs—having to program with distributed memory models and having to keep hundreds of processors working together efficiently—continue to limit the applicability of these systems. The early years of massively parallel processing were characterized by bold pronouncements from various proponents of the new approach. Rather than enter the fray, we continued to pursue our evolutionary approach to increasing parallelism. We had not ruled out an MPP architecture but wanted to make sure we could implement one in a marketable product.

Moreover, we were somewhat disappointed by the unwillingness of our customers (with a few noteworthy exceptions) to take advantage of the parallelism we already offered. Most application software vendors also were skeptical about parallel processing and resisted rewriting their codes to run in parallel. By now our customer base included many commercial industrial businesses that wanted reliable production environments. If they were unwilling to expend the energy to learn to take advantage of four- or eight-processor parallelism, then we could not assume they would expend the energy required to exploit higher levels of parallelism. With our evolutionary approach, we built a solid base of technical expertise in parallel hardware design, and we continued to enhance and refine our parallelizing system software, compilers, and tools.

5.0. THE CRAY Y-MP COMPUTER SYSTEMS

In 1988 we introduced the CRAY Y-MP system, which quickly evolved into the CRAY Y-MP series. The maximum CPU configuration on the largest system was twice that of the largest CRAY X-MP, raising the maximum from four to eight CPUs.

5.1. CPU enhancements

To facilitate parallel processing at this higher level, we also improved the efficiency of the shared registers responsible for interprocessor communication and synchronization. The CPUs were based on VLSI ECL gate-array technology developed in-house. We also switched to 12-layer, large-format printed circuit boards. Incorporating these new technologies enabled us to stick to our strategy of achieving efficiency through high-density packaging. To help customers in the transition to the CRAY Y-MP systems, the CPUs could run in either CRAY Y-MP or CRAY X-MP-compatible mode.

5.2. Model E IOS

The Model E IOS was introduced with the dual-processor CRAY Y-MP 2E system in 1990. This IOS used CRAY Y-MP CPU technology to improve I/O performance and reduce cost. The new technology enabled us to integrate the IOS into the CPU cabinet, eliminating the need for a separate IOS cabinet. The use of this and other new technologies enabled us to introduce enhanced CRAY Y-MP products in 1991: the CRAY Y-MP8E and CRAY Y-MP8I systems and the CRAY Y-MP4E system. The largest of these, the CRAY Y-MP8E system, included a 256-Mword central memory, compared to 32 Mwords on the original CRAY Y-MP system. By this time our SSD technology had evolved to where we could offer 2 Gwords of additional solid-state storage.

5.3. Large-memory systems

We also later introduced the CRAY M90 systems, which were essentially CRAY Y-MP systems with much larger central memories implemented in DRAM technology.

6.0. THE CRAY C90 COMPUTER SYSTEMS

In 1992 we introduced the CRAY Y-MP C90 system, which evolved quickly into the CRAY C90 series. With this series we again increased the maximum number of processors, from 8 to 16.

6.1. Enhanced parallelism

With 16 CPUs, the CRAY C90 system provided parallelism comparable to what some vendors were claiming as MPP capability. MPPs typically offered more processors, but users rarely would parallelize their applications for more than a few, certainly usually less than 16. But MPP already had become so identified with the microprocessor-based systems, and "supercomputer" had become so closely associated with Cray Research systems, that analysts and other commentators often overlooked the reliable, scalable parallelism available with the CRAY C90 systems. We enhanced parallelism even further by doubling, from one to two,

the number of vector pipes per CPU, so that vector throughput was essentially quadrupled in the largest systems compared to the largest CRAY Y-MP systems.

6.2. Enhanced storage and memory

The SSD technology by this time enabled us to offer models with up to 4 Gwords (32 Gbytes) of storage capacity. CRAY C90 systems also were available in large-memory DRAM models, providing a maximum memory of 2048 Mwords.

7.0. THE CRAY T3D COMPUTER SYSTEMS

In 1993 we introduced our first MPP architecture in the CRAY T3D system. Our MPP development plans were predicated on delivering a teraflops (TFLOPS—one trillion floating-point operations per second) of peak performance by mid-decade and a sustained TFLOPS by 1997. The high-performance microprocessors that were coming on line in the early 1990s convinced us that we could design MPP systems capable of these levels of performance and justify the development of an MPP architecture.

The CRAY T3D system developed quickly, and the product was ready to ship after only a 26-month development cycle. Based on the shortcomings we observed in systems already on the market and knowing that we had to leapfrog the capabilities of these systems, we settled on a MIMD architecture implemented in a bidirectional 3-D torus topology. The system was a considerable departure from our established architecture.

7.1. PE technology

The CRAY T3D system uses the DECchip 21064 microprocessor from Digital Equipment Corporation. Each microprocessor is supported by either a 16- or 64-Mbyte local DRAM memory. Each processing element (PE) in the system comprises the microprocessor, local memory, and custom-designed support logic. CRAY T3D systems are available with from 32 to 2048 PEs.

7.2. Topology

The 3-D torus topology ensures short connection paths. The peak interprocessor communication rate of 300 Mbytes per second in every direction through the interconnect network equals up to 76.8 Gbytes per second of bisection bandwidth (the maximum rate at which one half of the system can exchange data with the other half). The system also includes multiple latency-hiding features that augment the interconnect topology, in addition to a variety of synchronization mechanisms that support fine-grain parallel programming.

7.3. Distributed shared memory

We knew that the shared memory architecture we had developed for our vector systems would be prohibitively expensive for an MPP architecture. We had to work in the distributed memory model of the MPP world, but we wanted to retain the benefits of shared memory. Our solution was a physically distributed but logically shared memory. One distinguishing feature of the CRAY T3D system is the ability of any processor to reference any address in the system with simple loads and stores; the Cray support logic automatically interprets incoming message

packets and arbitrates for DRAM cycles with local processor references. On most other MPPs, interprocessor communication is interrupt-driven, with communication processors dedicated to send and receive functions. The hardware shared memory approach taken in the Cray Research MPP family is a crucial part of our long term goal of handling ever finer grain parallelism with minimal programming effort. The system is front-ended by a parallel-vector system to leverage the latter's mature operating system, especially I/O capability.

7.4. MPP enhancements

The next-generation Cray Research MPP system will be an evolutionary improvement over the CRAY T3D system. The new system will be based on a new microprocessor from Digital Equipment which will run at 250 MHz, with a peak speed of 500 MFLOPS. The support logic surrounding the chip will be simplified and consolidated considerably compared to the CRAY T3D technology. The external support logic of the new system will provide for a vector-like loading or storing of data across the system. The new system also will be a stand-alone MPP system; no parallel-vector front-end will be required. The operating system will be newly rewritten for distributed execution; different portions of the operating system will run on different processors. Also, relative to the CRAY T3D system, the follow-on will have significantly more I/O capability per processor. These and other technical advances will improve the price/performance of the new system. As a result of the lower cost relative to performance, this system should be attractive to a broader range of commercial users, for data-intensive commercial applications as well as a variety of scientific and engineering applications.

8.0. THE CS6400 SUPERSERVER

By this time it had become clear that parallel processing was useful outside of the scientific and engineering arenas in which Cray Research systems primarily were being applied. In 1991 we entered into a technical and business partnership with Sun Microsystems to create parallel systems for these commercial applications. The new venture, Cray Research Superservers (CRS), combined the popular Sun SPARC implementation of the RISC microprocessor architecture with the parallel-processing expertise of Cray Research. CRS's current product, the CS6400 system, is a shared memory, symmetric multiprocessor scalable up to 64 processors. In 1994 CRS became absorbed into Cray Research's business systems division, which continues to develop parallel systems for financial and marketing analytics, database mining, and other data-intensive business applications.

9.0. THE CRAY T90 COMPUTER SYSTEMS

Our current parallel-vector systems, the CRAY T90 systems, are available with up to 32 CPUs, each of which delivers a peak theoretical speed of about 2 GFLOPS, for a system peak performance approaching 64 GFLOPS. Like the CRAY C90 systems, each CPU has two vector pipes.

Many new technologies significantly enhance the reliability of the CRAY T90 systems. Zero-insertion-force connectors eliminate twisted-pair wires between the memory and the CPUs. High-density integrated circuits and high-density,

52-layer, printed circuit boards also improve reliability. These enhancements serve a dual purpose. They also improve system performance by physically condensing the system and shortening signal paths. The high density of the electronic components required the use of the very successful immersion cooling technology that had been used in the CRAY-2 system.

9.1. New CPU features

Each CPU includes a 1-Kword scalar cache to accelerate A and S register read references. Use of the cache reduces load times, for data in cache, by a factor of three or more. New load/store instructions have been added to "go around" the cache if necessary. Several new vector instructions also have been added.

9.2. Memory

The CRAY T90 systems offer a maximum memory capacity of 1 Gword. The systems also feature a new memory management scheme in which the previous Base Address/Limit Address mechanism is replaced by a Logical Address Table. Support now is provided for 40-bit logical addresses and 35-bit physical addresses. In addition, spare bits in the memory modules allow operators to return failing modules to operation without physical maintenance.

9.3. System partitioning

The CRAY T90 systems can be logically partitioned, allowing clustering of CPUs within a system. Memory and I/O resources each are configurable into separate partitions, and each CPU can be assigned to a partition. This partitioning capability allows for running multiple operating systems, simplifying operating system upgrades. An upgrade can be run on a subset of the system's CPUs and expanded gradually to the remaining processors at the customer's discretion. The largest systems in the series, the CRAY T932 systems, can be configured physically as two completely independent computers—each half of the total system has its own cooling and power systems. This physical partitioning also allows maintenance to be performed on half the system without interrupting the operation of the other half. Additional flexibility is provided by the ability to configure individual modules into or out of the system. Consequently, the system can operate correctly even with modules physically absent.

10.0. NEXT STEPS

Early orders for the CRAY T90 series demonstrate the enduring value of parallel-vector technology, and we fully expect demand to continue for parallel-vector supercomputers. However, we also know that the environment in which we do business will continue to change. A central topic in the industry these days, and certainly at Cray Research, is the potential for integrating parallel-vector and massively parallel architectures. One would hope to develop a workable architecture that preserves the advantages of each approach and which could operate in either mode according to the type of work the system was performing. A useful concept in this regard may be that of clustering, which workstation vendors promote as an alternative to both parallel-vector and massively parallel architectures. In the workstation environment, the concept has limited applications because of the latencies involved in communications and synchronization among

the clustered nodes. Cray Research has had some notable success with this approach, however, by linking our supercomputers through very-high-bandwidth network connections. In a particular case of a supercomputer cluster installed at a customer site, four 16-processor CRAY C90 systems can communicate with each other and operate in parallel in a shared file system environment. This very successful installation has taken us well beyond the proof-of-concept stage with regard to clustering supercomputers, and this approach will continue to evolve.

Looking further down the road, I think the reconciliation of the parallel-vector and massively parallel competition will emerge from what might be called a scalable-node architecture. This architecture would refine the clustered-node concept through a scalable hierarchy of independently powerful nodes. A node in such a system might have four or eight processors with both vector and scalar capabilities and sharing a common memory to ensure efficiency within the node. This node could stand alone and function as our parallel-vector systems do today, or be linked to other nodes and function in a distributed memory mode, to add performance and throughput. For more highly parallel applications, this system of linked nodes could itself operate as a node within a larger system, with the application distributed across the entire larger system. This approach not only would help customers preserve their investments in vector-optimized codes, but also deliver scalability on those applications that benefit from higher levels of parallelism. Such an approach may have sufficient ease-of-use and cost-effectiveness to overcome the limitations of current architectures and establish a new standard for supercomputing.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Designing a Scalable Parallel System : the IBM SP2

Tilak Agerwala[a], Jamshed H. Mirza[a], and Marc Snir[b]

[a] Power Parallel Division, IBM Corp., Kingston, NY 12401.
[b] T. J. Watson Research Center, IBM Corp., Yorktown Heights, NY 10598.

1. INTRODUCTION

The IBM SP2 is a general purpose *scalable parallel system* designed to address a variety of application areas and customer environments. Generally available SP2 systems range from 2 to 128 nodes, although much larger systems, up to 512 nodes have been delivered and are successfully being used today. The nodes are the latest POWER2 technology RS/6000 processors, interconnected with a high performance multi-stage packet-switched network for inter-processor communication. Standard AIX operating system is complemented with a set of software products for system management, job management, and application development and execution in the parallel environment. The system is designed for flexibility and availability, and addresses a wide range of application areas in the high-end Unix technical and commercial computing area.

This paper first discusses the underlying philosophy that guided the design of the SP2 system and then gives an overview of the architecture and structure of the system and some of the primary system components, including wherever relevant the rationale for significant system design decisions. Also discussed briefly are system performance and future system challenges.*

2. SYSTEM DESIGN PHILOSOPHY

Massively Parallel Processors (MPPs) have been around for several years now. These systems have typically been designed to apply the combined capacity of hundreds and even thousands of low-cost, low-performance processing elements to solving single large problems. However, until recently these systems were not adopted for mainstream supercomputing applications. Since the individual processors were of very low performance, considerable effort was required up front to parallelize a code sufficiently even to get performance equivalent to the mainstream uniprocessors of that day. That was a major inhibitor. In addition, limited processor memory, limited I/O, poor reliability, primitive, non-standard software development environment and

* Much of the material in this paper was previously published in [1]. Copyright 1995 International Business Machines Corporation. Used with permission from IBM Systems Journal, Volume 34, Number 2.

tools, and programming models that were closely tied to the underlying hardware (such as the interconnection structure) all contributed to their failure to be generally accepted. They remained, at best, special purpose machines for very narrow niche applications.

We realized (as did others [2]) that for massively parallel systems to succeed they must be more general purpose and less intimidating to use than they have been in the past. For one thing, they must allow users to do just about everything they are used to doing on more conventional systems. The basic nodes must be powerful enough, and the underlying operating system must be full-function so that users can move their current work over to the system with little effort and run his current applications in serial mode with acceptable performance; this way users can begin productive use of the system with little *up front* effort, and gradually parallelize and optimize the code over time. The system must support familiar interfaces, tools and environments, support existing standards and languages, and must have common applications available. It must be properly balanced in compute, memory, communication, and I/O capabilities and flexible enough to be configured suitably to address a wide variety of application areas and environments. It must address both throughput as well as response time requirements. Finally, the systems must be reliable enough to run production codes.

From the inception of the SP project, our goal was to design *general purpose* scalable parallel systems to address existing user demands for performance, price/performance, high availability and usability. We have designed our systems to be used in a variety of environments from very large stand-alone configurations dedicated to solving extremely complex and large single applications, to smaller systems that coexist with mainframes and traditional supercomputers and are used to offload some of the work for price/performance reasons, down to consolidated servers for mid-range LAN server environments. While our initial focus with SP1 was on technical computing, with the SP2 we have now broadened it to include commercial application areas.

2.1. Guiding Principles

In designing the SP2 as a flexible, general purpose *scalable parallel system*, we followed a set of guiding principles which are discussed below. We arrived at these principles after an analysis of current technology trends in both hardware and software, as well as an understanding of the requirements in the different technical and commercial application areas and the diverse customer environments we expected to address.

1. A high performance scalable parallel system must utilize standard microprocessors, packaging, and operating systems.

Major technology advances in recent years have primarily come from the workstation and distributed systems marketplace. High volumes and competitive pressures in that marketplace have prompted significant investments, resulting in significant advances being made in all aspects of the technology - processors, I/O, communications technology, compilers, system software, tools, and applications. It is generally accepted that microprocessor performance is doubling roughly every 18 to 24 months and that this trend will continue in the foreseeable

future. In fact, the performance of standard microprocessors is rapidly converging towards that of special custom-designed processors built for traditional supercomputers, and the cost of the two is rapidly diverging. Our design approach is to ride this technology curve; we will use standard components (both hardware and software) from the workstation environment as much as possible, and develop custom hardware and software components only where standard technology cannot meet some unique requirements of a scalable parallel system at the desired performance levels.

2. Time to market with the latest technology is critical to achieving leadership performance and price-performance.

The rate of improvement mentioned above creates both an opportunity and a challenge. Since performance and time can essentially be traded, it is imperative that the SP systems be able to incorporate the latest microprocessor technology very rapidly. This emphasizes the need to exploit this technology essentially *as is*, and place as few dependencies as possible on our technology suppliers for special features to support parallel processing.

3. Required levels of latency and bandwidth will require custom interconnect networks and communication subsystems over the next few years.

For parallel applications, a key determinant of performance is the process-to-process communication latency and bandwidth and the corresponding overhead on the processor for executing the communications protocol. While several interesting "commodity" network technologies (such as Fiber Channel Standard (FCS) and Asynchronous Transfer Mode (ATM)) have emerged recently, these alternatives are optimized for a very different environment and do not provide the right levels of latency, bandwidth and processor overhead to meet the stringent performance requirements of parallel systems.

4. The system must present a programming and execution environment identical to a standard open, distributed Unix environment.

Figure 1 shows the full stack of software (explained in the rest of the section) that is required for enabling various technical and commercial applications. It is not viable to develop all or even the bulk of this software from scratch for a scalable parallel system. Much of the software for systems management, job management, storage management, databases, and message passing libraries exists for distributed Unix environments. Our goal is to accommodate and leverage this software. This is one dominant personality of the system and it allows all software (except for that requiring special device drivers and adapters) written for a distributed Unix environment, and running on the underlying base node, to run on the SP2 machine.

5. The system should provide a judiciously chosen set of high performance services in areas such as the communications system, high performance file systems, parallel libraries, parallel databases, and high performance I/O to provide state-of-the-art execution support for supercomputing, parallel query, and high performance transaction solutions.

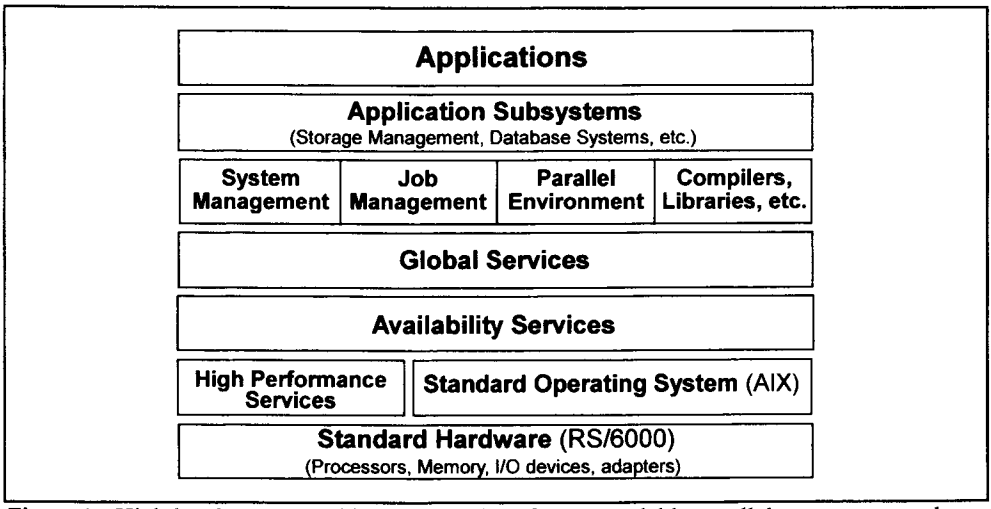


Figure 1: High level system architecture required for any scalable parallel system targeted at a *general purpose* environment. (For the SP2, the base nodes are RS/6000 microprocessors and the base operating system is AIX.)

Scalable Parallel Systems must provide a second dominant personality for the high performance supercomputing environment. This consists of a set of *high performance services*, and a development environment and tools to enable, develop, and execute new parallel applications and subsystems which cannot execute efficiently in conventional distributed systems environments.

The combination of principles 4 and 5 allows us to overcome a significant limitation of prior highly parallel solutions and dispel a commonly held misconception that massively parallel machines can provide only niche solutions. In fact, it is our contention that scalable parallel systems can provide the most general purpose solutions. Principle 4 lets us execute all distributed open systems software and Principle 5 at the same time allows us to provide competitive solutions for traditional MPP grand challenge and high-performance commercial applications.

The first five principles lead us to the high level system structure shown in Figure 2. The nodes consist of robust, high function, high performance RS/6000 processors, each running a full AIX operating system. The nodes are interconnected by a high performance switch through communication adapters attached to the node I/O bus (the microchannel). For the current systems, using the microchannel as the interface to the switch subsystem was a practical decision; the standard microchannel interface allows us to rapidly introduce new node technologies into the system while achieving targeted latencies and bandwidth.

A full AIX image on each node, together with support for standard communication protocols on the switch (ie., IP), provide full logical support of all standard distributed services. The high performance switch, together with the communications subsystem software and a parallel file

system implemented as kernel extensions to AIX, and complemented with a parallel program development and execution environment, provide the core of the high performance services. This system architecture allows us to achieve state-of-the-art performance, price/performance, and scalability in supercomputing environments.

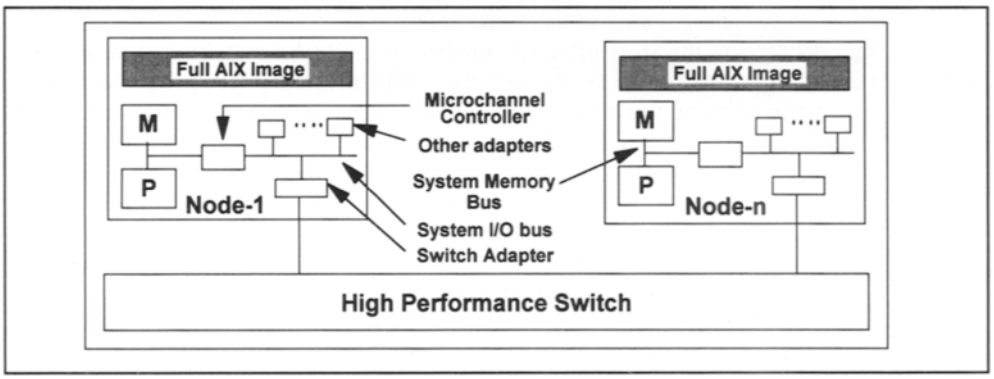


Figure 2: High-level system structure for the IBM SP2

6. Desired system availability can be achieved cost-effectively with standard commodity components by systematically removing single points of failure that make the entire system unusable, and by providing very fast recovery from all failures.

The structure described so far consists of a large fraction of commodity components which are produced for workstations rather than large system environments. In a very large system with hundreds or thousands of commodity parts, failures in the node hardware, node software, and switch will occur frequently enough that the system must be designed to continue functioning in the presence of failures.

The distributed operating system architecture has some inherent advantages compared to most symmetric multiprocessors. The failure of an operating system image does not have to bring the entire system down since the active operating system images can continue to function. Our system approach to high availability relies on this.

This approach requires sufficient replication (of data, compute resources, adapters, and access to data/adapters), and a software infrastructure for failure detection, failure diagnosis, reconfiguration of the system, and invoking recovery action. This software infrastructure consists of a set of *availability services*. The goal of these services is to allow a system to gracefully degrade from N resources to M resources (where $M < N$), and reintegrate the N-M resources later in a non-disruptive manner.

It should be noted that this is merely an infrastructure. To provide real benefit to an end user, all higher level subsystems such as the program development and execution environment, job scheduling, data base and transaction subsystems must exploit the N --> $(N\text{-}M)$ --> N infrastructure and take the appropriate recovery actions non-disruptively.

7. Selected support for single system image through the globalization of key resources and commands, together with single point of control for systems management and administration, is preferred compared to true single system image.

At the level just above the high availability services, the system software view is that of *N* AIX images, each of which manages a set of local resources and provides a set of local services. A critical design decision is the level of *single systems image* to be supported. Two extreme approaches are possible: the first is to stay with the totally distributed view; the second is to implement a layer of software that makes the N images appear to be one in *all* respects (true single system image).

This is a complex decision since different environments need different views. An interactive user would generally prefer a true single system image. On the other hand, database subsystems have been written for a distributed environment and expect to see the totally distributed view; these subsystems explicitly manage the different images for performance, and provide single system image at the database subsystem level. Finally, for a technical computing user, a single system image at the source code level and at the Unix shell level is desirable.

Since providing true (or complete) single system image in an efficient manner is a complex undertaking and not required in general, we have taken a more pragmatic approach. There are clearly key resources (such as disks, tapes, directory services) that need to be globally known and accessible. Similarly, there are key commands that should be globalized for ease of use. Our approach is to stage in the globalization of these selective resources and commands over time based on the critical requirements of the applications and subsystems we expect to support. Furthermore, our approach is to provide hardware and software support for controlling, administrating and managing *N* AIX images and nodes in an SP2 from a single point, i.e. we will also provide single system image at the system management level.

3. SP2 SYSTEM OVERVIEW

In this section we give a brief overview of the overall SP2 system architecture. The focus is on high-level design choices that were made and, where appropriate, the rationale behind them or the implications of those choices.

The SP2 hardware primarily consists of standard components (processors, memory, I/O devices, adapters) designed for the RS/6000 workstation market. Hardware enhancements were incorporated to make the system more reliable and available, and easier to manage and service. The processors are interconnected by a custom designed switching network.

The SP2 software provides a structured, integrated environment for application development and execution, as well as system management and administration. The basic approach was to build SP2 system services on standard open interfaces and platforms as much as possible, and only performance critical services should be implemented using proprietary protocols and

interfaces. Consequently, the environment is based primarily on industry standards and, where necessary, merges in required enhancements for parallel processing to provide both high performance and ease-of-use.

3.1. SP2 System Organization

One of the fundamental decisions in the design of a parallel system is the underlying architecture. In parallel MIMD machines, there are essentially three alternatives based on two primary design decisions - the memory structure and the communication architecture, as shown in Figure 3 and discussed below.

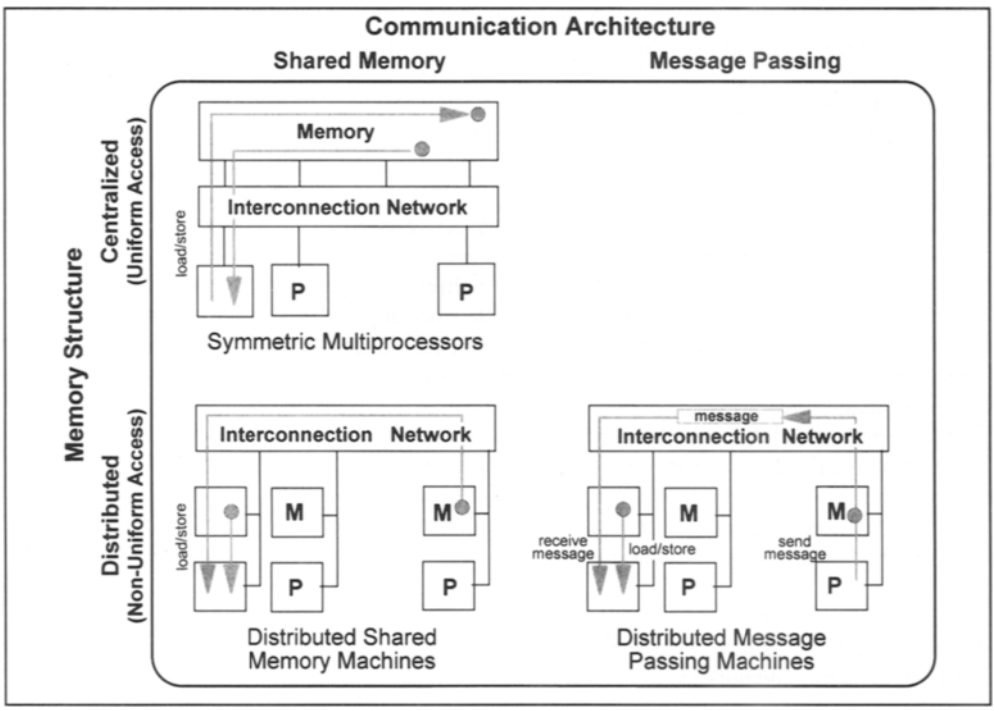


Figure 3: System organization alternatives for parallel (MIMD) processors.

The most common parallel structure today is a *symmetric multiprocessor* (or SMP) with centralized memory which is equally addressable from all processors. Such a structure presents a Uniform Memory Access (UMA) architecture and thus preserves data location transparency; data distribution or data location is typically not critical for performance. The shared memory *architecture* allows the familiar shared memory *programming model* to be supported easily. There is a single operating system image that manages the whole system. This preserves a *single systems image*, and furthermore, centrally managed shared resources (memory, I/O, external connectivity) facilitate good load balancing. However such structures have limited scalability, which today is typically less than 20 processors. SMPs based on a shared bus

architecture (typically found in microprocessor SMPs today) are limited by memory bandwidth that does not scale with additional processors. Custom designed SMPs (typically found in mainframe and traditional supercomputers) use a switch based organization which allows memory bandwidth to scale to some degree; beyond that, hardware complexity and cost become limiting factors. In cache based systems, the cache coherence hardware does not scale beyond a small number of processors because of complexity, cost and performance. Similarly, the single operating system managing all of the system resources can generally become a performance bottleneck and a limit to scalability. Finally, the single, system-wide operating system image is a critical single point of failure; an operating system failure can result in the loss of the total system.

Systems that must scale beyond a small number of nodes must necessarily be structured as *distributed memory* machines. In such systems, a portion of the memory is packaged close to each processor. Access to local memory is fast and remains constant with the size of the system, while access to remote memory is slower. This leads to a Non-Uniform Memory Access (NUMA) architecture. In these machines, data location transparency is essentially given up for higher system scalability.

The next design decision is how data is shared between different processors. A distributed memory machine can have a *shared memory* or *message passing* architecture. In distributed *shared memory* machines, a single global *real* address space exists across the whole system. All of the physical memory is directly addressable from any node. A node can perform a *load* or a *store* instruction to any part of the real address space. Typically there is a separate operating system (or micro-kernel) images on each node, but they are not independent; the different images are tightly connected at least at the Virtual Memory Manager (VMM) level so as to present a single global real address space. In such systems, address and data coherence must be maintained - in hardware (which makes the hardware complex and costly, and a fundamental limit to performance and scalability) or in software (which adds to programming or compiler complexity, further exposes program correctness, and can potentially affect performance because of conservative or safe coherence management actions).

Alternatively, a distributed memory machine may support a *message passing* architecture. Here, a processor has direct access (i.e., can perform *load* or *store* operations) to only its local memory. Remote memory is not directly addressable and data is shared by explicitly sending and receiving messages. Address and data coherence across nodes are not an issue here.

The SP2 is a *distributed memory message passing* machine. Two primary reasons, and a host of secondary reasons, led us to select this architecture as opposed to the distributed shared memory structure.

A globally shared *real* (as opposed to *virtual*) address space implies fundamental changes in the operating system running on these nodes - primarily in the virtual memory management area, but permeating into several other areas of the operating system. That goes contrary to our guiding principles 1 and 2.

Even more important is our contention that an underlying distributed memory architecture without global *real* memory addressability is the right one for cost-effective scalable parallel systems. Such systems are inherently more scalable at the system level because they do not require tight coordination between the operating system images on the different nodes to provide common address space management and maintain address coherence; nor do they require tight coordination at the hardware level to maintain data coherence. Further, message passing structures with loose coupling between the operating systems have inherently more availability since it is easier to localize failures to the failing node. Finally, a message passing architecture, if properly designed, allows for significant offload of the overhead associated with communication and overlap it with computation, especially when one can aggregate the data to be communicated into a few large messages. In a real shared memory structure, data is shared implicitly by referencing it, and the referenced data from a remote memory is accessed on a cache line by cache line basis. Although a single line miss may result in a latency and overhead that is very small compared to message passing, many such line misses would be required to share the data that could have been passed with just one large message.

3.2. Programming Model

Fundamentally, there are two styles of parallel programming. With the message passing programming style, processes have private address spaces and share data by explicit messages. With the shared memory programming style, processes share a common address space, and data is shared by a process directly referencing it. Note that we are making a distinction here between the underlying system *architecture* and the supported *programming style* or *programming model*. It should come as no surprise that with the right software and hardware support, both shared memory and message passing programming models can be supported on a system with either of the underlying architectures.

The dominant parallel program structure for large scalable systems today is Single Program Multiple Data (SPMD) where the same basic code executes against partitioned data. Such programs are generally written in a loosely synchronous style with *computation* phases alternating with *communication/synchronization* phases. During the *computation* phase, each process computes on its local data; during the *communication/synchronization* phase, the processes typically exchange data (in the case of message passing programming style, using a message passing library) or synchronize with each other (in the case of a shared memory style, using a synchronization library).

The essential difference between a shared memory style of programming and a message passing style on a distributed memory machine is the following. In the message passing style of programming, data sharing is *explicit* while synchronization is *implicit*. A programmer must identify when and what data is to be shared between processes and must explicitly insert *send* and *receive* commands to cause the sharing of that data. However, synchronization is implicit since the sharing will occur only when the source process is ready to send and the destination process is ready to receive. This is in contrast to shared memory programming style where data sharing is implicit; no special action is necessary since it is accomplished by a process merely referencing the data. However, process synchronization is explicit; since there is no

restrictions on referencing shared data, a programmer must identify when and what data is being shared and must properly synchronize the processes using special synchronization constructs in order to ensure program correctness. In either case (given that today any system scalable beyond a small number has a distributed memory structure) proper data distribution is critical for performance.

Since the SP2 has an underlying message passing architecture, clearly a message passing programming style is the preferred one for performance. Several message passing libraries callable from FORTRAN and C are supported on the SP2 - IBM's advanced message passing library, MPL [3], which is finely tuned and optimized for the underlying communication hardware; Express [4], PVM [5], and Linda [6]; as well as PVMe, which is IBM's version of PVM communication library optimized for the SP2. An optimized implementation of MPI [7], a recently defined message passing standard, will be supported on the SP2 sometime in 1995.

The SP2 can also be programmed in the Data Parallel model using High Performance Fortran (HPF) [8]. HPF programs can be written using sequential Fortran to specify the computations on the data (using either iterative constructs or the vector operations provided by Fortran 90), and HPF data mapping annotations to specify the distribution of large arrays across processors. An HPF preprocessor or compiler translates the HPF source code into an equivalent SPMD program with message passing calls. This approach has the advantage of freeing the user from the need to explicitly distribute global arrays onto local arrays, change names and indices accordingly, and explicitly manage the data sharing using message passing constructs. This allows the programmer to program in a single global name space.

In the commercial environment, standard database and OLTP subsystems have been enabled for some time now for a clustered systems environments by running multiple instances of the subsystem and providing a layer of software that provides a single system image to the higher level application programmer. This model fits the distributed memory message passing architecture of the SP2 well. Thus for the typical commercial computing user, the parallelism in the SP2 is effectively hidden or encapsulated; the underlying subsystem takes care of decomposing the individual query to run on multiple nodes or of routing the transaction to the right server.

3.3. Flexible Architecture

SP2 was designed to address a diverse set of application areas and customer environments. Consequently, it was of fundamental importance that SP2 have a *flexible architecture* that allowed a customer to customize the system to site-specific requirements and situation.

Figure 4 shows two views of an SP2 system illustrating its *configuration flexibility* and *operational flexibility*.

Configuration Flexibility: The SP2 system consists of from 2 to 512 RS/6000 processor nodes (0.5 to 137 peak system GFLOPS), each with its own private memory and its own copy of the AIX operating system, interconnected by a high performance switch (Figure 4(a)). Each

SP2 system also requires a *control workstation* which is a separate RS/6000 workstation that serves as the SP2 system console.

As per principles 1, 2 and 4, we have made no fundamental change to the base RS/6000 processor and the AIX operating system. This means that any of the major hardware or software options available on the base RS/6000 workstations can be installed on an SP2 node. Similarly, several thousand RS/6000 applications are available from day one to an SP2 customer.

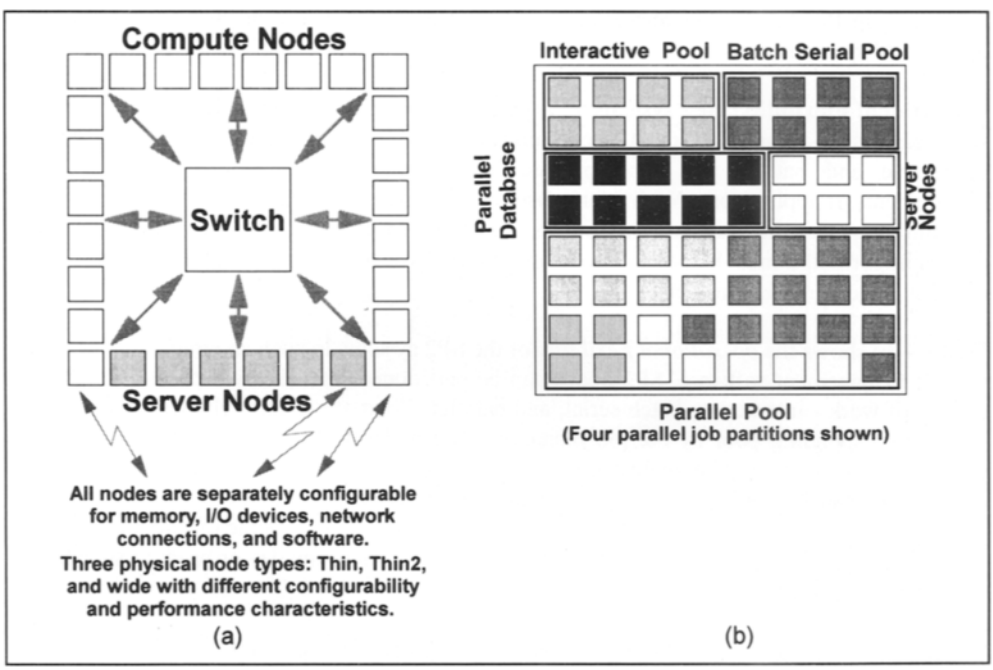


Figure 4: Flexible architecture features of the SP2 System.

SP2 nodes can be configured to have one of two fundamental *logical* personalities. Some nodes are configured as *compute nodes* and are used for executing user jobs. Other nodes are configured as *server nodes* which provide various services required to support the execution of user jobs on compute nodes; these could be integrated file servers, gateway servers for external connectivity, database servers, backup and archival servers, etc.

The requirements for node performance and configurability can be very different for compute and server nodes, and for different application areas. As a result, SP2 provides three different *physical* node types - *thin*, *thin2*, and *wide* nodes. Thin nodes occupy half the space of a wide node, so they can be packaged more densely. But they are less robust in their configurability (memory and microchannel slots) than wide nodes. The three node types provide optimal price/performance points for different environments. Furthermore, SP1 nodes can co-exist

with SP2 nodes in the same system, thus providing considerable flexibility in upgrading a system and providing investment protection to a customer.

SP2 is designed for maximum *configuration flexibility* so that a system can be tuned in a cost-effective manner to a customer's particular requirements. The system can scale up over a very wide range (4 to 512 nodes) in very small increments (one or two nodes). There can be any mix of compute and server nodes within the system. Similarly, there can be any mix of thin and wide nodes (within limits of frame configuration options). Furthermore, each SP2 node can be individually configured for memory, adapters, internal hard disk, and software. This configuration flexibility is an important distinguishing characteristic of the SP2 system from other MPP systems.

Operational Flexibility: SP2 systems will typically be used as enterprise-wide or department-wide servers and therefore must accommodate a mix of job types (serial and parallel, interactive and batch), and accommodate both throughput as well as response time requirements of applications. With a large user community sharing the system, it must function as a throughput engine, concurrently processing unrelated user jobs; however, when response time is important, it must be possible to usurp required system resources and apply them to one or a few large, long-running jobs.

Figure 4(b) shows the *operational flexibility* of the SP2 systems in such an environment. Using the job management software, SP2 nodes can be partitioned into different pools for different classes of work - interactive, batch serial, and parallel. Jobs of different classes are channeled to the corresponding pool by the job manager. The nodes within a pool may be shared by several jobs, or they may be dedicated to a single application. As shown in the figure, several other nodes can be configured as system-wide servers (File Servers and a Parallel Database Server in this case) providing shared service to applications running in the different pools. The size and nature of the pools can be changed at different times of the day or week to support the differing requirements.

The nodes in the parallel pool are further partitioned dynamically on a job basis. A parallel job requests and is granted the required number of nodes from the parallel pool; these nodes form a *parallel job partition.* Parallel programs generally execute in a dedicated partition of processors; this allows the individual processes to remain loosely synchronized and thus achieve good parallel speedup. It also allows the processes to use direct user-mode communication which bypasses the operating system, thus resulting in better performance. Multiple parallel jobs can be running on the machine simultaneously, each controlling a disjoint set of the nodes. Because of the topology of the interconnection network, there are no restriction on the size of a parallel job partition or on the topological location of the nodes in the partition; this provides significant flexibility in application design as well as in job management and resource management. This is discussed further in section 4.2.

4. MAJOR SYSTEM COMPONENTS

Given the overall system architecture and features described in the previous section, we now look at some of the primary system components in more detail - the nodes, communication subsystem, availability services, globalization services, and the parallel programming environment. These components have a direct impact on the essential character of the system - scalability, flexibility, general purpose nature, performance and price/performance, system availability, and usability.

Not described in this paper are other system components such as system management and job management. The philosophy in both cases was to leverage available interfaces and tools and incorporate extensions where necessary for the parallel nature of the system.

4.1. SP2 Processor Nodes

A first order determinant of performance for both serial and parallel jobs is the individual node performance. Our decision was to use the fastest and most robust processors available to us. This way a user can get reasonable performance without ever parallelizing the application.

The SP2 nodes are standard POWER2 architecture RS/6000 processors [9]. They are super-scalar RISC processors (rather than merely super-pipelined), and incorporate many sophisticated organizational techniques to achieve high sustainable performance.

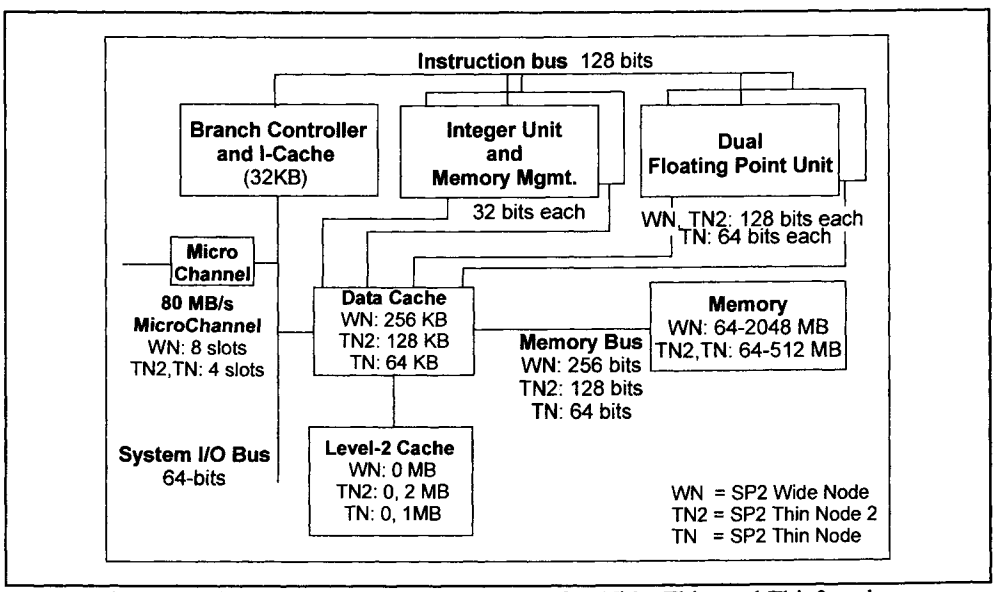


Figure 5: The SP2 Processor Node structure for Wide, Thin, and Thin2 nodes.

All three POWER2 node types have two fixed point units, two floating point units (each capable of a multiply and add every cycle) and an instruction and branch control unit. A 66.7 MHz clock speed gives the processors a peak performance of 267 MFLOPS. The instruction unit can decode and issue multiple instructions every cycle to keep all the units busy. This is matched up with a large capacity high performance memory hierarchy. For example, in the wide node equivalent (Figure 5), the memory can be up to 2 GB and has a bandwidth of 2 GB/s; a 256KB 4-way set associative cache can supply data at the rate of 4 64-bit operands per cycle. As a result, within a tight loop one can effectively sustain a performance equivalent to a two-pipe vector processor (two loads/stores, two floating-point multiple-and-add, index increment and conditional branch). Features such as short instruction pipelines, sophisticated branch prediction techniques and register renaming techniques, large cache, large memory, high cache and memory bandwidth all add up to a very robust processor capable of good performance on both vectorizable as well as scalar code. As a result, it can consistently sustains a very high percent of its peak capability. The SP2 thin nodes are similar to the wide nodes but have a less robust memory hierarchy and I/O configurability.

The robustness of the nodes, both in configurability and performance is a primary advantage of the SP2 system. In fact, on a processor to processor basis, SP2 has the most powerful node in any contemporary scalable parallel system; this contributes greatly to the overall SP2 system performance.

4.2. SP2 Communication Subsystem

Because of the high request rate, communication services between tasks in a parallel job partition are required to be high bandwidth and low-latency. Traditional AIX (or Unix) interfaces cannot provide the required performance, and hence SP2 has created unique facilities in the Communications Subsystem (CSS) to match the performance requirements for communications within a parallel job partition. Further, since the CSS is the heart of the system, a lot of design effort went into making it reliable and transparently recoverable from failures.

The SP2 nodes are interconnected by a *High Performance Switch* (HPS) [10] designed for scalability, low latency, high bandwidth, low processor overhead, and reliable and flexible communication between the nodes.

Topologically, the switch is an any-to-any packet-switched multi-stage or indirect network similar to an Omega network [11]. This topology allows the bisection bandwidth to scale linearly with the size of the system, which is critical for system scalability. In contrast, the bisection bandwidth of direct networks (such as rings, meshes or multi-dimentional toroids) increases much more slowly (or not at all as in the case of simple rings).

A consequence of the HPS topology is that the available bandwidth between any pair of communicating nodes remains constant irrespective of where in the topology the two nodes lie. This is not the case with direct networks. In general, the bandwidth available to a node is equal to lxk/h, where l is the link bandwidth, k is the number of links in the switch per node, and h is

the average number of hops through the switch required for a communication operation. destination. In a multi-stage network such as the HPS, the average number of hops, *h*, and the number of links per node, *k*, both scale logarithmically with the number of nodes. Thus the bandwidth between any communicating pair of nodes remains constant. This lends considerable flexibility in programming and managing such systems. In contrast, in a direct network the number of hops, *h*, for a random communication pattern increases with number of nodes, but the number of links per node, *k*, stays constant. As a result, the average bandwidth per pair of communicating nodes decreases. Because of this problem, programmers have an additional optimization parameter to worry about; they must carefully design the application so as to minimize *h* by trying to limit inter-node communications to close neighbors. Additionally, the resource manager and job scheduler must be designed to allocate nodes for a parallel application that are topologically in close proximity with each other, and carefully map the processes to the nodes so as to minimize the average number of hops required by the inter-node communication in the application. These optimizations are not critical with a topology such as the HPS.

Figure 6 shows the switch structure for a 64-way SP2. The switch is designed to be scalable, with the building block being a two-staged 16x16 switch board, made up of 4x4 bi-directional crossbar switching elements. Each link is bidirectional and has 40 MB/s bandwidth in each direction. In small systems (up to 64-way) only one switch board is required per 16 nodes. For the 64-way system shown in the figure, the required switch boards are packaged within the processor frames (one per frame) and connected via inter-frame cables to get a 4-stage switch network. For larger systems additional switch stages are required. The extra switch boards for these additional stages are packaged in a special switch frame. The switch uses buffered cut-through worm-hole routing for maximizing performance [10].

An SP2 node connects to the switch board through an intelligent microchannel adapter. The adapter has an onboard microprocessor that offloads some of the work associated with moving messages between nodes. The adapter can move messages to and from processor memory directly via DMA, thus reducing the overhead on the processor node for message processing and significantly improving the sustainable bandwidth. The switch adapter provides protection support for secure user-mode communication between nodes within a parallel partition without requiring system calls. This allows lower application-to-application message latency by avoiding kernel calls. Packet CRC checking and generation is also done by the adapter, further reducing the overhead on the SP2 node. Normal message-passing between the processor node and the adapter is polling driven to avoid the overhead of interrupt processing. The CSS allows shared use of the communications system by both user and kernel tasks; both user-space and IP traffic is concurrently supported over the switch.

The Communication Subsystem hardware and software are designed for reliability and transparent recovery from hard and soft failures.

The switch hardware is *fully checked.* Each switching element is in fact shadowed by a duplicate switching element so that any error in the switching elements is detected. Similarly,

packets on all links carry CRC codes; the code is generated at one end and checked at the other to detect errors in the links.

The switch always contains *at least one extra stage* more than necessary for full connectivity. Since the basic switching element is a 4x4 bi-directional crossbar, this extra stage guarantees that there are at lease 4 different paths between every pair of nodes. The redundant paths provide for recovery in the presence of failures (as well as reduce congestion in the switch). In the presence of hard errors in the switch (due to a failing switching element or link), the switch can be reinitialized with regenerated routes to avoid the failing components.

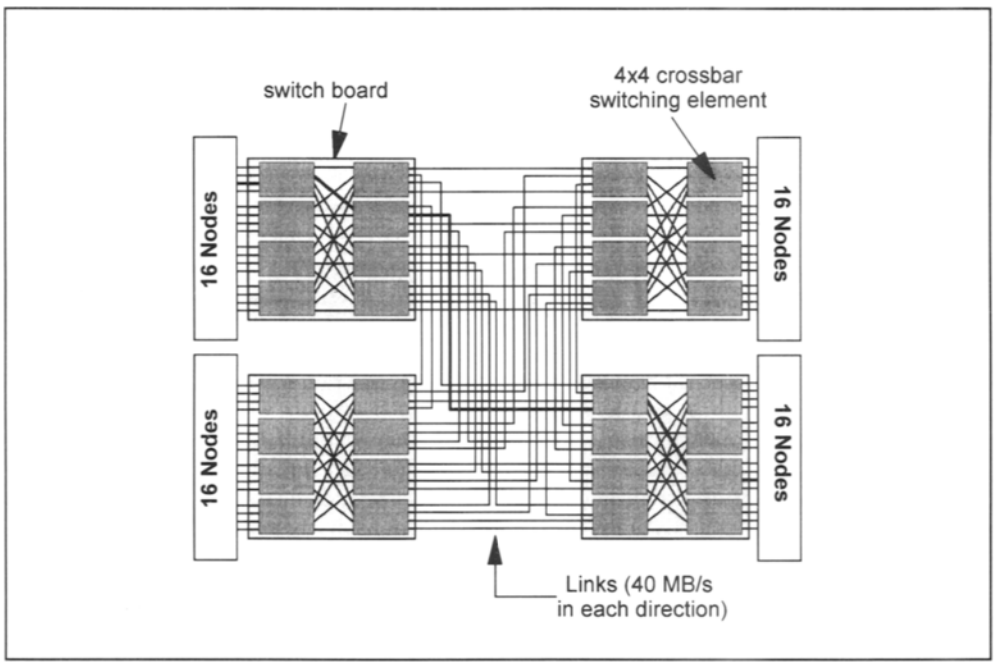


Figure 6: SP2 switch structure for a 64-node system.

The CSS software complements the error detection capability in the hardware to provide transparent recovery of lost or corrupted messages. The message protocol supports end-to-end packet acknowledgment. For every packet sent by a source node, there is a returned acknowledgment after the packet has reached *and* been received by the destination node. Thus the loss of a packet is detected by the source node. The CSS software automatically resends packets if an acknowledgment is not received within a preset interval of time.

When an error is detected by the switch subsystem hardware, the switch enters the *diagnostic* mode during which the cause of the error is identified. Thereafter new inter-node routes are generated, avoiding any failing component or link that was detected, and then the switch is brought back into *run* mode. At this time, the CSS code times out for the packets that were lost in transit when the switch took a fault. These packets will therefore be retransmitted by the

CSS software transparently to the application. Thus the full error detection support of the hardware, coupled with the full error recovery of the software result is a very reliable and robust communication subsystem for the SP2.

4.3. SP2 Availability Services

The availability services provided by the SP2 system and described below are today used by a very limited and controlled set of subsystems. In the future, as the application programming interfaces (APIs) and the function of these services stabilize, our intention is to make them more generally available. These services would then form a *scalable* infrastructure that can be used by software developers to build recoverable subsystems and servers for the SP2. Using these services, critical subsystems can be designed to gracefully degrade from N resources to M resources (where $M < N$) without disruption of service; later, the N-M resources can be reintegrated into the system in a non-disruptive manner after they have been serviced.

Today, the SP2 availability services include:

Heartbeat services allow processors in the system to be monitored for normal operation.

Membership services allow processors and client processes to be identified as belonging to some defined group; a group is generally formed to include members providing some known and related application service.

Notification services allow members of the group to be notified when membership in the group changes. Membership within a group changes as processors join or leave the group as a result of various *node events*, such as node restart, shutdown, or failures.

Recovery coordination services provide a mechanism for initiating recovery procedures within the active group in reaction to node events that change the membership. These services are used to coordinate the running recovery procedures across the members of the currently active group in response to membership changes. The scripts are specified by the subsystem that is to be recovered.

These services are implemented via user-level daemons running on the nodes. The daemons exchange periodic heartbeat messages to determine which nodes are up, and propagate new membership lists constituting a group. During initialization, groups are formed by individual nodes identifying themselves to each other as belonging to specific groups. User-level client processes register with the daemon on their local node to get notified about remote node status.

In the presence of a communications connection failure, it is possible that there will be two subgroups that do not know of each other, both of which try to access a shared resource. To avoid such integrity problems there must be a way to ensure that at most one active group can exist in the system at any time. The solution used in the SP2 availability services is that a group

only becomes *active* after it contains a *quorum* (or a majority of the potential members of the group).

Currently these services are used by some of the SP2 system software components (such as the Virtual Shared Disk described later) as well as by some third party subsystems. As the APIs and the functions for these services stabilize, we intend to make them more generally available. Over time we expect that all of the critical SP2 system subsystems and other third party software will use them to provide transparent recovery.

4.4. SP2 Global Services

As discussed earlier, full single system image is not seen as a critical requirement for SP2. Instead, much of the benefit associated with single system image can be derived from providing elements of this functionality as *global services*. Global access to specific resources such as disk tape, files, and communication networks are the primary requirement.

In SP2, global access to files is provided today by networked file solutions such as NFS and AFS. These provide for concurrent shared access to file data and are the basis for the globalization of this resource.

Global network access is provided via normal network routing functions and TCP/IP and UDP/IP support over the switch. In this way, SP2 nodes which are not physically attached to an external network still have the ability to communicate through nodes which are physically attached (i.e., through *gateway* nodes).

Global access to disks is provided by the *Virtual Shared Disk* support. Using VSD, an application running at any SP2 node can transparently access any disk, physically located on any other SP2 node, as if it were locally attached to that node. This is done by trapping a request for a remote shared disk at the disk driver level and shipping the request to the corresponding node. In effect, VSD is a device driver layer that sits on top of the AIX Logical Volume Manager (LVM) and exports a raw device interface. If the access is to a shared disk that is locally connected, the VSD layer passes the request directly on to the LVM on that node. If however the access is to a shared disk attached to a remote node, the VSD layer routes the request to the VSD on that remote node (through the switch), which in turn passes it on to the LVM for access. The response is returned to the VSD on the originating node and on to the requesting application.

VSD provides *transparent failover* capability. For this feature, each VSD server is logically paired with an alternate secondary server. The disks must be twin-tail attached to both the primary and secondary server nodes. However only the primary tail is active normally. The heartbeat service is used by the primary and secondary nodes to monitor each other. If the secondary node detects a primary node failure, control of the disks attached to the failing node is transparently switched over to the secondary server, and requests to those disks continue to be serviced by the secondary server node transparently to the application. The SP2 availability services are used by VSD to implement this transparent recovery capability.

Currently, VSD is primarily required by database subsystems based on the data sharing architecture which require access to disks from any node [12].

4.5. Parallel Environment

The SP2 Parallel Environment [13] is an integrated set of components that allow a user to develop, debug, and tune parallel FORTRAN or C programs, and to initiate, monitor, and control their execution.

Principles 5 and 6, and to some extent 2, resulted in a decision to base the Parallel Environment as much as possible on standard AIX tools and techniques and conform to established standards. This allows us to reduce the learning time and maximize ease-of-use for customers. Most commands use familiar UNIX syntax and various AIX tools are enabled to use with the Parallel Environment. Program compilation, scheduling, execution and monitoring are done in manners familiar to Unix programmers.

The SP2 Parallel Environment has four primary components:

The *Parallel Message Passing Library* (MPL) is an advanced communications library that supports the explicit message passing programming model for Fortran and C programs. It provides a rich and comprehensive set of functions and subroutines for implementing simple pair-wise communication between processes as well as more powerful collective communications operations involving user-definable groups of processes.

MPL is implemented to exploit the High Performance Switch using an optimized, light-weight *user-space* protocol that does not require a kernel call. Alternatively, a user can elect to have the MPL run using the IP protocol over a LAN connection or over the switch.

The *Parallel Operating Environment* provides a user environment for initiating, monitoring and controlling the execution of a parallel application.

The Parallel Operating Environment can be used to compile and link parallel code with message passing libraries; to create a parallel partition with the required nodes (nodes can be either explicitly specified by the user or selected by the SP2 *resource manager* based on user specified job requirements); to load the parallel job on the nodes in the parallel partition; and to communicate with and monitor the job while it is executing. The parallel application is controlled by a *partition manager* process created by POE on the node or workstation that was used to initiate the application.

The *Visualization Tool* (VT) provides Performance Monitoring and Trace Visualization for a parallel application. Performance Monitoring displays system activity in real-time while the application is running. Trace Visualization is a postmortem process that allows the user to view in detail the interactions between parallel processes, using traces collected during run time. VT can be used to debug an application by identifying deadlock situations and analyzing inter-

process communications; and it can be used to analyze and tune a parallel application by identifying performance bottlenecks and load imbalances.

The *Parallel Debugger* is a source level debugger with both a command line and a graphical interface. It is an enhancement of the familiar Unix *dbx* debug tool and incorporates additional functions specific to parallel program debugging.

4.6. SP2 Reliability and Availability Features

In the design of the system we paid particular attention to minimizing the probability of *catastrophic* failures. A catastrophic failure is defined as one that brings the whole system down and restart is not possible, even in degraded mode, until service is applied. In particular, the system is designed to be tolerant of single node failures; the effect of a failing node is isolated to applications using that node. Independent operating system images on each node make this possible. As discussed earlier, the switch subsystem which is the heart of the SP2 system, is designed for recovery from most common hard and soft errors. Support for RAIDs, multi-tailed devices and mirroring is provided for storage media recovery.

In most common cases, hardware and software service can be applied to a node or a part of the system while the rest of the system remains in operational mode. A node in an SP2 system can be powered off and disconnected from the rest of the system to perform repairs on the node, and later replaced and powered-up, while the rest of the system is operational.

The SP2 frames are designed for reliable, remotely controlled operations with concurrent maintenance and upgrade capability. Each frame has redundant main power supplies to reduce the chance of a system outage due to a power supply failure.

Support for transparent recovery of software subsystems and servers is provided by the SP2 availability services, as described earlier. These services provide an infrastructure for subsystem developers to write recoverable software. Currently these services are used by VSD and the SP2 resource manager as well as one of the database subsystems; over time many of the SP2 subsystems and other third party software will use them to provide transparent recovery.

5. PERFORMANCE

In standard benchmark tests as well in customer specific benchmarks, SP2 systems have delivered superior performance and price-performance. This is primarily attributable to very powerful nodes complemented with a flexible, robust communication subsystem and system software.

In the technical computing area, the NAS Parallel Benchmark Suite [14] from NASA is the most widely used standard benchmark. Figure 7 shows the performance of SP2 system (using wide nodes) and several other massively parallel systems as reported in [15]. The results are for 128 node systems in most cases, except for some of the benchmarks where the Intel Paragon

data is available for a different number of processors (as noted in the figure). The SP2 significantly outperforms other equivalent size systems, especially on the pseudo applications which are closely related to full applications in use by NASA scientists.

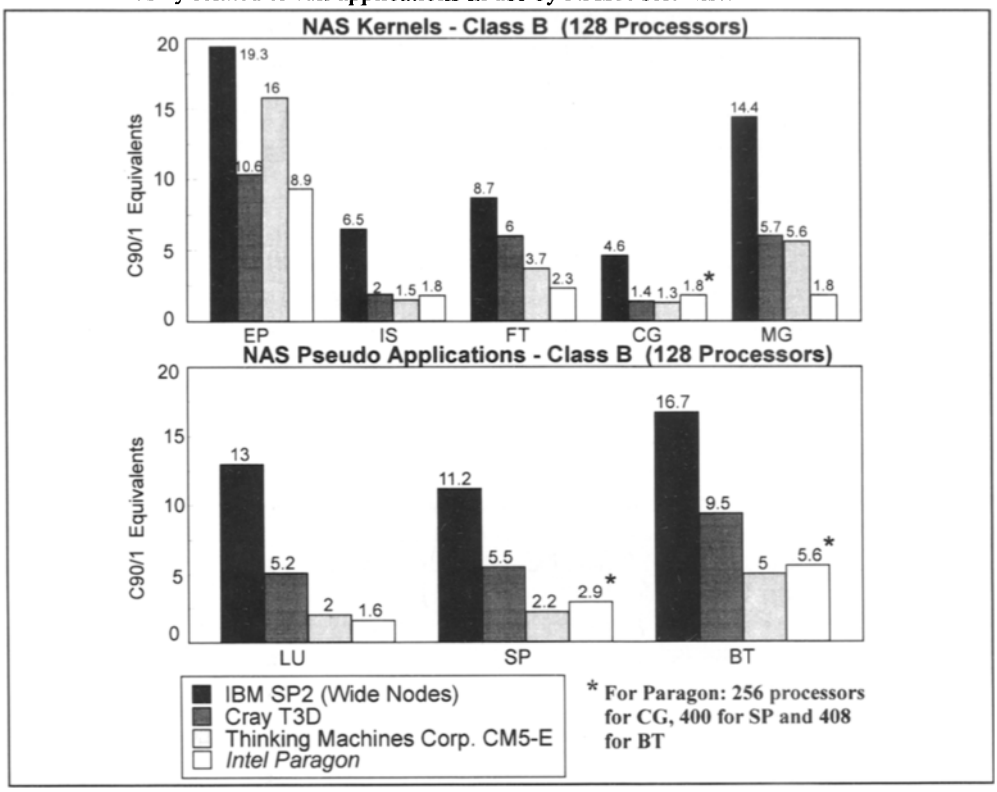


Figure 7: NAS Parallel Benchmark performance (in equivalent single processor Cray C90) for IBM SP2 and selected other scalable parallel systems.

6. KEY FUTURE CHALLENGES

We expect that the future evolution of SP systems will continue to be guided by the principles that we articulated in Section 2. In particular, we will continue to exploit workstation technology as much as possible. However, as the market for the scalable parallel systems expands, we will continue to assess technologies that are specifically targeted to scalable parallel computing.

A symmetric multiprocessor as the basic node in a scalable parallel system is an attractive option for future systems. It can provide significant improvement in peak node performance at a cost that is not much higher than the cost of a uniprocessor with the same amount of storage. However, effective use of this performance is a challenging problem. In particular, one needs to

manage two levels of parallelism, with big differences in performance and function between intranode communication and internode communication. Hiding the complexity of two levels of parallelism by providing transparent compiler exploitation will be crucial.

The communication subsystem in scalable parallel systems must continue to improve in performance and capability. Efficient support of a shared memory programming model requires aggressive bandwidth increase and latency reduction. This precludes interfacing to the switch via the I/O bus in future SP systems. Direct connection to an internal memory bus will lead to improved performance, but will require a more specialized package, and interlocking of adapter technology and processor technology.

Aggressive latency reduction will require a communication architecture that will allow a processor to access a remote memory without involving the remote processor. Such functionality is already available in one form or another on some systems. However current implementations either compromise on performance (latency), or require a global real address space. The challenge is to provide such functionality with very low latency on a virtual memory system with minimal impact on system interfaces, while still maintaining the distributed message passing system model we have adopted for scalability and availability.

Very large and scalable information servers (including video and multimedia servers) require the sharing and movement of massive amounts of data among nodes. A key metric here is the CPU overhead for moving data. Minimizing CPU overhead and supporting large throughput, rather than low latency, is critical.

There is limited experience today with scalable storage systems and scalable file systems, and hence limited knowledge to guide system architects in the design of future parallel I/O subsystems. Yet low-cost, scalable I/O is key to the success of scalable parallel computing. The High Performance Storage Subsystem (HPSS) initiative (jointly between key government labs and several vendors of high performance systems) has generated solutions to some of the key issues in this area. The challenge is to quickly incorporate these solutions into mainline products.

When applications are parallelized for speedup, their I/O bandwidth requirement from a single file also increases proportionately. Such bandwidths cannot be satisfied by standard network file systems, even with inboard I/O servers. A parallel file system is required, and one is being developed for the SP2 that allows files to be distributed across multiple I/O server nodes and accessed in parallel to increase the bandwidth to that file from a parallel application.

Efficient utilization of future scalable parallel computers will require major advances in software technology. Efficient compiler technology is needed to support more convenient programming models, while preserving portability to other platforms. Mechanisms (such as a parallel shell) are required that allows a user to control a parallel computation with the same ease as controlling a serial computation today. Parallel system services are needed for resource allocation and job management. Techniques for efficient time sharing of parallel applications

are required. Also needed are parallel system libraries that can be invoked by parallel applications (for example, a parallel I/O library).

The availability of parallel application codes and subsystems is critical for the success of scalable parallel computing. The development of such codes is hampered not only by the difficulties of parallel programming, but also by the lack of accepted standards in this arena. High Performance Fortran (HPF) and the Message Passing Interface (MPI) are two successful examples of establishing such standards. Much more is needed. In particular standards are needed in the area of parallel system services such as parallel I/O.

Another aspect, particularly in commercial applications, is that of providing coordinated and consistent recovery between a large number of subsystems running on any scalable parallel system. These could include multiple database systems, a transaction monitor, a parallel file system, a multimedia server, resource managers, the operating system, the global I/O subsystem, etc. Each subsystem may detect several software failure conditions on its own, and these need to be combined with other failure detection mechanisms (such as the heartbeat service), for overall diagnosis. Ordering, synchronization, and parallelism during recovery of the various subsystems is needed. Providing a high availability infrastructure that addresses these issues is a challenging problem.

Meanwhile, the software technology used in sequential systems is progressing. Object Oriented (OO) programming and object oriented storage (object oriented data bases, object oriented file systems) are becoming more prevalent; operating system technology is evolving to better support OO. Future scalable parallel systems will have to cope with this evolution, both because of its impact on the basic uniprocessor software technology, and because of the importance of OO for future applications. Scalable OO technologies are yet to emerge from the research community. Yet such technologies are likely to have a major impact on the future of scalable parallel systems.

7. SUMMARY

The support for standard distributed AIX program development and execution, standard message passing libraries, specific required globalization of resources, and availability services has allowed independent software vendors to enable key applications and subsystems on SP2 without diverting from their mainstream development strategies. There is now a broad portfolio of parallel applications available today for the SP2, in addition to the several thousands of RS/6000 applications that can execute serially on the SP2.

A familiar programming environment, availability of applications and enablers, performance, price/performance, robustness, and flexibility have made SP2 a leading player in the scalable parallel systems market today.

SP1 and SP2 systems from 2 to 512 processors are today being used productively in many different application areas, including computational chemistry, crash analysis, electronic, design

analysis, seismic analysis, reservoir modeling, decision support, data analysis, OLTP, LAN consolidation, and as workgroup servers. They are being used for large scale research at universities and government labs, for serial and parallel production jobs and process control in many different industry areas, and as business support systems. This broad based acceptance attests to the flexible and general purpose nature of the architecture and validates the principles we used to design the SP2. However significant issues remain to be addresses in order to provide truly usable scalable parallel systems.

REFERENCES

1. T. Agerwala, D. M. Dias, J. Martin, J. H. Mirza, D. Sadler, and M. Snir, *The SP2 System Architecture*, IBM Systems Journal, Volume 34, Number 2, May 1995.
2. D. E. Nielsen, *A Strategy for Smoothly Transitioning to Massively Parallel Computing*, Energy and Technology Review, University of California, Lawrence Livermore National Laboratory, November 1991.
3. V. Bala et al, *IBM External User Interface for Scalable Parallel Systems*, Parallel Computing, Volume 20, Number 4, 445-462, April 1994.
4. J. Flower and A. Kowala, *Express is not just a Message Passing System: Current and Future Directions in Express*, Parallel Computing, Volume 20, Number 4, 597-614, April 1994.
5. V. S. Sundcram, G. S. Geist, J. Dongarra, R. Manchek, *PVM Concurrent Computing System: Evolution, Experiences and Trends*, Parallel Computing, Volume 20, Number 4, 531-545, April 1994.
6. S. Ahuja, N. Carriero, and D. Gelernter, *Linda and friends*, Computer 19(8):26-34, Aug. 1986.
7. MPI Forum, *Document for a message passing interface*, Technical Report CS-93-214, University of Tennessee, Nov. 1993.
8. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr. and M. E. Zosel, *The High Performance Fortran Handbook*, MIT Press, 1994.
9. S. White and S Dhawan, *POWER2: Next generation of the RISC System/6000 Family*, in PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, Prentice Hall, 1994.
10. C. B. Stunkel et al, *The SP2 Communication Subsystem*, IBM Systems Journal, Volume 34, Number 2, May 1995.
11. T. Feng, *A survey of interconnection networks*, IEEE Computer, December 1981, 12-27.
12. P. S. Yu, D. M. Dias, J. T. Robinson, B. R. Iyer, and D. W. Cornell, *On Coupling Multi-Systems Through Data Sharing*, Proc. IEEE, 75(5) 573-587, 1987.
13. M. Snir, P. Hochschild, D. Frye, K. Gildea, *The SP2 Communication Software and Parallel Environment of IBM SP2*, IBM Systems Journal, Volume 34, Number 2, May 1995.
14. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, V. Venkatakrishnan, and S. Weeratunga, *The NAS Parallel Benchmarks*, Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
15. D. Bailey, E. Barszcz, L. Dagum, and H. Simon, *NAS Parallel Benchmark Results*, Technical Report RNR-94-001, NASA Ames Research Center, October 1994, and supplements.

III. PROGRAMMING AND SOFTWARE

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

An Integrated Environment for the Support of Automatic Compilation *

M. Pantano, H. Zima, Institute of Software Technology and Parallel Systems, University of Vienna, Brünnerstr. 72, A-1210 Vienna
e-mail: {pantano,zima}@par.univie.ac.at

In this paper, we propose an integrated program development environment for the purpose of automatically compiling Fortran programs for massively parallel distributed-memory architectures. Programs are subjected to an iterative tuning process, involving three major components: a restructuring system, which provides a catalog of analysis and transformation services, a performance analysis subsystem that computes parameters characterizing program behavior, and a parallelization support environment, which generates data distributions and an associated transformation strategy. Many of the components in this environment have already been implemented in the framework of the Vienna Fortran Compilation system. The paper focuses on the performance analysis subsystem and its relationship to the other components.

1. Introduction

In recent years, virtually all major computer manufacturers added *massively parallel distributed-memory multiprocessing systems (DMMPs)* to their product lines. Examples include Intel's hypercubes and the Paragon, the nCUBE, Thinking Machine's CM-5, the Meiko Computing Surface, the Cray T3D, Convex Exemplar, and IBM's SP series.

These architectures are potentially scalable to very large numbers of processors and display an impressive theoretical peak performance approaching the 100 GFLOPS mark. They are widely seen as capable of not only solving large scientific problems such as the *Grand Challenges* but also as playing an increasingly important role in business applications as well as being essential components in the information infrastructure of the future.

However, up to now, DMMPs have not entered the main stream of computing and their potential has not nearly been exploited, except for relatively few applications ported to these machines by dedicated experts. The main reasons for this failure are the lack of a suitable standard programming model, the lack of adequate software support, and the difficulties of achieving high performance for a broad range of applications with moderate

*The work described in this paper was partially supported by the Austrian Research Foundation (FWF Grant P8989-PHY) and by the Austrian Ministry for Science and Research (BMWF Grant GZ 308.9281-IV/3/93).

effort. The main source of these problems is the non-uniformity of memory in these machines: the time required for the access to non-local data may be significantly higher than the time taken for the access to locally stored data items. As a consequence, the successful management of the tradeoff between the goals of achieving maximum locality and balancing the workload of the system is crucial for obtaining adequate performance.

Over eight years experience with the use of such systems in Europe and the USA has led to the development of the data-parallel **Single-Program-Multiple-Data (SPMD)** programming model, in which data are distributed across the processors of the machine, and each processor essentially executes the same code, on different portions of the data domain. The current programming paradigm associated with this model is the *message passing approach*, in which existing high-level sequential languages such as Fortran and C are combined with a standard interface to a message passing subsystem: Parmacs [6] and PVM [3] have been widely used, and now MPI is likely to become a de facto standard [2].

The disadvantages of the message passing paradigm are well-known: the cost of producing a message passing code may be between 5 and 10 times that of its serial counterpart, the length of the code grows significantly, its portability is limited, and it is much less readable and less maintainable than the sequential version. For these reasons, it is widely agreed that a higher level programming paradigm is essential if parallel systems are to be widely adopted.

This was the rationale behind the definition of Fortran language extensions which allow the user a high-level specification of data distribution and alignment that provides the basis for the *automatic generation* of message passing programs [9,13,36]. These languages, which we will collectively refer to as *High Performance Fortran Languages (HPFLs)*, include Vienna Fortran [13,37] and Fortran D [19], both of which contributed to the definition of the de-facto standard High Performance Fortran (HPF)[1].

Although a number of commercial compilers for Subset HPF have been developed in Europe and the USA, the problems related to the *automatic and optimizing compilation of HPFLs for DMMPs* are far from being fully solved. Moreover, although HPFLs simplify the programming process, they still leave the intellectually most demanding task – the specification of data distribution and alignment – to the programmer. This means that automatic support for deriving efficient data distributions in Fortran programs is required; in short, we have to solve the problem of **automatic compilation of Fortran programs for DMMPs.**

This requires the realization of two complementary strategies:

- The development of a rich set of **heuristics** that reflect properties of real programs, the programming language, the compiler, and the characteristics of the underlying machines, and

- the creation of an **analysis infrastructure.** Based upon the reduced search space obtained by the application of heuristics to the program, the application of precise

or approximate analysis becomes feasible. The resulting information can be used to control automatic compilation.

This paper proposes an advanced **integrated program development environment** for the purpose of automatic compilation. Fortran programs are subjected to an iterative tuning process, involving

- a **restructuring system**, which provides a catalog of analysis and transformation services,
- a **performance analysis subsystem** that computes parameters characterizing program behavior, and a
- a **parallelization support environment**, which generates data distributions and transformation strategies tuned to the target DMMP.

The paper focuses on the problems related to the requirements of performance analysis. In Section 2, we give an overview of the salient features of an integrated program environment, followed by a more detailed discussion of performance issues in Section 3. Section 4 discusses a concrete proposal for a research prototype, based on the Vienna Fortran Compilation System (VFCS) [12,38]. A coherent framework for performance analysis and prediction is presented, including tools for static and dynamic profiling, static performance prediction (P^3T), and dynamic workload characterization (*MEDEA*). The paper finishes with an overview of related work (Section 5), followed by the conclusion.

2. Problem Statement

In this section, we discuss some of the problems related to the automatic translation of Fortran programs for DMMPs in more detail, and outline the major components of an integrated program environment addressing these issues.

The compiler has to determine data distributions and find a transformation sequence that restructures the program and parallelizes it in such a way that it performs well on the target DMMP architecture. A large number of program transformations has been used in actual systems [39], including

- *normalization and standardization transformations*, such as do-loop normalization, scalar forward substitution, induction variable substitution, constant propagation, and if-conversion,
- *loop transformations* such as loop distribution, loop unrolling and rerolling, and loop interchange,
- *array privatization*,

- *communication generation and optimization*,
- *idiom recognition*, and
- *in-line expansion* and *cloning* of procedures.

Transformations can be only applied if an associated *validity condition* is satisfied, which guarantees the preservation of the program's semantics depending on the data dependence relation and possibly other program properties.

Each transformation changes the program in a predefined way, which may result in a significant modification of its execution behavior. Some transformations may only be performed in order to create the validity conditions necessary for the application of other transformations, while others are at the heart of a complex program restructuring strategy. As a consequence, the effect of a transformation on the efficiency of the final program cannot in general be expressed as a local function depending on the actual program state and the immediate properties of the transformation. More generally, the **restructuring problem** can be formulated as follows:

Given

1. A Fortran program Q_0,
2. A target machine M,
3. A catalog, $\mathcal{T}$, of program transformations, and a
4. cost function h related to the desired performance characteristics,

determine a valid transformation sequence $\tau \in \mathcal{T}^*$ such that

1. τ transforms Q_0 into the *final program* Q, and
2. $h(Q)$ is optimal on machine M with respect to all programs that can be validly obtained from Q_0.

It is well known that the restructuring problem is algorithmically undecidable, even if restricted to seemingly simple subproblems such as constant propagation. Thus, a compiler cannot solve the problem *precisely*, but must rely on determining an approximate solution. Two key elements for determining such a solution are **heuristics** and **analysis**.

To make this discussion more concrete, we take a short look at an important subproblem of fully automatic compilation, which is the automatic determination of data distributions and alignments for the arrays of the program.

We speak in this context of the **automatic data distribution** problem.

The quality of a data distribution depends on:

- array access patterns,
- program phases,
- loop parameters,
- problem size, and
- architectural characteristics of the target machine such as the number of processors and the communication parameters.

Clearly, an optimal solution cannot be derived algorithmically in the general case. However, there are many programs and program kernels for which an optimal solution is known. The implication is again that a combination of heuristics and analysis is the right way to approach the problem.
Figure 1 shows the structure of an environment that is being designed along these lines. As discussed above, it includes a restructuring system, a performance analysis subsystem, and a parallelization support environment. The major objective of such an environment is the support of fully automatic compilation in the framework of an iterative restructuring process applied to the program, guided by analysis information and controlled by strategies implemented in the parallelization support environment. However, there are situations where the final result of such an automatic transformation process does not satisfy the performance constraints that must be met for critical applications. In order to cope with this, we allow the user to *interactively* influence the restructuring process by supplying the system with knowledge concerning the structure and behavior of the program that cannot be automatically derived. More specifically, a user may, for example, provide the system with data dependence information, specify the distributions of important data structures, or enforce the in-line substitution of procedures. On the other hand, a user may query the system for certain information, such as performance parameters of selected program components, or options for transformation strategies in a given state of the tuning process.

The *performance analysis subsystem* of the environment plays a key role in guiding the restructuring process, by providing essential parameters for the control strategies implemented in the parallelization support system, and, if an interactive mode is selected, by supplying the user with crucial information supporting an understanding of the program behavior. We examine this topic in more detail in the following section.

3. Performance Indices

In this section, we discuss a range of **performance indices** which reflect certain aspects of a parallel program's behavior. Performance indices can be derived at different *program*

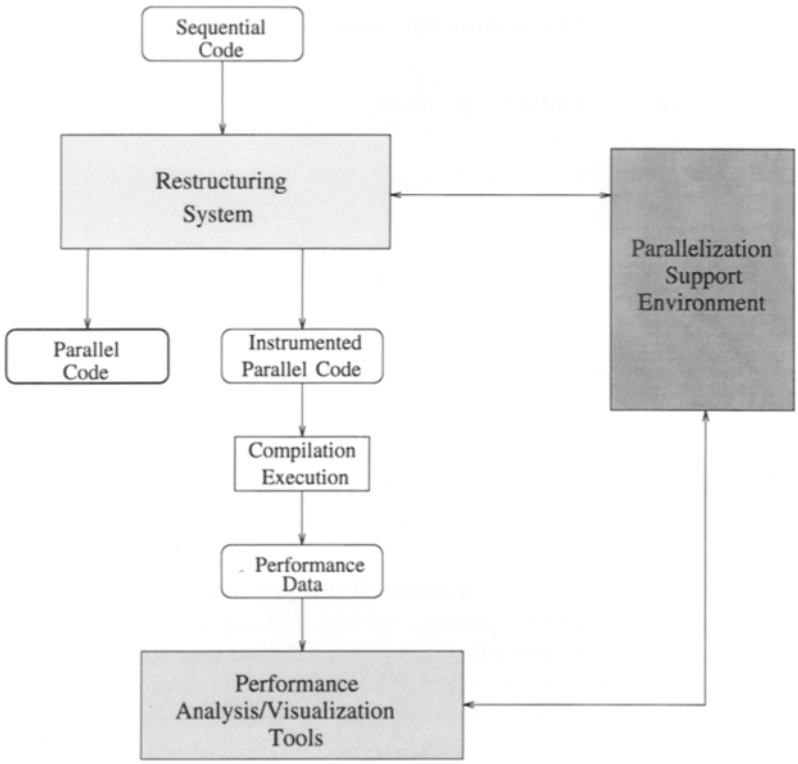


Figure 1. Structure of the Environment

levels: programmers often use performance tools tools like microscopes [26], starting with a global view and then gradually descending to smaller program components where sources of performance loss have been discovered. Roughly, at least three levels of analysis can be distinguished in a Fortran program:

- *interprocedural analysis*, providing a global characterization of program properties,
- *intraprocedural analysis*, which is applied to a single procedure, and
- *local analysis*, related to individual statements, statement sequences, and loops within a procedure.

For each level, performance indices can be determined using a combination of static and dynamic methods. The list below describes a set of indices that provide useful quantitative information concerning properties of SPMD programs when being executed on a DMMP [32]:

- *number of transfers*: number of Send and Receive operations.
- *send time*: time spent executing Send operations.
- *receive time*: time spent executing Receive operations.
- *communication time*: *send time* + *receive time*.
- *computation time*: the time spent in computation.
- *execution time*: *communication time* + *computation time*.
- *amount of data transferred*: the data volume transferred by the Send and Receive operations.
- *network contention*: an indication of the number of channel contentions induced by a communication operation on the underlying physical communication network.
- *cache misses*: number of cache misses.

Another important set of performance indices, on a higher level of abstraction, is the following:

- *Degree of Parallelism*: the number of processors that are busy computing at a given instant of time, assuming an unbounded number of processors [33].
- *Execution Profile*: a function representing the number of processors that are busy at each instance of the program execution. In a similar way, the *communication profile* and the *computation profile* can be defined [32].
- *Processor Working Set* (PWS): the number of processors associated with the knee of the execution time-efficiency profile [20]. Each point of such a profile represents the combination of execution time and efficiency achieved by a particular number of processors [14]. The knee – the point where the ratio between the execution time and efficiency is maximized – represents an optimal system operating point.

The computation of performance indices parameterized by varying data distribution, numbers of processors, and input data sizes, is a complex task which has to apply a combination of different tools. These include

- *static tools*, which performing analysis based only upon statically derivable information,
- *simulation tools*, and

- *dynamic tools*, which are based on the execution of the program on the actual DMMP.

Static tools are generally the least expensive in terms of resource usage, while being able to provide valuable information on important performance characteristics. They are, however, inherently limited in situations where the values of relevant symbolic objects – such as the upper limit in the range of a loop variable – cannot be bound at compile time.

Simulation tools [5] avoid this drawback by executing an abstract version of the program, usually on a workstation. A problem with this approach is related to scalability: as a consequence of increasing the number of processors, the resulting increase in simulation time and memory usage may render the simulation infeasible.

The easiest, but in terms of resource usage most expensive way to obtain performance information consists of *measuring* the code and collecting *dynamic indices*. A measurement system integrated into the compiler instruments the code and, during the program execution on a real system, collects performance data.

Among the various statistical techniques that should be applied to a data set, the analysis of distribution and variability are the most useful for the behavioral analysis of parallel programs. The variability analysis of the parameters *communication time* and *computation time*, for example, gives an indication of the way in which communication and computation are distributed across the processors. Experimental studies [15] [29] show that the coefficients of variation (COV) [22] of those two parameters are good indices expressing the "goodness" of work distribution (computation) and communication distribution.
Statistical clustering analysis [18] is a useful technique for automatically identifying behavioral equivalence classes [30]. Clustering analysis can be applied for identifying processor outliers, that is the set of processors on which the behavior of the program or a selected program region is different from the typical ones. This is often a key point for performance optimization.
For large codes, programmers will sometimes sacrifice performance for *scalability*. Scalability, characterized by a set of indices that relate performance to the number of processors in a DMMP [34], can be used to calculate the performance gain or loss when executing a parallelized program on systems with different sizes. The most common scalability indices are: the *speedup*, which gives the performance gain obtained by using parallel processing versus sequential processing, the *efficiency* that represents the speedup per processor, and the *efficacy* which indicates how well the processors are used.

An important aspect in the design of a performance analysis subsystem is its design as a **coherent toolkit** in the framework of a modular system with well-defined interfaces between tools. Since users are not willing to deal with the idiosyncrasies of a large number of different tools, it is important that such a toolkit offers a *standardized user interface* which is largely invariant under the insertion of new tools or the modification of existing ones. The often significant differences between the structures and functionalities

of individual tools must be hidden behind an interface that gives the user the look and feel of a *single tool.*

These requirements are difficult to implement but essential for providing a robust environment acceptable to a broad user community.

4. An Integrated Programming Environment Based on Vienna Fortran

In this section we propose a research prototype along the lines of the general discussion conducted in Section 2 (see Figure 1). We will focus on a more detailed design of the performance analysis subsystem, the interrelation of its components, and its interface with the restructuring system as well as the parallelization support system. The system described here is based on the Vienna Fortran Compilation System (VFCS).

4.1. Overview

Figure 2 gives an overview of the proposed environment.

It consists of

- the *restructuring system* (VFCS),
- a set of *performance prediction tools*, integrated into the compiler,
- a *code measurement system*, which inserts code instrumentation,
- a *performance analysis toolkit* composed of tools useful for the analysis of the generated parallel program and the visualization of performance data, and
- an *expert system adviser (XPA).*

VFCS provides a set of analysis services and a transformation catalog. The analysis features include intra- and interprocedural control and data flow analysis as well as data dependence analysis. The results of analysis can be applied within the system – for example, in order to verify validity conditions associated with transformations – and can also be presented graphically to the user. VFCS allows the choice between an automatic mode and an interactive mode, in which the user controls the application of transformations. More detailed information on VFCS can be found in [38,12].

P^3T is a performance prediction tool integrated into the VFCS, which can be used to statically calculate a set of performance indices, based on a given input data set.

The *code measurement system*, another performance tool integrated into VFCS, automatically instruments the parallel code in such a way that performance data for selected regions of the code can be gathered during program execution on a parallel system.

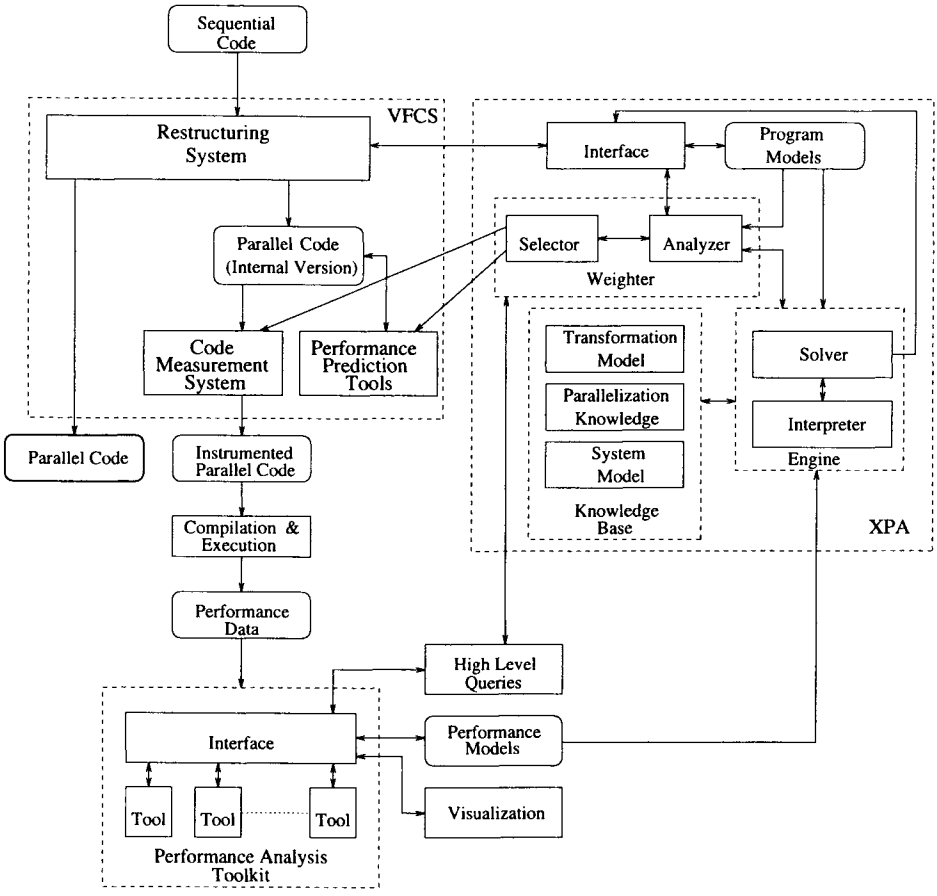


Figure 2. Automatic parallelization Environment

The raw data collected have to be analyzed in order to obtain meaningful and easy to interpret information that can be used to identify the real behavior of the program. The *Performance Analysis Toolkit*, composed of a set of performance evaluation tools with a single user interface, is the component of the environment which deals with this issue. On the basis of the information provided by the toolkit, characteristic features of the parallel program execution can be visualized, thus giving the programmer possible clues to the source of performance problems. On the other hand, this information can also be used directly by the XPA for establishing a relationship between performance indices and the code measured.

XPA is an "intelligent" parallelization support environment. It is designed to provide explicit knowledge in a number of areas, extending from properties of the source language

to target machine parameters. The XPA uses heuristics to reduce the search space for data distribution and transformation strategy selection; it constructs a program model, which provides information required to control the tuning of programs with respect to a given architecture. Furthermore, in order to reduce the measurement space, the measurement system can be directed by the XPA to focus on only those parts of the program which are relevant for the current analysis and/or dominate the overall computation time.

The entire transformation process is of a cyclic nature and may require a number of iterations before converging by yielding a program satisfying the performance constraints put forth by the user. The user may interact at any point by providing information to the tuning process and/or querying the system for selected analysis information.

The following subsections describe some of the components mentioned above in more detail.

4.2. P^3T

The P^3T ([15–17]) is an interactive performance estimator that assists users in performance tuning of scientific Fortran programs. Its main purpose is the detection of performance bottlenecks in the program.
Three of the most critical performance aspects of parallel programs are estimated: load balance, cache locality, and communication overhead. The integration of the P^3T into the VFCS enables the estimator to exploit considerable knowledge about the compiler's analysis information and code restructuring strategies.
The P^3T is based on a single profile run to obtain concrete data for statement and loop execution counts and branching probabilities. This means that all parameters as estimated by the P^3T are strongly based on these values. While program changes due to the application of transformations can often be handled by incrementally updating the analysis information, the effect of changes in the input data set is not yet well understood.

An inherent restriction of this approach is related to its inability to deal with runtime-dependent data values. As a consequence, programs displaying irregular data accesses and dynamic data and work distributions cannot be handled effectively.

4.3. Code Measurement System

Measuring large programs that have hundreds of modules and thousands of procedures is a difficult task, which requires the availability of a sophisticated monitoring tool. Monitoring the performance of parallel applications necessarily introduces perturbation. The monitoring tool must strive to minimize the intrusion and filter the monitoring effects.
For this purpose, a *program measurement tool* has been integrated into VFCS. The system allows the instrumentation of various code regions, such as procedures, loops, single statements, or arbitrary sequences of executable statements. The measurement system is event-driven, collecting time stamps only for the parts of the code previously selected, and permits the construction of different trace file formats.

It has two working modes: interactive and automatic.

In the interactive mode, the user can drive the measurement process through a graphical, Motif-based interface. Parts of the code can be selected with a point-and-click facility, and the system inserts automatically the measurement statements. The automatic mode is based on a formal specification of those parts of the code that have to be measured; this information is used by the system to generate the required instrumentation.

4.4. Performance Analysis Toolkit

The Performance Analysis Toolkit provides a coherent set of performance tools with a single interface to the user and the XPA. It has a modular structure which permits the insertion of new tools, the replacement of existing tools by others with enhanced functionality, and the removal of tools in a way essentially transparent to the user.

Based upon the requirements specified in a user query, the interface selects an appropriate set of tools to derive the needed information set. Similarly, the XPA-related part of the interface reacts to commands of the XPA by appropriately structuring the required information in the form of a *performance model* which provides data for all relevant parts of the measured code. An important component of the interface is a visualization module.

Below we discuss MEDEA, one of the tools included in the toolkit. Other tools such as Paragraph could be easily inserted as well.

MEDEA

The MEasurements Description Evaluation and Analysis tool (MEDEA) [10] is a general-purpose environment which contains features for the analysis of performance data collected by measuring/monitoring tools.

MEDEA has a modular structure. The various operations required for the program analysis have been logically subdivided into modules, each performing a specific manipulation of input data and the intermediate results.
The tracefile is pre-processed in order to perform a preliminary analysis of the data collected and to extract the values of performance parameters.
The amount of performance data collected and the large number of indices considered for program analysis can be manipulated through the application of statistical techniques. MEDEA allows the computation of standard statistical indices such as moments (mean, variance, standard deviation), parameter distribution, range, skewness, median, mode and percentiles. Simple data transformations are also provided.
The analysis of parameter distribution may lead to the application of various types of transformations in order to obtain comparable ranges for parameter values [11] [22]. Parameters with a highly skewed distribution can be transformed by taking the logarithm of the values. Static properties of the performance data set can be examined through a multidimensional *clustering analysis* technique which identifies groups of elements (units of work considered) having homogeneous characteristics with respect to predefined parameters.
As part of the clustering module, MEDEA computes the correlation matrix which helps

in selecting the most appropriate parameters for the analysis. The clustering algorithm implemented in MEDEA is the iterative non-hierarchical algorithm, the *nearest centroid method* also called *k-means* [21]. The basic algorithm partitions the elements into a specific number of clusters in order to minimize the distance (Euclidean or non-Euclidean) between each component and the centroid (center of mass) of the cluster it belongs to. The final result of clustering depends upon the values and ranges of different parameters [22]. A robust clustering analysis can be obtained if the parameters values are scaled so that they lie in a common interval.

In addition to the quantitative description of the program obtained by the statistical analysis, the *functional analysis* gives its "logical" description. This approach can identify, for example, the membership of particular components in a specific cluster.
Moreover, the application of various techniques such as numerical *fitting*, permits the reproduction of time-dependent program characteristics. Finally, MEDEA allows the computation of the most common scalability indices.

4.5. Expert System Adviser

The theoretical foundations and realization of an expert system adviser supporting automatic code parallelization are difficult issues. In this section, we describe only a few relevant aspects.

The expert system adviser is connected with the VFCS through a software module called *Interface*. The connection between the two tools has a double functionality: getting information from the compiler and returning, after an XPA-internal analysis process, solutions for the parallelization of the submitted program.

XPA generates a *program model* which plays a key role in the tuning process. The program model specifies, for example, the structure of the communication involved in a procedure or loop, as well as data dependence and data flow information. The analysis performed by the XPA is based upon this model.
Proposing a suitable data distribution for the arrays of the program is one of the most crucial problems which have to be managed during the parallelization process. The XPA generates, analyzing the program model, a "first" data distribution. This represents a starting point; depending on performance information derived from this choice, distributions may have to be subsequently modified.

For real codes, which may be very big, decisions relating to the parallelization strategy cannot, in general, be based upon performance information relating to the whole program. As a consequence, an appropriate reduction of the information space has to be performed by focusing only on the most relevant parts of the code. The *Selector* drives the measurement and performance prediction tools in instrumenting and analyzing these parts.

The realization of the entire parallelization process needs information from various sources.

Explicit knowledge must be provided in the following areas:

- properties of the source language,
- the target machine and its configuration,
- application domains,
- the restructuring system,
- transformation strategies, and
- the target machine compiler.

The *Engine*, composed of an *Interpreter* and a *Solver*, represents the core of the XPA. The Interpreter establishes the relationship between the performance model and the program model. The Solver performs a scalability analysis and uses asymptotic bounding analysis [24] to provide optimistic and pessimistic performance bounds. It also evaluates a cost function over a selected set of transformation sequences.

5. Related Work

The importance of representing *explicit knowledge* on both architectures and their software systems (including the compilers), and the identification of *rules for the application of optimization strategies* in terms of this knowledge, has been recognized by a number of workers in the field.

Paalvast, Sips and colleagues [28] have developed a calculus as a language-independent framework for expressing programs and their data, which enables the representation and reasoning about code transformations in the form of rewrite-rules. They have also shown that the language Booster [27] can be converted to their calculus.

A few attempts have been made to develop advanced programming environments as a collection of knowledge-based subsystems [7],[8],[23],[35]. This method has been explicitly applied to the problem of generating code for distributed-memory computers by Ko-Yang Wang at Purdue University. He has constructed a prototype system which has a description of a target machine, a set of program transformations, and rules for applying the transformations which are written in terms of the machine parameters. This approach permits code to be transformed for a variety of different machines by a single compilation system [35].

The MPP Apprentice performance monitoring tool helps the user to find and correct performance anomalies and inefficiencies in programs that are designed for Cray MPP systems. It is a post execution performance analysis tool that provides information about programs by examining data files that were created either at compile time or at runtime. The MPP Apprentice is a robust environment in which the user can navigate through

the code structure; it provides information regarding problems such as load imbalances, excessive synchronization or communication. General advice on ways to improve the code is also provided.

We conclude this discussion by mentioning a few performance analysis toolsets.

The Paragon Performance Monitoring environment is an integrated toolset that provides a consistent programming environment for monitoring and visualizing the performance of parallel programs. The performance is analyzed using application and system oriented tools sharing a common interface [4].

Pablo [31] is a performance analysis environment designed to provide features for the collection, analysis and presentation of performance data. It contains components for performance instrumentation and a graphical performance data reduction and analysis module.

Many performance visualization tools have been proposed. One of these is ParMon [25], an environment composed of a distributed event-driven measurement system for a class of DMMPs and an analysis component combined with a visualization facility. For debugging purposes, an animation feature of the program execution is provided.

6. Conclusion

In this paper, we described the salient features of an integrated program development environment for the purpose of automatically compiling Fortran programs for massively parallel distributed-memory architectures. Such an environment is currently being developed in the framework of VFCS: important components such as the restructuring system, P^3T, the code measurement tool, and MEDEA have already been implemented, whereas the XPA is still in a preprototype status. Key questions that have yet to be solved relate to interface issues, in particular within the performance analysis toolkit, and the implementation of model building and strategy selection in the expert system adviser.

REFERENCES

1. High Performance Fortran Forum. Technical Report, Rice University, Houston, TX, High Performance Fortran Language Specification Version 1.0, May 1993.
2. Message Passing Interface Forum. Document for a Standard Message-Passing Interface, University of Tennessee, Knoxville, Tennessee, October 1993.
3. A. Beguelin, J. Dongarra, A. Geist, B. Manchek, and V. Sunderam. A User's Guide to PVM. Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, Tennessee, July 1991.
4. T. Bemmerl and B. Ries. Performance Tools on Intel Scalable High Performance Computing Systems. In *Proc. 7th Int. Confernce on Modelling Techniques and tools for Computer performance Evaluation, Lecture Notes in Computer Science LNCS 794*, pages 76–88, Vienna, Austria, May 1994.

5. R. Blasko. Process graph and tool for performance analysis of parallel processes. In *Proceedings of IMACS Symposium on Mathematical Modelling (1.MATHMOD), Vienna, Austria*, February 1994.
6. L. Bomans and R. Hempel. The Argonne/GMD Macros in Fortran for Portable Parallel Programming and Their Implementation on the Intel iPSC/2. *Parallel Computing*, 15:119–132, 1990.
7. P. Bose. Heuristic rule-based program transformations for enhanced vectorization. In *Proceedings of the International Conference on Parallel Processing*, 1988.
8. P. Bose. Interactive program improvement via eave: an expert adviser for vectorization. In *Proceedings of the International Conference on Supercomputing, St. Malo*, 1988.
9. D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2(2):151–169.
10. M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. MEDEA: A tool for Workload Chracterization of Parallel Systems. Technical Report R3/, Progetto Finalizzato C.N.R. "Sistemi Informatici e Calcolo Parallelo", November 1994. [Submitted to: IEEE Computer Theme issues on Performance Evaluation Tools for Parallel and Distributed Computer Systems].
11. M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proceedings of IEEE*, 81(8):1136–1150, August 1993.
12. B. Chapman, S. Benkner, R. Blasko, P. Brezany, M. Egg, T. Fahringer, H. M. Gerndt, J. Hulman, B. Knaus, P.Kutschera, H. Moritsch, A. Schwald, V. Sipkova, and H. P. Zima. VIENNA FORTRAN Compilation System - Version 1.0. User's Guide, January 1993.
13. B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, August 1992.
14. D. L. Eager, E. D. Lazowska, and J. Zahorjan. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.
15. T. Fahringer. Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers. Ph.D. Thesis, University of Vienna, Institute for Software Tecnology and Parallel Systems, October 1993.
16. T. Fahringer. Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. In *IEEE Proc. of the Scalable High Performance Computing Conference 1994, Knoxville, TN*, May 1994.
17. T. Fahringer and H. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Proc. 7th ACM International Conference on Supercomputing*, Tokio, Japan, July 1993. [Invited Paper].
18. D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems.* Prentice-Hall, Engleewood Cliff, NJ, 1983.
19. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report,COMP TR90079, Rice University, Department of Computer Science Rice University, March 1991.
20. D. Ghosal, G. Serazzi, and S.K. Tripathi. The Processor Working Set and its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

21. J. A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, New York, 1975.
22. R. Jain. *The Art of Computer System Performance Analysis.* John Wiley & Sons, New York, 1991.
23. Wang Ko-Yang and D. Gannon. Applying AI Techniques to Program Optimization for Parallel Computers. In *Book: Parallel Processing for Supercomputers and Artificial Intelligence*, pages 441–485. McGraw-Hill, 1989.
24. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance, Computer System Analyis Using Queueing Network Models.* Printice-Hall, Englewood Cliffs, 1984.
25. P. Lenzi and G. Serazzi. ParMon: Parallel Monitor. Technical Report R3/95, Progetto Finalizzato C.N.R. "Sistemi Informatici e Calcolo Parallelo", October 1992. [in italian].
26. V. Natarajan, D. Chiou, and B. S. Ang. Performance Visualization on Monsoon. *Journal of Parallel and Distributed Computing*, (18):168–180, 1993.
27. E. Paalvast and H. Sips. A high-level language for the description of parallel algorithms. In *Proceedings of Parallel Computing 89, Leyden, The Netherlands*, August 1989.
28. E. Paalvast, A. van Gemund, and H. Sips. A method of parallel program generation with an application to booster language. In *Proceedings of the 4th International Conference on Supercomputing, Amsterdam*, June 1990.
29. M. Pantano. Integration of Workload Characterization Techniques and Tool in a Compiler for Parallel Systems. Ph.D. Thesis, Università di Pavia, February 1994. [in italian].
30. D. A. Reed. Experimental Analysis of Parallel Systems: Techniques and Open Problems. In *Proc. 7th Int. Confernce on Modelling Techniques and tools for Computer performance Evaluation, Lecture Notes in Computer Science LNCS 794*, pages 25–51, Vienna, Austria, May 1994.
31. D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyastha, T. Birkett, D. W. Jensen, B. A. Nazief, and B. K. Totty. Scalable performance environments for parallel systems. In *Proceedings of the Sixth Distributed Memory Computing Conference, Portland, Oregon*, April 1991.
32. E. Rosti and G. Serazzi. Workload Characterization for Performance Engineering of Parallel applications. *IEEE*, (9):457–462, January 1994.
33. Xian-He Sun. Parallel Computation models: Representation Analysis and Applications. Ph.D. Thesis, Michigan State University, Department of Computer Science, 1990.
34. Xian-He Sun and D. T. Rover. Scalability of Parallel Algorithm-Machine Combinations. *IEEE Transactions on Parallel and Distributed Systems*, pages 599–613, June 1994.
35. K. Y. Wang. A framework for intelligent parallel compilers. Technical Report CSD-TR-1044, Computer Science Dept., Purdue University , 1990.
36. H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.
37. H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - A Language Specification, Version 1.1. Technical Report ACPC/TR92-4, Austrian

Center for Parallel Computation, March 1992. Also published as: NASA Contractor Report 189629, ICASE Interim Report 21, NASA Langley Research Center, Hampton Virginia, March 1992.

38. H. Zima and B. Chapman. Compiling for Distributed-Memory Systems. In *Proc. of the IEEE, Special Section on Languages and Compilers for Parallel Machines*, pages 264–287, February 1993. Also published as: Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation, November 1992.
39. H. P. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

The Design and Implementation of the Reduction Routines in ScaLAPACK

Jaeyoung Choi, [a] Jack Dongarra[a,b], Susan Ostrouchov[a], Antoine P. Petitet[a], David W. Walker, [b] and R. Clint Whaley[a] *

[a]Department of Computer Science, University of Tennessee,
107 Ayres Hall
Knoxville, TN 37996-1301

[b]Mathematical Sciences Section, ak Ridge National Laboratory,
P. O. Box 2008, Bldg. 6012 , Oak Ridge, TN 37831-6367

This chapter discusses the design of linear algebra libraries for high performance computers. Particular emphasis is placed on the development of scalable algorithms for MIMD distributed memory concurrent computers. A brief description of the EISPACK, LINPACK, and LAPACK libraries is given, followed by an outline of ScaLAPACK, which is a distributed memory version of LAPACK currently under development. The importance of block-partitioned algorithms in reducing the frequency of data movement between different levels of hierarchical memory is stressed. The use of such algorithms helps reduce the message startup costs on distributed memory concurrent computers. Other key ideas in our approach are the use of distributed versions of the Level 3 Basic Linear Algebra Subprograms (BLAS) as computational building blocks, and the use of Basic Linear Algebra Communication Subprograms (BLACS) as communication building blocks. Together the distributed BLAS and the BLACS can be used to construct higher-level algorithms, and hide many details of the parallelism from the application developer.

The block-cyclic data distribution is described, and adopted as a good way of distributing block-partitioned matrices. Block-partitioned versions of the Cholesky and LU factorizations are presented, and optimization issues associated with the implementation of the LU factorization algorithm on distributed memory concurrent computers are discussed, together with its performance on the Intel Delta system. Finally, approaches to the design of library interfaces are reviewed.

*This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400, and by National Science Foundation Grant number ASC-ASC-9005933.

1. Introduction

Current advanced architecture computers are NUMA (Non-Uniform Memory Access) machines. They possess hierarchical memories, in which accesses to data in the upper levels of the memory hierarchy (registers, cache, and/or local memory) are faster than those in lower levels (shared or off-processor memory). One technique to more efficiently exploit the power of such machines is to develop algorithms that maximize reuse of data in the upper levels of memory. This can be done by partitioning the matrix or matrices into blocks and by performing the computation with matrix-vector or matrix-matrix operations on the blocks. A set of BLAS (Level 2 and 3 BLAS) [12,13] were proposed for that purpose. The Level 3 BLAS have been successfully used as the building blocks of a number of applications, including LAPACK [1,2], which is a successor of LINPACK [11] and EISPACK [16]. The LAPACK is a software library that uses block-partitioned algorithms for performing dense and banded linear algebra computations on vector and shared memory computers.

The scalable library we are developing for distributed-memory concurrent computers will be fully compatible with the LAPACK library for vector and shared memory computers, and is therefore called ScaLAPACK ("Scalable LAPACK") [5]. ScaLAPACK also makes use of block-partitioned algorithms. It can be used to solve the "Grand Challenge" problems on massively parallel, distributed-memory, concurrent computers [4,15].

The Basic Linear Algebra Communication Subprograms (BLACS) [3] comprise a package that provides ease-of-use and portability for message-passing in parallel linear algebra applications. The Parallel BLAS (PBLAS), which provide a simplified interface around the Parallel Block BLAS (PB-BLAS) [7], are intermediate level routines based on the sequential BLAS and the BLACS. The PBLAS provide all the functionality supported by parallel versions of the Level 2 and Level 3 BLAS on a restricted class of matrices having a block cyclic data distribution. The ScaLAPACK routines are built using the sequential BLAS, the BLACS, and the PBLAS modules. ScaLAPACK can be ported with minimal code modification to any machine on which the BLAS and the BLACS are available.

This paper presents the implementation details, performance, and scalability of the ScaLAPACK routines for the LU, QR and Cholesky factorization of dense matrices. Throughout the implementation of ScaLAPACK, we have tried to follow the LAPACK programming style by hiding most of the communications inside of the PBLAS and the ScaLAPACK auxiliary routines. We want to demonstrate how to make it simple to implement the complicated parallel routines without sacrificing performance.

Currently ScaLAPACK includes factorization routines with their solvers, routines to refine the solution to reduce the error, and routines to estimate the reciprocal of the condition number. ScaLAPACK also includes routines to reduce a real general matrix to Hessenberg or bidiagonal form, and a symmetric matrix to tridiagonal form. These reduction routines are considered in our separate paper [10].

The design philosophy of the ScaLAPACK library is addressed in Section 2. In Sec-

tion 3, we describe the ScaLAPACK factorization routines by comparing them with the corresponding LAPACK routines. Section 4 presents more details of the parallel implementation of the routines and performance results on the Intel family of computers: the iPSC/860, the Touchstone Delta, and the Paragon. In Section 5, the scalability of the algorithms on the systems is demonstrated, and conclusions and future work are presented.

2. Design Philosophy

In ScaLAPACK, algorithms are presented in terms of *processes*, rather than the processors of the physical hardware. A process is an independent thread of control with its own nonshared, distinct memory. Processes communicate by pairwise point-to-point communication or by collective communication as necessary. In general there may be several processes on a physical processor, in which case it is assumed that the runtime system handles the scheduling of processes. For example, execution of a process waiting to receive a message may be suspended and another process scheduled, thereby overlapping communication and computation. In the absence of such a sophisticated operating system, ScaLAPACK has been developed and tested for the case of one process per processor.

2.1. Block Cyclic Data Distribution

The way in which a matrix is distributed over the processes has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. The block cyclic distribution provides a simple, yet general-purpose way of distributing a block-partitioned matrix on distributed memory concurrent computers. The block cyclic data distribution is parameterized by the four numbers P, Q, r, and c, where $P \times Q$ is the process template and $r \times c$ is the block size. Blocks separated by a fixed stride in the column and row directions are assigned to the same process.

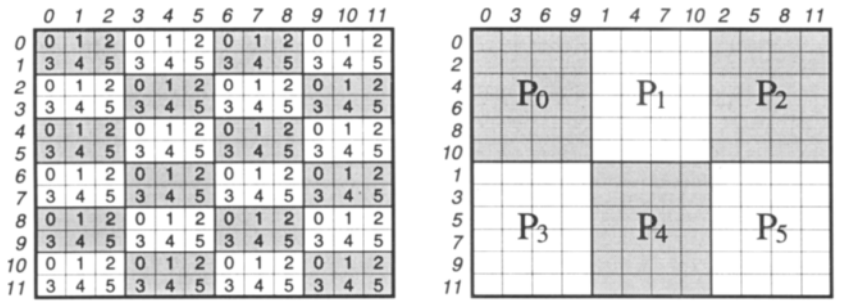
Suppose we have M objects indexed by the integers $0, 1, \cdots, M-1$. In the block cyclic data distribution the mapping of the global index, m, can be expressed as $m \longmapsto \langle p, b, i \rangle$, where p is the logical process number, b is the block number in process p, and i is the index within block b to which m is mapped. Thus, if the number of data objects in a block is r, the block cyclic data distribution may be written as follows:

$$m \longmapsto \left\langle s \bmod P,\ \left\lfloor \frac{s}{P} \right\rfloor,\ m \bmod r \right\rangle$$

where $s = \lfloor m/r \rfloor$ and P is the number of processes. The distribution of a block-partitioned matrix can be regarded as the tensor product of two such mappings: one that distributes the rows of the matrix over P processes, and another that distributes the columns over Q processes. That is, the matrix element indexed globally by (m, n) can be written as

$$(m, n) \longmapsto \langle (p, q), (b, d), (i, j) \rangle .$$

Figure 1 (a) shows an example of the block cyclic data distribution, where a matrix with 12×12 blocks is distributed over a 2×3 template. Therefore each process has 6×4 blocks



(a) block distribution over 2 x 3 template (b) data distribution from processor point-of-view

Figure 1. A matrix with 12 × 12 blocks is distributed over a 2 × 3 process template. (a) The shaded and unshaded areas represent different templates. The numbered squares represent blocks of elements, and the number indicates at which location in the process template the block is stored - all blocks labeled with the same number are stored in the same process. The *slanted* numbers, on the left and on the top of the matrix, represent indices of a row of blocks and of a column of blocks, respectively. (b) It is easier to see the distribution from the process point-of-view in order to implement algorithms. Each process has 6 × 4 blocks.

as in Figure 1 (b). The block cyclic data distribution is the only distribution supported by the ScaLAPACK routines. The block cyclic data distribution can reproduce most data distributions used in linear algebra computations. For example, one-dimensional distributions over rows or columns are obtained by choosing P or Q to be 1.

The nonscattered decomposition (or pure block distribution) is just a special case of the cyclic distribution in which the block size is given by $r = \lceil M/P \rceil$ and $c = \lceil N/Q \rceil$. That is,

$$(m,n) \longmapsto \left\langle \left(\left\lfloor \frac{m}{r} \right\rfloor, \left\lfloor \frac{n}{c} \right\rfloor \right), (0,0), (m \bmod r, n \bmod c) \right\rangle.$$

Similarly a purely scattered decomposition (or two dimensional wrapped distribution) is another special case in which the block size is given by $r = c = 1$,

$$(m,n) \longmapsto \left\langle (m \bmod P,\ n \bmod Q), \left(\left\lfloor \frac{m}{P} \right\rfloor, \left\lfloor \frac{n}{Q} \right\rfloor \right), (0,0) \right\rangle.$$

In factorization routines, such as the LU, QR and Cholesky factorizations, in which the distribution of work becomes uneven as the computation progresses, a larger block size results in greater load imbalance, but reduces the frequency of communication between processes. There is, therefore, a tradeoff between load imbalance and communication startup cost which can be controlled by varying the block size.

In addition to the load imbalance that arises as distributed data are eliminated from a computation, load imbalance may also arise due to computational "hot spots" where certain processes have more work to do between synchronization points than others. This is the case, for example, in the LU factorization algorithm in which partial pivoting is performed over rows, and only a single column of the process template is involved in the pivot search while the other processes are idle. Similarly, the evaluation of each block row of the U matrix requires the solution of a lower triangular system which involves only processes in a single row of the process template. The details of the implementation are described in Sections 3.1 and 4.1. The effect of this type of load imbalance can be minimized through the choice of P and Q.

2.2. Building Blocks

The ScaLAPACK routines are built out of a small number of modules. The most fundamental of these are the sequential BLAS, in particular the Level 2 and 3 BLAS, and the BLACS, which perform common matrix-oriented communications tasks. ScaLAPACK can be ported with minimal code modification to any machine on which the BLAS and the BLACS are available.

The BLACS comprise a package that provides ease-of-use and portability for message-passing in a parallel linear algebra program. The BLACS efficiently support not only point-to-point operations between processes on a logical two-dimensional process template, but also collective communications on such templates, or within just a template row or column.

Future software for dense linear algebra on MIMD platforms could consist of calls to the BLAS for computation and calls to the BLACS for communication. Since both packages will have been optimized for each particular platform, good performance should be achieved with relatively little effort. The BLACS have been implemented for the Intel family of computers, the TMC CM-5, the CRAY T3D, the IBM SP1 and SP2, and for PVM.

The Parallel BLAS (PBLAS) provide a simplified interface to the Parallel Block BLAS (PB-BLAS) [7] – the PBLAS are essentially C wrappers around the PB-BLAS, which in turn are intermediate level routines based on the BLACS and the sequential BLAS. The PBLAS provide all the functionality supported by parallel, distributed versions of the Level 2 and Level 3 BLAS, however, the PBLAS can only be used in operations on a restricted class of matrices having a block cyclic data distribution. These restrictions permit certain memory access and communication optimizations that would not be possible (or would be difficult) if general-purpose distributed Level 2 and Level 3 BLAS were used [6,8].

The sequential BLAS, the BLACS, and the PBLAS are the modules from which the higher level ScaLAPACK routines are built. The PBLAS are used as the highest level building blocks for implementing the ScaLAPACK library and provide the same ease-of-use and portability for ScaLAPACK that the BLAS provide for LAPACK. Most of the

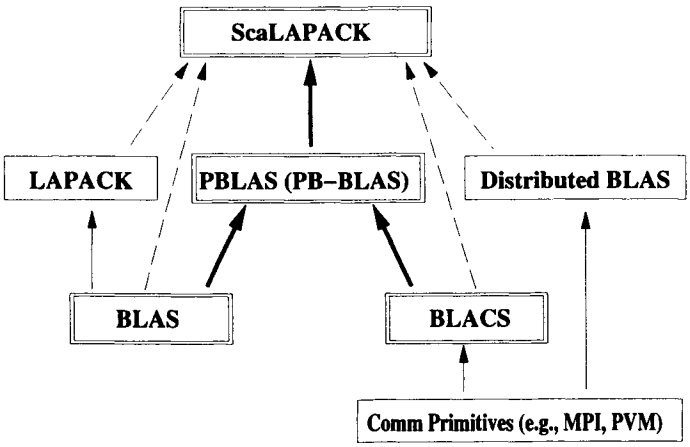


Figure 2. Hierarchical view of ScaLAPACK.

Level 2 and 3 BLAS routines in LAPACK routines can be replaced with the corresponding PBLAS routines in ScaLAPACK, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK. Thus, the ScaLAPACK code is modular, clear, and easy to read.

Figure 2 shows a hierarchical view of ScaLAPACK. Main ScaLAPACK routines usually call only the PBLAS, but the auxiliary ScaLAPACK routines may need to call the BLAS directly for local computations and the BLACS for communication among processes. In many cases the ScaLAPACK library will be sufficient to build applications. However, more expert users may make use of the lower level routines to build customized routines not provided in ScaLAPACK.

2.3. Design Principles

ScaLAPACK is designed to be the message-passing version of LAPACK. By maximizing the size of the submatrices multiplied in each process, that is, by maximizing the data reuse in the upper level of memory, it is possible to maximize the performance of the sequential BLAS. Similarly, by maximizing the size of the submatrices communicated among processes, the frequency of communication among processes can be reduced, thereby minimizing the communication startup cost. These two factors ensure that the ScaLAPACK routines have good performance and scalability characteristics.

The ScaLAPACK routines perform correctly for a wide range of inputs. For example, in the process of computing the elementary Householder vector in the QR factorization, the Euclidean norm needs to be computed without causing overflow and underflow problems. A PBLAS routine, `PDNRM2`, takes care of the problem (see Section 4.2). Similarly

if the data matrix is not positive definite in the Cholesky factorization, a process, which computes the Cholesky factorization on a diagonal block, halts its computation, yet other processes would keep waiting to finish their jobs. This problem can be avoided by broadcasting a flag to other processes to abort the computation (see Section 4.3).

3. Factorization Routines

In this section, we first briefly describe sequential, block-partitioned versions of the dense LU, QR, and Cholesky factorization routines of the LAPACK library. We use the right-looking versions of the routines for implementing them on distributed-memory concurrent computers, since this minimizes data communication and distributes the computation across all processes [14]. Then the parallel versions of these routines will be described.

For the implementation of the parallel block partitioned algorithms in ScaLAPACK, we assume that a matrix A is distributed over a $P \times Q$ process template with a block cyclic distribution and a block size of $n_b \times n_b$. Thus each column (or row) panel lies in one column (row) of the process template.

3.1. LU Factorization

The LU factorization applies a sequence of Gaussian eliminations to form $A = LU$, where A and L are $M \times N$ matrices, and U is an $N \times N$ matrix. L is unit lower triangular (lower triangular with 1's on the main diagonal), and U is upper triangular.

At the k-th step of the computation, it is assumed that the $m \times n$ submatrix of A ($m = M - k \cdot n_b$, $n = N - k \cdot n_b$) is to be partitioned as follows,

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}$$

where the block A_{11} is $n_b \times n_b$, A_{12} is $n_b \times (n - n_b)$, A_{21} is $(m - n_b) \times n_b$, and A_{22} is $(m - n_b) \times (n - n_b)$. L_{11} is a unit lower triangular matrix, and U_{11} is an upper triangular matrix.

At first, a sequence of Gaussian eliminations is performed on the first $m \times n_b$ panel of A (i.e., A_{11} and A_{21}). Once this is completed, the matrices L_{11}, L_{21}, and U_{11} are known, and we can rearrange the block equations,

$$\begin{aligned} U_{12} &\Leftarrow (L_{11})^{-1} A_{12}, \\ \tilde{A}_{22} &\Leftarrow A_{22} - L_{21}U_{12} = L_{22}U_{22}. \end{aligned}$$

The LU factorization can be done by recursively applying the steps outlined above to the $(m - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$. Figure 3 shows a snapshot of the block LU factorization. It shows how the column panel, L_{11} and L_{21}, and the row panel, U_{11} and U_{12}, are computed,

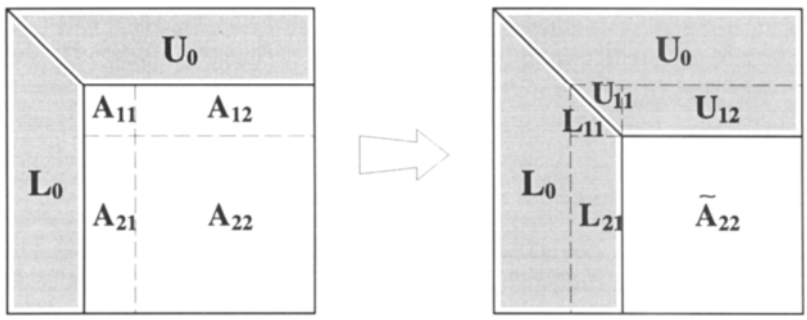


Figure 3. A snapshot of block LU factorization. It shows how the column panel, L_{11} and L_{21}, and the row panel, U_{11} and U_{12}, are computed, and how the trailing submatrix A_{22} is updated. The shaded areas represent data for which the corresponding computations are completed, that is, no more changes for these data will occur.

and how the trailing submatrix A_{22} is updated. In the figure, the shaded areas represent data for which the corresponding computations are completed.

The computation of the above steps in the LAPACK routine, `DGETRF`, involves the following operations:

1. `DGETF2`: Apply the LU factorization on an $m \times n_b$ column panel of A (i.e., A_{11} and A_{21}).
   - [Repeat n_b times ($i = 1, \cdots, n_b$)]
     - `IDAMAX`: find the (absolute) maximum element of the i-th column and its location
     - `DSWAP`: interchange the i-th row with the row which holds the maximum
     - `DSCAL`: scale the i-th column of the matrix
     - `DGER`: update the trailing submatrix
2. `DLASWP`: Apply interchanges to the rest of columns.
3. `DTRSM`: Compute the subdiagonal block of U,
   $$U_{12} \Leftarrow (L_{11})^{-1} A_{12}.$$
4. `DGEMM`: Update the rest of the matrix, A_{22},
   $$\tilde{A}_{22} \Leftarrow A_{22} - L_{21}U_{12} = L_{22}U_{22}.$$

The corresponding parallel implementation of the ScaLAPACK routine, `PDGETRF`, proceeds as follows:

1. PDGETF2: A column of processes performs the LU factorization on an $m \times n_b$ panel of A (i.e., A_{11} and A_{21}).
   - [Repeat n_b times $(i = 1, \cdots, n_b)$]
     - PDAMAX: find the (absolute) maximum value of the i-th column and its location (pivot information will be stored on the column of processes)
     - PDLASWP: interchange the i-th row with the row which hold the maximum
     - DSCAL: scale the i-th column of the matrix
     - PDGER: broadcast the i-th row columnwise ($(n - i)$ elements) and update the trailing submatrix
   - Broadcast the pivot information rowwise
2. PDLASWP: Apply interchanges to the rest of columns
3. PDTRSM: L_{11} is broadcast along a row of the processes, which compute the row panel U_{12}.
4. PDGEMM: The column panel L_{21} and the row panel U_{12} are broadcast rowwise and columnwise, respectively. Then, processes update their local portions of the matrix, A_{22}.

3.2. QR Factorization

Given an $M \times N$ matrix A, we seek the factorization $A = QR$, where Q is an $M \times M$ orthogonal matrix, and R is an $M \times N$ upper triangular matrix. At the k-th step of the computation, we partition this factorization to the $m \times n$ submatrix of A as

$$A = \begin{pmatrix} A_1 & A_2 \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = Q \cdot \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

where the block A_{11} is $n_b \times n_b$, A_{12} is $n_b \times (n - n_b)$, A_{21} is $(m - n_b) \times n_b$, and A_{22} is $(m - n_b) \times (n - n_b)$. A_1 is an $m \times n_b$ matrix, containing the first n_b columns of the matrix A, and A_2 is an $m \times (n - n_b)$ matrix, containing the last $(n - n_b)$ columns of A (that is, $A_1 = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$ and $A_2 = \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$). R_{11} is a $n_b \times n_b$ upper triangular matrix.

A QR factorization is performed on the first $m \times n_b$ panel of A (i.e., A_1). In practice, Q is computed by applying a series of Householder transformations to A_1 of the form, $H_i = I - \tau_i v_i v_i^T$ where $i = 1, \cdots, n_b$. The vector v_i is of length m with 0's for the first $i - 1$ entries and 1 for the i-th entry, and $\tau_i = 2/(v_i^T v_i)$. During the QR factorization, the vector v_i overwrites the entries of A below the diagonal, and τ_i is stored in a vector. Furthermore, it can be shown that $Q = H_1 H_2 \cdots H_{n_b} = I - VTV^T$, where T is $n_b \times n_b$ upper triangular and the i-th column of V equals v_i. This is indeed a block version of the QR factorization, and is rich in matrix-matrix operations.

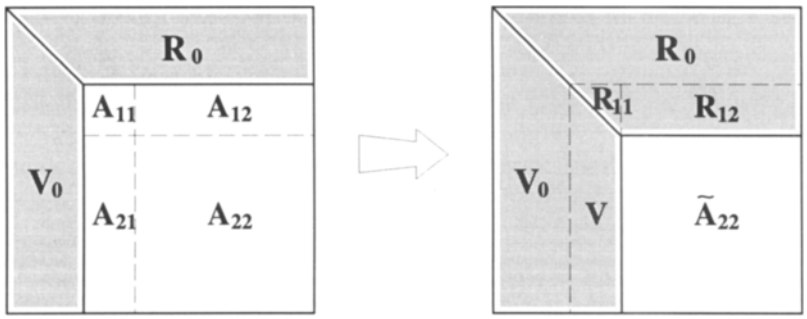


Figure 4. A snapshot of block QR factorization. During the computation, the sequence of the Householder vectors V is computed, and the row panel R_{11} and R_{12}, and the trailing submatrix A_{22} are updated.

The block equation can be rearranges as

$$\tilde{A}_2 = \begin{pmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{pmatrix} \Leftarrow \begin{pmatrix} R_{12} \\ R_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^T V^T\right) A_2.$$

A snapshot of the block QR factorization is shown in Figure 4. During the computation, the sequence of the Householder vectors V is computed, and the row panel R_{11} and R_{12}, and the trailing submatrix A_{22} are updated. The factorization can be done by recursively applying the steps outlined above to the $(m - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$.

The computation of the above steps of the LAPACK routine, `DGEQRF`, involves the following operations:

1. `DGEQR2`: Compute the QR factorization on an $m \times n_b$ panel of A (i.e., A_1)
   - [Repeat n_b times $(i = 1, \cdots, n_b)$]
     - `DLARFG`: generate the elementary reflector v_i and τ_i
     - `DLARF`: update the trailing submatrix
       $\tilde{A} \Leftarrow H_i^T A = (I - \tau_i v_i v_i^T) A$
2. `DLARFT`: Compute the triangular factor T of the block reflector Q
3. `DLARFB`: Apply Q^T to the rest of the matrix from the left

$$\tilde{A}_2 \Leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = Q^T A_2 = \left(I - VT^T V^T\right) A_2$$

   - `DGEMM`: $W \Leftarrow V^T A_2$

- DTRMM: $W \Leftarrow T^T W$
- DGEMM: $\tilde{A}_2 \Leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW$

The corresponding steps of the ScaLAPACK routine, PDGEQRF, are as follows:

1. PDGEQR2: A column of processes performs the QR factorization on an $m \times n_b$ panel of A (i.e., A_1)
   - [Repeat n_b times $(i = 1, \cdots, n_b)$]
     - PDLARFG: generate elementary reflector v_i and τ_i
     - PDLARF: update the trailing submatrix
2. PDLARFT: A column of processes, which has a sequence of the Householder vectors V, computes T.
3. PDLARFB: Apply Q^T to the rest of the matrix from the left
   - PDGEMM: The column of blocks V is broadcast rowwise and then saved in other processes. The transpose of V is locally multiplied by A_2, then the products are added to one row of processes $(W \Leftarrow V^T A_2)$.
   - PDTRMM: T is broadcast rowwise and multiplied with the sum $(W \Leftarrow T^T W)$.
   - PDGEMM: The row of blocks W is broadcast columnwise. Now, processes have their own portions of V and W, then they update the local portions of the matrix A $(\tilde{A}_2 \Leftarrow \begin{pmatrix} \tilde{R}_{12} \\ \tilde{R}_{22} \end{pmatrix} = A_2 - VW)$.

3.3. Cholesky Factorization

Cholesky factorization factors an $N \times N$, symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, i.e., $A = LL^T$ (or $A = U^T U$, where U is upper triangular). It is assumed that the lower triangler portion of A is stored in the lower triangle of a two-dimensional array and that the computed elements of L overwrite the given elements of A. At the k-th step, we partition the $n \times n$ matrices A, L, and L^T, and write the system as

$$\begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{pmatrix}$$
$$= \begin{pmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix}$$

where the block A_{11} is $n_b \times n_b$, A_{21} is $(n - n_b) \times n_b$, and A_{22} is $(n - n_b) \times (n - n_b)$. L_{11} and L_{22} are lower triangular.

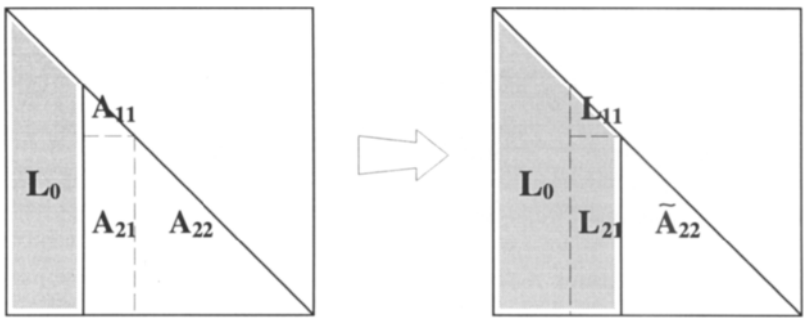


Figure 5. A snapshot of block Cholesky factorization shows how the column panel L (L_{11} and L_{21}) is computed and how the trailing submatrix A_{22} is updated.

The block-partitioned form of Cholesky factorization may be inferred inductively as follows. If we assume that L_{11}, the lower triangular Cholesky factor of A_{11}, is known, we can rearrange the block equations,

$$L_{21} \Leftarrow A_{21}(L_{11}^T)^{-1},$$
$$\tilde{A}_{22} \Leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T.$$

A snapshot of the block Cholesky factorization algorithm in Figure 5 shows how the column panel L (L_{11} and L_{21}) is computed and how the trailing submatrix A_{22} is updated. The factorization can be done by recursively applying the steps outlined above to the $(n - n_b) \times (n - n_b)$ matrix $\tilde{A}_{22}$.

In the right-looking version of the LAPACK routine, the computation of the above steps involves the following operations:

1. `DPOTF2`: Compute the Cholesky factorization of the diagonal block, A_{11}.

   $$A_{11} \Rightarrow L_{11}L_{11}^T$$

2. `DTRSM`: Compute the subdiagonal block of L,

   $$L_{21} \Leftarrow A_{21}(L_{11}^T)^{-1}$$

3. `DSYRK`: Update the rest of the matrix,

   $$\tilde{A}_{22} \Leftarrow A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$$

The parallel implementation of the corresponding ScaLAPACK routine, `PDPOTRF`, proceeds as follows:

1. `PDPOTF2`: A process P_i, which has the $n_b \times n_b$ diagonal block A_{11}, performs the Cholesky factorization of A_{11}.

- P_i performs $A_{11} \Rightarrow L_{11}L_{11}^T$, and sets a flag if A_{11} is not positive definite.
- P_i broadcasts the flag to all other processes so that the computation can be stopped if A_{11} is not positive definite.

2. PDTRSM: L_{11} is broadcast along a column of the processes, which compute the column of blocks of L_{21}.

3. PDSYRK: the column of blocks L_{21} is broadcast rowwise and then transposed. Now, processes have their own portions of L_{21} and L_{21}^T. They update their local portions of the matrix A_{22}.

4. Results and Discussion

We have outlined the basic parallel implementation of the three factorization routines. In this section, we describe a little more detail of the parallel implementation of the routines and performance results on the Intel iPSC/860, Touchstone Delta, and Paragon systems. Further we have investigated possible variations of the routines for the better performance.

The Intel iPSC/860 is a parallel architecture with up to 128 processing nodes. Each node consists of an i860 processor with 8 Mbytes of memory. The system is interconnected with a hypercube structure. The Delta system contains 512 i860-based computational nodes with 16 Mbytes /node, connected with a 2-D mesh communication network. The Intel Paragon located at Oak Ridge National Laboratory has 512 computational nodes, interconnected with a 2-D mesh. Each node has 32 Mbytes of memory and two i860XP processors, one for computation and the other for communication. The Intel iPSC/860 and Delta machines both use the same 40MHz i860 processor, but the Delta has a higher communication bandwidth. Significantly higher performance can be attained on the Paragon system, since it uses the faster 50 MHz i860XP processor and has a larger communication bandwidth.

On each node all computation was performed in double precision arithmetic, using assembly-coded BLAS (Level 1, 2, and 3), provided by Intel. Communication was performed using the BLACS package, customized for the Intel systems. Most computation by the BLAS and communication by the BLACS are hidden within the PBLAS.

The optimal block size of a routine could be determined by the algorithm itself and characteristics of the target computer system, such as the ratio of computation speed over communication speed and the process mesh aspect ratio of P/Q. The block size, n_b, of the routines was selected to produce the best performance of the routines for the given target machines. The numbers of floating point operations for an $N \times N$ matrix were assumed to be $2/3\,N^3$ for the LU factorization, $4/3\,N^3$ for the QR factorization, and $1/3\,N^3$ for the Cholesky factorization.

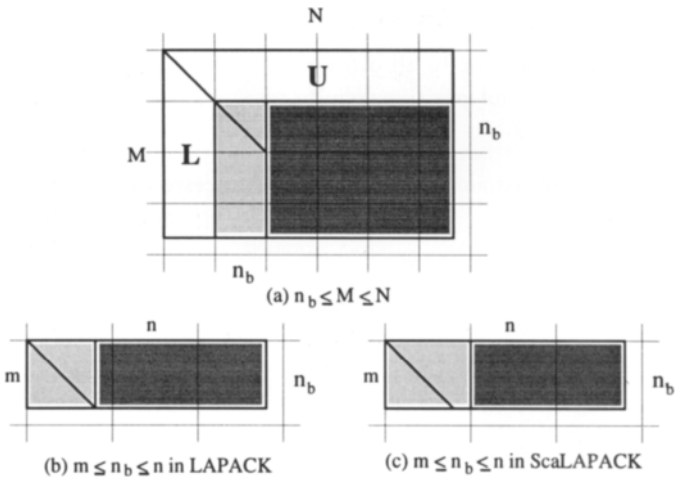


Figure 6. A snapshot of the block LU factorization when the matrix is a fat matrix ($M < N$)

4.1. LU Factorization

In LAPACK, the block size can be arbitrarily chosen to achieve optimal performance of a routine. But with the block cyclic data distribution in ScaLAPACK, the block size affects how the matrix is distributed over the 2-D process grid, and hence impacts load balance and communication overhead. In this section we investigate how the block distribution affects the implementation of the ScaLAPACK routine.

In LAPACK, the block size can be arbitrarily chosen to achieve optimal performance of a routine. But with the block cyclic data distribution in ScaLAPACK, it has a physical meaning how the matrix is distributed over the 2-D process grid, and it affects load balance and communication overhead. In this section we investigate the effects of the physical meaning of the block size on the implementation of the ScaLAPACK.

Figure 6 (a) shows a snapshot of the block LU factorization when the matrix is a fat matrix ($M < N$). As explained in Section 3.1, `DGETF2` (or `PDGETF2`) applies the LU factorization on the $m \times n'$ column panel of A, where $n' \leq n_b$. Let us consider the computation of the last row of blocks carefully, where $m \leq n_b \leq n$.

`DGETF2` takes n' as MIN(m, n, n_b), and applies the factorization on the $m \times m$ square portion of the matrix (lightly shaded area), as shown in Figure 6 (b). The rest of the matrix (darkly shaded area) is updated later with `DTRSM`. However, it is assumed that the level-3 PBLAS routine, `PDTRSM`, cannot deal with a matrix starting from the middle of the block [7]. As illustrated in Figure 6 (c), `PDGETF2` in ScaLAPACK takes n' as MIN(n,

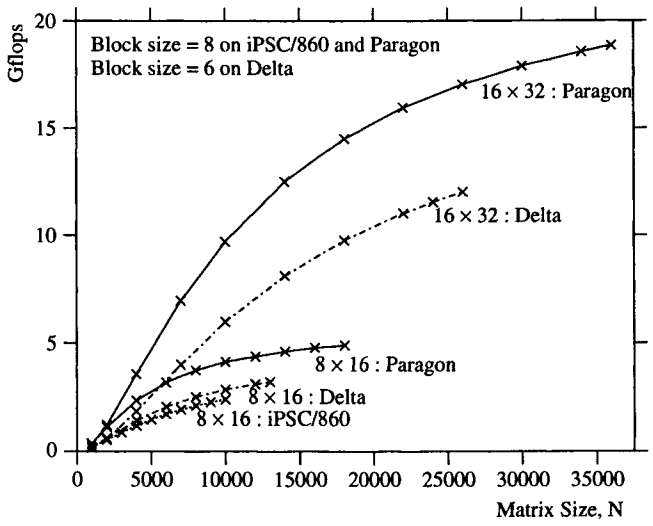


Figure 7. Performance of the LU factorization on the Intel iPSC/860, Delta, and Paragon

n_b), and it computes the factorization on $m \times n_b$ portion of the matrix. The rest of the matrix satisfies the preassumption of using PDTRSM. Thus in ScaLAPACK, computations are generally aligned with block boundaries.

Figure 7 shows the performance of the ScaLAPACK LU factorization routine on the Intel iPSC/860, the Delta, and the Paragon in Gflops (gigaflops per second) as a function of number of processes. The optimal block size on the iPSC/860 and the Paragon was 8, and on the Delta was 6, and the best performance was attained with a process aspect ratio, $1/4 \leq P/Q \leq 1/2$. The LU routine attained 2.4 Gflops for a matrix size of $N = 10000$ on the iPSC/860; 12.0 Gflops for $N = 26000$ on the Delta; and 18.8 Gflops for $N = 36000$ on the Paragon.

The LU factorization routine is a little more complicated than the other routines because it requires column pivoting. In other words, many possible different implementations exist. We describe briefly other possible variations, which will destroy the modularity and simplicity of the implementation, but attain a slightly better performance.

In the unblocked LU factorization routine (PDGETF2), after finding the maximum value of the i-th column (PDAMAX), the i-th row will be exchanged with the pivot row containing the maximum value. Then the new i-th row is broadcast columnwise ($(n_b - i)$ elements) in PDGER. Instead, the communications of PDLASWP and PDGER can be combined. That is, the pivot row is directly broadcast to other processes in the column, and the pivot row is replaced with the i-th row later.

The column of processes, which has the $m \times n_b$ column panel of A (i.e., A_{11} and A_{21}),

applies interchanges twice in order not to swap the data in the column panel (PDLASWP). These two separate communication processes also can be combined.

Finally, after completing the factorization of the column panel (PDGETF2), the column of processes, which has the column panel, broadcasts rowwise the pivot information for PDLASWP, L_{11} for PDTRSM, and L_{21} for PDGEMM. It is possible to combine the three messages to save the number of communications (or combine L_{11} and L_{21}), and broadcast rowwise the combined message.

It takes a non-negligible time to broadcast the column panel of L across the process template. It is possible to increase the overlap of communication with computation by broadcasting each column rowwise as soon as they are evaluated, rather than broadcasting all of the panel across after factoring it. With these modified communication schemes, the performance of the routine will be increased, but in our experiments we have found the improvement to be less than 5 %.

4.2. QR Factorization

It is required to compute an Euclidean norm of the vector, $A_{:i}$, to get the elementary Householder vector v_i. The sequential LAPACK routine, DLARFG, calls the Level-1 BLAS routine, DNRM2, which computes the norm without causing underflow or overflow problems. In the corresponding parallel ScaLAPACK routine, PDLARFG, each process in the column of processes, which holds the vector, $A_{:i}$, computes the global norm safely using the PDNRM2 routine.

The triangular factor T of a block Householder reflector V can be computed inside of PDGEQR2, and the routine can generate T directly, instead of τ. But keeping T might cause problems if the block size is subsequently changed by redistributing the matrix. However a separate routine PDLARFT needs only one communication over a column of processes to compute T on the fly, so we chose the LAPACK implementation style to compute T, i. e., we store τ and V, and generate T when necessary.

The $m \times n_b$ lower trapezoidal part of V, which is a sequence of the n_b Householder vectors, will be accessed in the form,

$$V = \begin{pmatrix} V_1 \\ V_2 \end{pmatrix}$$

where V_1 is $n_b \times n_b$ unit lower triangular, and V_2 is $(m - n_b) \times n_b$. In the sequential routine, the multiplication involving V is divided into two steps: DTRMM with V_1 and DGEMM with V_2. However, in the parallel implementation, V will be broadcast rowwise to other processes, and all columns of processes have their own copies of V. The upper triangular part of V (including the main diagonal) will not be accessed by the other columns of processes. By sending $\bar{V}$, which is a copy of V, but with 0's on the upper triangle and 1's on the main diagonal, the multiplications involving $\bar{V}$ can be done in one step (DGEMM) as illustrated in Figure 8. This one step multiplication not only simplifies the implementation of the routine (PDLARFB), but also increases the overall performance of the routine (PDGEQRF).

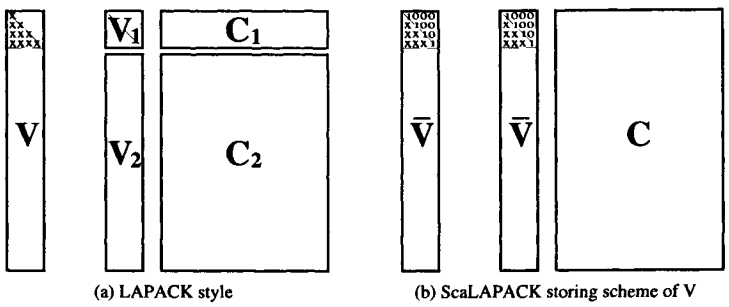


Figure 8. The storage scheme of the lower trapezoidal matrix V in ScaLAPACK QR factorization. By broadcasting $\bar{V}$, which is a copy of V, but with 0's on the upper triangular and 1's on the main diagonal, the computation involving V can be done in one step. (a) $V^T \cdot C = V_1^T \cdot C_1$ (DTRMM) $+ V_2^T \cdot C_2$ (DGEMM) (b) $V^T \cdot C = \bar{V} \cdot C$ (DGEMM)

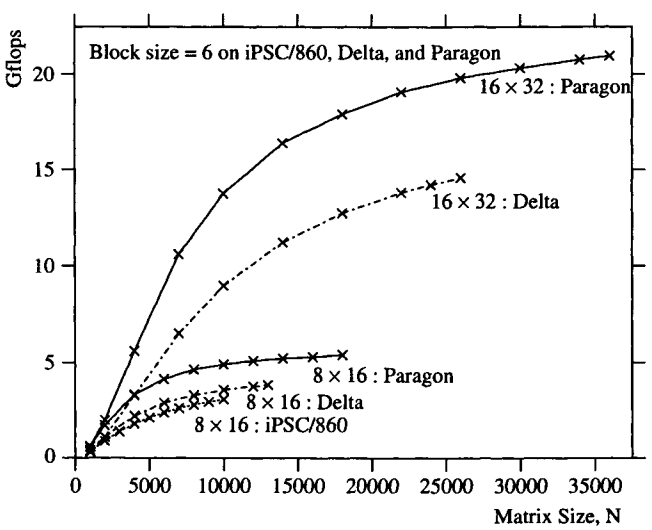


Figure 9. Performance of the QR factorization on the Intel iPSC/860, Delta, and Paragon

Figure 9 shows the performance of the QR factorization routine on the Intel family of concurrent computers. The optimal block size of $n_b = 6$ was used on all the machines. Best performance was attained with an aspect ratio of $1/4 \leq P/Q \leq 1/2$. The highest performances of 3.1 Gflops for $N = 10000$ was obtained on the iPSC/860; 14.6 Gflops for $N = 26000$ on the Delta; and 21.0 Gflops for $N = 36000$ on the Paragon.

4.3. Cholesky Factorization

The PDSYRK routine performs rank-n_b updates on an $(n - n_b) \times (n - n_b)$ symmetric matrix A_{22} with an $(n - n_b) \times n_b$ column of blocks L_{21}. After broadcasting L_{21} rowwise and transposing it, each process updates its own portion of A_{22} with its own portion of L_{21} and L_{21}^T. The globally lower triangular matrix A_{22} is not stored in the lower triangular form in the local processes as shown in Figure 10, thus it is complicated to update. The simplest way to do this is to repeatedly update one column of blocks of A_{22}; but if the block size is small, this updating process will not be efficient.

It is possible, and more efficient, to update several column blocks at a time. It is desirable to compute a multiple of LCM/Q blocks simultaneously since the processes easily determine their own physical data distribution of the lower triangular matrix of A, where LCM is the least common multiple of P and Q. Figures 10 (b) and (c) show how to update 2 ($= LCM/Q$) and 4 ($= 2 \cdot LCM/Q$) columns of blocks of A at a time, respectively. In the argument list of the PDSYRK routine, MULLEN specifies an approximate length of multiplication to update A_{22} efficiently. The multiple factor is computed by $k = \lceil MULLEN/((LCM/Q) \cdot n_b) \rceil$, and $k \cdot (LCM/Q)$ columns of blocks are updated simultaneously inside of the routine. For details, see [7]. The optimum number is determined by processor characteristics as well as the size of the matrix and the block size. The optimum number was found to be about 40 on the Intel iPSC/860 and Delta computers.

The effect of the block size on the performance of the Cholesky factorization is shown in Figure 11 on 8×16 and 16×16 processors of the Intel Delta. The best performance was obtained at the block size of $n_b = 24$, but relatively good performance could be expected with the block size of $n_b \geq 6$, since the routine updates multiple column panels at a time. With $n_b = 24$, and $MULLEN = 40$, the routine updates 2 ($= \lceil MULLEN/n_b \rceil = \lceil 40/24 \rceil$, where $LCM = Q$) column panels simultaneously.

Figure 12 shows the performance of the Cholesky factorization routine. The best performance was attained with the aspect ratio of $1/2 \leq P/Q \leq 1$. The routine ran at 1.8 Gflops for $N = 9600$ on the iPSC/860; 10.5 Gflops for $N = 26000$ on the Delta; and 16.9 Gflops for $N = 36000$ on the Paragon.

If A is not positive definite, the Cholesky factorization should be terminated in the middle of the computation. As outlined in Section 3.3, a process P_i computes the Cholesky factor L_{11} from A_{11}. After computing L_{11}, P_i broadcasts a flag to other processes so that all processes stop the computation if A_{11} is not positive definite. If A is guaranteed to be positive definite, the process of broadcasting the flag can be skipped, and another performance increase can be expected.

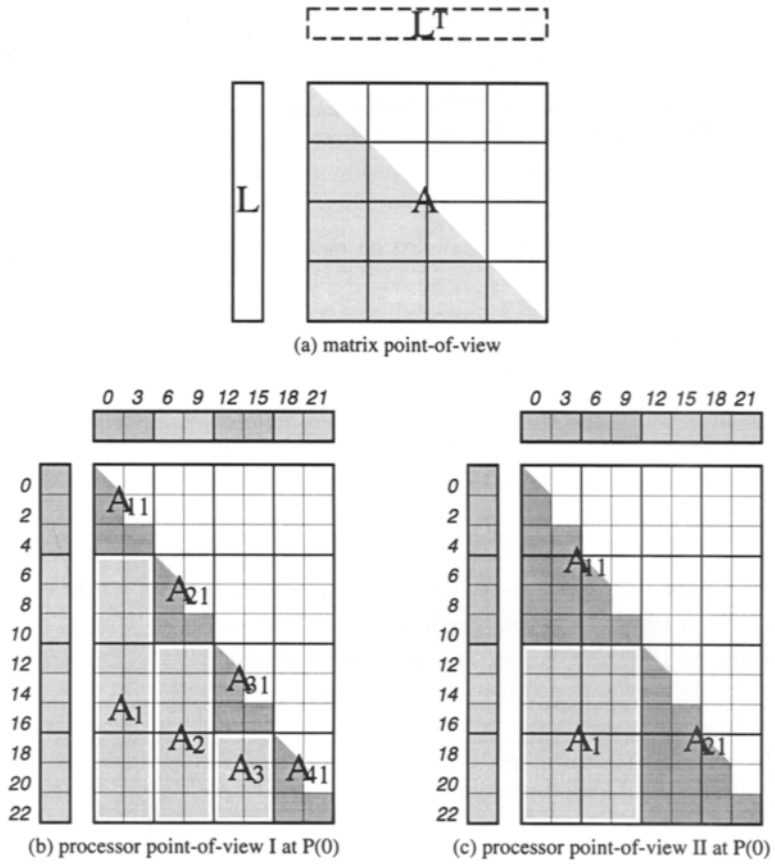


Figure 10. PDSYRK performs a rank-k update on a symmetric matrix. It is assumed that 24×24 blocks of A are distributed over a 2×3 process template. (a) A is a globally symmetric lower triangular matrix. (b) It is possible to update 2 ($= LCM/Q$) columns of blocks of A at a time. (c) It is more efficient to update 4 ($= 2 \cdot LCM/Q$) columns of blocks of A simultaneously.

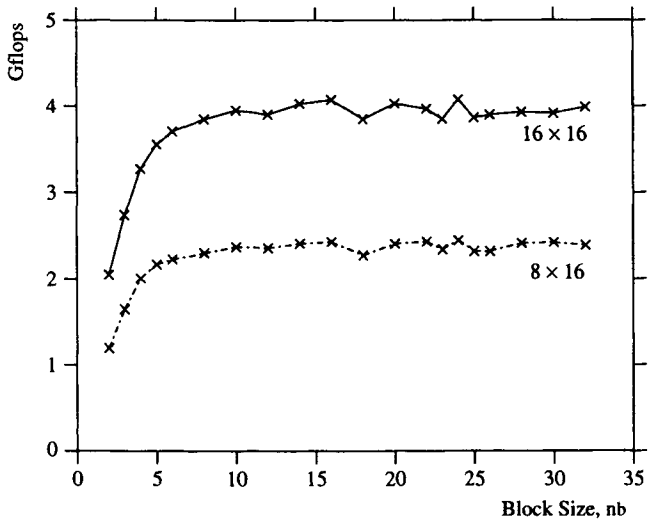


Figure 11. Performance of the Cholesky factorization as a function of the block size on the Intel Delta ($N = 10000$). The best performance was attained with the block size of $n_b = 24$ on 8×16 and 16×16 processes, but was quite insensitive to block size for $n_b \geq 8$.

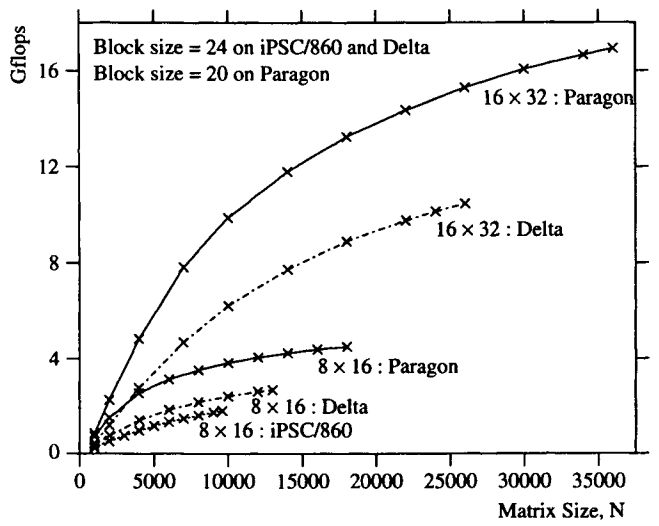


Figure 12. Performance of the Cholesky factorization on the Intel iPSC/860, Delta, and Paragon

5. Scalability and Conclusions

The performance of the three factorization routines on 128 nodes of the iPSC/860 and 512 nodes of the Delta and the Paragon were compared in Figures 13, 14, and 15, respectively. Generally the QR factorization routine has the best performance since the updating process of $Q^T A = (I - VTV^T)A$ is rich in matrix-matrix operation, and the number of floating point operations is the largest $(4/3\,N^3)$. The Cholesky factorization involves operations on a symmetric matrix, and the total number of floating point operations $(1/3\,N^3)$ is less than the other routines, thus its performance is poorer. The LU factorization routine seems to have more communication overhead since the routine contains column pivoting and row swapping operations. On the Delta, which has faster communication than the iPSC/860, the LU routine is slower than the Cholesky routine for small problem size ($N \leq 12000$).

The performance results in Figures 7, 9, and 12 can be used to assess the scalability of the factorization routines. In general, concurrent efficiency, ϵ, is defined as the concurrent speedup per process. That is, for the given problem size, N, on the number of processes used, N_p,

$$\epsilon(N, N_p) = \frac{1}{N_p} \frac{T_s(N)}{T_p(N, N_p)}$$

where $T_p(N, N_p)$ is the time for a problem of size N to run on N_p processes, and $T_s(N)$ is the time to run on one process using the best sequential algorithm. Another approach to investigate the efficiency is to see how the performance per process degrades as the number of processes increases for a fixed grain size, i. e., by plotting isogranularity curves in the (N_p, G) plane, where G is the performance. Since

$$G \propto \frac{T_s(N)}{T_p(N, N_p)} = N_p\, \epsilon(N, N_p),$$

the scalability for memory-constrained problems can readily be accessed by the extent to which the isogranularity curves differ from linearity.

Figures 16, 17, and 18 show the isogranularity plots for the ScaLAPACK factorization routines on the iPSC/860, the Delta, and the Paragon, respectively. The matrix size per process is fixed at 5 Mbytes on the iPSC/860, 9 Mbytes on the Delta, and 5 and 20 Mbytes on the Paragon. The linearity of the plots in the figures indicates that the ScaLAPACK routines have good scalability characteristics on these systems.

We have demonstrated that the LAPACK factorization routines can be parallelized fairly easily to the corresponding ScaLAPACK routines with a small set of low-level modules, namely the sequential BLAS, the BLACS, and the PBLAS. The PBLAS are particularly useful for developing and implementing a parallel dense linear algebra library relying on the block cyclic data distribution. In general, the Level 2 and 3 BLAS routines in the LAPACK code can be replaced on a one-for-one basis by the corresponding PBLAS routines. Parallel routines implemented with the PBLAS have good performance,

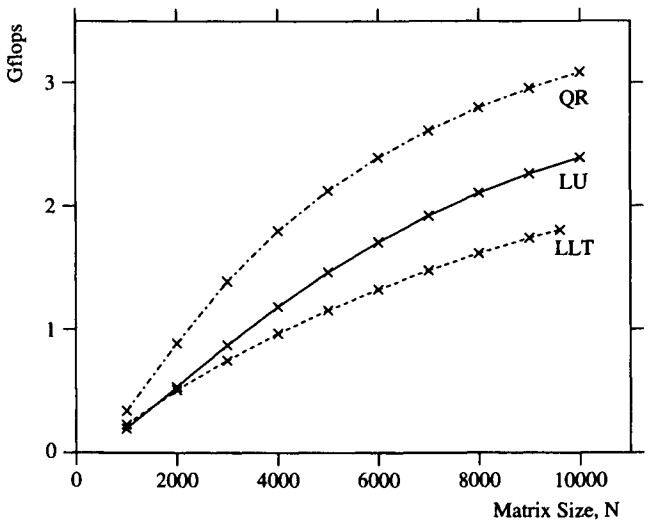


Figure 13. Performance of factorization routines on Intel iPSC/860 (128 nodes)

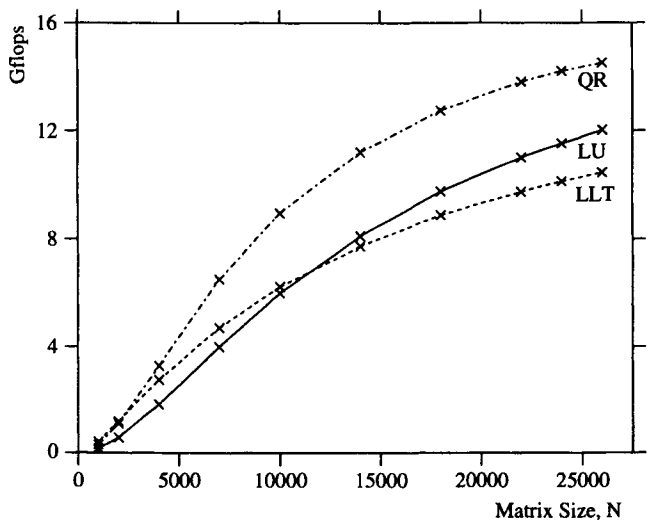


Figure 14. Performance of factorization routines on Intel Delta (512 nodes)

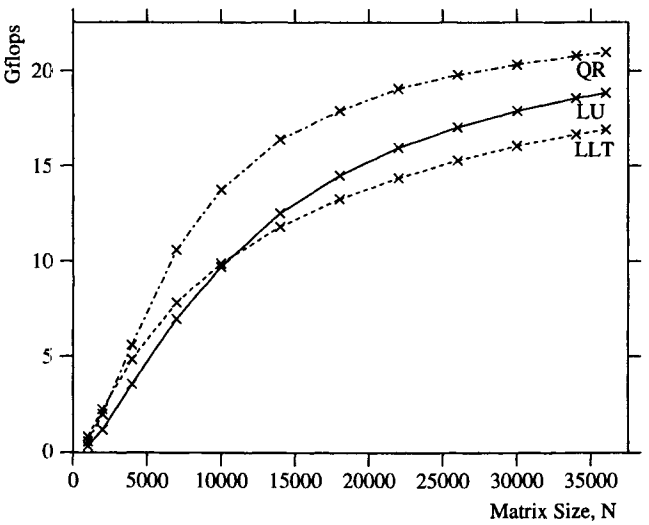


Figure 15. Performance of factorization routines on Intel Paragon (512 nodes)

since the computation performed by each process within PBLAS routines can be itself performed using the assembly-coded sequential BLAS.

There is a tradeoff between performance and software design considerations, such as modularity and clarity, in designing and implementing software libraries. As described in Section 4.1, it is possible to combine communications to reduce the communication costs in several places, such as in factorizing the column panel (`PDLASWP` and `PDGER`), in swapping data in the column panel (`PDGER`), and in broadcasting the column panel (`PDTRSM` and `PDGEMM`), and to replace the high level routines, such as the PBLAS, by calls to the lower level routines, such as the sequential BLAS and the BLACS. However, we have found that the performance gain is too small to justify the resulting loss of software modularity.

We have shown that the ScaLAPACK factorization routines have good performance and scalability on the Intel iPSC/860, the Delta, and the Paragon systems. Similar studies will be performed on recent machines, including the TMC CM-5, the Cray T3D, and the IBM SP1 and SP2.

Currently the ScaLAPACK library includes not only the LU, QR, and Cholesky factorization routines, but also factorization solvers, routines to refine the solutions to reduce error, and routines to estimate the reciprocal of the condition number. The ScaLAPACK routines are currently available through *netlib* only for double precision real data, but in the near future, we intend to release routines for other numeric data types, such as single precision real and complex, and double precision complex. To obtain the routines, and the ScaLAPACK Reference Manual [9], send the message "`send index from scalapack`" to

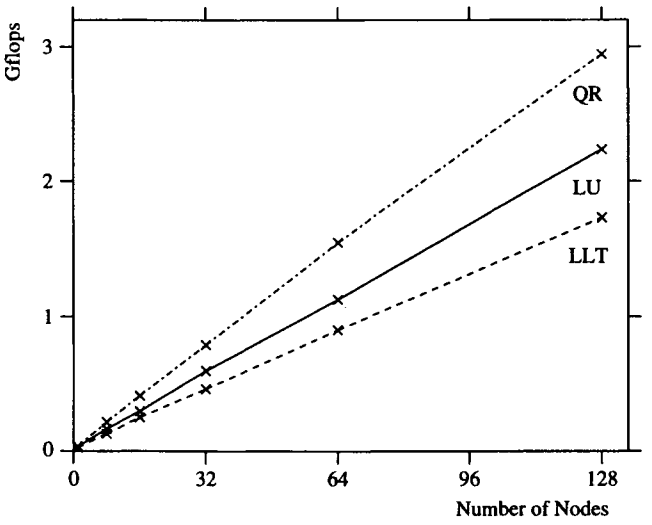


Figure 16. Scalability of factorization routines on the Intel iPSC/860 (5 Mbytes/node)

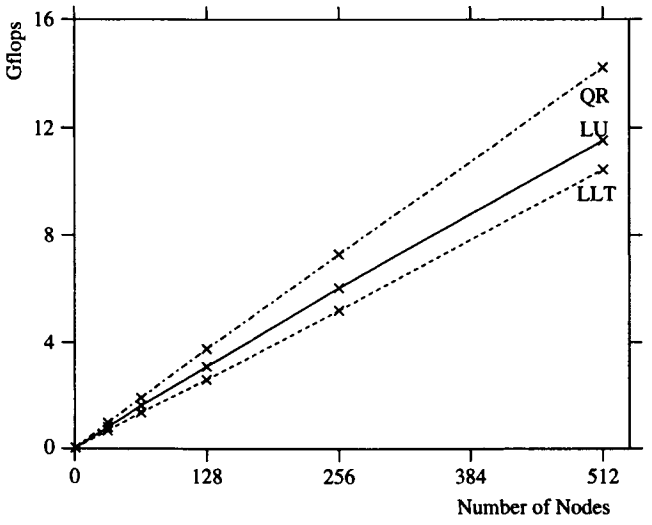


Figure 17. Scalability of factorization routines on the Intel Delta (9 Mbytes/node)

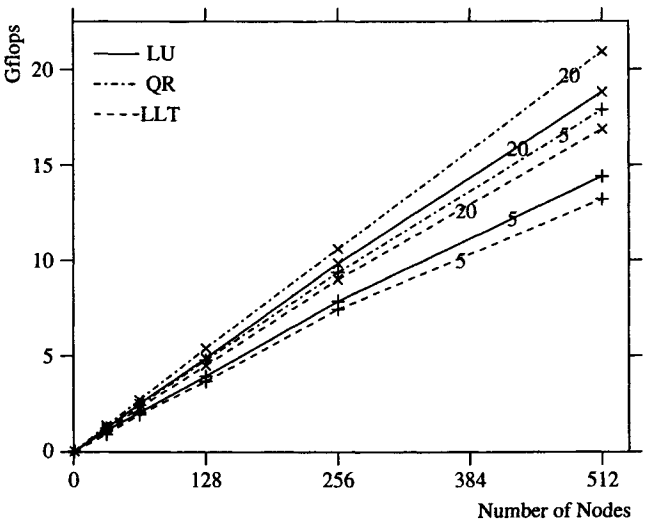


Figure 18. Scalability of factorization routines on the Intel Paragon (5, 20 Mbytes/node)

`netlib@ornl.gov`.

Acknowledgements

This research was performed in part using the Intel iPSC/860 hypercube and the Paragon computers at Oak Ridge National Laboratory, and in part using the Intel Touchstone Delta system operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to the Delta system was provided through the Center for Research on Parallel Computing.

REFERENCES

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A Portable Linear Algebra Library for High-Performance Computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.
2. E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM Press, Philadelphia, PA, 1992.
3. E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

4. J. Choi, J. J. Dongarra, R. Pozo, D. C. Sorensen, and D. W. Walker. CRPC Research into Linear Algebra Software for High Performance Computers. *International Journal of Supercomputing Applications*, 8(2):99–118, Summer 1994.
5. J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. In *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*. IEEE Computer Society Press, Los Alamitos, California, October 19-21, 1992.
6. J. Choi, J. J. Dongarra, and D. W. Walker. Level 3 BLAS for Distributed Memory Concurrent Computers. In *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France)*, pages 17–29. Elsevier Science Publishers, September 7-8, 1992.
7. J. Choi, J. J. Dongarra, and D. W. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. *Submitted to Concurency: Practise and Experience*, 1994. Also available on Oak Ridge National Laboratory Technical Reports, TM-12268, February, 1994.
8. J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurency: Practise and Experience*, 1994. Accepted for publication. Also available on Oak Ridge National Laboratory Technical Reports, TM-12252, August, 1993.
9. J. Choi, J. J. Dongarra, and D. W. Walker. ScaLAPACK Reference Manual: Parallel Factorization Routines (LU, QR, and Cholesky), and Parallel Reduction Routines (HRD, TRD, and BRD) (Version 1.0BETA). Technical Report TM-12471, Oak Ridge National Laboratory, February 1994.
10. J. Choi, J. J. Dongarra, and D. W. Walker. The Design of a Parallel, Dense Linear Algebra Software Library: Reduction to Hessenburg, Tridiagonal, and Bidiagonal Form. *Submitted to SIAM J. of Sci. Stat. Computing*, 1994.
11. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1979.
12. J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 18(1):1–17, 1990.
13. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1988.
14. J. J. Dongarra and S. Ostrouchov. LAPACK Block Factorization Algorithms on the Intel iPSC/860. LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee, October 1990.
15. A. Edelman. Large Dense Linear Algebra in 1993: The Parallel Computing Influence. *International Journal of Supercomputing Applications*, 7(2):113–128, Summer 1993.
16. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*. Springer-Verlag, Berlin, 1976. Vol.6 of Lecture Notes in Computer Science, 2nd Ed.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Software Tools for Parallel Computers and Workstation Clusters

Wolfgang Gentzsch, Uwe Block, and Friedrich Ferstl

GENIAS Software GmbH, Erzgebirgstr. 2, 93073 Neutraubling, Germany

Abstract: This contribution presents the experience of the authors on the use of software tools for developing and porting parallel codes to a variety of parallel platforms at GENIAS, such as the Intel iPSC/860, nCUBE/2, Parsytec Multicluster, and clusters of workstations. Software tools like xHPF, FORGE 90, EXPRESS, PVM/MPI and others are used to port seismic, fluid dynamics, grid generation, and other applications to the parallel machines. New benchmark results for xHPF and some comparisons to hand-tuning efforts are also given.

1. PARALLELIZING AND PORTING PROGRAMS

In parallel computing, two topics are strongly related to each other: Developing new parallel programs and porting existing parallel programs. Both topics are crucial for the further acceptance of parallel computing: Today, the number of commercially available application programs for parallel computers is still rather limited. This handicap might be overcome by speeding up the processes of creating parallel programs and subsequent porting of parallelized applications to different parallel computers, as is done new e.g. in the EUROPORT project supported by the CEC. With reduced time and cost for these processes, users and software vendors will increasingly be attracted by the promising raw performance of parallel computers. This goal can be achieved through human expertise and parallelization tools, and by effective interaction of both.

1.1. Parallelization of sequential programs

Most of the parallelization projects start with an existing sequential program. Even when intending to implement a parallel algorithm from scratch, writing first the corresponding sequential program for one processor is recommended, but with the parallel structure of algorithm and code already in mind. The computational kernel then should be tested, corrected, and optimized without the additional difficulties arising from the implementation on a parallel computer. Also, timing results and output data should be collected for later comparisons with the parallel version. This computational kernel then is a solid starting point for the parallel version.

At first, we always check an existing sequential program, especially "dusty decks", carefully for correctness, and improve the code quality where possible (e.g. dead code elimination). Anyhow, we need a deep understanding of the inner data and control structures of the program to make the right restructuring and parallelization decisions. Therefore, the parallelization process should consist of the following steps:

1. Check consistency of code
2. Understand control and data structures
3. Improve code quality
4. Gather runtime execution statistics
5. Optimize most time consuming parts
6. Choose appropriate data decomposition
7. Select loops or blocks etc. for parallelization
8. Modify the code, e.g. loops, according to decomposition
9. Insert communication statements into program code
10. Test parallel program on target machine
11. Profile parallel execution
12. Optimize/restructure parallel program.

Performing steps 1 to 5 should give the necessary insight into the structures of the program required for the parallelization steps 6 to 12.

1.2. FORGE: interactive program analysis and parallelization

FORGE is a very powerful interactive Fortran parallelization environment for global analysis and parallelization of Fortran programs from Applied Parallel Research, APR, [12], and distributed by GENIAS. FORGE/DMP is an add-on module for interactive parallelization for distributed memory MIMD architectures. The tools support the user in performing each of the steps of the parallelization process. FORGE SMP does the same for shared memory architectures.

1.2.1. Baseline FORGE Browser

The interactive parallelizers for distributed and shared memory systems are built upon the Fortran program browser, FORGE Baseline. This tool analyzes large, complex Fortran application programs for parallelization on both shared and distributed memory systems.

The Baseline Browser utilizes database capable of analyzing even the 'dusty deck" program. FORGE's database viewing tools provide facilities for fast reference tracing of variables and constants, consistency checking of COMMON blocks and subprogram calls, and exposing variable aliasing through COMMON and calls, as well as displaying COMMON block usage, data flow through calls, and data dependencies between routines and basic or arbitrary code blocks.

Baseline FORGE's interprocedural database provides the global view of a program that one needs before optimization. Additional facilities for program maintenance and reformatting, and an advanced instrumentation module and runtime library for gathering serial execution performance statistics are also included.

1.2.2. The Distributed Memory Parallelizer (DMP)

The DMP adds onto Baseline the Distributed Memory Parallelizer to spread loops and distribute data arrays for MIMD architectures interactively. The parallelized program is fully scalable, with calls to the parallel run- time library, interfacing any of the popular communication packages such as PVM, Express, and Linda, or native message passing systems.

With FORGE's SPMD (Single Program, Multiple Data) parallelization strategy, the same program runs on each processor while selected DO loops are rewritten to automatically distribute their iterations across the processors.

The first step in parallelizing a Fortran program is identifying the critical data arrays, proposing an array decomposition scheme, and then restructuring the program to decompose these arrays over the processors. FORGE DMP's Data Decomposition facility offers an interactive way to specify decompositions and select arrays for partitioning while viewing the implications of these decisions. The Data Decomposer implements either BLOCK or CYCLIC distributions along any single dimension, with either FULL or SHRUNK memory allocation. With full allocation, an array is allocated its original size on each processor. With shrunk allocation, each processor is allocated only enough memory for an array to hold the elements that it owns.

The next step is identifying which loops to parallelize. DMP's Loop Spreader allows interactive or automatic selection of loops. Under automatic selection, DMP uses actual runtime execution statistics to determine the best loops to parallelize to obtain higher parallelization granularity and reduced communication costs.

DMP checks for parallelization inhibitors, rewrites dimension declarations and array subscripts on distributed arrays to reflect the partitioning, modifies DO loop control counters to operate dynamically depending on the loop distribution scheme, and insures that all restructurings are consistent through subroutine calls. Data communication calls to the parallel runtime library are inserted automatically around and within distributed loops to move the data as it is needed. DMP's interactive displays allow fine tuning of the communications. The resulting parallel program is dynamically scalable at runtime. DMP can also be used to interactively view the parallelizations developed by the batch parallelizing pre-compilers dpf and xhpf.

1.2.3. Parallel Performance Profiler and Simulator

Programs parallelized by FORGE DMP can utilize the Parallel Profiler to gather runtime performance statistics of CPU utilization and communication costs. The Performance Simulator can be used to predict performance on various MPP systems or configurations. Performance instrumentation options generate parallelized programs with calls to the runtime timing library to accumulate data on each node for loop and subprogram execution times, communication costs, and program wait times. The post-processor polytime is provided to analyze the results over all nodes and produce a composite report of a program's true performance on the parallel system. By linking with the runtime simulation library, performance of a parallelized program running on a single node can be extrapolated to report CPU and communication performance on a variety of scalable MPP systems.

1.2.4. The Shared Memory Parallelizer (SMP)

Another add-on to Baseline FORGE is the Shared Memory Parallelizer. Unlike parallelizing compilers that fail to parallelize the most important DO loops in a program, SMP's interprocedural analysis can handle loops that call subroutines. SMP's strategy is to parallelize for high granularity by analyzing outermost loops first. It analyzes array and scalar dependencies across subprogram boundaries by tracing references through the database up and down the call tree. The result is a parallelized source code with compiler-specific directives inserted for scoping variables and for identifying Critical and Ordered regions of code.

DO loops are selected for parallelization interactively. Using execution performance timings as a guide, FORGE SMP will suggest the most significant loop as a starting point, working through the code from the highest CPU-intensive loops down to some threshold, below which parallelization does not produce a performance gain.

SMP's interprocedural analysis makes scoping of variables passed through subprogram calls and COMMON possible. In a parallel region of code, SMP analyzes all variable references within a loop, including those enclosed in routines called from the loop. Proceeding down the call chain, SMP identifies variables as PRIVATE or SHARED, and GLOBAL or LOCAL, displaying them interactively and allowing you to modify its decisions. SMP also identifies Critical and Ordered Regions in the code that will give rise to synchronization calls. On some systems these regions cannot be parallelized. These are also displayed interactively.

Following successful analysis of a loop nest, FORGE SMP inserts directives that are specific for the target system and compiler on which the program is to be run. SMP knows about a number of shared memory systems and generates the correct directives. And, the program's parallel analysis is saved and can be recalled again later to generate a parallel program for some other target system.

1.2.5. MAGIC

FORGE Magic/DM Parallelizer (dpf) for distributed memory is able to automatically partition data arrays and distribute loops based upon a static analysis of the source program. Or, one can supply a serial timing profile to direct the automatic parallelization right to the hot spots in the code. With FORGE Magic/SM (spf) for shared memory systems, data arrays are automatically padded and aligned for optimal cache management, and DO loops are parallelized by target system compiler-specific directives.

With Magic one gets a first rough sketch at a parallelization. This may be especially useful with large, unwieldly codes when, most likely, one would not have a clue as to where to begin. A parallelization report shows in great detail which loops parallelized and which data arrays were partitioned, and how this was done. More importantly, it shows which loops/arrays could not be parallelized and the inhibitors in the program that prevented this. An output option annotates the original Fortran 77 program with parallelization directives to refine the parallelization.

Also, Magic can be used to generate instrumented programs for serial runtime execution timing. FORGE Magic/DM (dpf) can also instrument the generated

parallelized code to produce parallel runtime performance profiles that identify communication bottlenecks and losses due to poor load balancing.

1.2.6. High Performance Fortran, HPF

The FORGE HPF batch pre-compiler, xhpf, with an optional MAGIC automatic parallelization mode as well, is able to take a serial Fortran 77 program and automatically generate a parallelized code with Fortran 90 array syntax and HPF directives. The HPF consistency checker assures that the parallelization directives are legal HPF and are consistent with themselves and the program. Also the parallelization is viewable from the interactive FORGE/DMP Parallelizer through is compatible database. And, if the target system does not yet have an HPF compiler, xhpf, like dpf, will generate a SPMD Fortran 77 code with explicit message passing calls interfacing to PVM, Express, Linda, etc.

Some advantages of these parallelization tools over other HPF compilers are:

- They generate efficient Fortran 77 code from the HPF that is immediately compilable and optimizable by most native f77 compilers.
- They parallelize Fortran DO loops as well as Fortran 90 array syntax. (HPF compilers only parallelize array syntax.)
- MAGIC on xhpf will generate an initial HPF parallelization automatically to start with.
- One can review and analyze the parallelization with the FORGE/DMP interactive tool.
- One can instrument the parallel code and obtain a parallel runtime performance profile that includes measurement of all communication costs and bottlenecks.
- With the parallel runtime library, one can interface to all major multiprocessor systems and workstation clusters running PVM, Express, Linda, IBM EUI, nCUBE, Intel NT, etc.
- They are available today.

FORGE Explorer, based upon the FORGE Baseline Browser, presents an easy to use and mostly intuitive approach to interprocedural program data and control flow analysis, tracing, and global context searching.

1.3. Portable parallel programs with EXPRESS

Today, some people deny the necessity of standards for parallel programs. They argue that the Message Passing principle for data communications has been widely accepted in the parallel computing community, and that the required calls to communication subroutines, e.g., "send", "receive", look very similar on different machines and often can be converted without great pain (cf. section 1.5.).

This might be true for distributed memory parallel computers, see Table 1. But, with a portable environment for parallel programs like EXPRESS, we can port a parallel application between the following three different types of parallel architectures without any conversions:

- Distributed memory parallel computers
- Shared memory parallel computers
- Networks of workstations.

Why are parallel programs for EXPRESS portable between these different hardware platforms? The reason is that EXPRESS employs a parallel programming model which is common to all platforms: The Message Passing paradigm.

EXPRESS is essentially a library of communication subroutines which is linked with standard FORTRAN and C programs and a set of tools like "ndb", a source level debugger, or performance monitors., see e.g. [1], [3], [11]. The library includes:

- subroutines for low-level Message Passing communication, e.g., exsend, exreceive,
- utilities for global data operations, e.g., global sum, broadcast,
- runtime domain decomposition tools,
- parallel graphics routines,
- parallel I/O routines.

Of course, the implementations of this communication library are dedicated to and optimized for the respective target platform, but the application programming interface looks always the same (cf. Table 1).

With EXPRESS, we usually write one program which is complemented by calls to EXPRESS' communication library. After compiling and linking, the program is loaded into each node of the parallel machine, and each node executes the program independently.

Parallelization of I/O is one of the most difficult and time-consuming tasks in parallel programming. I/O nearly always degrades the parallelism of a program and is usually the least portable part of a code. EXPRESS helps also in these issues. Actually, EXPRESS supports two parallel programming models: host-node and cubix. In the first model, all I/O is done through a host program and is communicated to the node programs. The conversion of an existing sequential program would require the splitting of the program into a host and a node part and heavy modifications to both. Cubix is a generic host program which handles the I/O requests from the node programs. Only a few additional subroutine calls must be inserted, and EXPRESS' pre-processor takes care of all necessary changes to the node program. Using cubix, we end up with one node program (and no host program) which is easily ported to other platforms despite the different hardware architectures.

1.4. PVM

Some tools with similar functionality as EXPRESS are available in the public domain, [4], [5], [13]. The probably most commonly used is called PVM (Parallel

Virtual Machine), with more than 10.000 users worldwide. PVM puts a strong emphasis on distributed heterogeneous network computing. Although the principal structure and usage of PVM is comparable to EXPRESS, there are, however, some considerable differences:

- There is no parallel I/O facility available at present with PVM. There is especially no generic host program like Cubix under EXPRESS. This greatly deteriorates the applicability of PVM for I/O-intensive programs.
- PVM to date still lacks a comprehensive development environment with tools for debugging and monitoring. In the last few months, however, the first release of the graphical monitoring environments xPVM is completed and work is ongoing.
- Although PVM of course supports the "Message Passing core" presented in the following paragraph, its set of operations is significantly smaller than that of EXPRESS.

1.5. Message passing and portability

Today, there are some ten Message Passing tools available. There are vendor specific Message Passing environments such as those from Intel, nCUBE, and Parsytec, and the vendor independent ones like EXPRESS (cf. section 1.3.), PVM (cf. section 1.4.), or P4. Although different in detail the referred tools share a common subset of Message Passing operations which can be grouped into nine categories:

Initialize Processes:
Allocation of processing units at program start.

Collapse Processes:
Disallocation of the processing units at program end.

Inquire Node Id:
Retrieving the local node number.

Inquire Number of Nodes:
Retrieving the number of compute units.

Synchronization:
Synchronize processes.

Post Messages:
The send mechanism.

Read Messages:
The receive mechanism.

Probe for Messages:
Searching the message queue for specific messages.

Global Operations:
Broadcast, global sum etc.

Table 1 shows the concerning user callable subroutines within each category for all of the above mentioned Message Passing programming environments.

Category	Tools					
	iPSC/860	nCUBE/2	EXPRESS	PVM	PICL	P4
Initialize Processes	getcube	rexec	kxinit	enroll, initiate	open0	p4init
Collapse Processes	relcube	(not needed)	kxexit	leave	close0	p4cleanup
Inquire Node Id	mynode	npid	kxpara	whoami	who0	p4myid
Inquire # of Nodes	numnodes	ncubesize	kxpara	(simulate)	who0	p4nslaves
Synchronization	gsync	nsync	kxsync	barrier	sync0	(simulate)
Post Messages	csend	nwrite	kxwrit	initsend, put[type], snd	send0	p4send
Read Messages	crecv	nread	kxread	get[type], rcv	recv0	p4recv
Probe for Messages	cprobe	ntest	kxtest	probe	probe0	p4probe
Global Operations	gdsum, ...	ndsumn, ...	kxcomb	(simulate)	gsum0, ...	p4globop

Table 1 Common subset of Message Passing operations

Building parallel applications on this subset will allow for a high level of portability between different Message Passing implementations. But even porting between different Message Passing implementations is not necessary, if one decides to program parallel applications on top of one of the portable Message Passing libraries (EXPRESS, PVM, ...), which have become very popular in the past few years. These libraries support a wide range of commonly used parallel computers including shared memory machines. As an extra, most of them provide the opportunity to develop and execute parallel applications on clustered workstation networks.

2. MODEL APPLICATIONS

To illustrate the state of the art concerning source code portability on distributed memory MIMD architectures, we present some model applications, which we ported between a considerable variety of parallel machines. The ported codes are 15 benchmark kernels representing the computational intensive part of real world applications. They have been developed in the PARANUSS project which has been funded by the German Ministry for Science and Technology [16]. Three of the kernels are: MIGRA, FIRE, and TOMCATV.

2.1. MIGRA Kernel, a seismic application

The first example is migration, a seismic application, [10]. In oil and gas exploration, reflection seismic surveying is used to gain a geologic model of the subsurface. Artificial seismic waves which have been reflected by geologic boundaries are measured at the surface. Unfortunately, the image obtained in this way can be heavily distorted, especially in geologically most interesting areas. In the migration process,

the subsurface is reconstructed from the measured and pre-processed wavefield by extrapolating the wavefield back in time and depth using the wave equation. Considering the huge amount of data collected through seismic experiments, this migration process is one of the most time-consuming steps in seismic processing.

The program MIGRA uses a finite difference scheme for the monochromatic wave equation to extrapolate the wavefield which has been Fourier transformed with respect to time, [9]. The monochromatic wave equation has the advantage that the frequencies are independent of each other which is favourable for parallelization.

2.2. FIRE Kernel, computational fluid dynamics

FIRE (Flow In Reciprocating Engines) is a fully interactive computer program system (from AVL in Graz, Austria, [2]) for solving problems on compressible turbulent fluid flow as found in the engineering environment. Three- or two-dimensional unsteady or steady simulations of flow and heat transfer within arbitrary complex geometries with moving or fixed boundaries are possible. It provides the engineer with the capability to analyze complex fluid flow problems without requiring previous expertise in computational fluid dynamics or computer programming.

This computer application package is widely distributed in Europe and in the U.S., e.g. in the automotive industry, and is a key program to be selected for parallel computers.

2.3. TOMCATV Kernel, grid generation

Another test case is the generation of computational meshes with the TOMCATV-code, [8], which is also part of the SPEC benchmark suite. For the numerical solution of systems of partial differential equations, like the Navier-Stokes equations, the computational domain is decomposed into many subdomains (finite volumes). On a uniform grid the equations are easily discretized. However, in the case of a body with arbitrary shape, the numerical treatment of the boundary conditions is greatly simplified by the use of a curvilinear or even irregular computational mesh system, which is conformable to the body surface, as well as to other boundaries, in such a way that the boundaries are mesh lines.

Among the many possible choices for the partial differential equations to be solved in the grid generation process most investigations to date have used sets of elliptic equations derived from Laplace's or Poisson's equations. The mesh generation code TOMCATV is a 200-line kernel based on these ideas. The principle is to solve, in a transformed space with an orthonormal coordinate system (u,v), two nonlinear Poisson equations for the coordinates x and y, respectively, in the physical space. The equations are discretized by a finite difference approximation and the resulting discrete systems of algebraic equations (in every iteration step linearized) are solved by a line SOR (successive overrelaxation) method.

3. RESULTS AND EXPERIENCES

Our model applications were parallelized and ported at GENIAS to

- the Intel iPSC/860,
- the nCUBE/2,
- the Parsytec Multicluster, and
- networks of SUN, SGI and RS/6000 ethernet connected workstations

using the portable environments EXPRESS and PVM, and the parallelization tool package FORGE for global analysis and parallelization.

3.1. Portable parallel applications on clustered workstations

In general, porting between present state parallel computers has proven to be easy resulting in sufficiently performing codes, regardless of porting by hand (e.g., using the common Message Passing subset from section 1.5.) or to the EXPRESS platform. The main drawback of porting by hand is, that you can not use special operations of one machine, which are absolutely incompatible with the related mechanisms of the other ones. This can be partly avoided by using EXPRESS, since EXPRESS takes care of the efficient representation of it's own kind of specialities on each supported machine. Using EXPRESS, however, introduces some small amount of overhead for nearly every kind of Message Passing operation due to the additional EXPRESS top level software layer.

The commonly used data communication hard- and software of networked workstations is not very well suited to the requirements of parallel processing. In particular, the set-up time for data tranfer operations is too high. Therefore, running parallel applications on networked workstations requires a more subtle analysis of the algorithm's communication structure. TOMCATV, and MIGRA use a regular decomposition of the internal representation of the computational domain. Thus, the communication scheme is well defined, which gives a good multiprocessor performance. Table 2 shows the observed runtime behaviour of TOMCATV (matrix size 512*512) running on a cluster of SUN Sparc2 workstations. Figure 1 depicts the related speed-up graph.

Workstations	Elapsed Time [sec]
1	808
2	409
4	206
8	111

Table 2 TOMCATV (512*512) on up to 8 SUN Sparc2 workstations

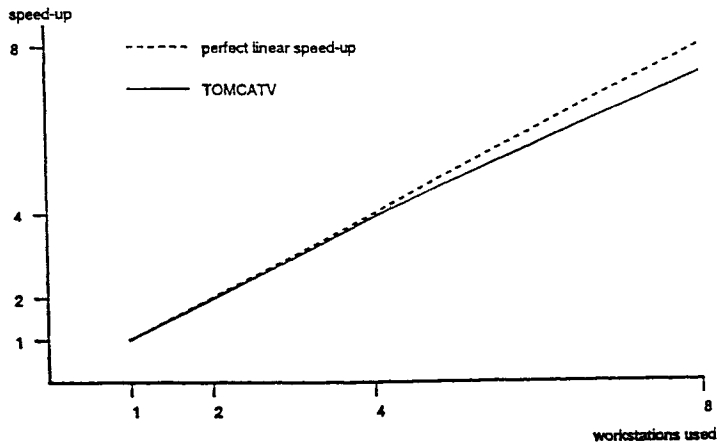


Figure 1 TOMCATV (512*512) speed-up behaviour

The parallel FIRE benchmark version, which we used first, incorporated a kind of data decomposition, which can be considered as equal to a random distribution of compute cells (finite volumes) to processing units. This scheme enforces a large amount of data to be exchanged and a large number of communication operations, too. Therefore, it does not perform sufficiently well on networked workstations.

Therefore, an improved version of the FIRE benchmark has been developed, utilizing a regular decomposition of the computational domain comparable to the schemes used with TOMCATV and MIGRA. Since the computations within FIRE (some thousands of floating point operations for each grid element) are much more expensive as compared to TOMCATV (approximately 50 FLOP per grid point), at least the same speed-up behaviour for FIRE can be predicted as was observed for TOMCATV.

Note, that the presented experiences hold true both for EXPRESS or PVM for parallel computation in workstation networks. Furthermore, the speed-up figure for MIGRA is comparable to the one given for TOMCATV and to those under usage of RS/6000 or SGI instead of SUN Sparc2 workstations. Therefore, we omit presenting further details.

Nodes	Elapsed Time [sec]
2	201
4	112
8	64

Table 3 FIRE benchmark on the iPSC/860

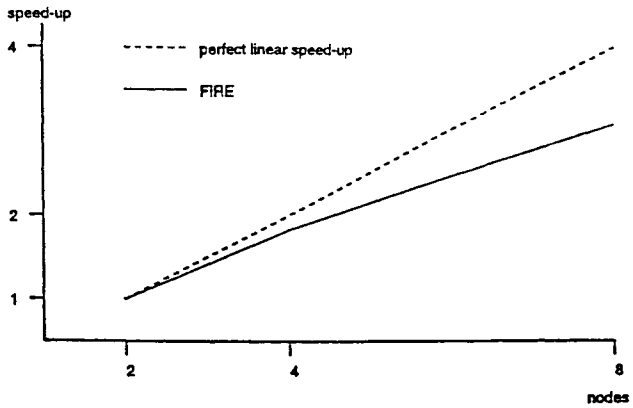


Figure 2 FIRE benchmark speed-up on iPSC/860 (figures are relative to the 2-node run)

3.2. Parallelization with FORGE

MIGRA has been parallelized in two different ways. One project was aimed at parallelization by hand and optimal implementation of MIGRA on the iPSC/860, while the other project focused on the use of FORGE for the parallelization of application programs.

The analysis of MIGRA with FORGE/DMP showed that MIGRA is nearly ideal suited for parallelization. One of the main loops which, by far, accounts for most of the computational load can easily be split up and spread over the processors. The subtasks can be executed nearly independently in parallel. Only the computation of a global sum and the storing of intermediate results require communication.

Table 4 shows that the parallel code generated by FORGE/DMP, although without any further optimizations, achieved a maximum speed-up of 3.7 on a 4-processor iPSC/860 for a problem size of 256 (256 frequencies, 256 depth levels, 256 x-steps). The overhead introduced by the calls to the run time library is less than 1%, as can be seen from runs of the sequential and parallel versions on one processor. I/O operations need a constant amount of time irrespective of the number of processors. This degrades the speed-up. We expect to see improved speed-up figures from optimizing the sequential and I/O parts of the code and from taking full advantage of FORGE's capabilities to tune communications interactively.

# Nodes	manually parallelized and optimized	parallelized with FORGE no optimizations applied
1	1	1
2	1.995	1.95
4	3.935	3.69

Table 4 Speed-Up for MIGRA on the iPSC/860, problem size 256.

The interactive parallelization with help of FORGE DMP took only about 3 hours, including sequential and parallel test runs to collect run time statistics. Without FORGE, much more time was needed to understand the control and data structures. Also, the modification of the program code and the insertion of communication statements by hand turned out to be time-consuming and caused some errors in the coding.

Several optimizations have been applied manually to the sequential and parallel parts of the code and the following effects have been observed:

- Hand-tuning in two places of the program code, i.e., reducing the number of complex divisions, reduced the sequential runtime by 22%.
- Optimization of I/O was crucial for reducing the elapsed time and for achieving high speed-ups. On the iPSC/860 we used
  - the built-in Concurrent File System (CFS),
  - unformatted instead of formatted I/O, and
  - high-speed synchronous I/O-routines for CFS files.
- Altogether, the optimized code performed 3-4 times better than the unoptimized.

With these optimizations we observed a speed-up of over 3.9 on a 4-processor iPSC/860 (cf. Table 4), resulting in an efficiency of 98.4%. Because the algorithm contains no inherent bottlenecks growing with the number of processors, we expect a speed-up of over 100 on a 128-processor iPSC, for a more realistic problem size of 1024.

Other finite difference migration methods have structures which are similar to TOMCATV, requiring much more communication than MIGRA. The results for TOMCATV shown above indicate that for those migration methods, speed-ups might be achieved which are comparable to MIGRA for realistic problem sizes. The geophysical industry already has recognized the benefits of parallel computing and is using parallel computers for commercial seismic processing.

The comparison of both approaches to parallelization shows that the use of FORGE DMP gives good results for a moderate number of processors at very reasonable cost (time). The results obtained for the iPSC easily can be transferred to other target machines. However, achieving optimal performance on a certain computer system always requires fine-tuning program codes and/or using machine-specific features which means a substantial investment and the loss of portability.

4. THE HPF BENCHMARK SUITE

High Performance Fortran (HPF) is an extension to Fortran that utilizes compiler directives for specifying the distribution of data across a distributed memory parallel system. The following results were obtained using Applied Parallel Research' s (APR) High Performance Fortran compilation system, xHPF, to parallelize a set of benchmark codes on the Intel PARAGON, the IBM SP1 and the CRAY T3D, [14], [15]. Although HPF is intended as extensions to Fortran 90, APR's xHPF implementation supports HPF's data distribution directives over either Fortran 90 or

Fortran 77. (E.g. xHPF will parallelize both F90 array syntax and F77 DO loops.) The suite of benchmarks includes both Fortran 90 and Fortran 77 based HPF programs.

Built into the xHPF system is an automatic parallelization capability that can be used very effectively to create an initial parallelization strategy for a complex application. Semi-automatic and explicit parallelization methods are also possible with xHPF whereby directives placed in the source code are used to derive or completely specify the parallelization strategy.

It should be noted that the process of porting and parallelizing of the benchmark codes described in this paper all began with the original serial code. In all cases the time to complete the parallelization of a code was less than a few hours, and in some cases the parallelization was done using xHPF's automatic capabilities. The benchmark suite available via anonymous FTP contains the original programs with comment line directives representing the parallelization strategy.

The xHPF system is a sophistication parallelization facility for distributed memory parallel systems. Using the HPF approach allows the user to maintain a portable Fortran program which can be converted for parallel execution on all of the popular MPP systems as well as clusters of workstations. The performance one can obtain using the system is dependent upon the communication parameters of the target system. Even high latency, low bandwidth systems will perform well if the granularity of the parallel work is large enough to overcome the overhead of performing the communication.

4.1. General description of the HPF benchmarks

As reference value the timing of the original program is used without any message passing on a single processor. The column labelled 1 in Tables 5, 6, and 7 is the parallel version executed on a single processor. In many cases this version of the program ran faster due to the way xhpf linearizes the subscripts. Some compilers perfer the re-write and in some cases cache utilization may be better.

No optimization was performed on these benchmarks. In a few cases when the results looked very poor, parallel statistics were obtained and a few additional directives were used to direct xhpf 's parallelization.

Timings on the IBM SP1 fluctuated quite a bit. It seemed like other users on the system could effect the timings by factors of two to three. In addition, the processor cache is another reason for this. The SP1 timings in the table represent the minimum time obtained over a series of runs.

The poor timings on two processors on the SP1 is repeatable and of unknown origin. We expect some detrimental effect of cache utilization.

Some of the benchmarks have a very poor communication-to-granularity ratio. These will not parallelize well under even the best conditions.

All of the programs contain HPF data distribution directives supplemented by APR directives in areas that HPF directives couldn't cover the situation.

The entire program was timed, not just the important loops

4.2. Description of the HPF benchmark programs

Shallow_77

This is a Fortran 77 version of the shallow water benchmarks from NCAR. It is a two dimensional finite difference algorithm on a grid that is 512x512. In the parallelization second of the dimensions is distributed across the processors. This algorithm requires nearest neighbor communication and as more processors are used the calculation performed on each processor is not sufficient to hide the overhead of the communication.

Shallow_90

This is a conversion of the Fortran 77 program to Fortran 90. The differences in timings between the Fortran 77 and the Fortran 90 version are due to inefficiencies in the conversion from Fortran 90 to Fortran 77 with message passing for running on the target system.

Chem3d

This is a three dimensional model of the middle atmosphere. The dimensions of the computational grid was 32x19x24 and the parallelization was performed on the second dimension of 19. Unfortunately this experiences very bad load balancing beyond 4 processors.

Embar

This is one of the NASA benchmarks. In our version, no data is distributed. The calculation consists of performing global summations across the processors.

Appsp

Another NASA benchmark. This one requires a transpose of many of the major work arrays. That overhead of the transpose degrades the speedup since the granularity of the computation is not large enough to cover the transpose.

Baro

This is one of the original Raul Mendez benchmarks that was re-written to allow for parallelization of the outer DO loop for a shared memory system. That re-write allowed the program to be distributed across the parallel system. In this program the major computational routine COMP contains DO loops of sufficient granularity. Several miscellaneous routines with small granularity and significant communication caused a degradation in the performance.

X42

This is an explicit modeling system using fourth order differencing. It demonstrates a super-linear speedup due to the effective utilization of cache. In fact, on 32 processors all of the major work arrays fix in the cache of the 32 processors on the IBM SP1.

grid

This is a simple test case with very little calculation. One should make the grid larger than the current 100x100 in order to get better results. The table says that max= 500; however, max really equals 100 in the input.

ora

One of the SPEC benchmarks. This one is similar to the embar benchmarks from the NASA benchmark suite. No data is distributed and global sums are performed across the processors.

swm256

This benchmark is very similar to the shallow_77 benchmark; however, the loops are written in such a way that much more communication is required.

tomcatv

This SPEC benchmarks is simply a grid generation program.

scalgam

This is a monte carlo transport toy program from LANL. Once again no data is distributed and only global summations performed across the processors in order to tabulate the contributions from the particles being traced.

Program	Size	From/Lines	Language
Shallow_77	512 ITMAX=50	NCAR 660	Fortran 77
Shallow_90	512 ITMAX=50	NCAR/APR 501	Fortran 90
Chem3d	32x19x24	NCAR 3272	Fortran 77
Embar	M=20	NASA 135	Fortran 77
Appsp	Nitmax=2 NX=32	NASA 4634	Fortran 77
Baro	IT=20	Mendez 927	Fortran 77
X42	DIM=512	 345	Fortran 77
grid	max=500	APR 114	Fortran 77
ora	152000	Spec 460	Fortran 77
swm256	ITMAX=200	Spec 597	Fortran 77
tomcatv	257	Spec 199	Fortran 77
Scalgam	12800 Particles	Lanl 840	Fortran 77

Table 5: Characteristics of the HPF benchmarks

	Number of Processors					
Program	1	2	4	8	16	32
Shallow_77	1.00	1.97	3.87	7.53	12.74	21.59
Shallow_90	1.00	1.87	3.59	7.02	12.42	19.50
Chem3d	1.37	2.23	4.14	2.45		
Embar	1.02	1.91	3.80	7.71	20.45	28.23
Appsp	0.96	1.77	3.23	5.49	7.02	
Baro	0.97	1.78	3.04	4.28	6.36	3.83
X42	0.99	1.90	3.91	8.43	23.27	
grid	1.00	1.99	3.80	7.10	12.35	17.22
ora	1.00	1.99	3.99	7.92	15.63	
swm256	1.01	1.67	2.53	3.08	3.93	
tomcatv	1.00	1.88	3.23	5.16	6.82	
Scalgam	0.52	1.99	3.89	5.15	14.10	

Table 6: Benchmarking results with xHPF, results expressed as ratio of wallclock seconds of original to wallclock seconds of parallel run, on an IBM SP1.

	Number of Processors					
Program	1	2	4	8	16	32
Shallow_77	1.12	2.15	4.22	7.86	15.04	26.60
Shallow_90	1.00	1.95	3.86	7.47	13.44	21.00
Chem3d	0.93	1.41	2.14	2.29	2.29	1.62
Embar	0.99	1.94	3.77	6.84	10.68	15.96
Appsp	0.84	1.18	1.94	3.09	4.37	4.89
Baro	0.85	1.36	2.51	4.35	6.44	8.53
X42	1.72	2.78	5.34	10.11	18.35	32.11
grid	0.96	1.89	3.65	6.74	11.44	15.10
ora	1.00	2.00	3.99	7.91	15.49	29.85
swm256	0.99	1.76	2.97	4.55	5.89	6.97
tomcatv	1.01	1.78	3.12	5.25	7.65	9.58
Scalgam	1.00	1.97	3.78	7.10	11.41	16.62

Table 7: Benchmarking results with xHPF, results expressed as ratio of wallclock seconds of original to wallclock seconds of parallel run, on an Intel Paragon.

	Number of Processors					
Program	1	2	4	8	16	32
Shallow_77	SIZE	SIZE	SIZE	1.00	1.66	2.43
Shallow_90	SIZE	SIZE	SIZE	1.00	1.54	2.15
Embar	0.81	1.62	3.23	6.37	12.35	23.29
Appsp	SIZE	SIZE	SIZE	1.00	1.60	2.12
Baro	0.70	1.12	1.84	2.61	3.15	3.21
X42	SIZE	SIZE	1.00	1.86	3.15	4.89
grid	1.12	2.22	4.30	8.31	15.54	26.02
ora	1.01	2.01	4.02	8.04	16.08	31.84
swm256	1.46	2.21	3.43	4.58	5.20	4.93
tomcatv	1.40	2.41	3.93	4.27	7.33	8.47
Scalgam	0.88	1.83	3.57	6.78	12.11	19.94

Table 8: Benchmarking results with xHPF, results expressed as ratio of wallclock seconds of original to wallclock seconds of parallel run, on a Cray T3D.

Note: Due to the amount of memory available on each processor of the T3D, some programs could not be run on less than 8 processors. In these cases, the performance on 8 processors is taken as 1.00. (This situation is indicated by "SIZE")

5. CONCLUSION

Summarizing the results, one can state, that portability is no longer an issue for a more widespread use of parallel computers because of the portable platforms like EXPRESS and PVM/MPI. For well structured algorithms an alternative to dedicated parallel computers are networks of workstations. They are especially well suited as development environment for portable parallel programs, and as a powerful production environment during night and weekend, as long as one uses a moderate number of workstations, say up to 16. Our experiences with xHPF and FORGE/DMP shows that the use of these tools enormously improves productivity and quality in developing, optimizing, parallelizing, porting, and documenting FORTRAN applications.

6. REFERENCES

1. J. Ahmad, Min-You Wu: EXPRESS versus iPSC/2 primitives: a performance comparison. Report CRPC - TR 91147, Rice University, April 1992.
2. G. Bachler, W. Brandstätter, A. Ennemoser, K. Pachler. H. Steffan, K. Wieser: FIRE, main program description. AVL List GmbH, 1990.

3. L. Beernaert, D. Roose, W. Verhoeven: Parallel CFD on a Meiko Transputer system with EXPRESS in comparison to iPSC systems. Report TW 152, KU Leuven, May 1991.
4. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam: A users' guide to PVM Parallel Virtual Machine. Report ORNL/TM-11826, Oak Ridge Nat. Lab., July 1991.
5. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam: Graphical development tools for network-based concurrent supercomputing. Proceedings Supercomputing '91, Albuquerque 1991.
6. G. C. Fox: Achievements and prospects for parallel computing. Report CRPC - TR 90083, NPAC Syracuse University, Jan. 1991.
7. W. Gentzsch: Vectorization of computer programs with applications to computational fluid dynamics. Vieweg 1984.
8. W. Gentzsch, K. W. Neves, H. Yoshihara: Computational fluid dynamics: algorithms and supercomputers. NATO AGARDograph No. 311, 1988.
9. W. Gentzsch: Vectorization and parallelization techniques for supercomputers. In: K. Fujii (Ed.), Supercomputers and their performance in computational fluid dynamics. Vieweg 1991.
10. E. Kjartansson: Modeling and migration with the monochromatic wave equation - variable velocity and attenuation. Stanford Exploration Project, Nr. 15, 1979.
11. A. Kolawa: The EXPRESS way to distributed processing. Supercomputing Review, May 1991.
12. J. Levesque: FORGE/MIMDizer: Parallel programming environment for distributed memory MIMD machines, Applied Parallel Research 1992.
13. V. S. Sunderam: PVM, a framework for parallel distributed computing. Concurrency, Practice and Experience. Vol 2 (4), 315-339, 1990.
14. J. Levesque: Using High Performance Fortran on the Intel Paragon, Intel User's Group Meeting 1994.
15. J. Levesque: xHPF Benchmark results, APR 1994.
16. U. Block, F. Ferstl, W. Gentzsch: PARANUSS Report 01 IR 203 A 0, GENIAS 1994.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Mentat – Applying the Object-Oriented Paradigm to Parallel Processing*

A. S. Grimshaw
Department of Computer Science, University of Virginia,
Charlottesville, VA

1. Background

Mentat is an object-oriented parallel processing system designed to address three problems that face the parallel computing community, the difficulty of writing parallel programs, the difficulty achieving portability of those programs, and the difficulty exploiting contemporary heterogeneous environments. Writing parallel programs by hand is more difficult than writing sequential programs. The programmer must manage communication, synchronization, and scheduling of tens to thousands of independent processes. The burden of *correctly* managing the environment often overwhelms programmers, and requires a considerable investment of time and energy. If parallel computing is to become mainstream it must be made easier for the average programmer. Otherwise, parallel computing will remain relegated to specialized applications of high value where the human investment required to parallelize the application can be justified.

A second problem is that once a code has been implemented on a particular MIMD architecture, it is often not readily portable to other platforms; the tools, techniques, and library facilities used to parallelize the application may be specific to a particular platform. Thus, considerable effort must be re-invested to port the application to a new architecture. Given the plethora of new architectures and the rapid obsolescence of existing architectures, this represents a continuing time investment. One can view the different platforms as one dimension of a two dimensional space, where the other dimension is time. One would like the implementation to be able to cover a large area in this space in order to amortize the development costs.

Finally there is heterogeneity. Today's high performance computation environments have a great deal of heterogeneity. Many users have a wide variety of resources available, traditional vector supercomputers, parallel supercomputers, and different high performance workstations. The machines may be connected together with a high speed local connection such as FDDI, ATM, or HIPPI, or they may be geographically distributed. Taken together these machines represent a tremendous aggregate computation resource. There are currently no software systems available that permit the user to treat such a collection as a single high performance computer that manages the heterogeneous resources for the user, and allows applications to execute across heterogeneous hardware platforms.

Mentat was originally designed to address the first two of these issues, implementation difficulty and portability. The primary Mentat design objectives were to provide 1) easy-to-use parallelism, 2) high performance via parallel execution, 3) system scalability from tens to hundreds of processors, and 4) applications portability across a wide range of platforms. The premise underlying Mentat is that writing programs for parallel machines does not have to be hard. Instead, it is the lack of appropriate abstractions that has kept parallel architectures diffi-

*This work was partially supported by NSF grant ASC-9201822, NASA NGT-50970, and NLM grant LM04969.

cult to program, and hence, inaccessible to mainstream, production system programmers. The third issue, heterogeneity, is the focus of the Legion project which is an outgrowth of Mentat.

To date Mentat has been ported to a variety of platforms that span the latency and bandwidth spectrum; from heterogeneous workstation networks, to distributed memory MPP's, to shared memory machines. Workstations include generations of Sun workstations, IBM RS 6000's, Hewlett Packard's, and Silicon Graphics workstations. MPP's platforms have included the BBN Butterfly, the Intel iPSC/2, iPSC/860, and Paragon, and the IBM SP-1 and SP-2. For the most part we have had applications source code portability between platforms. The only exception being the iPSC/860.[1] Applications performance has been good as well, though not always good. Mentat is not appropriate for fine-grain applications that require either very frequent communication, or large volumes of communication. This is not surprising, as even hand-coded parallel programs often have difficulty with such applications.

The objective of this chapter is to provide the reader with a solid introduction to Mentat and to provide intuition as to the performance that can be expected from Mentat applications. This will be accomplished by first examining the Mentat philosophy to parallel computing and reviewing the Mentat programming language basics. We will then move onto applications performance. For each of five applications we will address two questions. 1) What is the shape of the Mentat solution? 2) How did the implementation perform? For more information please see [14,15,17, 18] or visit our WWW page at http://uvacs.cs.virginia.edu/~mentat/. The web page has pointers to examples, technical reports, and the Mentat binaries.

2. Related Work

There are many other projects with objectives similar to Mentat's. One important dimension along which solutions to the parallel software problem can be placed is the level of programmer awareness and control of the parallel environment. Solutions can be placed along a spectrum that ranges from fully automatic solutions to completely explicit solutions. In fully automatic the programmer is completely freed from the responsibility of managing any aspect of the parallel environment. A parallelizing compiler, often acting in concert with a sophisticated run-time system, finds and *safely* exploits opportunities for parallelism in the application. The user is not responsible for threads, synchronization, or other details. At the other extreme are the explicit, manual systems where the programer writes programs in the assembly language of parallelism. The programmer must decompose the problem, distribute data structures, manage scheduling and communication, and *safely* manage synchronization between tasks.

Both automatic techniques and explicit techniques have advantages and disadvantages. The advantages of automatic techniques are that the parallel programs are deterministic and semantically equivalent to the sequential program. Compilers are very good at finding large amounts of parallelism in programs, often down to the statement level. Further, automatic techniques can be applied to "dusty deck" programs, usually Fortran programs, to leverage the huge existing software base. A disadvantage of automatic techniques is that often the granularity of the detected parallelism is too small to efficiently exploit on contemporary distributed memory machines. To the compiler, the program is a large directed graph. It must attempt to partition and schedule the graph onto the parallel computer. This classic scheduling problem is NP-complete. The compiler is not exploiting any of the programmer's domain knowledge as it

1. The iPSC/860 only allows one process per processor. Many applications require more than one object per processor.

attempts to partition the problem. Finally, automatic techniques are often applied to dusty decks. When this is the case they are often defeated by spurious dependencies in the program that require programmer intervention. Despite these difficulties there have been significant advances in automatic solutions [26,30, 32].

Explicit approaches require the programmer to manage all of the details of parallel programming. This is an advantage in that the programmer has total control of the program and can tune the program to the particular machine. Another advantage is that explicit techniques are an easy add-on to existing programming languages. They are often implemented as libraries, e.g., send and receive functions in a library, or shared memory and semaphores in a library. The downside of explicit approaches is that the programmer has total control of the program; including the opportunity to make synchronization errors leading to non-deterministic deadlock or non-deterministic program behavior, and the opportunity to do a very poor job of decomposing and scheduling the problem. Examples of explicit systems include [2,5,38].

There are systems that operate in the middle of the spectrum rather than at the extremes. By doing so they attempt to capture some of the benefits of both explicit and implicit approaches. Mentat, Linda [7], Fortran D [13], HPF Fortran [28], and Dataparallel C [33] are just a few of these systems.

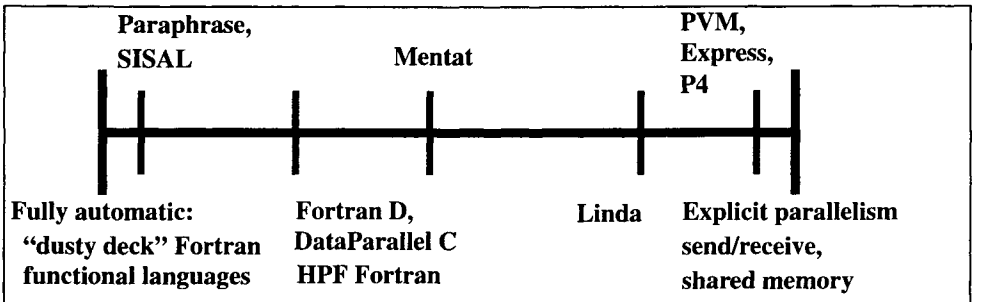


Figure 1. The level of programmer control and intervention varies significantly between different solutions to problem of writing parallel software.

Another dimension of the solution space is the programming language style, e.g., object-oriented versus functional. There are a variety of object-oriented parallel processing systems. Examples include Charm++ [22],CC++ [8], pC++ [4], ESP [35], and Presto [3]. Mentat differs from systems such as [2,3] (shared memory object-based systems) in its ability to easily execute on both shared memory MIMD and distributed memory MIMD architectures, as well as hybrids. pC++ [4] and Paragon [10] on the other hand are data-parallel derivatives of C++. Mentat accommodates both functional and data-parallelism, often within the same program. Mentat differs from other distributed object based systems and languages [9] in our objectives, we strive for performance via parallelism rather than distributed execution.

Applications portability across parallel architectures is an objective of many projects. Examples include PVM [38], Linda [7], the Argonne P4 macros [5], and Fortran D [13]. Our effort shares with these and other projects the basic idea of providing a portable virtual machine to the programmer. The primary difference is the level of the abstraction. Low-level abstractions such as in [5,7,38] require the programmer to operate at the assembly language level of parallelism. This makes writing parallel programs more difficult. Others [13,27,28,33] share our philosophy of providing a higher level interface in order to simplify applications development. What differentiates our work from other high-level portable systems is that we support both functional and data-parallelism as well as support the object-oriented paradigm.

3. Mentat

The Mentat philosophy on parallel computing is guided by two observations. The first is that the programmer understands the application domain and can often make better data and computation decomposition decisions than can compilers. The truth of this is evidenced by the fact that most successful applications have been hand-coded using low-level primitives. The second observation is that management of tens to thousands of asynchronous tasks, where timing-dependent errors are easy to make, is beyond the capacity of most programmers. Compilers, on the other hand, are very good at ensuring that events happen in the right order and can more readily and correctly manage communication and synchronization than programmers. The Mentat solution is driven by those two observations, that there are some things people do better than compilers, and that there are some things that compilers do better than people. Rather than have either do the complete job, we exploit the comparative advantages of each.

Mentat has also been heavily influenced by pioneering work in object-based operating systems [21,41] and by the object-oriented revolution [39]. The object-oriented paradigm has proven to be a powerful software engineering tool for sequential software. Attributes such as programming in the large, encapsulation, polymorphism, fault containment, and software reuse have all made the task of constructing complex sequential software more tractable. The belief that these same attributes can help manage the complexity of parallel software led to an early decision to adopt an object-oriented approach and to provide parallel processing support for object-oriented languages rather than Fortran.

Mentat exploits the object-oriented paradigm to provide high-level abstractions that mask the complex aspects of parallel programming, communication, synchronization, and scheduling from the programmer. Instead of managing these details, the programmer is free to concentrate on the details of the application.

There are two primary components of Mentat: the Mentat Programming Language (MPL) [16] and the Mentat run-time system [18]. The MPL is an object-oriented programming language based on C++. The programmer uses application domain knowledge to specify those object classes, called Mentat classes, that are of sufficient computational complexity to warrant parallel execution. The granule of computation is the Mentat class member function. Mentat classes consist of contained objects (local and member variables), their procedures, and a thread of control.

Mentat extends object implementation and data encapsulation to include parallelism encapsulation. Parallelism encapsulation takes two forms that we call *intra-object* and *inter-object* encapsulation. Intra-object encapsulation of parallelism means that callers of a Mentat object member function are unaware of whether the implementation of a member function is sequential or parallel. Inter-object encapsulation of parallelism means that programmers of code fragments (e.g., a Mentat object member function) need not concern themselves with the parallel execution opportunities between the different Mentat object member functions they invoke. Thus, the data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization are automatically detected and managed by the compiler and run-time system without further programmer intervention.

Mentat classes are denoted by the inclusion of the keyword "mentat" in the class definition, as in the mentat class `sw_worker` shown below. The keyword `mentat` tells the compiler that the member functions of the class are worth executing in parallel. Mentat classes may be defined as either `persistent` or `regular`. Instances of regular Mentat classes are logi-

```
regular mentat class sw_worker {
// private data and function members
public:
result_list*compare(sequence*,libstruct*,paramstruct);
};
```

Figure 2. A regular mentat class definition. The keyword "mentat" tells the compiler to treat instances of this class differently. The "regular" modifier indicates that instances of this class are stateless, i.e., they are pure functions.

cally stateless, thus the implementation may create a new instance to handle every member function invocation. `Persistent` Mentat classes maintain state information between member function invocations. This is an advantage for operations that require large amounts of data, or that require persistent semantics.

A Mentat object is an instance of a Mentat class, and possesses a name, a thread of control, and an address space. Because Mentat objects each have their own address space they are address space disjoint. Therefore, *all* communication between Mentat objects and between Mentat objects and main programs is via member function invocation.

The set of Mentat objects visible to a Mentat object defines that objects *object subspace*. Similarly the transitive closure of the names visible to a Mentat program is the programs object subspace. Two or more Mentat programs may have intersecting object spaces. Interactions between the shared subspaces are unconstrained, i.e., no consistency or serialization properties are enforced by the language.

MPL also provides mechanism to allow the programmer to direct scheduling decisions, and an ADA-like select/accept. These and other features are addressed in the programming language manual [16], available via our World Wide Web page at *http://uvacs.cs.virginia.edu/~mentat/*. Details of the run-time implementation appear in [18]. We will confine the remainder of our discussion to member function invocation semantics and the execution model.

Member function invocation on Mentat objects is syntactically the same as for C++ objects. Semantically there are important differences. First, Mentat member function invocations are non-blocking, providing for the parallel execution of member functions when data dependencies permit. Second, each invocation of a regular mentat object member function causes the instantiation of a new object to service the request. This, combined with non-blocking invocation, means that many instances of a regular class member function can be executing concurrently. Finally, Mentat member functions are always call-by-value because the model does not provide shared memory. All parameters are physically copied to the destination object. Similarly, return values are by-value. Pointers and references may be used as formal parameters and as results. The effect is that the memory object to which the pointer points is copied. Variable size arguments are supported as well, as they facilitate the writing of library classes such as matrix algebra classes.

Instances of Mentat classes are used exactly like C++ classes. In general the execution of a code fragment proceeds normally, i.e., sequentially, until a statement is reached which contains a Mentat expression, e.g. line 4, Figure 3. At this point all arguments are marshalled and a program graph node is generated which contains the name of the object and the member function invoked. (Mentat programs are represented at run-time by program graphs. Program execution is driven by the arrival of data on the arcs. See [18], available via the web, for more details.) The variable on the left hand side, *i* on line 4 of Figure 3, is marked as delayed. When a delayed value is used as a parameter to a Mentat member function, e.g., *i* on line 9, an arc is constructed from the graph node generating *i* to the consumer. Execution proceeds without

blocking until either a delayed value is needed or a value is returned using rtf(). At that point

```
1:  // A,B,C,D,E are Mentat objects
2:  // y is a previously defined integer
3:  int i,j,k,m;
4:  i=A.op1(4.5); // execution does not block
5:  if (y > 5)
6:     j=B.op1(y)
7:  else
8:     j=C.op1(2.3);
9:  k=D.op2(i,j);
10: m=E.op1(j);
11: rtf(m);
```

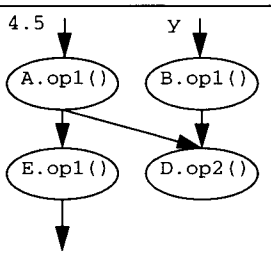


Figure 3. Code fragment illustrating the execution model. When executed with a value of y>5 the fragment on the left constructs the graph on the right. The arguments 4.5 and y are marshalled and passed to the object member functions A.op1() and B.op1() respectively. The rtf() is analagous to a C return. The final rtf(m) has the effect of starting the subgraph on the right and arranging for the results to be forwarded directly to where they are needed.

the program graph is executed in parallel and return values are sent where they are needed to satisfy data dependencies. Note that program graph construction, argument marshalling, and graph execution are the responsibility of the MPL compiler and Mentat run-time systems. Programmers never see the program graphs, much as most programmers never see the assembly code generated for sequential programs.

In summary, we believe that by splitting the responsibility between the compiler and the programmer, we can exploit the strengths and avoid the weaknesses of each. The assumption is that the programmer can make better decisions regarding granularity and partitioning, while the compiler can better manage synchronization. This simplifies the task of writing parallel programs and makes parallel architectures more accessible to non-computer scientists.

4. Applications Experience

The bottom line for parallel processing systems is performance on real applications. Our goals include both performance and ease-of-use. Over the past three years we have tried to answer three questions about Mentat and our approach to object-oriented parallel processing.

- Is MPL easy to use?
- Is the performance acceptable to users?
- What is the performance penalty (if any) with respect to hand-coded implementations?

To answer these questions we set out to implement a set of both real and synthetic applications. Our application selection criteria was that the application must be representative of a class of applications, and that the application must be of genuine interest to identifiable users. Further, we wanted variety in our applications, not just linear algebra applications. In all cases the implementation processes has been carried out in collaboration with domain scientists (e.g., biochemists) who had an interest in the codes. For each application we are interested in the level of difficulty in implementing the code, and in the resulting performance. For some of the applications there already existed a hand-coded parallel C or Fortran implementation of the application. In those cases we compare to the hand-coded version.

The set of Mentat applications is very diverse and includes matrix algebra libraries for dense linear systems, DNA and protein sequence comparison (biochemistry), automatic test pattern generation (electrical engineering), genetic algorithms (searching a combinatorial space), image processing (both libraries and for target recognition), command and control, and

parallel databases (computer science) to name a few. Only a few of Mentat application have been implemented by our research group at the University of Virginia. Below results for five of those applications are presented. The five applications are DNA and protein sequence comparison, the dense linear algebra libraries, the stencil libraries, ATPG (automatic test pattern generation), and a 2D electromagnetic finite element method. For each of the four applications (or libraries that are used in several applications) we will address two questions. 1) What is the shape of the Mentat solution? 2) How did the implementation perform?

4.1 DNA and Protein Sequence Comparison

Our first application is DNA and protein sequence comparison. With the advances in DNA cloning and sequencing technologies, biologists today can determine the sequence of amino acids that make up a protein more easily than they can determine its three-dimensional structure, and hence its function. The current technique used for determining the structure of new proteins is to compare their sequences with those of known proteins. DNA and protein sequence comparison involves comparing a single query sequence against a library of sequences to determine its relationship to known proteins or DNA. For the computer scientist, the basic problem is simple: DNA and protein sequences can be represented as strings of characters that must be compared. Biologists want to know the degree of relationship of two sequences. Two given sequences are compared and, using one of several algorithms, a score is generated reflecting commonality. Three popular algorithms are Smith-Waterman [36], FASTA [31], and Blast [1]. The latter two algorithms are heuristics; the quality of the score is traded for speed. Smith-Waterman is the benchmark algorithm, generating the most reliable scores although at considerable time expense. FASTA is less accurate but is twenty to fifty times faster than Smith-Waterman.

An important attribute of the comparison algorithms is that all comparisons are independent of one another and, if many sequences are to be compared, they can be compared in any order. This natural data-parallelism is easy to exploit and results in very little overhead.

A common operation is to compare a single sequence against an entire database of sequences. This is the *scanlib* problem. In *scanlib*, a source sequence is compared against each target sequence in the sequence library. A sorted list of scores is generated and the sequence names of the top n, usually 20, sequences and a score histogram are generated for the user.

A second common operation is to compare two sequence libraries, i.e., to compare very sequence in a source library against every sequence in the target library. For each sequence in the source library statistics are generated on how the sequence compares to the target library as a whole. This is known as the *complib* problem.

The Mentat implementation of *scanlib* uses `regular` Mentat class workers to perform the comparisons. The class definition for the Smith-Waterman worker is given in Figure 4.The private member variables have been omitted for clarity. The single member function, `compare()`, takes three parameters: `sequence`, the source sequence to compare, `libstruct`, the structure containing information defining a subrange of the target library, and `paramstruct`, a parameter structure containing algorithm-specific initialization information. The member function `compare()` compares the source sequence against every sequence in its library subrange and returns a list of result structures. Each result structure has a score and the library offset of the corresponding sequence.

The important features of the main program are shown in Figure 5 . Note that we only had to declare one worker, and that the code from lines 7-10 looks as though the single worker

```
1  regular mentat class sw_worker {
2     // private member data and functions
3  public:
4     result_list *compare(sequence, libstruct, paramstruct);
5     // Compares sequence against a subset of the library. Returns
6     // a list of results (sequnce id, score).
7  }
```

Figure 4. Class definition for scanlib worker. Note it is a regular class indicating that the compare function is a pure function, and that the system may instantiate new instances as needed.

is being forced to do its work sequentially. Recall, however, that since the worker is a `regular` Mentat class, each invocation instantiates a separate copy of the worker object, so each copy is actually doing the comparisons in parallel.

```
1  // master program routine -- initialization details omitted
2  // -- get the number of workers
3  // -- divide the library into a partition for each worker
4
5  sw_worker worker;
6  // invoke the comparison for each partition of the library
7  for (i = 0; i < num_workers; i++) {
8     // compute library parcel boundaries
9     results[i] = worker.compare(the_seq, libparcelinfo, param_rec);
10 }
11 // for each partition's result, and for each comparison
12 // within the partition, compute statistics
13 for (i = 0; i < num_workers; i++) {
14    for (j = 0; j < results[i]->get_count(); j++) {
15       // for each result
16       // record mean, stdev, and histogram information
17    }
18 }
```

Figure 5. Code for the master program routine for scanlib.

Performance on the iPSC/2 is given in Table 1. Performance information is given for both the Mentat implementation and for a handcoded message passing C implementation. .A

Table 1 Scanlib execution times on the iPSC/2 in minutes: seconds.

Workers	1		3		7		15	
Sequence/Algorithm	M	HC	M	HC	M	HC	M	HC
LCBO - FASTA	2:47	2:42	0:59	0:57	0:28	0:26	0:17	0:16
LCBP - SW	-	-	95:49	95:19	41:12	41:00	19:22	19:14
RNBY3L[a]- FASTA	7:19	6:56	2:30	2:33	1:07	1:04	0:35	0:32

a. The Smith-Waterman times are not shown. They provide no additional insight.

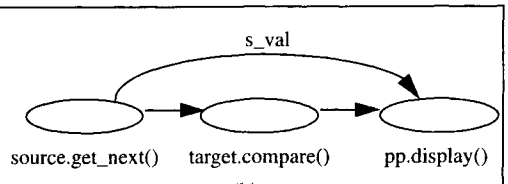
3.8-Mbyte target library containing 9633 sequences was used. CCHU, RSMD, LCBO, and RNBY3L are four different source sequences. They are 104, 229, 229, and 1490 bytes long respectively. Execution times for both the Smith-Waterman and much faster FASTA are given.

Performance on this application clearly demonstrates that for naturally data-parallel applications with no inter-worker communication and little worker-master communication, neither the object-oriented paradigm nor dynamic management of parallelism seriously affect performance. Indeed, the performance of the Mentat version is always within 10% of a handcoded version, and usually within 5%

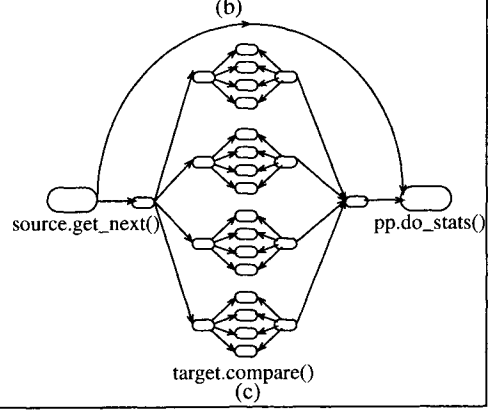
The complib implementation is more complex, though the main program remains straight forward. The main program manipulates three objects, the source genome library, the target genome library, and a recorder object that performs the statistical analysis and saves the results. See [17] for more detail. The main program for loop is shown in Figure 6 below. The effect is that a pipe is formed, with sequence extraction from the source, sequence comparison in the target, and statistics generation are executed in a pipelined fashion. Each high-level sequence comparison is transparently expanded into a fan-out, fan-in program graph where the "leaves" are the workers, the source sequence is transmitted from the root of the tree to the leaves, and the results are collected and sorted by collators.

```
for(i=0;i<num_source_seq;i++) {
    //for each sequence
    s_val = source.get_next();
    //Compare against target library
    result = target.compare(s_val);
    //Do statistics
    post_process.do_stats(result,s_val);
}
```

(a)



(b)



(c)

Figure 6 Mentat implementation of *complib*. The main loop of the program is shown in (a). Three objects are manipulated, the source, the target, and the post_processor. The pipelined program graph is shown in (b). *Target.compare()* has been expanded showing sixteen workers in (c). The fan-out tree distributes the source sequence to the workers. The internal nodes of the reduction tree are collator objects. The reduction tree sorts and merges the results generated by the workers.

Table 2 presents results for five complib implementations on a network of 16 Sun IPC workstations. The execution times for four hand-coded implementations are compared to the Mentat time. To reduce the effect of the perturbations caused by other users five runs for each were taken, and the *best* times reported. Keep in mind that *complib* is not a low communication application. At fifteen workers over fourteen megabytes of data are moved through the pipeline shown in Figure 6. We used the faster FASTA algorithm, a twenty sequence source library and a 10,716 sequence target library. The same kernel C code to actually perform the comparisons was used by all five implementations. The sequential time (without any parallel constructs) is 583 seconds. The most surprising aspect of the results is that the performance of the tools is so similar. With the exception of Express most of the execution times are within 5% of each other.

Table 2- Complib performance on Sun IPC workstations. Execution time in seconds.

Workers	Express	Linda	Mentat (2.6)	P4 (1.3b)	PVM (3.2.6)
3	220	211	**202**	218	206
7	98	95	**91**	98	92
11	80	62	63	65	**60**
15	NA	50	**48**	49	49

4.2 Matrix Algebra

Linear algebra is the basis for a wide variety of applications. Because it is so often the computational kernel of applications and so well understood by users we chose to construct a set of non-optimized linear algebra libraries to package with Mentat as examples. We begin with a discussion of an important support class. For more information see [29].

The foundation for many Mentat applications, including the linear algebra library, is a C++ library that provides classes for two dimensional arrays and vectors. Much of the functionality of these classes can be found in numerous matrix library classes [11,34]. The Mentat implementation environment (disjoint address spaces) placed the requirement on our class that parameters passed as arrays must be memory contiguous. The class *DD_array* provides an interface to provide the user with a class instance that is contiguous in memory without binding the size of the array involved or its orientation in memory. The *DD_array* methods provide for the decomposition of an array into subarrays, the creation of new arrays by overlaying or extracting from existing arrays, the access of array elements, and so on. Finally, *DD_array* serves as a base class from which arrays of any type can be created. Currently, arrays of floating point, double precision floating point, integer, and character have been developed. In addition, sparse and dense vectors of integers, floating point, and double precision floating point are derived from the class *DD_array*. The class *DD_array* also provides functions to decompose the array into one of five forms: by row, by column, cyclically by row, cyclically by column, or by block. For example, if the array consists of 100 rows, a row-cyclic decomposition into four pieces would allot rows 0, 3, 7,.. to the first subarray, 1,4,8,..to the second subarray, etc. A column-cyclic decomposition is analogous to a row-cyclic decomposition.

Using the *DD_array* class as a basis we then developed a set of Mentat linear algebra classes, one persistent, and one regular. The interface for the persistent class is shown below in Figure 7. The class implementations were coded in a classic master-worker/data parallel manner. Each matrix is internally sub-divided into several sub-matrices (*matrix_sub_block*s) which are themselves Mentat objects. Operations performed on the entire matrix are then implemented in the member functions by manipulating the sub-matrices.

Performance in the libraries is presented below in Table 3. The same network of 16 Sun IPC's was used. In [29] we present more complete results, including Intel Paragon performance. Because of performance difficulties with the NFS file system the run-time was measured from a point after the matrices had been loaded into memory[2]. The results are for single precision numbers. The matrix multiply uses a naive algorithm requiring $O(n^3)$ operations. The

2. We found that a large number of simultaneous bulk I/O requests will bring NFS to it's knees. The packet loss rate, combined with other factors that we do not understand, often causes the I/O to fail.

```
persistent mentat class p_matrix {
    int total_rows, total_cols;   //Total rows and columns in the array
    int num_pieces;   //Number of pieces into which the array is divided
    int decomp_type;  //Decomposition method
    p_matrix_sub_block *sub_arrays; //Pointers to the objects holding the decomposed matrix
public:
initialize(DD_floatarray *data, int decomp_type, int pieces);
initialize(string *file_name, int decomp_type, int pieces);
/*Creates instances of p_matrix_sub_block and distributes the array data by  decomp_type.*/
void scalar_add_mat(float scalar);                  /*Add a scalar to each of the subarrays.*/
int transpose();                                    /*Perform the transpose of the array.*/
sp_dvector *vec_mult(sp_dvector *arg_vect);         /*Performs matrix-vector multiplication.*/
DD_floatarray *mat_mult(DD_floatarray *data);
p_matrix  mat_mult(p_matrix);                       /*Performs matrix matrix multiplication. */
sp_dvector *solve(sp_dvector *rhs_vector);          /* Gaussian Elimination using partial pivoting.*/
};
```

Figure 7. Function prototypes for the `persistent mentat class p_matrix`. Only a subset of the interface is shown.

solver uses Gaussian elimination with partial pivoting which is also $O(n^3)$.

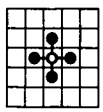
Table 3 Sun 4 Network Execution Times for 1024 x 1024 Matrices

Problem	Sequential Time	4	6	8	10	12	14
Regular Class							
Matrix Multiply	1380	363	258	202	163	206	193
Gauss	626	179	131	109	108	118	126
Persistent Class							
Matrix Multiply	1380	356	258	228	151	137	137
Gauss	626	172	123	102	99	107	126

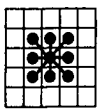
4.3 Stencil Libraries

Stencil algorithms are used in a wide range of scientific applications, such as image convolution and solving partial differential equations (PDE's). Stencil algorithms are a class of algorithms that have several features in common: (1) the input data set is an array of arbitrary dimension and size, (2) there is a *stencil* that defines a local neighborhood around a data point, (3) some function is applied to the neighborhood of points that are "covered" when the stencil is centered on a particular point, and (4) this function is applied to all points in the data set to obtain a new data set. Figure 8-*a* shows a two dimensional 3×3 stencil that indicates that each output value will depend *only* on the "north," "east," "west," and "south" (called NEWS) neighboring points of the corresponding point in the input array. The associated function is an example of a stencil function that uses NEWS neighbors.

We have defined a base stencil class, `Stenciler`, that is designed to manage those issues that are common to all stencil algorithms while providing a framework for the user to create derived classes that can be tailored to specific applications [23]. The base class contains



$$F_{i,j} = \frac{I_{i-1,j} + I_{i+1,j} + I_{i,j-1} + I_{i,j+1}}{4}$$



$$F_{i,j} = \sum_{k=0}^{2} \sum_{l=0}^{2} (I_{i+k-1,j+l-1} \times M_{k,l})$$

Figure 8. Typical 2 dimensional stencils. *F - final matrix, I - input matrix, M - convolution mask.* (*a*) 2D 3 × 3 NEWS stencil and sample function. (*b*) 2D 3 × 3 eight-connected stencil and sample function.

built-in member functions to perform common tasks, such as managing data communication between pieces. The base class also contains well-commented stubs for member functions that the user must define, such as the stencil function. This approach minimizes the effort needed to create new stencil applications through reuse of common code while supporting flexibility in creating parallel stencil applications. The user creates a new class derived from `Stenciler`. The derived class inherits all of the member functions of the base class, so instances of this new class have all of the built-in common functions provided with the `Stenciler` class. The user then supplies the application-specific code by overloading certain member functions.

An instance of a `Stenciler` or derived class is designed to handle one piece of the total array. Each `Stenciler` instance can create additional workers to split the work-load into smaller pieces. These pieces, in turn, may be further divided, creating a general tree structure of pieces as shown in Figure 9. Each new level of the tree has a "contained-in" relationship to the previous higher level. The pieces at leaves of this tree structure are the workers who perform the stencil function. The interior instances are managers for the workers below them; the managers distribute and synchronize the work of their sub-piece and collect the results. This hierarchical tree structure of processes is a powerful and flexible tool for decomposing a stencil problem, especially when running on different hardware platforms.

To illustrate the use of the stencil framework we describe our experience with two sample implementations: an image convolver and a PDE solver using Jacobi iteration.

Image convolution is a common application in digital image processing. In two dimensional image convolution, a small 2D stencil, also called a *filter* or *mask*, defines a region surrounding each picture element (pixel) whose values will be used in calculating the corresponding point in the convolved image. Each element of the filter is multiplied by the corresponding neighbor of the current pixel, and the results are summed and normalized. Figure 8-*b* shows a stencil function for a 3 × 3 mask.

Another common class of stencil algorithms are iterative methods. Jacobi iteration is a method for solving certain systems of equations of the form $A\vec{x} = \vec{b}$, where A is a matrix of coefficients, $\vec{x}$ is a vector of variables, and $\vec{b}$ is a vector of constants. The general procedure for using Jacobi iteration is to first guess a solution for all variables, and then to iteratively

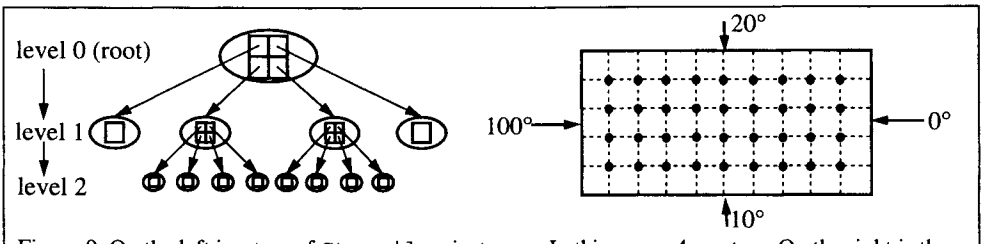


Figure 9. On the left is a tree of `Stenciler` instances. In this case a 4-ary tree. On the right is the cannonical grid approximation of a heated plate used in our PDE example.

refine the solution until the difference between successive answers is below some pre-determined threshold.

The application of Jacobi iteration used here is the classic heated plate problem. The heated plate problem consists of a plate or sheet of material that has constant temperatures applied around the boundaries, and the goal is to determine the steady-state temperatures in the interior of the plate (Figure 9). The temperature in the interior region is approximated by dividing the plate into a regular 2D grid pattern and solving for each of the grid points. The values at each point are approximated by the average of the values in the NEWS neighboring points. This transforms the problem into a system of linear equations which can be solved using Jacobi iteration. The form of the stencil for Jacobi iteration is shown in Figure 8-*a*.

To test performance we created two versions of each class, a sequential version written strictly in C++, and a parallel version written in C++/MPL. We executed the sequential versions on a Sun IPC and recorded the best of the wall-clock execution times. Similarly, we ran the parallel versions on a network of 16 Sun IPCs connected via ethernet. The parallel versions were executed decomposing the problem into from two to fourteen row pieces. Each decomposition was run several times and the best time for each decomposition was recorded.

For the `Convolver` tests, both the sequential and parallel versions were executed using identical problems: a 2000 × 2000 8-bit grey scale image convolved with three successive 9 × 9 filters. The `PDE_Solver` problem used a 1024 × 1024 grid of floating point numbers to estimate the interior temperatures of the heated plate problem. Table 4 shows the raw best execution times.

Table 4 Stenciler Performance Results

#Pieces	Convolver		PDE_Solver	
	Best Execution Time (min:secs)	Speedup	Best Execution Time (min:secs)	Speed up
1	49:33	N/A	43:39	N/A
2	24:33	2.0	23:01	1.9
4	12:37	4.0	11:37	3.8
6	8:47	5.6	7:53	5.5
8	7:07	7.0	7:08	6.1
10	5:53	8.4	6:47	6.4
12	5:08	9.7	6:23	6.8
14	4:57	10.0	6:10	7.1

4.4 APTG

Our next application is automatic test pattern generation, ATPG, an electrical engineering application where the objective is to generate a test pattern for sequential circuits that "covers" potential faults in the circuit [25]. The program we used implements a subset of the total ATPG problem. Each test vector is a vector of ones and zeros used as input to the circuit. Associated with each test vector is a corresponding expected output. A set of vectors covers the circuit if the application of all test vectors will expose any faulty element, i.e., whether the circuit is faulty. ATPG is an NP-complete problem. The complexity is exponential in circuit size [24].

The ATPG application was not developed in my research group. Instead we used code developed (and used) at the Center for Semicustom Integrated Circuits at the University of Virginia. The code was developed by Sanjay Srinivasan under the direction of Jim Aylor. The description here of the implementation is from [37], as are the graphics in Figure 10.

The basic idea is simple. There are two Mentat classes, `head_objects` and `support_objects`. There is a single `head_object` and many `support_objects` (Figure 10). The circuit is first partitioned. The `head_object` receives a portion of the circuit, and each `support_object` receives the remaining portion. The `head_object` generates vectors of assignments to evaluate. The `support_objects` request a vector from the `head_object`, evaluate the vector, and returns results. The computation required for the `support_objects` to perform their function is variable, some vectors may require much more computation than others. The structure of the application is essentially a bag of tasks.

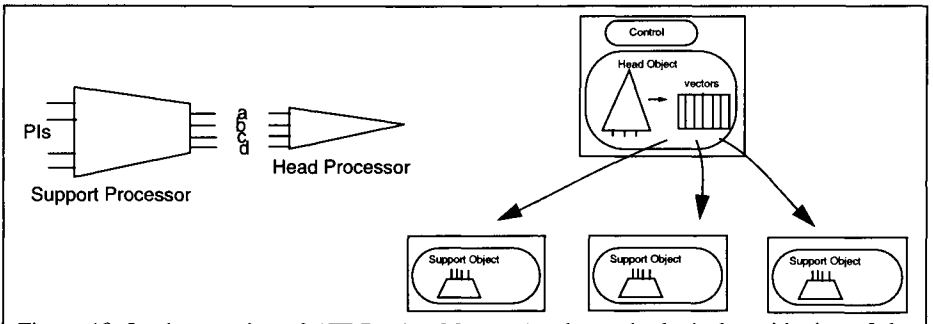


Figure 10. Implementation of ATPG using Mentat. (a) shows the logical partitioning of the problem, (b) shows the head object and the workers. This figure is from [37].

Because of the nature of the computation it is naturally load-balanced, when a `support_object` needs more work it gets more instead of waiting. A `support_object` with a particularly difficult piece of work will not request any more until it is done. Indeed, they have found that it is the one worker that gets a very difficult piece of work that limits performance.

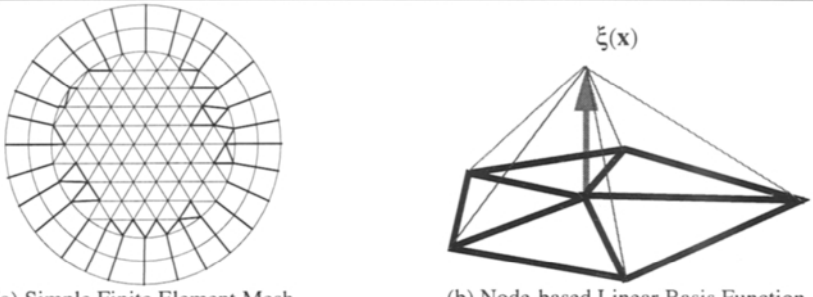
Table 5 ATPG results for s1423 on a network of 24 Sun IPC's.

Circuit	#support objects	Time (Level 1)
s1423	1	945.2min
	2	471.5min
	4	237.6min
	7	139.7min
	11	95.5min
	16	82.7min
	24	64.7min

4.5 2D Electromagnetic Finite Elements

The finite element method (FEM) has been in use for many years in structural mechanics and has become popular in recent years as a technique for use on electromagnetic problems. FEM has the advantage of being able to deal with the specific geometry of objects by using unstructured gridding which follows an object's shape. This can be of particular importance in EM scattering problems, where the correct representation of a scatterer's surface is necessary for accurate computation. Finite elements are used in 2 and 3 dimensional electromagnetic scattering problems to model objects of complex composition. A "hand-coded" FEM version of the code has been implemented on several MIMD computers by my collaborator using explicit message passing. A complete description of this code, along with parallel implementation description and performance, is found in [12]. A description of the Mentat implementation can be found in [40].

To solve the FEM problem a 2D integral equation is transformed into a set of linear equations by decomposing the problem domain into a set of finite elements. The problem domain is meshed with nodal points at which the solution is to be found, matching the geometry of the objects. These nodes are then tiled with a set of finite elements as in Figure 11-a. In



(a) Simple Finite Element Mesh (b) Node-based Linear Basis Function

Figure 11.

2D, the elements might be triangles or quadrilaterals. A set of basis functions are defined at each node in the mesh, which have nonzero value only within the elements of which it is a part. These basis functions are generally some polynomial function which is 1 at the node defining it, 0 at all other nodes in the element, and 0 along the edges of the element opposite the defining node. An example of a linear basis function at a node in a section of finite element mesh is given in Figure 11-b. The function is continuous inside and across elements, dropping to zero at the element edges which do not intersect the node. On all other elements in the grid, the basis function is identically zero.

The EM finite element application consists of two primary computation phases, matrix assembly and matrix solve. In matrix assembly, the finite elements compute contributions (i.e. matrix values) that are assembled (i.e. added) into the stiffness matrix $\overline{\mathbf{K}}$. The stiffness matrix is banded, symmetric, and very sparse. During assembly, the force vector $\mathbf{F}$ is also computed by the elements. This vector is the right-hand-side vector during the matrix solve computation.

In matrix solve the system of equations represented by the stiffness matrix with the force vector as the right-hand-side is solved by a conjugate-gradient algorithm known as Biconjugate gradient [20]. The algorithm uses three basic operations: matrix-vector multiplica-

tion, vector dot product, and vector saxpy. The solve phase poses challenges to achieving good performance on parallel machines due to the sparse nature of the matrix-vector operations.

The Mentat EM code was developed at JPL and run on a 64-node Intel iPSC/860 at Caltech. The data collected are from a data set that consisted of 2304 9pt quadrilateral elements (9313 nodes). This is considered a small problem. We computed speedups with respect to the sequential C++ EM code run on a single i860 node (Figure 12). The results are divided into the two dominate phases: 1) assembly is the time taken to complete the matrix assembly operations, and 2) solve is the time taken for the matrix solve operation. The sequential C++ is a factor of two slower for the solve phase, and a factor of three slower for the assembly phase. This is expected as the C++ compiler technology is much less mature than the Fortran technology; in particular the C++ compiler does not exploit the vector unit.

We compared the results with a hand-coded optimized parallel Fortran EM implementation that has been in development for some time. We expected the performance to be worse than the hand-coded version, but how much? The results indicate that this is indeed the case, though neither implementation achieves good speedup on the solve phase. They also indicate that the optimized version is scaling in a manner similar to the hand-coded Fortran. The assembly phase scales identically to the hand-coded while the solve phase scales almost as well. The slight discrepancy is due to Mentat overheads often seen for small problems. We expect the performance of the Mentat version to more closely match the hand-coded for larger problems.

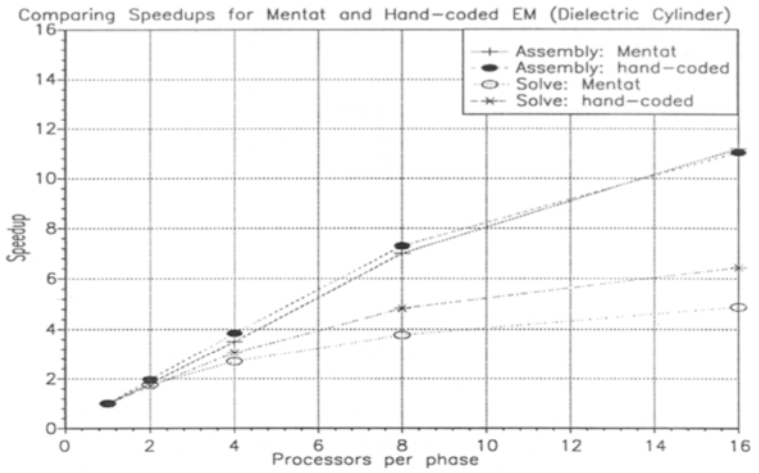


Figure 12. Comparison of optimized Mentat version to hand-coded EM version

5. The Future

In the three years that the MPL compiler has been operational we have developed several applications and learned much about the Mentat approach. In this chapter we introduced Mentat and presented results for five of those applications. The results are not all good. Like many other parallel processing systems Mentat performs well on some applications and not so well on others. The primary factors that influence performance are application granularity and application load balance. Mentat performs poorly when application granularity is small or when the application has load imbalances that Mentat cannot correct. This is not unexpected.

There is not much that can be done about the granularity restrictions, overheads can be reduced to lower the minimum effective grain size. However there is a limit on how low we can drive the overhead. Similarly, the underlying communication system of MPP's favors large grain computations. That is unlikely to change. With respect to load imbalance, there is room for significant improvement in the areas of dynamic scheduling of objects, and dynamic re-distribution of data parallel objects. We are working on both.

On the plus side we have learned that the use of the object-oriented paradigm combined with compiler-based parallelism detection and management can be performance competitive with hand-coded implementations using send and receive. This is significant because it means that the future of parallel processing is not limited to send and receive, and that the benefits of the object-oriented paradigm can be realized in high-performance parallel environments.

Now that we have demonstrated the efficacy of the Mentat approach in a homogeneous parallel processing environment our next challenge is to tackle first campus-wide, and then nation-wide, wide-area heterogeneous parallel processing. Three technological changes make nation-wide computing possible. The first is the much heralded information superhighway or national information infrastructure (NII) and the gigabit networks which are its building blocks. These networks can carry orders of magnitude more data than current systems. The effect is to "shrink the distance" between computers connected by the network. The second technological change is the development and maturation of parallelizing compiler technology for distributed memory parallel computers as exemplified by Mentat, DataParallel C, and HPF Fortran. The third technological change is the maturation of heterogeneous distributed systems technology. After twenty years of research, solutions have been found to many of the difficulties that arise in heterogeneous distributed systems.

The combination of mature parallelizing compiler technology and gigabit networks means that it is possible for applications to run in parallel on an nation-wide system. The gigabit networks also permit applications to more readily manipulate data regardless of its location because they will provide sufficient bandwidth to either move the data to the application *or* to move the application to the data. By utilizing heterogeneous distributed system technology problems such as data representation and alignment, processor faults, and operating system differences can be managed.

The Legion project at the University of Virginia is working toward providing system services that provide the illusion of a single virtual machine to users, a virtual machine that provides *both* improved response time and greater throughput. Legion is targeted towards nation-wide computing. Rather than construct a complete system from scratch we have chosen to construct a campus-wide testbed, the campus-wide virtual computer (CWVC), by extending Mentat. Even though the CWVC is smaller, and the components closer together, than in a full scale nation-wide system, it presents many of the same challenges. The processors are heterogeneous, the interconnection network is irregular, with orders of magnitude differences in bandwidth and latency, and the machines are currently in use for on-site applications that must not be negatively impacted. Further, each department operates essentially as an island of service, with its own file system.

The CWVC is both a working prototype and a demonstration project. The objectives are to demonstrate the usefulness of network-based, heterogeneous, parallel processing to computational science problems; provide a shared high-performance resource for university researchers; provide a given level of service (as measured by turn-around time) at reduced cost; and act as a testbed for the large scale Legion. The prototype is now operational and consists of over eighty workstations from four different manufacturers in five different buildings.

Acknowledgments

No project of the scope of Mentat is the product of a single individual. I would like to thank my collaborators James Aylor, Robert Ferraro, William Pearson, and Tim Strayer. I would also like to thank all of the students over the last six years who have made Mentat possible by implementing components of the system and by working on all save one of the above applications. Gorrell Cheek, Ed Loyot, John Karpovich, Laurie MacCallum, Dave Mack, Mark Morgan, Padmini Narayan, Ahn Nguyen-Tuong, David Shiflet, Sherry Smoot, Virgil Vivas, Emily A. West, and Jon Weissman.

Portions of this chapter, in particular the performance data, have appeared elsewhere [17,23,29,37,40] and appear by permission. Portions of this research were performed using the Intel Gamma operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Jet Propulsion Laboratory.

6. References

1 S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, "Basic local alignment search tool", *J. Mol. Biol.*, 215, pp. 403-410, 1990.

2 B. Beck, "Shared Memory Parallel Programming in C++," *IEEE Software*, 7(4) pp. 38-48, July, 1990.

3 B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Presto: A System for Object-Oriented Parallel Programming," *Software - Practice and Experience*, 18(8), pp. 713-732, 1988.

4 F. Bodin, et. al., "Distributed pC++: Basic Ideas for an Object Parallel Language," *Proceedings Object-Oriented Numerics Conference*, pp. 1-24,Sunriver, Oregon, April 25-27, 1993.

5 J. Boyle et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.

6 J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, pp. 111-120, vol. 16, no. 2, Feb., 1990.

7 N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April, 1989.

8 K. M. Chandy, and C. Kesselman, "CC++: A Declarative Concurrent Object Oriented Programming Notation,", Technical Report CS-TR-92-01, Department of Computer Science, California Institute of Technology, 1993.

9 R. Chin and S. Chanson, "Distributed Object-Based Programming Systems," *ACM Computing Surveys*, pp. 91-127, vol. 23, no. 1, March., 1991.

10 A.L.Cheung, and A.P. Reeves, "High Performance Computing on a Cluster of Workstations," *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 152-160, Syracuse, NY, Sept., 1992.

11 J. Dongarra, R. Pozzo, and D. W. Walker, "An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures," *Proceedings Object-Oriented Numerics Conference*, pp. 257-264, Sunriver, Oregon, April 25-27, 1993.

12 R. D. Ferraro, "Solving Partial Differential Equations for Electromagnetic Problems on Coarse-Grained Concurrent Computers," *Pier 7*, T. Cuik and J. Patterson, Eds., pp. 111-155, EMW Publishing, Cambridge, MA, 1993.

13 G. C. Fox, et al., "Fortran D Language Specifications," Technical Report SCCS 42c, NPAC, Syracuse University, Syracuse, NY.

14 A. S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, pp. 39-51, May, 1993.

15 A. S. Grimshaw, W. T. Strayer, and P. Narayan, "Dynamic Object-Oriented Parallel Processing," *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 34-47, May, 1993.

16 A. S. Grimshaw, et. al., "The Mentat Users Manual," available on the world wide web, *http://uvacs.cs.virginia.edu/~mentat/*, 1994.

17 A. S. Grimshaw, E. A. West, and W.R. Pearson, "No Pain and Gain! - Experiences with Mentat on Biological Application," *Concurrency: Practice & Experience*, pp. 309-328, Vol. 5, issue 4, July, 1993.

18 A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," *to appear ACM Transactions on Computer Systems.*

19 T. J. R. Hughes, *The Finite Element Method*, Prentice Hall, Englewood Cliffs, NJ, 1987.

20 D. A. H. Jacobs, *Sparse Matrices and their Uses*, Academic Press, London, 1982.

21 A. K. Jones, "The Object Model: A Conceptual Tool for Structuring Software," in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller Eds., Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1979.

22 L. V. Kale, and S. Krishnan, "CHARM++: A Portable Concurrent Object-Oriented System Based on C++", *Proceedings of OOPSLA '93*, pp. 91-108, Washington DC, 1993.

23 J.F. Karpovich, M. Judd, W.T. Strayer, and A.S. Grimshaw, "A Parallel Object-Oriented Framework for Stencil Algorithms," *Proceedings of the Second Symposium on High-Performance Distributed Computing*, pp. 34-41, Spokane, WA, July, 1993.

24 R. H. Klenke, et. al.,"Workstation Based Parallel Test Generation," *1993 IEEE International Test Conference,* pp. 419-428.

25 R.H Klenke, R.D. Williams, and J.H. Aylor, "Parallel Processing Techniques for Automatic Test Pattern Generation, IEEE Computer, pp. 71-84, January, 1992.

26 K. Kennedy."Compiling for Parallel Computers", *Strategic Directions in Computing Research*, ACM Press, pp. 15-17, Arlington, VA., Oct. 11-13, 1989.

27 J. R. Larus, B. Richards, and G. Viswanathan, "C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language," UW Technical Report #1126, Computer Sciences Department, Univerrsity of Wisconsin, November 1992.

28 D. B. Loveman, "High Performance Fortran,"*IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25-42, February, 1993.

29 L. MacCallum and A.S. Grimshaw, "A Parallel Object-Oriented Linear Algebra Library," *Proceedings Object-Oriented Numerics Conference*, pp. 233-249, Sunriver, Oregon, April 25-27, 1994.

30 J. McGraw, et. al., "Sisal: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2," Lawrence Livermore National Laboratory Manual M-146, Lawrence Livermore National Laboratory, Livermore, CA, 1984.

31 W. R. Pearson and D. Lipman, "Improved tools for biological sequence analysis", *Proc. Natl. Acad. Sci. USA*, 85, pp. 2444-2448, 1988.

32 C. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

33 M. J. Quinn and P. J. Hatcher, "Data-Parallel Programming on Multicomputers," *IEEE Software*, pp. 69-76, September 1990.

34 Rogue Wave Software, "Math.h++, Matrix.h++, and Linpack.h++," C++ based math libraries from Rogue Wave Software, Corvallis, OR.

35 S. K. Smith, et al., "Experimental Systems Project at MCC," MCC Technical Report Number: ACA-ESP-089-89, March 2, 1989.

36 T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *J. Mol. Biol.*, 147, pp. 195-197, 1981.

37 S. Srinivasan and J.H. Aylor, "Digital Circuit Testing on a Network of Workstations," *Proceedings of the International Conference on Parallel Processing*, August, 1994.

38 V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

39 P. Wegner,"Dimensions of Object-Based Language Design," *Proceedings of the 1987 Object-Oriented Programming Systems, Languages and Applications Conference*, ACM, pp. 168-182, October, 1987.

40 J. B. Weissman, A. S. Grimshaw, and R. Ferraro, "Parallel Object-Oriented Computation Applied to a Finite Element Problem," *Scientific Computing*, vol. 2, no. 4, pp. 133-144, Feb., 1994.

41 W. A. Wulf, et. al., "Hydra: The Kernel of a Multiprocessor Operating System," Communications of the ACM, pp. 337-345, vol. 17, no. 6, June, 1974.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Parallel Loop Scheduling for High Performance Computers

Kelvin K. Yue[a*] and David J. Lilja[b]

[a]Department of Computer Science,
[b]Department of Electrical Engineering,
University of Minnesota, Minneapolis, MN 55455, USA

Executing loop iterations in parallel on a multiprocessor system is one of the many ways to improve the execution of a program. However, due to the scheduling overhead and the potential for load imbalance among processors, maximum performance might not be attained. This article reviews current loop scheduling algorithms and studies their scheduling overhead versus load balancing tradeoffs. Using analytical models, simulations, and experimental measurements, the performance and the scalability of chunk scheduling, self-scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling are compared.

Keywords: Parallel loop scheduling; Scalability; Shared memory multiprocessor; Analytical modeling; Performance analysis

1. INTRODUCTION

With a *smart* compiler, the loop-level parallelism of many existing application programs can be exploited without rewriting the sequential code. This approach not only increases the portability of the code, but also lessens the programmer's burden to develop new parallel algorithms. Although loops are a rich source of parallelism, one is confronted with the problem on how to fully utilize the processors to execute the loop iterations efficiently in a multiprocessor environment.

Valuable resources will be wasted if workloads are distributed among the processors unevenly, that is, if some processors are busy executing the loop iterations while others are idling. There are many possible reasons for this load imbalance, one of which is the structure of the loop. For instance, conditional branching statements may cause the iterations of a loop to have different numbers of operations. Another possible reason for uneven workload is system interrupts, such as page faults or context switches, which cause some iterations to take longer to execute than others. To balance the processors' workloads, a variety of algorithms have been developed to dynamically assign iterations to idle processors [7,9,16,18].

Since the performance of these algorithms depends on the execution environment, the loop structure, and the specific implementation details, finding the best algorithm for

*This work was supported in part by the National Science Foundation under grant no. MIP-9221900, and by Army Research Office contract number DAALO3-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center.

scheduling a specific loop on a given system can be difficult. To compare their performance potential, this article reviews current scheduling algorithms using a common notation and consistent terminology. An analytical framework is used to compare the basic characteristics of these algorithms and the strategies they use to tradeoff a balanced workload and scheduling overhead.

As today's high-performance multiprocessor systems employ more and more processors and the problem sizes become larger and larger, scalability becomes one of the important issues for algorithm design. This article discusses and compares the scalability of these algorithms using the analytical models. To make the comparisons more realistic, the algorithms were also implemented on a Silicon Graphics Onyx multiprocessor system. The scheduling algorithms we study include chunk scheduling [9], self-scheduling [3], guided self-scheduling [16], factoring [7], and trapezoid self-scheduling [18].

This article is organized as follows: Section 2 reviews some techniques for exploiting loop-level parallelism and provides some useful background information. Section 3 describes the scheduling algorithms and develops the analytical models. It also discusses the scalability aspects of each algorithm. The performance comparison of the algorithms is presented in Section 4, and Section 5 discusses some current research issues. Section 6 concludes this article.

2. LOOP LEVEL PARALLELISM

Before discussing the algorithms for scheduling loop iterations to execute on multiprocessor systems, it is useful to review other techniques for exploiting loop-level parallelism. First, we review trace scheduling and software pipelining, techniques that are used at compile-time to exploit fine-grained parallelism. Then we discuss some strategies for exploiting coarse-grained parallelism, such as *doacross* scheduling and *doall* loop scheduling.

Fine-grained parallel architectures, such as the VLIW (Very Long Instruction Word) processors [14], exploit parallelism at the instruction-level. The *trace-scheduling* compilation technique [4] statically predicts the outcome of conditional branches and unrolls the loops in a program to generate a long sequence of instructions, called a *trace*. The trace is then optimized using standard compilation techniques, and operations from the trace are packed into the long instruction words for execution. Compensation code is added around branches to ensure correct execution when a branch outcome is incorrectly predicted.

Another compilation technique for parallelizing loop execution is *software pipelining* [11] in which parallelism is achieved by overlapping the execution of instructions from several different iterations. At compile-time, a schedule for the iterations is generated so that there is a fixed interval between the start of one iteration and the next. The maximum performance is attained by using the smallest interval that can satisfy the cross-iteration dependences [12]. As an example, consider the following loop:

```
DO i = 1, N
   a(i) = e(i) + 6
   c(i) = b(i) * d(i)
END DO
```

If the arrays are stored in different memory modules, software pipelining allows the loading of the variables to be done simultaneously. Then the adder and the multiplier can start

the calculations in the next cycle. If an addition takes two cycles and a multiplication takes three, the memory loads for the next iteration can begin two cycles later for variable `e(i)` and three cycles later for `b(i)` and `d(i)`. The store operations for the first iteration can also be done in parallel with the loads from the next iteration.

Both trace scheduling and software pipelining are techniques that use only compile-time information to schedule instructions. Since both compile-time information and physical resources are limited, assumptions about cross-iteration dependences must be conservative to preserve the correctness of the program execution. As a result, these techniques might not be able to achieve the maximum performance. Run-time profiling is one approach to compensate for this shortcoming. By running an instrumented version of the target program, statistics such as branch execution frequencies, average execution times of loops, and average iteration counts can be obtained and fed-back to the compiler to assist the optimization.

Coarse-grained parallel architectures exploit parallelism by executing entire loop iterations in parallel. The iterations of a loop are considered as independent tasks and are scheduled for execution on a shared-memory multiprocessor system, such as that shown in Figure 1, using some loop scheduling strategy. For loops with cross-iteration depen-

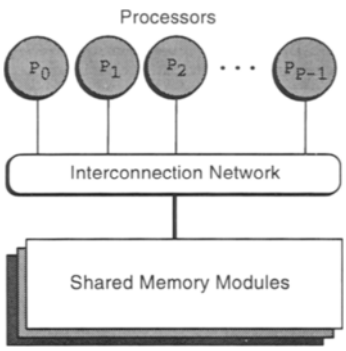


Figure 1. Block diagram of a shared memory multiprocessor architecture.

dences [12], the execution order for the iterations must be maintained to ensure program correctness. Doacross scheduling [2] uses explicit synchronization to enforce the order of execution by ensuring that each iteration starts at least d cycles after the previous iteration. An example of a doacross scheduled loop is:

```
DO i = 1, N
   delay d * (i - 1)
   {loop body for iteration i}
END DO
```

Although the idea of delaying execution in doacross scheduling is similar to software pipelining, the delay is introduced at a different level. Doacross scheduling uses explicit

synchronization operations to delay independent processors. It allows variable delays between processors to prevent dependence violations. Software pipelining, on the other hand, introduces a fixed delay at the instruction level and, thus, requires knowledge of the exact instruction execution times to produce the loop schedule.

For loops without cross-iteration dependences, which are called *doall* loops, the iterations can be executed in any order. Thus, the iterations of a doall loop can be viewed as independent tasks and can be executed concurrently without any restriction. An example of a doall loop is:

```
DO i = 1, N
   a(i) = b(i) + c(i)
   z(i) = x(i) + y(i)
END DO
```

There are two main categories of algorithms to schedule doall loop iterations for execution: static and dynamic [12]. Static scheduling, or pre-scheduling, assigns iterations to the processors at compile-time. Each processor *knows* exactly which iterations it should execute before the program is invoked and, therefore, there is no scheduling overhead. An example of static scheduling is a simple round-robin approach in which the compiler assigns iterations to the processors based on the processor number. For example, processor 0 executes iterations 1, $P+1$, $2P+1$, $\cdots$, processor 1 executes iterations 2, $P+2$, $2P+2$, $\cdots$, and so on, where P is the number of processors. The main disadvantage of static scheduling is load imbalance [1] which causes some processors to remain idle while others are busy. This imbalance can be caused by differences in iteration execution times, or by differences in the number of iterations each processor executes.

Dynamic scheduling attempts to reduce load imbalances by having idle processors assign iterations to themselves at run-time. Examples of this type of scheduling algorithm are self-scheduling, chunk scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling. The remainder of this article reviews these dynamic scheduling algorithms in detail.

3. LOOP SCHEDULING ALGORITHMS

In this section, we compare current parallel loop scheduling algorithms using simple analytical models. These analytical models are then used to study the scalability of each algorithm. The analysis is based on a shared-memory multiprocessor system with P processing units, all of which execute at the same speed. The processing units are connected to the global memory via some interconnection network. We let T_p be the parallel execution time of a parallel loop when the loop is executed on the multiprocessor system. When the same loop is executed sequentially on a single processor, the execution time is T_s.

When the parallel loop is executed on the multiprocessor system, there is additional run-time overhead that is not required when the loop is executed sequentially. This overhead includes such things as synchronization delays, the time required to calculate loop index ranges, processor idling, and network contention to access shared variables. We let T_o be the total overhead of *all* the processors. Figure 3 shows the relationship

between T_p, T_s, and T_o for a four-processor system. Thus, we have

$$T_s + T_o = P \times T_p.$$

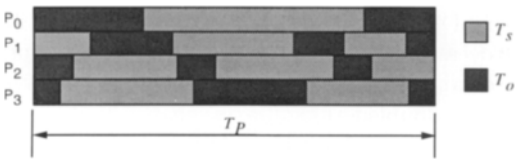


Figure 2. Parallel execution time of a four-processor system.

The *speedup*, S, of an execution of the parallel loop is defined to be the ratio of the sequential run-time to the parallel run-time giving

$$S = \frac{T_s}{T_p}.$$

The *efficiency*, E, is defined to be the speedup divided by the number of processors:

$$E = \frac{1}{P} \times \frac{T_s}{T_p} = \frac{T_s}{T_s + T_o}.$$

Thus, we have

$$E = \frac{1}{1 + T_o/T_s}.$$

Using this expression, the efficiency of an algorithm can be estimated by determining the sequential execution time, T_s, of the algorithm and the total overhead, T_o, required if the algorithm is executed on a parallel system.

To use this equation to compare the parallel loop scheduling algorithms, we let μ be the average execution time of an iteration of a loop with a total of N iterations. Thus, the sequential run-time of the loop, T_s, is $N\mu$. The total overhead, T_o, on the other hand, is quite difficult, if it is not impossible, to quantify as it depends on many factors, such as the variance in iteration execution times, the execution environment, and other unpredictable system events. Therefore, instead of using extensive statistical analyses based on some simplified assumptions, we calculate the number of scheduling steps required for the algorithms and use that as a first-order approximation of T_o. Then we introduce the variable δ to represent other varying overheads. Thus we have

$$T_o = T_{sch} + \delta.$$

Although this method might not give us the *exact* efficiency value for a scheduling strategy, it is sufficient for comparing their algorithmic characteristics. The efficiency function becomes:

$$E = \frac{1}{1 + (T_{sch} + \delta)/N\mu}.$$

According to the above equation, the efficiency of an algorithm is dependent on the ratio of the total overhead to the total computation time. This ratio has been called the *isoefficiency function* [10]. This isoefficiency function is used to measure the *scalability* of an algorithm, which is an indication of its ability to effectively utilize an increasing number of processors. An algorithm is *scalable* if its speedup increases in linear proportion to the increase in the number of processors [5]. For instance, when a loop with N iterations is executed on a P-processor system using some scheduling algorithm, this scheduling algorithm is *perfectly* scalable if the parallel execution time is halved when the number of processors in the system is doubled.

In the following subsections, the algorithmic design of chunk scheduling, self-scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling is studied and the efficiency of each algorithm is formulated based on the relationship between the number of processors, the number of iterations, the overhead, and the iteration execution time. These models are then used to study the scalability of these algorithms.

3.1. Chunk Scheduling

The simplest method of scheduling parallel loop iterations on shared-memory multiprocessor systems is to use chunk scheduling. The chunk size, or the number of iterations per scheduling step, is calculated at compile-time or at the beginning of the loop execution. Each idle processor obtains its share of work based on this chunk size. Single-chunk scheduling uses a chunk size of $\lceil N/P \rceil$ so that each processor is assigned the same number of iterations to execute.

For instance, if there are a total of 100 iterations and 4 processors, processor 0 executes iterations 1 to 25, processor 1 executes iterations 26 to 50, processor 2 executes iterations 51 to 75, and processor 3 executes iterations 76 to 100. We let t_{cs} be the time required for a processor to access the shared work queue and determine the chunk of iterations that it needs to execute. Single chunk scheduling requires each processor to determine its share of work only once, so the total scheduling overhead, T_{sch}, is Pt_{cs}. As the sequential run-time for executing this loop is $N\mu$, the efficiency for single chunk scheduling is:

$$E = \frac{1}{1 + (Pt_{cs} + \delta_{cs})/N\mu}.$$

Assuming that the variance of the iteration execution times is nil, and that there is no system event to affect the execution, $\delta_{cs} \approx 0$. Then the efficiency of chunk scheduling is solely dependent on the term $Pt_{cs}/(N\mu)$. Therefore, the efficiency of chunk scheduling is high when the number of iterations, N, or the average iteration execution time, μ, is large compared to the number of processors, P, and the scheduling overhead, t_{cs}. Moreover, this term suggests that if the number of processors, P, is doubled, the problem size, N, needs only to be doubled to maintain the same efficiency value. Thus, chunk scheduling is scalable in terms of the isoefficiency definition.

If there are variances in iteration execution times, however, the total execution times for each processor may vary widely, although the number of iterations per processor is constant. Moreover, it is very unlikely that the execution will be unaffected by any system events. As a result, δ_{cs} typically will *not* be zero. Kruskal and Weiss [9] statistically analyzed load imbalances with this scheduling strategy and proposed the optimal chunk size to be $[(\sqrt{2}Nt_{cs})/(\sigma P\sqrt{\log P})]^{2/3}$, where N is the number of iterations, P is the number

of processors, σ is the standard deviation of the distribution of iteration execution times, and t_{cs} is the scheduling overhead. This expression shows the relationship between the chunk size, scheduling overhead, and iteration execution time variance and suggests that if the variance of the iteration execution times, σ, is large, a small chunk size should be used. On the other hand, if the scheduling overhead, t_{cs}, is large, a large chunk size will be better. However, using this method for real applications might not be practical since it is difficult to determine the variance in iteration execution times *before* the loop is executed. Moreover, the variance is affected by the execution environment and may change from one run of the program to another.

3.2. Pure Self-Scheduling

Although chunk scheduling distributes the iterations evenly among the processors, the workload for each processor might not be the same due to variances in iteration execution times. Pure self-scheduling, or simply *self-scheduling*, uses a chunk size of one iteration. As suggested by Kruskal and Weiss's equation, if the scheduling overhead is small, the optimal chunk size to use for perfectly balancing the workload should be one iteration per chunk. Since each processor obtains the next iteration to be executed when it is idle, scheduling a doall loop with N iterations using self-scheduling requires N scheduling steps. If each scheduling step requires t_{ss} units of time, the total scheduling overhead is $T_{sch} = Nt_{ss}$. The efficiency for self-scheduling then is:

$$E = \frac{1}{1 + (Nt_{ss} + \delta_{ss})/N\mu} = \frac{1}{1 + t_{ss}/\mu + \delta_{ss}/N\mu}.$$

According to this expression, the efficiency for self-scheduling is limited by the ratio between the scheduling overhead, t_{ss}, and the average iteration execution time, μ, since typically $\delta_{ss} \ll N\mu$. Therefore, self-scheduling should only be used in systems in which the overhead for accessing shared variables is small, such as systems with built-in hardware for fetch-and-add operations.

Even in such systems, the overall performance for self-scheduling might be unacceptable in many cases. Since self-scheduling requires the processors to access the shared loop index for each iteration, the chances of network and memory contention are very high. For instance, if P processors are trying to access the shared loop index at the same time, $(P-1)$ processors must wait for the first processor to complete its access. Then when the second processor is accessing the shared loop index, $(P-2)$ processors must wait, and so on. If the time to access the shared variable is t_{ss}, the waiting time for the second processor before it obtains an iteration is t_{ss}, for the third processor is $2t_{ss}$, and for the last processor is $(P-1)t_{ss}$. Thus, the total waiting time to distribute P iterations to P processors is $P(P-1)t_{ss}/2$. If there are a total of N iterations, then the total waiting time for all the P processors is $N(P-1)t_{ss}/2$. Using this for δ_{ss}, the efficiency for self-scheduling degrades to:

$$E = \frac{1}{1 + t_{ss}/\mu + (P-1)t_{ss}/2\mu} = \frac{1}{1 + (P+1)t_{ss}/2\mu}$$

This expression suggests that when the number of processors increases, the efficiency decreases. For a system with a large number of processors, self-scheduling does not perform well unless the scheduling overhead, t_{ss}, is very small. Therefore, self-scheduling does not scale very well.

3.3. Guided Self-Scheduling

Chunk scheduling requires very little scheduling overhead but it does not balance the workload very well. Pure self-scheduling, on the other hand, can balance the workload quite well but it generates a large overhead and possibly significant contention. To tradeoff load balancing and scheduling overhead, several algorithms that use a variable chunk size have been proposed [7,16,18]. These algorithms schedule iterations in chunks where the size of the chunk (i.e. the number of iterations per chunk) decreases as the loop execution proceeds.

Guided self-scheduling (GSS) [16] is one algorithm that uses this strategy to schedule the loop iterations on to the processors. For GSS, the current chunk size is calculated at each scheduling step to be $\lceil R/P \rceil$ where R is the number of iterations remaining to be executed. For instance, if there are a total of 100 iterations to be executed on a five-processor system, the size of the chunks is: 20, 16, 13, 11, 8, 7, 5, 4, 4, 3, 2, 2, 1, 1, 1, 1, and 1. This algorithm allocates large chunks at the beginning of a loop's execution to reduce the scheduling overhead. As the number of iterations remaining to be executed decreases, smaller chunks are allocated. At the end of the execution, GSS guarantees there are at least $(P-1)$ single-iteration chunks remaining that can be used to balance the load [16].

In the best case, guided self-scheduling requires only P scheduling steps. This happens only when N, the number of iterations, is approximately equal to P, the number of processors. For N much greater than P, it is quite difficult to predict the number of scheduling steps required due to the round-off operation. Therefore, we use the upperbound of the number of scheduling steps in our analysis. The maximum number of scheduling steps required is $PH_{\lceil N/P \rceil}$ steps, where $H_n \approx \ln(n) + \gamma + 1/(2n)$ is the nth harmonic number, and $\gamma = 0.5772157\ldots$ is Euler's constant [16]. Therefore, the total scheduling overhead for GSS is

$$T_{sch} \approx (P \ln\lceil \frac{N}{P} \rceil + P\gamma + \frac{1}{2}\lceil \frac{P}{N} \rceil) t_{gss}.$$

To simply this expression, we can consider the case when the number of iterations, N, is large so that the term $P \ln\lceil N/P \rceil$ dominates the expression. As a result, we can approximate the total scheduling overhead to be:

$$T_{sch} \approx P \ln\lceil \frac{N}{P} \rceil t_{gss}$$

Thus, the efficiency for GSS is approximately:

$$E \approx \frac{1}{1 + (P \ln\lceil \frac{N}{P} \rceil t_{gss} + \delta_{gss})/N\mu}.$$

As for chunk scheduling and self-scheduling, we also include the variable δ in the expression to represent the overhead introduced by load imbalance and unpredictable system events.

Based on this approximation, the efficiency of GSS depends primarily on the term $(P \ln\lceil N/P \rceil) t_{gss}/N\mu$. This term indicates that when the number of processors is doubled, the number of iterations must be doubled to maintain a constant ratio. Thus, GSS is quite scalable. However, recall that in the equation for calculating the number of scheduling

steps for GSS, there is the term $P\gamma$. This term, which we ignored in our approximation, degrades the efficiency when GSS is used in systems with large numbers of processors. Consequently, the scalability of GSS is more limited.

For load balancing, GSS claims that the finishing times for all processors are balanced within μ units of time, where μ is the average iteration execution time [16]. However, this claim is true only when the variance of iteration times is small [7]. To verify this claim, we measured the performance of GSS as shown later in this article.

3.4. Factoring

Factoring (FS) [7] is another scheduling strategy that uses variable chunk sizes. It is similar to GSS in that the chunk size is calculated at each scheduling step based on the remaining number of iterations, but, unlike GSS, it allocates P batches of equal-sized chunks before the chunk size is recalculated. The equation for calculating the chunk size is

$$\lceil \frac{R}{xP} \rceil$$

where R is the number of iterations remaining, P is the number of processors, and, based on some statistical analysis of iteration execution time variances, x is suggested to be two [7]. The chunk sizes for FS when $N = 100$ iterations in a five-processor system are: 5 chunks with 10 iterations each, 5 chunks with 5 iterations each, 5 chunks with 3 iterations each, and 10 single-iteration chunks. In general, when $x = 2$, the chunk size is determined by [7]:

$$K_j = \lceil (\frac{1}{2})^{j+1} \frac{N}{P} \rceil$$

where K_j is the chunk size for *factoring* step j, N is the total number of iterations, and P is the number of processors.

To determine the total number of scheduling steps required by FS, we set K_j to one and calculate j, which is the number of *factorings* required before the chunk size becomes one. We then find that

$$j = \frac{\ln(P/N)}{\ln(1/2)} - 1.$$

Since Factoring assigns P batches of chunks with the same number of iterations, the total number of scheduling steps before the chunk size becomes one is

$$P \times (\frac{\ln(P/N)}{\ln(1/2)} - 1).$$

In the worst case, there are $2P$ single-iteration chunks at the end of the loop's execution. The total scheduling overhead is

$$T_{sch} = P(\lceil \frac{\ln(P/N)}{\ln(1/2)} \rceil + 1)t_{fs} \approx P\lceil 1.44 \ln(N/P) \rceil t_{fs}.$$

Thus, the approximate efficiency of FS is:

$$E \approx \frac{1}{1 + (P\lceil 1.44 \ln(P/N) \rceil t_{fs} + \delta_{fs})/N\mu}.$$

The scalability of factoring is characterized primarily by the $P\lceil 1.44 \ln(P/N) \rceil t_{fs}/N\mu$ term, which indicates that when the number of processors is doubled, the number of iterations needs to be doubled to keep the ratio constant. Therefore, it is quite scalable. Another observation is that this term is 1.44 times the equivalent term in GSS. This relationship indicates that factoring allocates more smaller chunks than GSS to balance the workload—an example of trading-off scheduling overhead and load balancing. Later in this article, we justify this tradeoff with experimental results.

3.5. Trapezoid Self-scheduling

Instead of decreasing the chunk size non-linearly like guided self-scheduling and factoring, trapezoid self-scheduling (TSS) [18] decreases the chunk size linearly. The chunk size for TSS is reduced by a factor of $(f-1)/(C-1)$ at each scheduling step, where f is the number of iterations assigned in the first chunk, l is the number in the last chunk, and C, the total number of chunks, is equal to $2N/(f+l)$. TSS does not allocate chunks as large as GSS in the beginning, and it does not require as many scheduling steps as FS.

The number of scheduling steps for trapezoid self-scheduling is

$$\lceil \frac{2N}{f+l} \rceil$$

where typically $f = N/2P$ and $l = 1$ [18]. Therefore, the total scheduling overhead is $T_{sch} = \lceil 4NP/(N+1) \rceil t_{tss}$. For large N, the total scheduling overhead is approximately $T_{sch} \approx 4Pt_{tss}$. The efficiency for TSS then is approximately

$$E \approx \frac{1}{1 + (4Pt_{tss} + \delta_{tss})/N\mu}.$$

Similar to GSS and FS, the efficiency of TSS can be maintained by increasing the number of iterations, N, at the same rate as the number of processors, P. However, in cases when δ_{tss} is large, TSS generates large load imbalances for the last few scheduling steps due to the linearly decreasing properties of the algorithm.

4. SCHEDULING ALGORITHM COMPARISON

In the previous section, we formulated the efficiency equations for current scheduling algorithms and found that self-scheduling does not scale well, while chunk scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling have similar scalability properties. The issue then becomes to determine which strategy is the best for balancing the workload with the smallest overhead, δ. In this section, we first compare the analytical analyses of the scheduling algorithms. Then we present the results of experiments that measured the performance of these algorithms on an actual machine.

4.1. Analytical Comparison

In Section 3, we developed an efficiency equation for each of the scheduling algorithms. Based on the expressions, we briefly discussed the performance of the algorithms and their scalability. To compare their performance in greater depth, their algorithmic characteristics are summarized in Table 1 in three categories: scheduling overhead per scheduling step, the total number of scheduling steps, and the other overheads caused by load imbalance, network and memory contention, etc.

Table 1
Catagorized comparison of scheduling algorithms

Algorithm	CS	SS	GSS	FS	TSS
Scheduling overhead	t_{cs}	t_{ss}	t_{gss}	t_{fs}	t_{tss}
Scheduling steps	P	N	$P \ln \lceil \frac{N}{P} \rceil$	$P \lceil 1.44 \ln \frac{N}{P} \rceil$	$4P$
other overheads	δ_{cs}	δ_{ss}	δ_{gss}	δ_{fs}	δ_{tss}

The scheduling overhead of the algorithms is dependent on the system environment and the algorithms' actual implementation details. If a system provides a fetch-and-add operation for shared variables, both chunk scheduling and self-scheduling can be implemented using that operation. For chunk scheduling, the chunk size can be calculated at compile-time or when the parallel loop is first executed. A shared variable counts the number of chunks assigned, and when a processor needs work, it fetch-and-adds the shared variable and uses it to calculate its share of the work. For self-scheduling, the shared variable is simply the loop index. Therefore, t_{cs} and t_{ss} can be very small.

However, for guided self-scheduling, the size of the current chunk must be calculated at each scheduling step. An idle processor first obtains a lock to enter the critical section where it determines its share of the remaining work. It then releases the lock and starts executing its assigned iterations. Compared to the fetch-and-add operation, the locking and unlocking operations are usually more expensive. The calculation of the chunk size in the critical section further increases the overhead. Therefore, t_{gss} typically is much larger than t_{cs} and t_{ss}.

Factoring, similar to GSS, needs to calculate it chunk size in a critical section. However, since it assigns a batch of P chunks with the same size, the chunk size only needs to be calculated every P steps. Once the chunk size is calculated, a single fetch-and-add operation can be used to obtain additional work. Therefore, FS needs one critical section execution for every $(P-1)$ fetch-and-add operations and, on average, t_{fs} is smaller than t_{gss}, but larger than t_{cs} and t_{ss}.

Using both a critical section and fetch-and-add operations to implement factoring might require extra synchronizations and cooperation between processors. For instance, if there are two idle processors that need work at the same time after P chunks of the same size have been allocated, one will enter the critical section to calculate the new chunk size and reset the chunk counter. The other processor must wait until the first processor is done. If it does not wait, it might obtain some inconsistent information about the chunk size and the number of allocated chunks. Therefore, there is a need for extra synchronizations to ensure the correctness of the execution which can increase t_{fs}.

Since trapezoid self-scheduling decreases the chunk sizes linearly, the size difference between the current chunk and the next chunk is constant. As a result, the size of the current chunk can be determined simply by knowing how many chunks have already been allocated. Thus, it can be implemented by using a fetch-and-add operation to a shared variable that keeps track of the number of chunks assigned. At compile-time, or at the start of a loop's execution, the step size between chunks is calculated. Then the shared variable is initialized to count the number of chunks allocated. When a processor needs work, it fetch-and-adds the shared variable and uses this value with the step size to

calculate its share of work [18]. The calculation of the current chunk size can be done in parallel eliminating the need for a critical section. Therefore, t_{tss} is equal to t_{cs} and t_{ss}.

Eliminating the critical section and utilizing fetch-and-add operations to minimize the overhead in calculating the chunk sizes and obtaining the loop iterations are two of the many ways to improve the performance of a scheduling algorithm. As discussed in the previous section, self-scheduling requires so many scheduling steps that even if it uses efficient operations in obtaining work, the contention in the network and memory become a major problem.

From Table 1, we see that chunk scheduling requires the fewest scheduling steps while self-scheduling requires the most. Factoring requires slightly more steps than guided self-scheduling. Trapezoid self-scheduling requires only $4P$ steps which is fewer than that of GSS and FS when the iterations-to-processors ratio is larger than 55 and 16, respectively. Figure 3 shows the size of the chunk versus the scheduling step for chunk scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling when the number of processor, P, is 4 and the number of iterations, N, is 1000. Combining both the scheduling overhead per step and the number of steps, chunk scheduling is the most efficient algorithm, while TSS is the second.

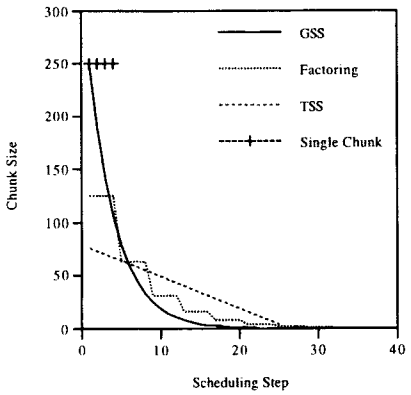


Figure 3. Chunk size at each scheduling step for the scheduling algorithms.

Although chunk scheduling is an efficient algorithm requiring only a small overhead, it may not balance the workload very well, which can cause a large δ_{cs}. Recall that we defined δ to be the overhead caused by load imbalance, network and memory contention, and other unpredictable system events. Self-scheduling also has a large δ_{ss} because of the contention it generates on the memory and the network. There is no simple way to quantify δ for GSS, FS, and TSS. Therefore, we have developed a probabilistic simulation to estimate the sensitivity of the algorithms to the variance in iteration execution times.

The simulator mimics a 16-processor shared-memory system at the clock cycle level. At each processor cycle, the simulator assigns iterations to the idle processors according to the selected scheduling algorithm. The execution times of the iterations are generated by a random number generator with a normal (Gaussian) distribution. The mean of the iteration executions times is 100 units and the variance is changed from 5 to 70 units. The overhead for allocating a chunk of iterations to a processor is 10 units. When two processors try to obtain work at the same time, one must wait until the other is done. The waiting time is the same as the scheduling overhead for the first processor, which is 10 units of time. This delay simulates the contention from accessing the shared variable. We simulated the performance of the algorithms for $N = 500$ and $N = 5000$. The results are shown in Figure 4.

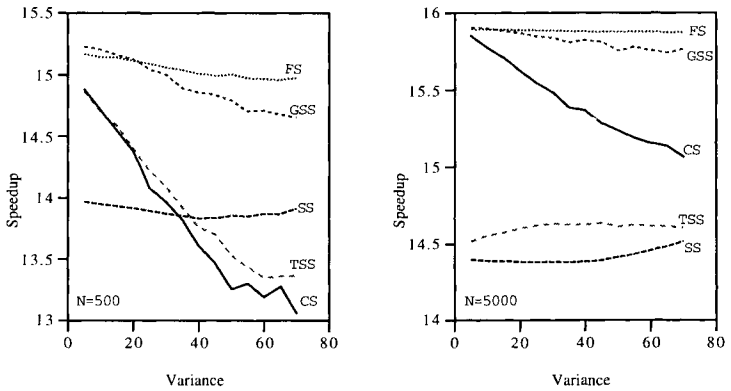


Figure 4. Comparison of scheduling algorithms as a function of iteration execution time variance.

These results confirm that the overhead and contention generated by self-scheduling limits its performance, but it is insensitive to changes in the variance. In fact, it performs better when the variance is large than when it is small. On the other hand, the performance of chunk scheduling (CS) and trapezoid self-scheduling (TSS) diminishes as the variance increases. When the number of iterations, N, is small and the variance is large, they perform worse than self-scheduling as the effect of load imbalance is more significant than the effect of the large scheduling overhead.

Although the speedup value for guided self-scheduling (GSS) also decreases as the variance increases, it decreases at a slower rate than CS and TSS. In general, GSS performs better than SS, CS, and TSS. However, it does not balance the workload as well as factoring when the variance is large. Since GSS allocates too many iterations at the beginning of a loop's execution, it does not have enough small chunks remaining to balance the load at the end.

Factoring (FS) maintains its performance even when the iteration execution time variance is large. However, when $N = 500$ and the variance is small, its performance is slightly below GSS's because of its extra scheduling steps.

Another observation based on these two graphs is that these algorithms require a large number of iterations to efficiently balance the workload. When $N = 500$, the speedup values for the algorithms decrease as the iteration execution time variance increases since there are not enough iterations to compensate for the large unevenness. When N is large, factoring maintains performance very well. GSS's speedup value decreases at a slower rate, and, most significantly, TSS improves its performance when the variance is large.

A note on the simulation is that we used a constant scheduling overhead time for all of the algorithms. As discussed earlier, some algorithms can be implemented more efficiently than the others, but the simulation results do not reflect this factor. To take that into consideration, we measured the performance of these algorithms on an actual machine. These results are presented in the next subsection.

4.2. Experimental Results

The simulations presented in the previous subsection showed how the overall performance of the algorithms compared. However, the simulations are very primitive and use some simplifying assumptions about the system architecture and implementation details. To further investigate the scheduling algorithms, we implemented them on an actual system and measured their performance while executing some real loops.

Our target machine is a Silicon Graphics Onyx multiprocessor system [8]. The system is based on the POWERpath-2 RISC system architecture, which uses the MIPS-designed R4400 processor with an R4010 Floating Point Chip as the math coprocessor. The R4400 is a 64-bit microprocessor with an eight-stage superpipelined architecture. It has 32 general-purpose registers and 32 floating-point registers.

The system in which the performance is measured has twenty processing elements (PEs) executing at a clock rate of 150MHz. Each PE has a 16-Kbyte data cache, a 16-Kbyte instruction cache, and a 1-Mbyte secondary unified instruction and data cache. The PEs are connected to the main memory through a single shared bus. The main memory is 512 Mbytes in size, and is four-way interleaved. The SGI Onyx system is running the IRIX operating system, version 5.2-ALPHA. This operating system uses a modified implementation of the UNIX preemptive multitasking scheduler to support the execution of multiple processes during each time slice.

The scheduling algorithms are implemented using the standard system routines. At the beginning of a loop, the parent process calls the `fork` routine to *spawn* a number of subprocesses as specified by the user. These subprocesses are then executed on separate processors. When a subprocess is idle and needs work, it calls the `lock` routine to request access to the shared work queue. After the subprocess obtains the lock, it accesses the loop index and calculates its share of work in the critical section based on the scheduling algorithm. Then it releases the lock and starts the execution. When all the work is completed, the subprocesses call the `barrier` routine to wait for the unfinished subprocesses. After every subprocess is done, the `join` routine is called and the subprocesses are destroyed. Then the parent process continues to execute the serial section of the program. The outline of this implementation is shown below:

```
fork(P); /* start P subprocesses */

lock;    /* determine its share of work */
/* calculate chunk_size based on the scheduling strategy */
start_i = next_iter;
end_i = next_iter + chunk_size;
next_iter = end_i + 1;
unlock;

while (not done) do
   do i = start_i, end_i
      /*  execute loop iterations from start_i to end_i */
      {loop body}
   end do

   lock;  /* get more work */
   /* calculate chunk_size based on the scheduling strategy */
   start_i = next_iter;
   end_i = next_iter + chunk_size;
   next_iter = end_i + 1;
   unlock;
end while

barrier;   /* wait for others to finish */
join(P);
```

Since the Onyx system does not provide a hardware fetch-and-add operation, we cannot employ the most efficient implementations for the algorithms discussed previously. Therefore, we implemented all the scheduling algorithms using the above pseudo-code with only lock and unlock operations.

Using this implementation, we measured the performance of the scheduling algorithms on two different types of loops. The first loop is a matrix multiplication program which is parallelized on the outer loop. The size of the parallel tasks is large, and the variance in iteration execution times is small.

```
doall i = 1, N
  do j = 1, N
     do k = 1, N
        C(i,j) = C(i,j) + A(j,k) * b(k,j)
     end do
  end do
end doall
```

The second loop we used is based on the adjoint-convolution process. It is parallelized on the outer loop and, since the number of iterations in the inner loop depends on the index of the outer loop, the variance in iteration execution times is large.

```
doall i = 1, N*N
  do j = i, N*N
     A(i) = A(i) + X*B(j)*C(i - j)
  end do
end doall
```

Figures 5 and 6 show the results of our measurements of these two programs. The figures do not include the performance of self-scheduling because it performs poorly on our system. Since the Onyx system does not have a built-in fetch-and-add operation, we had to lock and unlock the shared variable for every iteration causing the overhead to overwhelm any possible performance gain of executing the loop in parallel with self-scheduling.

Besides chunk scheduling, self-scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling, we also measured the performance of static chunk scheduling. Static chunk scheduling is similar to chunk scheduling except the iterations are assigned at compile-time. Therefore, static chunk scheduling does not require the processors to determine their share of work in a critical section. By comparing the performance of static chunk scheduling to that of dynamic chunk scheduling, we can estimate the overhead required to access the shared variable.

For the matrix multiplication program, all the iterations have the same number of operations so that the variance in iteration execution times is small. When the number of iterations, N, is small, static CS and dynamic CS perform better than the others, as shown in Figure 5, since both algorithms require minimum scheduling overhead. Static CS performs slightly better than dynamic CS because it eliminates the overhead in accessing the shared variable.

For large N, all algorithms perform similarly. When the problem size is large, the chance that the execution is interrupted by system events is higher, so algorithms that use variable chunk sizes can effectively balance this small unevenness. Since the benchmark program does not have a large variance, the effect of load imbalance is not significant.

Figure 5 also shows that when N is small, the speedup values for the algorithms leveled off when P is about 10. If the number of processors is further increased, the speedup actually decreases. This phenomenon is caused by the the overhead of creating subprocesses. At the beginning of the parallel loop, the parent process creates the children processes sequentially. When N is small, all the work might be completed, or at least assigned, *before* the last few processes are created. As a result, those processes remain idle. When N and P are large, the performance starts degrading due to the contention of accessing the shared variable. Since the variance in iteration execution times is small, the chance of the processors finishing their share of work and trying to access the critical section at the same time is large.

Figure 6 shows the results for the algorithms when executing the adjoint convolution program. Since each parallel iteration has a different number of operations, the variance in iteration execution times is large. Static CS and dynamic CS use a fixed chunk size and, thus, the first few chunks have much more work to do than the last few chunks. As a result, these algorithms produce only half of the possible speedup. GSS also does not perform well as it assigns too much work at the beginning of the execution and does not save enough work at the end for balancing the load. FS and TSS balance the workload

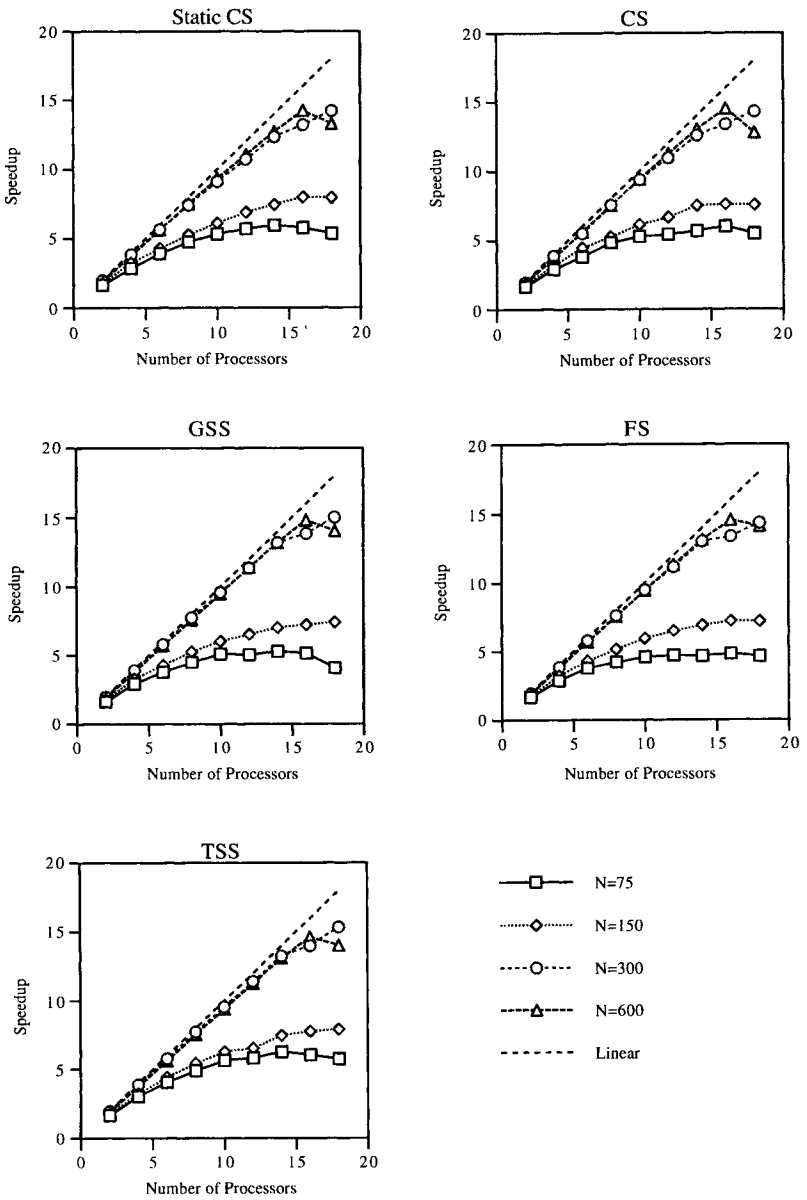


Figure 5. Performance of scheduling algorithms on matrix multiplication.

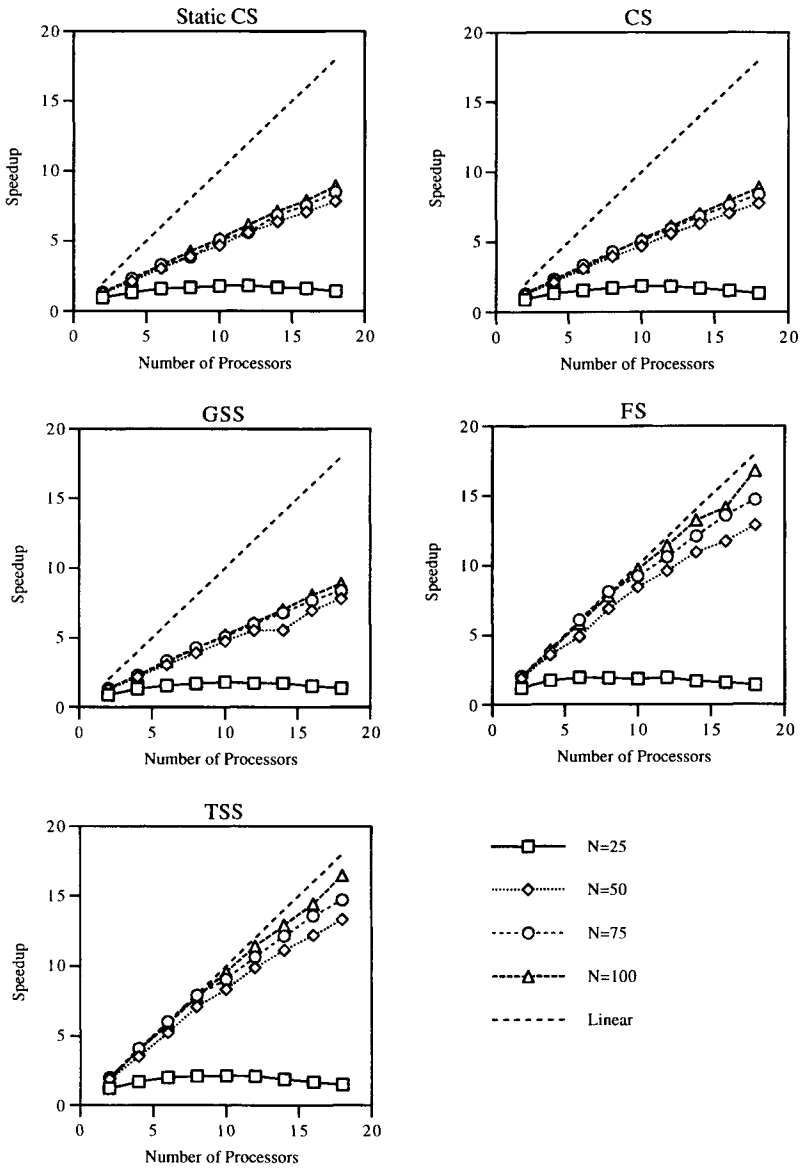


Figure 6. Performance of scheduling algorithms on adjoint convolution.

better than the others. The results also suggest that none of the algorithms perform well when N is small since there are not enough iterations to balance the overhead when executing the loop in parallel.

Another observation from Figure 6 is that the speedup does not degrade as the number of processors increases. Since the variance in iteration execution times is large, the probability of the processors accessing the critical section at the same time is small. Therefore, the contention is small and the performance does not degrade even when the number of iterations and the number of processors is large.

5. CURRENT RESEARCH

The previous sections have reviewed current parallel loop scheduling algorithms for shared memory multiprocessor systems. The performance of chunk scheduling, self-scheduling, guided self-scheduling, factoring, and trapezoid self-scheduling has been studied in detail. Performance was compared using analytical models, simulations, and actual experiments performed on an SGI Onyx multiprocessor system. Based on these results, we conclude that the performance of the scheduling algorithms depends not only on the characteristics of the loops, but also on the system architecture and the specific implementation details of the algorithms. In this section, we briefly discuss our current research towards improving the performance of parallel loop scheduling algorithms.

5.1. Generalized Parallel Loop Scheduling Algorithm

The algorithms reviewed in this article typically have been developed using statistical analyses with simplified assumptions or using trial-and-error experiments. As a result, they do not perform well in cases where the assumptions are not satisfied. The performance of an algorithm depends not only on the loop characteristics, but also on the system architecture and unpredictable system events, such as network and memory contention. Statistical analysis or trial-and-error methods might not be adequate to consider all these factors when developing a scheduling algorithm. Therefore, we have proposed a new technique for developing parallel loop scheduling algorithms which allows us to *invent* new algorithms, or to verify the existing ones, in a quick and systematic manner [19].

Our technique involves two main phases. In the first phase, we examined the strategies used by current algorithms and generalized them into a set of parameterized equations. This generalized scheduling algorithm is then used as a framework to develop new algorithms using a simulator that employs the Genetic Algorithm. Based on the loop characteristics, system architecture, and other factors we want to consider, the simulator estimates different parameter values in the generalized algorithm. Two new scheduling algorithms, CS-2 and FS-alt, were discovered using this technique.

The first algorithm we discovered, CS-2, is a hybridization of chunk scheduling and self scheduling. Each processor is assigned a chunk of $\lceil N/P - 2 \rceil$ iterations, while saving $2P$ iterations to later balance the workload. This strategy requires very small scheduling overhead and works very well for loops with small variance in iteration execution times, as shown in Figure 7.

For loops with large iteration execution times variances, a decreaing chunk size strategy, such as factoring, works better. Our algorithm, FS-alt, improves factoring by using a smaller factor, $x = 6/5$ instead of 2, and a smaller number of chunks per batch, $P/2$

instead of P. As a result, FS-alt balances the workload as well as factoring while requiring fewer scheduling steps. Figure 7 shows the performance of our new algorithms compared to the current algorithms on a 16-processor system with $N = 500$ iterations.

5.2. Scheduling for Multiprogrammed Multiprocessors

Another issue in multiprocessor scheduling that we are investigating is how to ensure high-performance for an application while maintaining high system utilization in a multiprogrammed multiprocessor system. Many of today's multiprocessor systems are multiprogrammed in which several independent applications share the system and compete for system resources. Currently the most widely used scheme to share the processors among the applications is an extension of the time-sharing scheme used by the Unix operating system. However, several studies have shown that this approach might cause the applications' performance to degrade drastically due to the overhead caused by context switching, poor cache utilization, and process thrashing [15,17].

Techniques such as gang scheduling [15] and dynamic processor allocation [17] have been proposed to improve both the application performance and the system utilization. However, these techniques usually require either a modification of the operating system or a special programming model. Our proposed strategy works at the loop level and has each application determine the number of processors available based on the system load at the beginning of each parallel section. This approach allows the application to fully utilizate the idle processors when the system is not busy. When the system is busy, however, the applications share the system while minimizing context switching overhead. Another advantage of our scheme is that it works with a standard *fork-join* parallel programming model and does not require any operating system modification. Consequently, it is compatible with existing systems and is portable.

Figure 8 shows the preliminary result of our strategy based on an eight-processor multiprogrammed shared-memory system simulator. We compared the speedup values of four applications from the Perfect Fortran benchmarks while they were executing 1) on a dedicated system, 2) on a shared system where each application is allocated eight time-shared processors, 3) on a evenly shared system (i.e. each application is allocated two processors), and 4) using our dynamic allocation scheme. As shown in the graph, our strategy produces the highest speedup for each application when executing in the multiprogrammed environment and maintains 67 to 97% of the performance obtained when running on a dedicated machine.

6. CONCLUSION

This article has reviewed current parallel loop scheduling algorithms for shared memory multiprocessor systems and has compared their performance. It has also briefly described a generalized scheduling algorithm to ease the task of finding a parallel loop scheduling algorithm based on the loop characteristics and the system architecture. We also discussed the effect of a multiprogrammed environment and proposed a solution to produce high-performance for each individual application while obtaining high system utilization.

Another issue in parallel loop scheduling that we did not discuss is the effect of data locality. Scheduling iterations without considering the locality of the data may cause false sharing on some systems which can result in poor performance[12]. *Affinity scheduling*

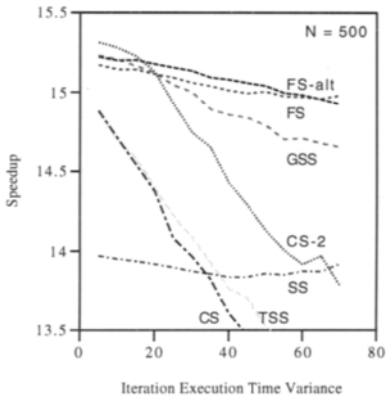


Figure 7. Performance of the generalized scheduling algorithm.

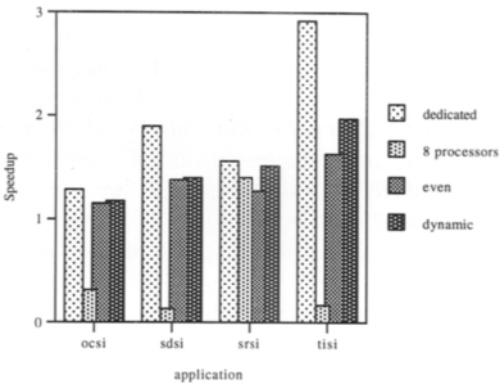


Figure 8. Performance comparison between different multiprogrammed scheduling schemes.

[13] and *self-adjusting scheduling* [6] are examples of algorithms that try to maintain both load balance and data locality while minimizing scheduling overhead. Their performance has yet to be studied in detail.

REFERENCES

1. C. Beckmann and C. Polychronopoulos. The effect of scheduling and synchronization overhead on parallel loop performance. In *Int'l Conf. on Parallel Processing*, volume II, pages 200–204, 1989.
2. R. Cytron. Doacross: beyond vectorization for multiprocessors (extended abstract). In *Int'l Conf. on Parallel Processing*, pages 836–844, 1986.
3. Z. Fang, P. Yew, P. Tang, and C. Zhu. Dynamic processor self-scheduling for general parallel nested loops. In *Int'l Conf. on Parallel Processing*, pages 1–10, Aug. 1987.
4. J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, 30(7):478 – 490, Jul. 1981.
5. A. Grama, A. Gupta, and V. Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, pages 12–21, 1993.
6. B. Hamidzadeh and D. Lilja. Self-adjusting scheduling: an on-line optimization technique for locality management and load balancing. In *Int'l Conf. on Parallel Processing*, volume II, pages 39–46, Aug. 1994.
7. S. Hummel, E. Schonberg, and L. Flynn. Factoring: A method for scheduling parallel loops. *Communciations of the ACM*, 35(8):90– 101, Aug. 1992.
8. Silicon Graphics Inc. Symmetric multiprocessing systems. Technical report, Mountain

View, CA 94043, 1993.

9. C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. on Software Engineering*, pages 1001–1016, Oct. 1985.
10. V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379–391, Sep. 1994.
11. M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *SIGPLAN 88 Conf. on Programming Language Design and Implementation*, pages 318–328. ACM, 1988.
12. D. Lilja. Exploiting the parallelism available in loops. *Computer*, 27(2):13 – 26, Feb. 1994.
13. E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 5(4):379–400, April 1994.
14. A. Nicolau and J. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. on Computers*, C-33(11):968–976, Nov. 1984.
15. J. Ousterhout. Scheduling techniques for concurrent systems. In *Distributed Computing Systems Conference*, pages 22–30, 1982.
16. C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Computers*, C-36(12):1425 – 1439, Dec. 1987.
17. A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *12th ACM Symposium on Operating System Principles*, pages 159–166, 1989.
18. T. Tzen and L. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–97, Jan. 1993.
19. K. Yue and D. Lilja. Parameter estimation for a generalized parallel loop scheduling algorithm. In *28th Hawaii Int'l Conf. on System Sciences*, volume II, pages 187 – 188, 1995. Also available as Technical Report TR-94-51, Dept. of Computer Science, U. of Minnesota, Minneapolis.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

The MPI Message-Passing Interface Standard: Overview and Status

William Gropp and Ewing Lusk[a*]

[a]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439, USA

MPI (Message-Passing Interface) has been developed over the last two years as a standard message-passing interface specification. This paper summarizes what MPI is, describes recent activities, particularly MPI implementation activities, and supplies sources for further information about MPI.

1. INTRODUCTION

During 1993 and 1994, the MPI Forum, a group of parallel computer vendors, library writers, and applications scientists (see Table 1, met regularly to develop a standard message-passing library interface definition. The objective was to agree on a single interface that would serve the needs of compiler writers, library writers, and application programmers, and provide a single library interface to be used on parallel supercomputers, networks of workstations, and large-scale, high-performance networks of computers of all kinds.

The MPI Standard [1,2,4], was finalized in May of 1994, and and is available from a number of sources. In this article we give an overview of MPI and describe implementation activities since the publication of the Standard.

2. WHAT IS MPI?

MPI is a *message passing library specification.* That is, it addresses the *message-passing* model of parallel computation, in which processes with separate address spaces communicate by sending and receiving messages. It is a *specification*, not a product or particular implementation. It specifies a *subroutine library* interface, in which the library routines are called from conventional C and Fortran programs; it is not a new parallel language.

3. MPI FEATURES

MPI has its roots in conventional message-passing libraries. It thus contains most send/receive features of popular vendor-supplied and portable libraries. It has gone beyond most of them, however, in its inclusion of several advanced features.

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Companies	Laboratories	Universities
ARCO	ANL	UC Santa Barbara
Convex	GMD	Syracuse U
Cray Res	LANL	Michigan State U
IBM	LLNL	Oregon Grad Inst
Intel	NOAA	U of New Mexico
KAI	NSF	Miss. State U.
Meiko	ORNL	U of Southampton
NAG	PNL	U of Colorado
nCUBE	Sandia	Yale U
ParaSoft	SDSC	U of Tennessee
Shell	SRC	U of Maryland
TMC		Western Mich U
		U of Edinburgh
		Cornell U.
		Rice U.
		U of San Francisco

Table 1
MPI Forum Participants

- A rich set of communication *modes*, to provide access to high-performance and user control of message buffering,
- Communication *contexts* to isolate layers of the communication fabric to provide security,
- Process *subgroups* for management of the process space, and virtual process topologies for organizing them in convenient application-oriented ways,
- *Communicators*, which encapsulate contexts and process groups to provide the modularity needed by parallel libraries,
- A rich set of global operations and data movement routines which can be applied on a communicator basis,
- Built-in and user-defined *datatypes* for all message buffers, which provide direct access to non-contiguous buffers on machines with advanced communication processors, and allow translation for heterogeneous communication,
- A mechanism for users to intercept MPI calls for the purpose of installing their own debugging or performance-analysis mechanisms without access to the MPI source code,
- Thread safety.

4. MPI IMPLEMENTATIONS

Recently a workshop was held at Argonne National Laboratory to facilitate communication among those currently engaged in MPI implementation projects. Many implementations are under way, as summarized in Table 2.

Vendor Implementations
IBM Research (MPI-F)
IBM Kingston
Intel SSD
Cray Research
Meiko, Inc.
Kendall Square Research
NEC
Fujitsu (AP 1000)
Convex
Hughes Aircraft
Portable Implementations
Argonne–Mississippi State (MPICH)
Ohio supercomputer Center (LAM)
University of Edinburgh
Technical University of Munich
University of Illinois

Table 2
Current MPI implementations

Other interested hardware and software companies who were represented at the Workshop were Sun, Hewlett-Packard, Myricom (makers of high-performance network switches) and PALLAS (a German software company). Sandia National Laboratory plans an implementation for their Intel Paragon running SUNMOS.

5. MPICH–A PORTABLE IMPLEMENTATION

One particular portable, freely available MPI implementation is MPICH, developed jointly by Argonne National Laboratory and Mississippi State University. It contains a number of unique features:

- It is the only one of the public implementations to focus on achieving performance comparable to that of proprietary systems on parallel supercomputers.

- It is a complete implementation of MPI, including heterogeneous communication (not using XDR).

- It is portable to a wide variety of parallel computers (IBM SP1 and SP2, Thinking Machines' CM-5, Intel IPSC860, Touchstone Delta, and Paragon, Meiko CS-2 and Ncube) and networks of workstations (Sun, IBM, SGI, Hewlett-Packard, DEC).
- It comes with a convenient programming environment including a portable startup mechanism, several profiling libraries, a parallel graphics library, and the `upshot` performance visualization tool.

Detailed information about `mpich` can be found in the *MPICH User's Guide*, available at `http://www.mcs.anl.gov/mpi`.

6. Further Information about MPI

Material about MPI is available from a variety of sources. Some of these, particularly WWW pages, include pointers to other resources.

- The Standard itself:
  - As a Technical report: U. of T. report [1]
  - As postscript for ftp: at `info.mcs.anl.gov` in `pub/mpi/mpi-report.ps`.
  - As hypertext on the World Wide Web: `http://www.mcs.anl.gov/mpi`
  - As a journal article: in the Fall issue of the Journal of Supercomputing Applications [4]
- MPI Forum discussions
  - The MPI Forum email discussions and both current and earlier versions of the Standard are available from `netlib`.
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum [3].
  - *MPI Annotated Reference Manual*, by Otto, et al., in preparation.
- Newsgroup:
  - `comp.parallel.mpi`
- Mailing lists:
  - `mpi-comm@cs.utk.edu`: the MPI Forum discussion list.
  - `mpi-impl@mcs.anl.gov`: the implementors' discussion list.
  - `mpi-bugs@mcs.anl.gov`: the address to report problems with `mpich` to.
- Implementations available by `ftp`:

  - MPICH is available by anonymous `ftp` from `info.mcs.anl.gov` in the directory `pub/mpi/mpich`, file `mpich.*.tar.Z`.
  - LAM is available by anonymous `ftp` from `bag.osc.edu` in the directory `pub/lam`.
  - The CHIMP version of MPI is available by anonymous `ftp` from `ftp.epcc.ed.ac.uk` in the directory `pub/chimp/release`.
  - The Fujitsu AP1000 version is available from `dcssoft.anu.edu.au`.

- Test code repository (new):
  - `ftp//:info.mcs.anl.gov/pub/mpi-test`

7. CONCLUSIONS

When the MPI Forum was creating the specification for MPI, there was some concern about the level of interest on the part of vendors and other implementors in providing robust versions of it so that application programmers and library writers could reap the benefits. One obvious conclusion from observing recent MPI-related activities is that both vendors and others have found MPI to be important and useful enough, and enough of an advance over existing message-passing systems, to invest substantial effort in producing high-quality implementations. It now seems clear that within a year there will a number of vendor implementations of MPI to choose from, fully supported by tools. In the meantime, complete, public implementations offer the opportunity for both library writers and applications scientists to begin using MPI now on many different parallel computing environments.

REFERENCES

1. Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, 1994.
2. The MPI Forum. MPI: A message-passing interface standard. World Wide Web. `http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html`.
3. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.
4. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Software Engineering Considerations in the Construction of Parallel Programs*

Jonathan Schaeffer and Duane Szafron

Department of Computing Science, University of Alberta, Edmonton, Alberta, T6G 2H1, CANADA

In many papers describing parallel programming tools, the authors illustrate the strengths of their approach by presenting some impressive speedup results. However, is this the only metric by which we should judge the quality of their tool? Many of these tools offer significant software engineering advantages that reduce program development time and increase code reliability. This paper uses the Enterprise programming environment for coarse-grained parallel applications to illustrate the advantages of these tools. For most users, high performance is not an important evaluation criteria; other criteria, such as tool usability and program development savings, are often far more important.

1. INTRODUCTION

All too often in the literature, the only evaluation metric given for the implementation of a parallel algorithm is the ubiquitous speedup. Speedup only measures the benefits of parallelization at execution time, and completely ignores the software development investment required to achieve this result. Parallel programming adds an extra dimension of complexity to software design, development, testing and debugging. This is a real cost that must be factored into any decision to allocate programming resources to building parallel solutions. A better measure of the effectiveness of a parallel implementation might be the *Utility*† :

Utility = F(S, SD, PD, I)

where

S: execution speedup achieved,
SD: total sequential program development time of the program,
PD: total parallel program development time,
I: importance of achieving the speedup, and
F a function that maps its arguments into a cost measure.

Utility augments the speedup metric by considering the importance of achieving a good speedup and the development resources required to achieve this. For example, code which is run frequently or for long periods of time, has a higher *I* value than programs that are run once. Essentially, *Utility* is a cost-benefit measure, reflecting the cost of human and computer resources, versus the benefits of a faster program.

* This research has been funded in part by NSERC grants OGP-8173 and OGP-8191, a grant from IBM Canada Limited and the Netherlands Organization for Scientific Research (NWO).

† A modified definition from a personal communication with Greg Wilson.

In the frequently occurring case of legacy code, where the sequential code already exists, SD is effectively 0, and PD is the additional effort required to convert the program to run in parallel. Consider a sequential legacy program that executes for a month when it is run. A programmer may convert this code to run on a network of workstations using a low-level tool (such as PVM [11]). It may take one month of programmer time to create a parallel version of the program that now takes only a day to execute. Another programmer may tackle the same problem using a high-level tool (such as Enterprise [19]). What if his solution takes only a week to develop, but the program takes 1.5 days to execute? Which is the better solution? Programmer time costs real dollars; compute cycles are often free or inexpensive. We argue that a high-level parallel programming tool allows the user to develop solutions quickly, freeing up programmer resources for other tasks. In the case of Enterprise, *if* the execution time is not acceptable, you can still use the tool to generate a correct parallel program, and then edit the generated code to provide your own application-specific enhancements.

This paper discusses the software engineering advantages of using the high-level parallel programming tool Enterprise for developing applications to run on a network of workstations. Enterprise inserts code into a program to handle all the parallel considerations, reducing the programming time required to turn a sequential application into a parallel one. This allows the user to concentrate on the domain-dependent code which the user understands best. Enterprise produces a correct parallel program structure that will often achieve performance close to that of a hand-crafted solution. In other words, there is a trade-off: better software engineering and shorter development time for (possibly) slower execution performance.

There is a community that demands high performance from their applications, but this community represent a small percentage of the potential user community for parallel computing technology†. With the increased availability of relatively low-cost multi-processor machines and the proliferation of networked single-processor machines, more people will be tempted to test-drive parallel software technology. However, these people will usually not demand high performance. A parallel tool that allowed them to quickly achieve improved throughput would be welcome, even if the performance was not as high as might be achieved through significantly increased programming effort.

In the past decade, there has been significant progress in developing tools to ease the pain of programming coarse-grained parallel applications. These tools adopt a variety of approaches to add parallelism to existing languages (primarily C and Fortran) such as providing library calls (e.g. PVM [11] and Linda [6]), loop parallelism (e.g. Myrias PAMS [14]), extending a language (e.g. HPF [16]) and expressing the parallelism graphically (e.g. Hence [4] and Enterprise [19]). More radical approaches include defining a new procedural language that supports parallelism (e.g. Orca [3]) or alternative programming paradigms (such as a functional language like Sisal [9]). However, given the large investment in C and Fortran software, we see these latter approaches as not being practical in the short-term. All of these approaches, referred to as parallel programming systems (PPS), offer software engineering advantages to the software developer. This paper discusses Enterprise because it offers us greater scope for illustrating many of the potential advantages of using higher-level tools. Enterprise is more than just a parallel programming model, it is also a software development environment.

This paper is not an introduction to Enterprise (see [19]), nor is it intended to be a sales presentation. Instead we concentrate on some of the major advantages of using a high-level parallel programming model and a complete integrated parallel programming environment. These advantages are available in a variety of tools and, rather than turn this paper into a literature survey, we concentrate on one representative system, Enterprise, and give a few sample references to other systems. We argue that from the software engineering point of view, high-level parallel programming tools can considerably reduce program development

† However, they do represent a large percentage of the parallel cycles used in the world.

time and increase the reliability of the resulting code. In many cases, these advantages far outweigh the benefits obtained from a faster hand-coded solution.

2. PROGRAMMING: COMMUNICATION

Multiple agents, all working together towards a common goal, must have a means of communication. In distributed systems, the method of communication is message passing. The traditional programmer's view of message passing involves four steps:

1. pack the data,
2. send the message,
3. receive the message and
4. unpack the data.

In some systems, such as PVM, these steps must be explicitly programmed in the user's code. Alternative systems, such as Sun RPC [22], only require the user to provide packing and unpacking routines. Both of these low-level approaches require the user to write additional lines of code that only increase the probability of introducing an error. In contrast, Linda handles the data transparently, but expects the programmer to explicitly call routines to do the communication [6]. Hence [4] allows the user to express this information graphically, but parameter information must be specified both in the interface and the code (causing a redundancy problem, see Section 6). Concert/C eliminates the packing/unpacking routines by having the user provide additional type information describing the data [12]. In all these cases, the higher-level tools can significantly reduce the programming effort.

One approach to message passing is to make this form of communication look like procedure calls that happen to be executed remotely (but without the programming effort and synchronous semantics of RPC). Ideally, the user would write the code without any knowledge of whether the procedure was to be executed in parallel or sequentially. In practice, this is not possible with a distributed memory; there are no globally shared variables unless additional facilities are added to the language (Orca is an example of a language that supports shared objects in a distributed environment [3]). However, modern software engineering practices limit the use of global variables anyway and procedures that use them can be rewritten using extra parameters.

For example, let's assume the procedure `Callee()` is to be executed in parallel with `Caller()` (a technique for specifying this is discussed in Section 4):

```
Caller()
{
   char a;
   USER_TYPE b;
   double d[100];
   int dnumb;
   int result[100];
   ...
   result[5] = Callee( a, b, &d[10], dnumb );
   ...
}

int Callee( char a, USER_TYPE b, double * d, int dnumb )
{
   ...
}
```

Since Enterprise includes a pre-compiler, we have access to symbol table information and can determine the size of data. Hence, Enterprise can automatically generate the code to marshal the parameters of `Callee()` into a message, rather than forcing the user to do this explicitly. In effect, the call to `Callee()` looks like sequential code, but may in fact be executed remotely. The consequence of this approach is that parallel code looks like ordinary sequential code.

Note that because of weak typing in C, it is not possible for the compiler to figure out all data typing information at compile-time. An example is the `d` pointer above. Although in this example, the compiler can compute the number of elements pointed to by `&d[10]` (100 - 10 = 90 elements), in general it may not be able to compute the number of elements since a pointer can be coerced to point to anything. Enterprise solves this problem by requiring each pointer parameter to be followed by an additional parameter that specifies the number of elements to pass (`dnumb` in the example). This approach is analogous to Fortran code, where the user must pass an array and the size of the array as parameters. Concert/C overcomes many of the C type ambiguity problems by having the user provide additional type information describing the data [12]. If a PPS wants to handle passing an arbitrary C data structure between processes, there appears to be no alternative than to follow the Concert/C approach.

By letting the compiler generate the code to do data packing and unpacking, the probability of programming errors is greatly reduced.

3. PROGRAMMING: SYNCHRONIZATION

There are a variety of ways that PPSs can introduce parallelism. A common method is to use compiler technology to automatically detect loops that can be executed in parallel. While this has been successful for detecting fine-grained parallelism, it seems to have limited utility for coarse-grained applications. Alternatively, some systems use a keyword, such as *pardo* [14], to explicitly tell the system which loop iterations (or which sections of independent code) should be executed in parallel. Both methods require users to reorganize code to make loop iterations independent of each other. However, these approaches have the advantage that the PPS does most of the work. In contrast, using library calls to introduce parallelism puts the onus solely on the shoulders of the programmer (PVM and a variety of *fork/join* models, such as [18]).

In Enterprise, designated functions can be executed in parallel. This means that users express parallelism by the way they write their functions, rather than the way they reorganize their loops. However, if `Callee()` is designated as a parallel call in the `Caller()` statement

```
result[5] = Callee( a, b, &d[10], dnumb );
```

it appears to have no concurrency. Even though `Callee()` may be invoked in parallel, it appears that `Caller()` must wait for the return value to set `result[5]` correctly. In Enterprise, *futures* are used to increase concurrency during parallel calls [13]. Enterprise relaxes the semantics of a parallel call to allow the calling routine to continue executing concurrently with the call, until it needs the result. For example, in this code fragment:

```
sum = 0;                                                    (1)
for( i = 0; i < 100; i++ )                                  (2)
   result[i] = Callee( a, b, &d[i], 1 );                    (3)
for( i = 0; i < 100; i++ )                                  (4)
   sum = sum + result[i];                                   (5)
```

`Caller()` is allowed to continue executing after each call to `Callee()` on line (3) and does not block until at least line (5). On line (5), when `result[i]` is accessed, `Caller()` blocks only if `Callee()` has not yet returned with the new value of `result[i]`. Thus, the semantics of sequential C are preserved. When pointers are used as arguments to parallel calls for the purpose of side effects, Enterprise treats them as return values. Enterprise provides optional macros for improving execution efficiency to specify that the data passed to a routine need not actually be copied back.

Futures are the synchronization tool of Enterprise. `Caller()` continues to execute and only blocks when it accesses a result that has not yet returned. This allows the user to increase concurrency by moving code around to keep `Caller()` busy doing other things until `Callee()` returns. With this model, the programmer does not need to specify any explicit synchronization point in their code. However, users have the responsibility of organizing their code into parallel functions, and generating enough futures to keep the computational resources busy.

A second technique is used to increase concurrency. In a situation where a series of calls are made to the same parallel function and the order of the results is not critical, the function may be declared to be *unordered*. This attribute is set in the user interface and is independent of the user's code. Unordered semantics means that the first reference to a particular return value for an asset will receive the first corresponding parameter value returned, regardless of whether the variable name matches or not. The second value returned will be placed in the second reference, etc. For example, in the previous code fragment, when statement (5) is executed, the program may have to block and wait for `result[0]`. However, other results may have already been returned, say `result[1]` and `result[4]`. Since the value of sum is independent of the order of summation of the results, more concurrency can be obtained by using `result[1]` or `result[4]` in place of `result[0]` and using `result[0]` later in place of another result. It is important to realize that the unordered option violates the semantics of sequential C, but does allow additional concurrency in some computations.

In Enterprise, great care has been taken to avoid explicit references to parallelism in the code. The goal is that the user should write code that will execute sequentially or in parallel, with no changes. For example, consider the parallel divide-and-conquer code of Figure 1. This code multiplies the coefficients of two polynomials together to produce the coefficients of the product. If `Mult()` is to be executed recursively in parallel, then in many systems the user must specify some condition for switching between parallel and sequential recursion, such as:

```
if( N >= threshold )
        Parallel_Mult( ... );
else Sequential_Mult( ... );
```

Defining `threshold` can be tricky since it is a function of the hardware used and must be changed when the execution environment changes. In Enterprise, each executable binary is capable of executing any parallel routine sequentially. Currently, the depth of the parallel recursion (the point where the code switches between parallel and sequential execution) is expressed graphically and is independent of the code generated (see next section). However, there is on-going research on having the system dynamically decide when to switch between parallel and sequential execution, based on run-time statistics. Again, the goal is to reduce the onus on the programmer.

```
/* Multiply two polynomials together, with coefficients in Pointer1 and */
/* Pointer2 arrays, and put the product coefficients in the Answer-     */
/* Pointer array.                                                       */

Mult( Pointer1, Pointer2, N, AnswerPointer )
{
      localvars Result1, Result2, Result3, Cross1, Cross2;

      if( N == 1 ) {
            AnswerPointer[0] = Pointer1[0] * Pointer2[0];
      } else {
            /* Multiply the low and high order terms */
            Mult(  Pointer1,       Pointer2,      N/2, Result1 );
            Mult( &Pointer1[N/2], &Pointer2[N/2], N/2, Result2 );

            /* Low and high crossover terms */
            Cross1 = CrossOverTerms( Pointer1, Pointer2, N/2 );
            Cross2 = CrossOverTerms( Pointer2, Pointer1, N/2 );
            Mult( Cross1, Cross2, N/2, Result3 );

            /* Sequentially combine results to give the answer */
            Combine( Result1, Result2, Result3, N, AnswerPointer );
      }
      return;
}
```

Figure 1. Pseudo-code for polynomial multiplication [20].

4. META-PROGRAMMING: ASSETS

There are several parallelization structures that are used frequently in writing code for coarse-grained parallel applications. Examples include pipelines, master-slave structures and parallel divide-and-conquer. There are a number of tools that provide easy generation of these structures (for example, PIE [21] and P3L [2]) that can significantly reduce the program development time for applications well-suited to these forms of parallelism.

Although these structures have been developed and named in an *ad-hoc* fashion, each can be considered a high-level specification for a parallelization mechanism. However, two fundamental concepts are required to translate these unrelated parallelization mechanisms into a suite of powerful software engineering components. The first requirement is a single unifying vehicle that will allow each of these disparate techniques to be documented, explained and applied in a consistent manner. This would allow novices to learn a second or third technique more quickly after the first technique is mastered. It would also allow different experts to communicate results more easily with fewer misunderstandings. Using a single vehicle to describe entities that appear different is not a new idea, just a powerful one. For example, BNF is used as a single mechanism to describe the syntax of a variety of programming languages.

It is not enough to develop a notation that regularly describes these parallelization techniques, but only allows them to be used in isolation from each other. The second requirement is some high-level mechanism that can be used to combine these techniques so they can be used as building blocks in the same application. For example, a flowchart not only provides a vehicle for representing different control structures, it also allows a variety of control structures to be combined in the same flow control diagram. Ideally however, this combining mechanism should allow the parallelization techniques to be combined in a hierarchical fashion, not just a linear one. Perhaps our ideal vehicle is closer to the data-flow

diagrams that are used in software engineering to describe a multitude of different software structures that can be combined in a hierarchical fashion.

Whereas other parallel tools stick with the traditional names for parallelization techniques, Enterprise uses an analogical metaphor to serve as our vehicle for documenting, explaining, applying and combining these different parallelization techniques at a high level. In fact, a major goal of the Enterprise PPS is to validate our metaphor. Business organizations are working systems that exploit parallelism. Every day, businesses receive orders and distribute tasks to different divisions, departments, assembly lines and individuals who work concurrently to fill these orders. We have chosen the structure of a business organization as our metaphor for parallel computation. We use the generic term *asset* to represent a resource that can be used to complete a task, and define a suite of assets that represent our high-level parallelization techniques by business units. Enterprise uses this metaphor both to represent parallelization techniques and to combine them in programs. Although experienced parallel programmers might scoff at the analogy, our experience shows that it is easy to learn, and is of significant benefit for explaining parallelism to sequential programmers [23]. Parallel programming still has a steep learning curve and anything that can reduce programmer startup time is beneficial.

4.1. Parallelization Techniques

Enterprise currently supports the assets whose icons are given in Figure 2. Each asset has a name and associated code, consisting of a function with the same name as that of the asset. Using the business analogy, each asset represents a named person and a definition of its role in the organization (or enterprise). The user can expand composite assets to see their underlying structure and use them to build hierarchical structures. It is also possible for the user to replicate an asset so that multiple calls to that asset do not have to wait until the first replica has finished. The mechanics of transforming, expanding and replicating assets are discussed in Section 4.2. All assets have unique names that can appear only once in the diagram, meaning that cyclic calls (possibly introducing deadlock) are not possible (such as in Hence [4]).

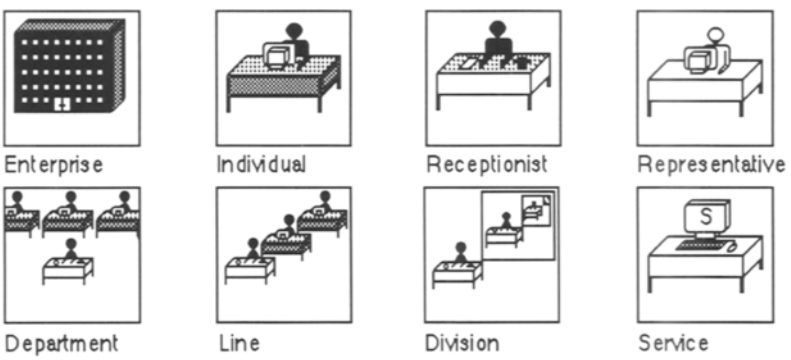


Figure 2. The icons for the Enterprise assets.

Enterprise: it represents a program and is analogous to an entire business organization. Every enterprise asset contains a single component.

Individual: it represents a slave in traditional parallel terminology and is analogous to a person in an organization. It does not contain any other assets. In terms of Enterprise's programming component, it represents a procedure that executes sequentially. An

individual has source code and a unique name. When an individual is called, it executes its sequential code to completion. Any subsequent call to that individual must wait until the previous call is finished. If a developer entered all the code for a program into a single individual, the program would execute sequentially.

Line: it represents a pipeline in traditional parallel terminology and is analogous to an assembly or processing line. It contains a fixed number of heterogeneous assets in a specified order. The assets in a line need not be individuals; they can be any legal combination of Enterprise assets. Each asset in the line refines the work of the previous one and contains a call to the next. The first asset in a line is the *receptionist.* A subsequent call to the line waits only until the receptionist has finished its task for the previous call, not until the entire line is finished.

For example, consider a graphical animation program. For each frame it creates objects, converts the objects to polygons and renders the frame. Enterprise could represent this program using a line asset that contains three individual assets (*CreateObjects*, *Polygon* and *Render*) that get called for each frame (i.e. three functions, with *CreateObjects* containing a call to *PolyGon*, and *PolyGon* containing a call to *Render*).

Department: it represents a master/slave relationship in traditional parallel terminology and is analogous to a department in an organization. It contains a fixed number of heterogeneous assets and a receptionist that directs each incoming communication to the appropriate asset. All assets execute in parallel.

For example, consider a program that solves a series of sets of linear equations. There are many different techniques for solving sets of equations, depending on the properties of the coefficient matrix. The program repeatedly reads a matrix, selects a technique and calls a solver that implements that technique. In Enterprise, this program would appear as a department whose receptionist reads each matrix and calls the appropriate individual asset to solve the equations.

Division: it represents a divide-and-conquer computation and contains a hierarchical collection of individual assets among which the work is distributed. When created, a division contains a receptionist and a *representative* that represents a leaf node. Divisions are the only recursive asset in Enterprise. Programmers can increase a division's breadth by replicating the representative. The depth of recursion can be increased one level at a time by transforming the representative (leaf node) into a division. This approach lets developers specify arbitrary fan-out at each level.

For example, consider the polynomial multiply program shown in Figure 1. A call to `Mult()` divides each polynomial into three pieces (lower, upper and cross-over) and makes recursive calls on each. By making `Mult()` a division, each recursive call can be done in parallel. This parallel recursion continues until the lowermost division asset is reached. Further recursive calls are executed sequentially so a division is a combination of parallel and sequential recursive calls.

Service: it represents a monitor and is analogous to any asset in an organization that is not consumed by use and whose order of use is not important. It cannot contain or call any other assets, but any asset can call it. A wall clock is an example of a service. Anyone can query it to find the time, and the order of access is not important.

For example, consider a program that uses shared memory. This can be simulated using a service asset that accepts two types of calls: one to set the value in shared memory and the other to retrieve the value. Any other asset can call this service to set/get the value. Alternately, a service could be used to perform a more complicated function. For example, a service could be used to queue requests to a shared resource, such as a printer, guaranteeing mutual exclusion for the requests.

4.2. Meta-Programming: Combining Parallelization Techniques

Each of the Enterprise assets can be regarded as a template, or algorithmic skeleton [8], for parallelizing a section of sequential source code. However, a mechanism must be provided to combine these assets into a working program. Some PPS's represent each process with a node and require the user to "connect-the-dots" to define communication paths ([1] and [4], for example). Although this approach produces a (possibly complex) diagram of the inter-process communication structure of a program, it does little to reveal the high-level parallelization techniques and algorithms that are being used. At the other extreme, some systems have templates defined for a class of problems, hiding most of the parallel implementation details. This requires little interaction on the user's part since everything is pre-packaged. These tools, such as PUL [7], can support powerful high-level structures, but with restricted domain applicability.

Enterprise provides a small set of building blocks from which the user can construct arbitrarily complex programs using a simple mechanism. In Enterprise, the user begins by representing a program as a single enterprise asset containing a single individual. This "one person business" represents a sequential program. Four basic operations are used to transform this sequential program into a parallel one: asset expansion, asset transformation, asset addition and asset replication. Using the analogy, the simple business grows into a (possibly complex) organization.

The initial enterprise asset can be expanded to reveal its internal structure; a single individual. The individual asset can then be transformed into a composite asset like a department, line or division and the composite asset can be expanded to reveal its default components. Component assets can be added to lines or departments. If there are more calls to an asset than it can handle in a reasonable time, the asset can be replicated to produce multiple identical copies. If a call to a replicated asset has not returned by the time a subsequent call is made to that asset, one of the replicas transparently handles the call. Finally, component assets at any level can be replicated and expanded so a program can consist of a hierarchy of assets to an arbitrary level.

For example, consider a program that consists of a department asset that contains a single receptionist, `DEPT()`, and calls to three individual component assets, say `A()`, `B()` and `C()`, in a loop. Assume that `A()` is not compute bound, but that assets `B()` and `C()` are. If `B()` has an internal structure that does not lend itself to further parallelization, it can still be replicated so that multiple calls to it can execute concurrently. Assume that `C()` has an internal structure that could be represented by a line. `C()` can then be transformed into a line asset with receptionist (also called `C()`) and additional components `D()` and `E()`. All three assets (`C()`, `D()` and `E()`) can then execute concurrently. If the line `C()` is still compute bound, it can also be replicated. Multiple copies of it (and all of its components) can then execute concurrently. Figure 3 illustrates the structure of this program as it would appear in Enterprise. Inside the double-line rectangle is the expansion of the enterprise asset. Inside the solid-line rectangle is the expansion of the department `DEPT()`. Asset `B()` has been replicated three times and asset `C()` two times. Inside the dashed-line rectangle is the expansion of the line `C()`, containing the receptionist `C()` and individuals `D()` and `E()`. In theory, there is no limit to the complexity of a parallel program. In practice, we have never had to build a diagram as complex as this one for any application.

An important point illustrated with this diagram is the separation of sequential code from parallel specification (Section 5). The user can change this diagram, for example by modifying a replication factor or making `C()` an individual instead of a line, *without having to change the code.*

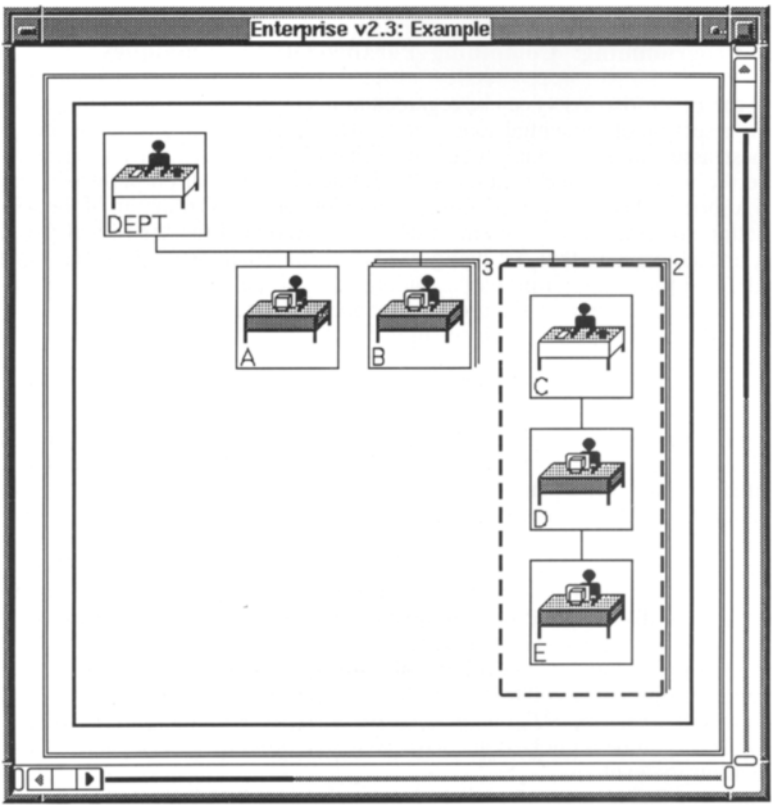


Figure 3. The hierarchical representation of a complex program.

Consider the parallel polynomial multiplication program of Figure 1. The program can be described as a line of two assets. `PolyMult()` reads in the coefficients of the polynomials to be multiplied. It then calls `Mult()` to recursively do the multiplication. Figure 4 shows the parallel structure of the program. Inside the double-line rectangle is the expansion of the enterprise asset. Inside the dashed-line rectangle is the expansion of the line of two. Inside the lightly-shaded rectangle is the expanded division for the second asset in the line. Inside the division is another division. Here the division is replicated three times (because of the three recursive calls to `Mult()`). The diagram shows the depth of the recursion (two levels). Figure 5 shows the results of expanding the diagram into a standard call graph showing all of the processes (except for some hidden Enterprise processes). The simple diagram of Figure 4 corresponds to a complex structure of 13 processes.

There are two important consequences to the way that legal Enterprise diagrams are constructed. The first is that not all parallel algorithms can be expressed using the Enterprise meta-programming model. For example, an algorithm that relies on a group of processes using peer-to-peer communication cannot be supported using current Enterprise assets. Although this problem can be alleviated somewhat as new Enterprise assets are designed, it will never really disappear. It is probably impossible to pre-determine a set of templates that can represent an arbitrary communications topology without reducing the level of the templates to a "connect-the-dots" approach.

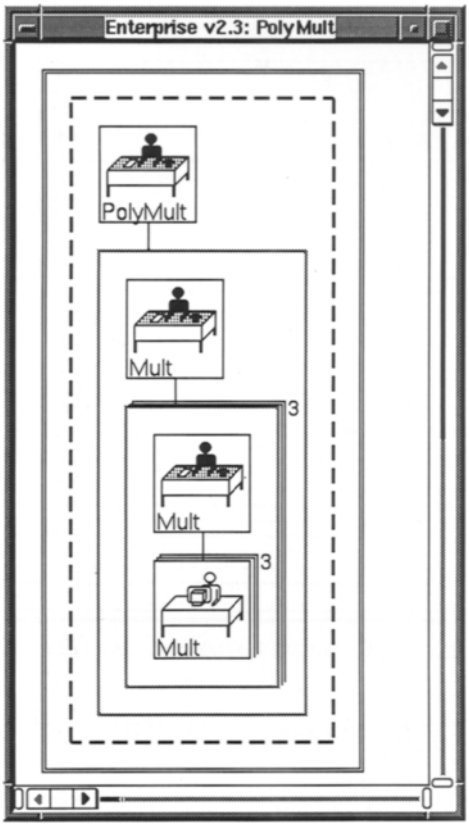


Figure 4. `PolyMult` asset diagram.

The second consequence is that it is impossible to draw a diagram that contains a logical parallelization error. For example, deadlock cannot occur in an Enterprise program because the only process graph cycles that can occur are strictly managed in the division asset†. Similarly, connection errors cannot occur due to missing communication channels, since these channels are established by Enterprise generated code, not user code.

Enterprise provides a consistent metaphor that allows parallelization techniques to be documented, explained, applied and combined.

† The user can create deadlock in their code by, for example, having one process go into an infinite loop or by having one process wait for an event (such as a file i/o) that never occurs. Enterprise guarantees that the communication structure cannot deadlock.

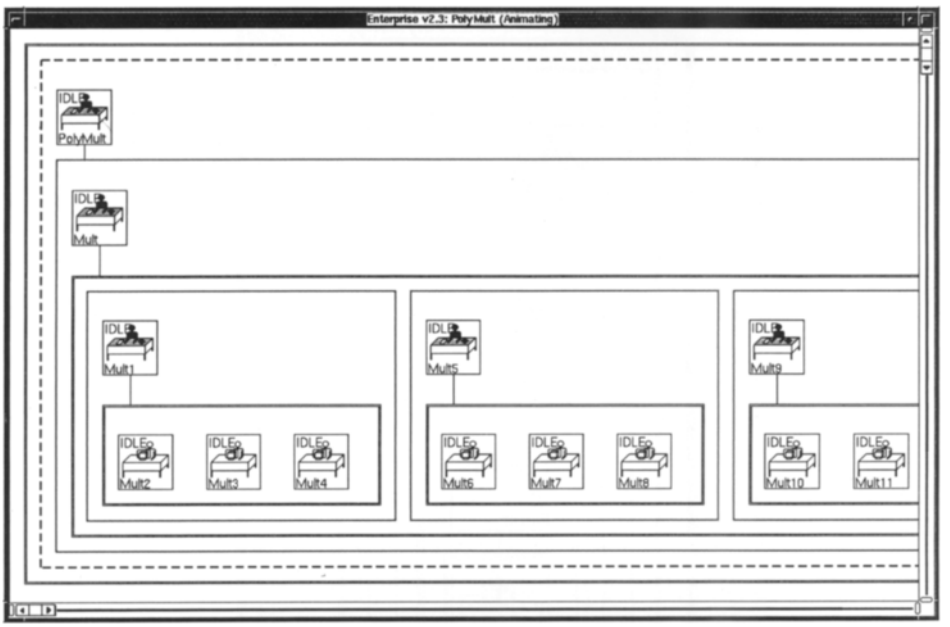


Figure 5. Expanded `PolyMult` asset diagram.

5. PROGRAMMING AND META-PROGRAMMING: ORTHOGONALITY

There are two aspects to executing a parallel program: the code used at compile time and the parameters needed to execute it at run-time. In older systems, these two properties were intertwined; changing the process/processor mapping meant editing the code and re-compiling. Newer systems abstract this information into a configuration file which specifies the process-processor mappings (for example, NMP [17] and P4 [5]). This file must be edited every time the execution environment of the program changes (such as machine availability) or run-time parameters are altered (such as a replication factor). This extra editing effort can be error-prone.

An Enterprise program consists of sequential C code and a diagram (meta-program) that represents the desired parallelization at a high level. What are the relationships between the program and its meta-program? First, every asset in the diagram must appear as a procedure call in the program. Second, the call structure of the diagram must match the call structure of the program. Third, every asset must be in its own disk file. All three of these requirements are checked by the Enterprise compiler and if there is a violation, the user is informed. Except for these three constraints, a program and its meta-program are completely orthogonal. That is, many parts of a meta-program can be changed without affecting the program code and visa-versa. For example, program and asset features (such as the replication factor, ordered or unordered attribute, which of its parameters are recorded during event logging, etc.) can be changed without changing the program source code. Alternatively, the internal code of any asset can be changed without affecting the asset diagram. Figure 6 illustrates this orthogonality.

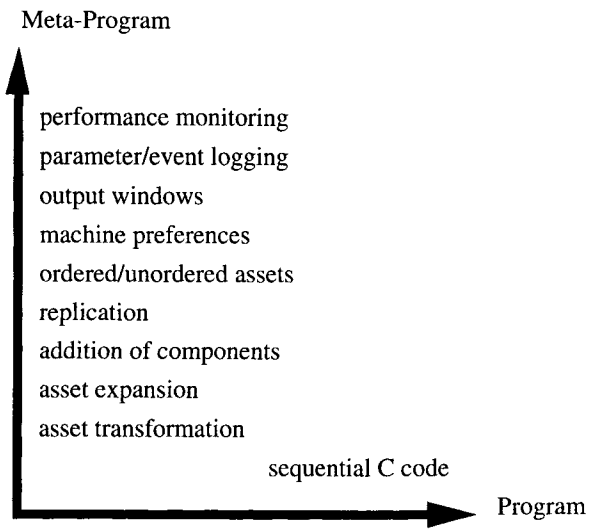


Figure 6. The orthogonality of a program and its meta-program.

This separation of a program's code and its parallel structure supports experimentation with different parallelization techniques and rapid performance tuning. The same program code can be mapped to a meta-program that uses a department or a line simply be re-drawing the diagram. Replication factors can be increased in locations where all replicas of an asset are busy and decreased where some replicas are idle. This can be done either manually by the user or automatically by the system.

As an example, consider the program of Figure 3 again. This program can be executed with the diagram as shown and performance information obtained (either by timing or by using some of the performance monitoring tools described in the next section). The meta-program (diagram) can then be edited to collapse the assets `C()`, `D()` and `E()` into a single individual asset. The program can then be recompiled without changing the source code (the asset calls to `D()` and `E()` are interpreted as local procedure calls now) and the program can then be re-executed. The performance of the modified meta-program can then be compared with the previous one and the meta-program with the better performance can be selected. Alternately, the replication factors of the replicated individual `B()` and the replicated line `C()` can be traded off, shifting available processors between `B()` and `C()` to obtain maximum performance. This modification to the meta-program can be made rapidly without changing the source code (or even recompiling in this case). The change to the meta-program that causes the biggest change to a user's program occurs when an individual is transformed into a composite asset like a line or a department. However, even in this case, the only change to the program is to shift the asset code for the component assets (some procedures) to separate files. The actual code itself may not have to change†!

The virtual orthogonality of sequential source code (a program) and its parallelization technique (a meta-program) allows users to focus on each of the two components independently. It is analogous in some respects to the art of programming with abstract data

† A change may be necessary if the parameters do not meet Enterprise requirements. For example, when executed sequentially the function may have relied on accessing values through global variables; they now have to be passed as parameters.

types where the use and implementation are independent. This approach significantly reduces the complexity of parallel programming.

6. ENVIRONMENT: INTEGRATED INTERFACE

Although a text editor and a compiler are the only tools necessary to write a computer program, the complexity of modern applications has led to the development of CASE tools to support the complete software development process. Unfortunately, in practice there are few CASE tools used to support the development of parallel applications. Those parallel tools that are used tend to concentrate on a single phase of the development process: coding, performance monitoring or debugging.

The two main goals for using tools are an increase in programmer productivity and a decrease in design and programming errors. Unfortunately, using an assortment of unrelated tools can have a negative effect on both of these goals.

Three problems are created by differences in tools. The first is a steep learning curve as programmers learn each new tool. This may exacerbate an already difficult situation if a programmer is simultaneously struggling with a switch from the sequential to parallel domain. The second problem is the number of context switches necessary when switching from tool to tool. Each tool reflects an underlying model or development philosophy, and switching models in mid-development not only requires additional time, it can also lead to errors that must be found and fixed. The third problem is that these tools often use different representations, and format translations can be time consuming and error-prone. For example, a post-mortem debugger and a performance monitor may both require an event log, but if they are in different formats then a third tool is required to translate the events. Fortunately, the third problem is being addressed by the parallel community, with the introduction of community standards (such as MPI [25]) and popular standards (such as PICL [10]).

The user interface of the Macintosh computer is a good example of the potential benefits of a common user interface. After a user learns one Macintosh application, it is easy to learn a second and a third because of the similarities in the user interface. It is also easy to switch from one application to another, and the clipboard provides a simple mechanism for moving information between them. Significant benefits can accrue from having a comprehensive programming environment that supports the entire software life-cycle of parallel applications using a common user interface.

Enterprise has a uniform interactive graphical user interface that supports a common programming metaphor across all of its components [15]. Enterprise provides three main views of a program: a design view, an animation view and a replay view. In all of the views, the program is represented as an asset diagram. The "look and feel" of the interface is identical across all three views.

6.1 Design View

The design view is used to edit the asset diagram (meta-program) of a program as shown in Figure 3. This view supports edit, dialog, compile and run windows that the user can use to edit source code, set asset attributes (like ordered/unordered), set compile/run options and to compile/run a program. Enterprise has an automatic *makefile* facility so that only the necessary files will be recompiled when a compilation is done (again, relieving programmers from the additional effort of constructing their own *makefiles*).

6.2 Animation View

The animation view is used to display an animation of program execution as recorded in an event log. The animation view is similar to the design view except that replicated assets are flattened so that each replica can be seen. Figure 7 shows the animation view of a program. In

addition to the assets, messages and message queues are shown. This view actually animates messages and shows them moving between assets. The state of each asset is shown (IDLE, BUSY, BLOCKED, DEAD) using a label and a color code. The user can start, stop, resume and single step the animation. At any time, a message queue can be opened to display a list of messages and the logged parameters of a message can be viewed as well.

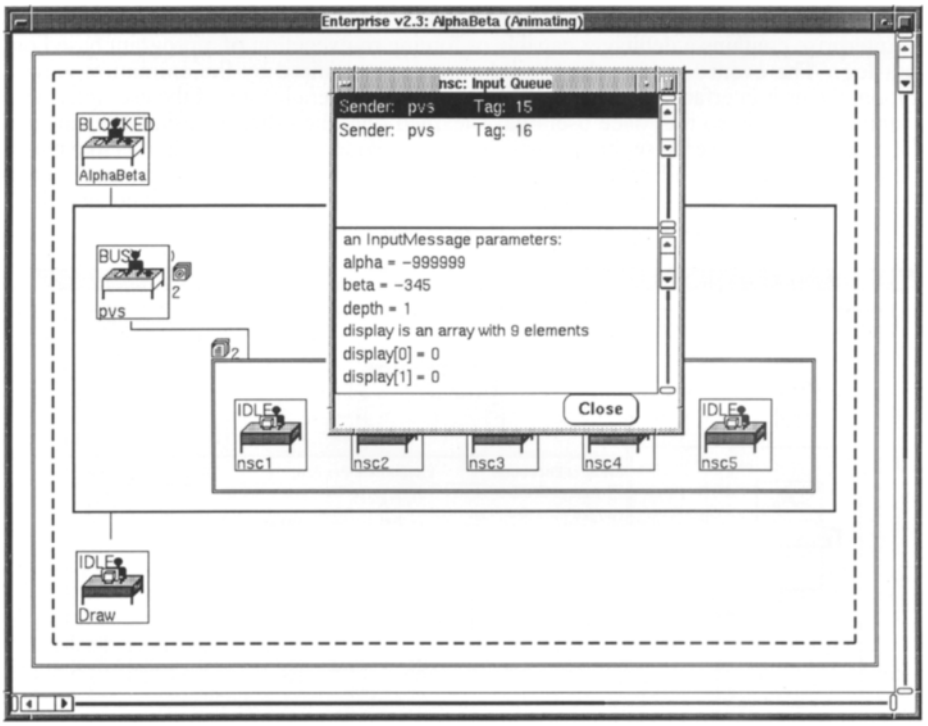


Figure 7. The animation view of Enterprise.

A variety of performance windows can be displayed in the animation view including graphs that show program speed-up as a function of execution time during a run (in analogy to a profits and loss chart for a business organization) and pie charts that show the relative idle, busy and blocked times for each asset. This information can be used to tune performance by quickly returning to the design view, changing the meta-program, re-running the program and viewing another animation. The common model and uniform user-interface make this a simple process.

6.3 Execution Replay View

Debugging distributed programs is a difficult endeavor. In most situations, sequential code is deterministic so that repeated execution with the same input data follows the same execution path. This makes it relatively easy to isolate and correct programming errors by executing the program repeatedly. Unfortunately, a parallel program may be non-deterministic. That is, it may follow different execution paths for the same input data due to race conditions between processors with different loads. A particular logic error may only occur on one run out of ten. To isolate and correct such an error, it may be necessary to reproduce the error run many times.

To help reproduce such non-deterministic errors, a parallel debugger can include an execution replay mechanism (such as [24]). Execution replay works as follows. The program is executed with event-logging until the error occurs. The debugger then forces the program to re-execute, following the same event order as the event-log for the error run. The program can be re-executed in this order as many times as are necessary to isolate the error. Of course, the debugger should also provide a breakpointing mechanism to isolate the problem during these re-executions.

Enterprise contains a debugger capable of forcing re-execution of a program based on an event-log. The event-log is in the same format as is required by the animation view. In addition, the user interface for the replay view is a direct generalization of the user interface for the animation view so that once users have learned one, the other is easy. The Enterprise debugger allows the user to set breakpoints based on a wide variety of conditions like message type (send message, send reply, receive message, receive reply), message collaborators (sender asset and receiver asset) parameter values (like `a[i] = 3`) and event counts (like the third message sent). Figure 8 shows a replay view and a breakpoint browser.

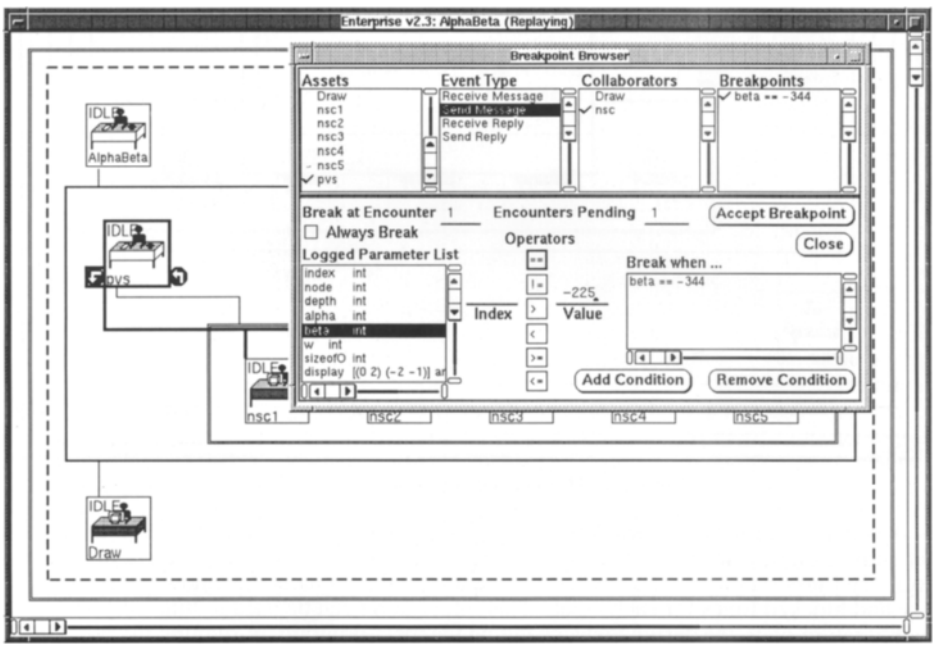


Figure 8. The replay view of Enterprise and a breakpoint browser.

The common user interface and uniform model make context switches between the design, animation and replay views completely seamless and increases the productivity of parallel programmers.

7. EXECUTION: RESOURCES

Most coarse-grained distributed applications are run in environments with changing conditions. The identities, types, configuration and availability of machines on the network can change frequently, even while a computation is running. The user should *not* be

responsible for knowing the up-to-date status of the network resources available. However, many systems make it the user's responsibility to know the names of all machines used in the computation beforehand (such as NMP [17]). In others, dynamic reconfiguration is possible, but it is the user's responsibility to identify when it should be done. For example, user code may check to see if the processors are busy and take corrective actions. Again, this does not seem to be a good way to spend programming resources.

When a program is started, Enterprise decides which machines to use (although there are options for the user to make the selection if desired) based on availability and load average. The user does not need to know the identities of the machines used in a computation beforehand. Current research has Enterprise monitoring the status of machines on the network and dynamically changing the resources used by the program. This moves the network management overhead from the user to the system where it belongs.

Finally, a PPS should show interoperability between different systems. For example, network computing must face issues of different machine architectures, operating systems and file systems. PVM has made great progress in hiding many of these details from the user. Systems like Enterprise only exploit what PVM provides.

8. CONCLUSIONS

In many parallel programming tool papers, the authors illustrate the strengths of their approach by presenting some impressive speedup results. Is this the only metric that we should use in judging the quality of the tool?

We have conducted a controlled experiment that compared programming in Enterprise to programming with a PVM subset (NMP) [23]. Half of a graduate student class did an assignment using Enterprise; the other half used NMP [17]. Enterprise users ended up writing 66% less code than did NMP users. This translated into fewer compiles, editing sessions and test runs. It was interesting to compare the errors contained in the programs submitted by the students. Some NMP students had problems with program correctness; some Enterprise students had problems with performance. We argue that correctness is far more important than performance. Performance issues should only be addressed once correctness has been established.

Enterprise allows the user to quickly develop a correct parallel structure so that potential errors may only lie in the application-specific sequential code. Only when the program is running properly should performance be a consideration, and only if the performance does not meet expectations. In our experience there are many applications which can benefit from tools like Enterprise, but there is a mental block in the computing community that equates parallel programming with being a difficult task. Users are delighted when they can use a tool like Enterprise and quickly obtain speedups for their application. That the amount of the speedup might not be as high as is achievable using some low-level tool is irrelevant; any performance gain is welcome.

Although this paper used Enterprise as an example, most of the features described can be found in other tools. The state of the art in parallel programming systems is improving rapidly and the user community is more receptive to these new products. Over the last few years, PVM has emerged as a *de facto* standard for writing distributed applications (although MPI may change this [25]). We are still waiting for a higher-level tool, one that builds on top of PVM/MPI, to emerge and gain wide-spread acceptance.

ACKNOWLEDGMENTS

Our thanks to the current Enterprise team members: Paul Iglinski, Steve MacDonald, Chris Morrow, Diego Novillo, Ian Parsons and David Woloschuk.

REFERENCES

1. O. Babaöglu, L. Alvisi, A. Amoroso, R. Davoli and L. Giachini. Paralex: An Environment for Parallel Programming Systems. Technical report UBLCS-92-4, Laboratory for Computer Science, University of Bologna, 1992.
2. B. Bacci, M. Danelutto and S. Pelagatti. Resource Optimization via Structured Parallel Programming. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 13-25, 1994.
3. H. Bal, M. Kaashoek and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190-205, 1992.
4. A. Beguelin, J. Dongarra, G. Geist, R. Manchek and V. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. *Supercomputing '91*, pp. 435-444, 1991.
5. R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, vol. 20, no. 4, pp. 547-564.
6. N. Carriero, D. Gelernter, T. Mattson and A. Sherman. The Linda Alternative to Message-passing Systems. *Parallel Computing*, vol. 20, no. 4, pp. 633-655, 1994.
7. L. Clarke, R. Fletcher, S. Trewin, R. Bruce, A. Smith and S. Chapple. Reuse, Portability and Parallel Libraries. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 171-182, 1994.
8. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Programming.* MIT Press, Cambridge, Mass., 1989.
9. J. Feo, D. Cann and R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, vol. 10, 1990.
10. G. Geist, M. Heath, B. Peyton and P. Worley. PICL: A Portable Instrumented Communication Library, Technical report ORNL/TM-11130, Mathematical Sciences Section, Oak Ridge National Laboratory, 1990.
11. G. Geist and V. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.
12. A. Goldberg. Concert/C: A Language for Distributed C Programming. IBM T.J. Watson Research Center, Yorktown Heights, New York, 1993.
13. A.R. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, 1985.
14. W. Karpoff and B. Lake. PARDO - A Deterministic, Scalable Programming Paradigm for Distributed Memory Parallel Computer Systems and Workstation Clusters. *Supercomputing Symposium '93*, Calgary, pp. 145-152, 1993.
15. G. Lobe, D. Szafron and J. Schaeffer. The Enterprise User Interface. *TOOLS (Technology of Object-Oriented Languages and Systems) 11*, R. Ege, M. Singh and B. Mayer (editors), pp. 215-229, 1993.
16. D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, vol. 1, no. 1, pp. 25-42, 1993.
17. T. Marsland, T. Breitkreutz and S. Sutphen. A Network Multiprocessor for Experiments in Parallelism. *Concurrency: Practice and Experience*, vol. 3, no. 1, pp. 203-219, 1991.

18. D. Ritchie and K. Thompson. The UNIX Timesharing System. *Communications of the ACM*, vol. 17, no. 7, pp. 365-375, 1974.

19. J. Schaeffer, D. Szafron, G. Lobe and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, vol. 1, no. 3, pp. 85-96, 1993.

20. R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, 1983.

21. Z. Segall and L. Rudolph. Pie (A Programming and Instrumentation Environment for Parallel Processing). *IEEE Software*, vol. 2, no. 6, pp. 22-37, 1985.

22. Sun Microsystems. Remote Procedure Call Programming Guide. Sun Microsystems, 1986.

23. D. Szafron and J. Schaeffer. Experimentally Assessing the Usability of Parallel Programming Systems. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 195-201, 1994.

24. D. Taylor. The Use of Process Clustering in Distributed-System Event Displays. *CASCON '92*, vol. 1, IBM Toronto, pp. 29-42, 1992.

25. D. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, vol. 20, no. 4, pp. 657-673, 1994.

This Page Intentionally Left Blank

IV. APPLICATIONS

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

A role for high-performance computing in surgery

Carl Diegert[1]

Mail Stop 1109, Sandia National Laboratories, Box 5800, Albuquerque NM, 87185-5800. diegert@cs.sandia.gov.

Introduction

New capacities for acquiring, processing and communicating digital data have already begun a revolution in surgery. The new focus is on a broad, and ever widening set of digital data to more completely describe the patient, the disease, the machines in the operating room, the surgical team, and other hospital processes. The result can be more effective intervention, and reduced net cost of intervention. Already, effective intervention often cannot be completed without first at least obtaining digital information from automated "scans" of a patient before surgery. Successful outcomes increasingly depend on managing and exploiting an exploding base of digital data.

With the new focus on large-scale, digital datasets, come abundant roles for automatic computation. Here we describe one role: how automatic computation can extract summary information that is both manageable in the operating room, and can improve a surgeon's speed and accuracy. More specifically, we outline a role where we apply a high-performance computer to obtain summaries of the three-dimensional extent of lesions, and other clinically-relevant structures. We then deliver these summaries to workstations at the clinical site in time to provide a basis for planning, for rehearsal, and for guidance during surgery.

Today, the computational role we describe demands high-performance computer hardware. We chose a role that does not demand that the supercomputer be online during surgery. The role's process flow demands only that the supercomputer receive, process, and return data, all within a few hours. Present, sub-gigabit computer networks are enough to allow a supercomputer remote to the clinical site to fill the role.

---

1. The role we present must be continually tuned and checked. Therefore, the computation cannot add value without demanding continued clinical expertise. We have been fortunate to collaborate with Dr. Sanders (MR specialist), Dr. Orrision (radiologist), and Dr. Baldwin (surgeon), all from the Albuquerque Department of Veteran's Affairs Medical Center.

This work was supported by the Laboratory Directed Research and Development Program, Sandia National Laboratories, United States Department of Energy under contract DE-AC04-94AL85000.

The marginal cost of using today's supercomuters and networks to enhance treatment of at least some patients may already be justified by lowered costs of surgery, and of better outcomes. But there is a more compelling reason to begin experimental clinical use now, despite the trouble and expense of using remote supercomputers. Within a year or two, technological advance and volume production will drive the cost of computing hardware for this role down to levels that allow economical deployment of the hardware at, say, the thousands of sites that already do clinical imaging. By getting through development and clinical trials now, the enhanced treatment can be ready for wide deployment as soon as decreased computing hardware costs make it cost effective. From a business perspective, projecting the cost of future computing hardware is low risk compared to the risk in predicting clinical utility. Clinical experience now can reduce the risk in business plans, allowing industry to have product ready to exploit new computing hardware soon after it becomes available.

The conclusion of this paper shows that an effective way to implement and deliver the computation is to code using explicit message-passing, and to execute the code on a distributed-memory computer.

Improving the surgeon's accuracy

We will present the computational role by beginning with the problem, and concluding with the computation and its implementation.

A case study illustrates the problem. A central surgical goal is to quickly and completely resect certain brain tissue, while minimally disturbing other tissue. Information observed by prior magnetic-resonance (MR) acquisitions is indispensable in planning and guiding the intervention. The problem is that the surgeon and radiologist work from a less-than-ideal presentation of a small subset of the MR data. The problem is even greater in a typical surgery because information from multiple MR acquisitions, as well as information from new types of functional imaging all should help plan and complete the intervention.

About two hours into the five-hour surgery, the surgeon has exposed the brain, and must decide where to make an initial incision with his scalpel. Figure 1a

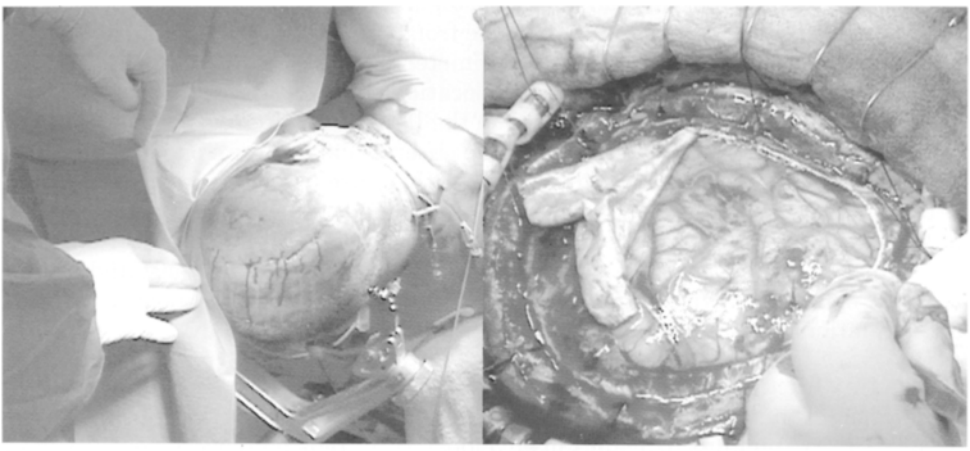


Figure 1a. Figure 1b.

shows how the patient presents for this surgery. Figure 1b is the surgeon's view of the patient just before making an initial incision.

Before making the incision, the surgeon completed a final review of the patient's MR data by stepping away from the patient, to stand beside five light boxes on the operating room (OR) wall. Each box displayed a black-and-white film mosaic of about 24 images. Figure 2a, a somewhat enhanced version of one of these images,

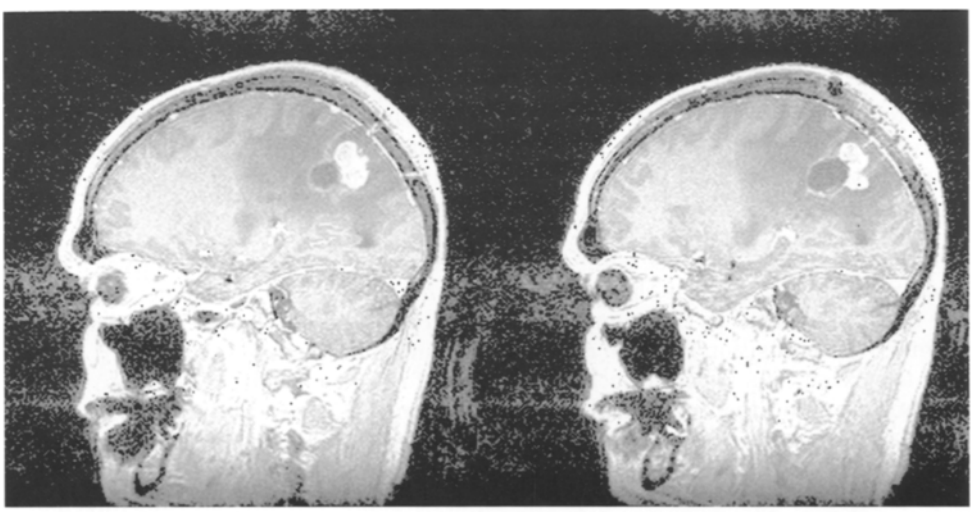


Figure 2a Figure 2b

shows a two-dimensional cross-section through the head in a plane perpendicular to the ear-to-ear axis. At the back of the head (at about 2:00 o'clock) are well-defined, adjacent bright and dark regions, together comprising a lesion, the tissue to be removed. Figure 2b is another image from the same volume acquisition, sliced perpendicular to a point a few millimeters closer to the patient's right ear. Other images (not presented here) show location of critical brain function, tissue to leave undisturbed. All of the images on the light boxes are two-dimensional sections. Some images show the same volume observation as figure one, but sliced at additional points along the ear-to-ear axis. Others are from acquisitions tuned to better show other features, acquisitions by other modalities (computed tomography X-ray and magnetic source imaging), and acquisitions taken at prior dates. The multiple acquisitions are taken at different sampling resolutions, and taken with the sampling lattice aligned with different axes. Understanding changes and other relations among these data demands skill and careful study.

Only a tiny fraction of the imaging acquisitions entered into the planning and completion of this surgery. In addition to the images on the five light boxes, hundreds more are on films in a stack of envelopes.Hundreds more were never printed to film — medical technicians chose only a few "best" images from each volume acquisition to print to film. However, the entire digital dataset is archived on computer media in the hospital's radiology department. The radiologist and surgeon did make at least some use of these additional films in planning the surgery, but not during surgery. Access to the patient's complete digital database was impractical while planning the surgery, and was impossible during surgery.

Relevant, usable summary information

A role for high-performance computing is to give the surgeon a more suitable presentation of the digital database. The computation provides two opportunities for improvement. First, by aggressively summarizing the whole digital dataset, we seek to provide summaries that include more relevant information, dramatically more than can be presented in a small collection of "best" image slices. Second, our presentation must actually allow a human surgeon to improve his speed and accuracy.

Applying high-performance computing, as described below, we can deliver a compact representation of the three-dimensional extent of a lesion to a workstation in the operating room. In this computation, we load all of the digital information from a MR acquisition into the memories of a high performance computer, then find the extent of a lesion (or other structure of interest) in three dimensions. The computation and result are at the full resolution of the MR acquisition. The summary product includes only the geometric extent of the lesion. Substituting the summary product for the entire MR acquisition discards most of the information in the acquisition, and retains just a compact summary of the relevant information. With the reduced dataset size, a workstation is all that is needed to manage the summaries in the OR.

The form of the data is also changed when the summary is substituted for the MR acquisition. When the summary correctly defines the clinically-relevant extent of a lesion, we are adding value (for use in surgery) to the acquisition data. The segmentation algorithms that make this summary must be continually tuned and checked. Therefore, the computation cannot add value without demanding some continued clinical expertise. We have been fortunate to collaborate with Dr. Sanders (MR specialist), Dr. Orrision (radiologist), and Dr. Baldwin (surgeon), and have added our expertise in statistics and high-performance computing, to experiment with this new computational role.

A workstation can easily load our summary information, and can generate realistic images of the lesion and related structures from any viewing position. Two views are shown in Figures 3a and 3b. These views may better convey the shape of

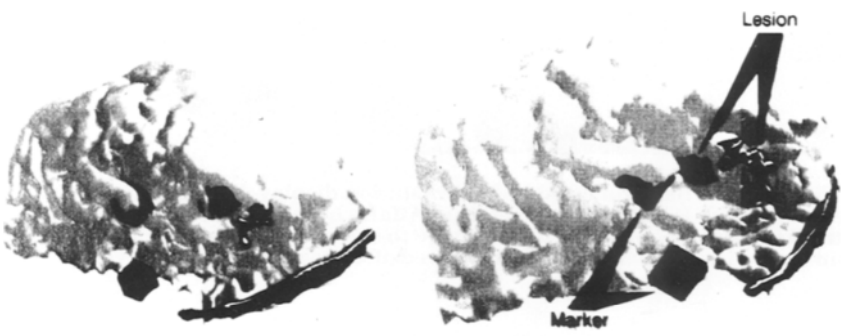


Figure 3a Figure 3b

Two views, rendered on a workstation from the supercomputer's summary information. The summary estimates the extent of a lesion, the extent of healthy brain tissue (this tissue is severely compressed by swelling at the lesion site), and the extent of the deep vein. The two octagon shapes mark the location of brain function. In presentations on the workstation, we use color

the lesion to the surgeon, but are probably not sufficient to dramatically improve the accuracy of locating his incision.

The final step to clinical impact

The second opportunity for high-performance computing is to provide the reduced data that can actually improve the speed and accuracy of a human surgeon. The groundwork for taking this step has been realized in a research protocol now in use at the Albuquerque Department of Veterans Affairs Medical Center. The groundwork comprises two components. First, a 3D tracking device in the OR makes the measurements that can align prior datasets with the patient, in the position he assumes during surgery. Second a workstation in the OR completes intraoperative computation.

The workstation is loaded with programs and data, then disconnected from the Internet, and rolled into the OR before surgery (Figure 4). The tracking system

Figure 4. Dr. Baldwin consults a workstation display during a surgery at the Albuquerque Department of Veterans Affairs hospital. The cart also carries a tracking system. We altered this image to show the display discussed below, an image Dr. Baldwin actually reviewed after completing the surgery.

(Pixsys, Boulder CO) is also moved to the OR before surgery, and is connected to the workstation. With this innovative setup, the surgeon uses a sterile probe equipped with infrared marker lights (part of the tracking system) to point to a few fiducial points on the patient, sending three-dimensional location of these points to the workstation. Workstation software, then, uses the fiducial information to establish a map between the coordinate system used by the surgeon's pointing device, and the coordinate system used in the prior MR acquisition. Using the map, the surgeon can point to any location, and automatically call up an annotated display of the three orthogonal "slices" of the MR acquisition that intersect this point.

By adding a video camera and workstation video interface to the equipment already in the OR, the surgical team can obtain a promising display from our three-dimensional summary information, as shown in Figure 5b.

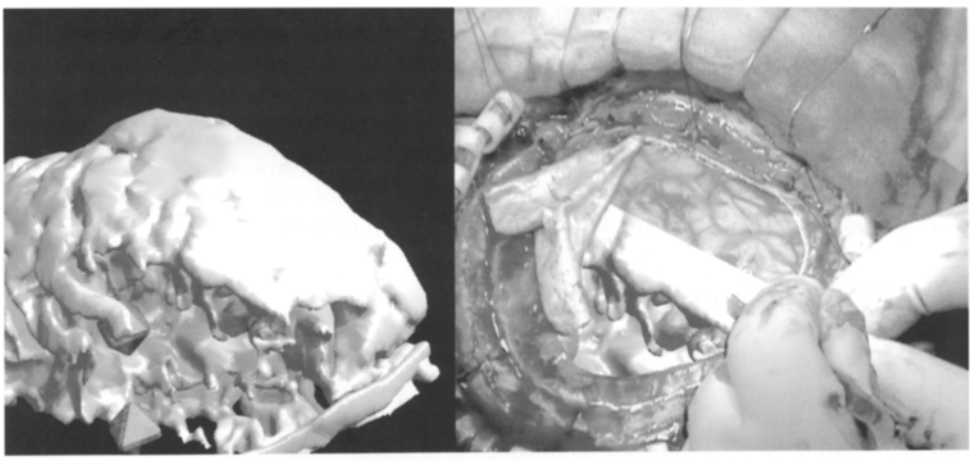


Figure 5a. Figure 5b.

A tracking system in the OR can observe the particular viewpoint (and other parameters) of an intraoperative video camera. Figure 5a is a rendering of the summary surfaces from the camera's viewpoint. Figure 5b is a composite of the camera's optical image (shown in figure 1b), and the rendered image from figure 5a. In the composite, some of the optical image has been painted away, revealing the location of the lesion underneath. In this case, the lesion's location relative to the bone cut could help locate the surgeon's

When the surgeon can obtain enhanced reality displays like Figure 5b on a workstation display screen in the OR (Figure 4), his accuracy and speed should show a dramatic improvement over the present standard of care, where he must rely on the films (Figure 2). Once the utility of these displays is established, the incremental improvements of substituting a heads-up display for the workstation screen may be worth some additional disruption to the clinical procedures. We anticipate using a scanned, visible laser mounted on the OR ceiling (together with the 3D tracking cameras) to "write" guiding marks directly on the patient.

Summary models are key to other roles

In the high-performance computing role just described, we use the summary models to improve the surgeon's speed and accuracy, as the models allow a workstation in the OR to guide the surgeon with enhanced-reality images. The key technology behind making the enhanced-reality pictures is an ability to segment and summarize the clinical MR data, producing compact, clinically-relevant summary models.

The same technology for making summary models can allow surgical rehearsal, training, and numerical simulations. To deliver a synthetic rehearsal environment using both a head-tracking visual display and a haptic manipulation device, we must generate both images and forces at interactive update rates. Without compact models, updates rates will be too slow to achieve useful fidelity. For rehearsal, we must extract the models after the patient's last MR acquisition, and in time to build the rehearsal environment before his surgery. High-performance computing offers speed and a potential for automation to meet this demanding schedule. Finally, numerical simulation involving patient anatomy depends not just on the 3D extents of the anatomy, but also on computational meshes on these regions. Our work on extracting summaries suitable for automatic mesh generation and subsequent numerical simulation are well underway.

Parallel implementation

The image segmentation and summary for the surgical role is still best accomplished on high-performance, general-purpose computers, where flexible segmentation and image generation are completed at interactive speeds. In contrast, it takes tens to hundreds of minutes for high-end workstations, equipped with about one-half gigabyte of main memory, to complete most segmentation and summary computations on single acquisitions. This is too slow to support the interactive data exploration that we still need to refine and tune the algorithms.

Our implementation delivers the required interactive performance by using explicit message passing software, executing on a scalable, distributed-memory computer. In fact, by executing on computers with more processors, the timings scale to execute much faster than demanded in this role.

For the specific computation timed in Table 1, we have a single MR acquisition from the case just presented loaded into the distributed memories of an Intel Par-

TABLE 1.
Mean Milliseconds Execution (+/- Gives Range of Two Measurements)
Speedup Percent (Relative to 32-Node Timing)
Efficiency Percent (Relative to 32-Node Timing)

Nodes	32	64	128	256
Trimmed Filter	2,710 +/- 3	1,365 +/- 3	694 +/- 1	357 +/- 1
	100	199	390	759
	100	99	98	95
Extract Geometry	2,242 +/- 1	1,217 +/- 0	659 +/- 0	361 +/- 0
	100	184	340	621
	100	92	85	78
Generate Image	2,074 +/- 1	1,428 +/- 1	979 +/- 1	697 +/- 1
	100	145	212	298
	100	73	53	37

agon XPS system. The data are represented as a regularly-sampled lattice of 32-bit floating-point numbers with 256 by 256 by 128 entries. (Figure 1 shows two slices from this acquisition.) Our code, VENGINE, builds on a portable, massively-parallel software library MPSLIB, written by Mark Sears (Mail Stop 1111, Sandia National Laboratories, Albuquerque NM). Mark wrote MPSLIB principally for completing electronic structure calculations. A secondary goal was to make MPSLIB useful in other applications, a goal he demonstrated both by broadening the library to better support VENGINE, and by writing parts of the VENGINE application.

An early step in exploring noisy MR data is to tune the parameters of a 3D, non-linear filter, apply the filter, and observe the effect on a subsequent extraction of 3D surfaces. The time to a new image in this interactive exploration is the sum of three times: filtering, surface extraction, and image generation. Table 1 shows a range in time-to-image from seven seconds down to 1.4 seconds, as executed by our code VENGINE (version of Jan-30-1995) on Paragon systems with between 32 and 256 processing nodes.

For the first step, input parameters for the trimmed filter are a threshold scalar value and a self-weight value. For each voxel, if the voxel value is below threshold, its value is unchanged by the filter. Otherwise, the filter considers the voxel and its six neighbors. If three or fewer neighbors are over threshold, the voxel value is unchanged by the filter. Otherwise, the filter replaces the voxel value with a weighted mean of itself (with weight self-weight), and the mean of a subset of the neighbor voxel values that are over threshold (with the complimentary weight). The subset is formed by discarding a highest and a lowest value.

The efficiency of the parallel filter implementation decreases slightly as more processors are used because the amount of information we must share between processors to allow access to neighbor voxel values becomes relatively more burdensome. The table shows that filtering of just a single MR acquisition is a large enough computation relative to this burden to allow extremely high efficiency on Paragon systems as large as 256 processing nodes.

The surface extraction step timed in Table 1 is a table-driven marching cubes algorithm[1], with two input parameters. The first input parameter is a fuzz value, to suppress generation of triangles too small to be useful. The second is the value of the isosurface to be approximated by the output triangle list. The execution timed in the table produces a list of 1,142,256 triangles.

A more sophisticated extraction algorithm, or further reduction of marching-cubes triangle list is needed before the summary is useful on the workstation in the OR.

---

1. To implement this parallel surface extraction step, we worked with Mark Sears to adapt the tables and serial code from part of Mike Krogh's isovis software from the National Center for Supercomputer Applications.

However, for exploration using a high-performance computer, working directly with the large triangle lists is often more effective than complicating the interaction by demanding compact models. Thus for exploration, the final step to an image is to just shade all of the triangles. The timings are for computing simple, flat-shaded triangles. They do not even benefit from an additional step of balancing the computation: the processor that generates the triangle in the last, surface-extraction step just shades it. The execution time for image generation also includes forming a composite image from all of the triangles, where the image is itself distributed across many processing nodes. The efficiency of image generation could be improved on the larger Paragon systems by adding code to dynamically balance the computation.

To best show how a current-generation Paragon system treats the real, clinical problem, Table 1 times processing of the fixed-sized problem. Efficiencies, of course, would be much higher if we scaled up the size of MR acquisitions to better fill the systems with larger numbers of processing nodes. This scalability of our implementation will be important in the future, as clinical instruments acquire data with more resolution, and with more attributes.

The next step

To date, we have developed the high-performance computing role without influencing patient outcomes. We did work with real patient cases, and with the clinical people treating the patient. We did not, however, make our results available in time to influence patient outcomes. Working in this mode was effective in focusing our development, and in demonstrating promising results. While we can continue to refine the computing role without influencing patient outcomes, we cannot directly measure the effectiveness of the computational role in this mode. Effective computation will not deliver benefits to health care unless it is used to treat patients.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Parallel Shortest Paths Methods for Globally Optimal Trajectories *

D.P. Bertsekas[a], F. Guerriero[b] and R. Musmanno[b]

[a]Laboratory for Information and Decision Systems, M.I.T.,
Cambridge, MA, 02139, U.S.A.

[b]Dipartimento di Elettronica, Informatica e Sistemistica,
Universita' della Calabria, 87036 Rende, Italy

In this paper we consider a special type of trajectory optimization problem that can be viewed as a continuous-space analog of the classical shortest path problem. This problem is approached by space discretization and solution of a discretized version of the associated Hamilton-Jacobi equation. It was recently shown by Tsitsiklis [1] that some of the ideas of classical shortest path methods, such as those underlying Dijkstra's algorithm, can be applied to solve the discretized Hamilton-Jacobi equation. In more recent work, Polymenakos, Bertsekas, and Tsitsiklis [2] have carried this analogy further to show that some efficient label correcting methods for shortest path problems, the SLF and SLF/LLL methods of [3] and [4], can be fruitfully adapted to solve the discretized Hamilton-Jacobi equation. In this paper we discuss parallel asynchronous implementations of these methods on a shared memory multiprocessor, the Alliant FX/80. Our results show that these methods are well suited for parallelization and achieve excellent speedup.

1. INTRODUCTION

We consider a trajectory optimization problem that arises in a variety of contexts involving the planning of a motion within a, perhaps irregular, region of two-dimensional or three-dimensional space. A vehicle starts at an initial point x_0 located in some region G of $\Re^m$ and follows a trajectory $x(t)$ such that

$$dx/dt = u(t), \tag{1}$$

where $u(t) \in \Re^m$ is a control vector that must satisfy the constraint $\|u(t)\| \leq 1$ for all t. After a certain time T the vehicle arrives at the boundary ∂G of G where a cost $q(x(T))$ is incurred. There is also a traveling cost $\int_0^T r(x(t))dt$ that depends on the trajectory followed by the vehicle. The objective is to find a trajectory $x(t)$ that starts at the given initial point $x(0)$, ends at the boundary ∂G of G, and minimizes

$$q(x(T)) + \int_0^T r(x(t))dt, \tag{2}$$

*Research supported by National Science Foundation under Grants 9108058-CCR, 9221293-INT, and 9300494-DMI

subject to the constraint $\|u(t)\| \leq 1$ for all t.

We assume that $\inf_{x \in G} r(x) > 0$, which forces the vehicle to reach the boundary of G in finite time. Note that this problem formulation includes the case where we want to reach a given destination point x_f with minimum traveling cost $\int_0^T r(x(t))dt$; we may just take $G = \Re^m - \{x_f\}$ and $q(x_f) = 0$. More generally, we can use the terminal cost $q(x)$ to provide a preference for reaching some portions of the boundary of G over others. In particular, points of the boundary of G with a very high value of $q(x)$ are essentially "forbidden". This allows the introduction of "obstacles" that the vehicle must avoid.

Note that when $r(x(t))$ is identically equal to 1, when G contains several obstacles, and when there is a fixed destination x_f, the problem becomes the shortest path problem of finding a minimum length trajectory that starts at x_0 ends at x_f and avoids the obstacles. This problem has been extensively studied in the robotics and theoretical computer science literature. More generally, the integral cost $\int_0^T r(x(t))dt$ may be viewed as a "generalized length" of the trajectory, and the problem may be viewed as a continuous-space shortest path problem.

Our problem can be approached by classical continuous-time optimal control techniques. However, when the region G contains many "obstacles" that the vehicle must avoid, and/or the cost functions $q(x)$ and $r(x)$ are nonconvex, the problem may be essentially combinatorial and may have multiple local minima. In this case, solution methods based on dynamic programming and discretization of the associated Hamilton-Jacobi equation, which provide globally optimal solutions, are typically preferable.

There is an interesting general method to address the discretization issues of continuous-time optimal control. The main idea in this method is to discretize, in addition to time, the state space $\Re^m$ using some kind of grid, and then to approximate the cost-to-go $V(x)$ of nongrid states by linear interpolation of the cost-to-go values of the nearby grid states. By this we mean that if a nongrid state x is expressed as

$$x = \sum_{i=1}^{m} \xi^i x^i \tag{3}$$

in terms of the grid states $x^1, \ldots, x^m$, where the positive weights ξ^i add to 1, then the cost-to-go of x is approximated by

$$\sum_{i=1}^{m} \xi^i V(x^i), \tag{4}$$

where $V(x^i)$ is the cost-to-go of x^i. When this idea is worked out, one ends up with a *stochastic* optimal control problem having as states the finite number of grid states, and transition probabilities that are determined from the weights ξ^i above. If the original continuous-time optimal control problem has fixed terminal time, the resulting stochastic control approximation has finite horizon. If the terminal time of the original problem is free and subject to optimization, as in the problem of this paper, the stochastic control approximation becomes a Markovian Decision Problem, known as a *first passage* or *stochastic shortest path problem* (see [5] and [6] for a discussion of such problems). We refer to the monograph [7], the papers [8] and [9], and the survey [10] for a description and analysis of continuous-time optimal control discretization issues.

Tsitsiklis [1] showed that the problem of this paper, when appropriately discretized, maintains much of the structure of the classical shortest path problem on directed graphs. In particular, a finitely terminating adaptation of the Dijkstra shortest path algorithm was developed in [1]. In [2], other shortest path methods were adapted to the problem of this paper. These adaptations are extensions to the SLF label correcting method of [3] and the SLF/LLL method of [4]. Computational results given in [3] show that these adaptations vastly outperform (by orders of magnitude) the classical dynamic programming methods, which are based on Jacobi and Gauss-Seidel value iterations.

In the present paper we focus on the efficient parallelization of the Dijkstra, and SLF/LLL methods of [3] and [4] on a shared memory machine. We draw motivation from our earlier work on parallel asynchronous label correcting methods, where we found that the SLF and SLF/LLL approaches lend themselves well to parallel computation, and result in very efficient shortest path solution methods. We concentrate on asynchronous algorithms. Generally, dynamic programming iterations, which contain label correcting methods as a special case, can be executed in a totally asynchronous fashion, as shown in [16]. The textbook [6] contains an extensive discussion and analysis of asynchronous algorithms, including theory that establishes the validity of the asynchronous implementations of the present paper.

Our computational results suggest that the Dijkstra and SLF/LLL approaches are well suited for parallel solution of the problem, and result in excellent speedup. The SLF/LLL method is faster for the problems that we tried, both in a serial and in a parallel environment. This is consistent with the serial computational results of [2]. Since the solution of the problem of this paper is very computationally intensive, we conclude that the gains from parallelization can be significant.

The remainder of the paper is organized as follows. In Section 2 we formulate the trajectory optimization problem, and we describe the discretized version of the Hamilton-Jacobi equation. In Section 3 we describe the Dijkstra method and the SLF/LLL method. In Section 4 we present parallel asynchronous implementations of these methods and computational results.

2. PROBLEM FORMULATION

A *trajectory* starting at $x_0 \in G$ is a continuous function $x : [0,T] \in \Re^m$, where T is some positive scalar, such that $x(t) \in G$, for all $t \in [0,T)$ and $x(T) \in \partial G$. The trajectory is said to be *admissible* if there exists a function $u : [0,T] \to \Re^m$ such that:

$$x(t) = x(0) + \int_0^t u(s)ds, \tag{5}$$

and

$$\|u(t)\| \leq 1, \qquad \forall\, t \in [0,T]. \tag{6}$$

Let G a bounded connected open subset of $\Re^m$ and let ∂G be its boundary. Let also $r : G \to (0,\infty)$ and $q : \partial G \to (0,\infty)$ be two positive-valued cost functions. The cost of an admissible trajectory is given by

$$q(x(T)) + \int_0^T r(x(t))dt. \tag{7}$$

The optimal cost-to-go function $V^* : G \cup \partial G \to \Re$ is defined as follows. If $x \in \partial G$, we let $V^*(x) = q(x)$, otherwise, if $x \in G$, we let $V^*(x)$ be the infimum of the costs of all admissible trajectories starting at x.

The cost-to-go function V^*, under appropriate conditions [11], satisfies the Hamilton-Jacobi equation

$$V^*(x) = \min_{u \in \Re^m,\ \|u\| \leq 1} r(x) + u' \nabla V^*(x), \qquad x \in G. \tag{8}$$

A discretized version of this equation was given and analyzed in [1]. We follow closely the framework of that reference.

Let $h > 0$ some discretization step. Let S and B be discretizations of the sets G and ∂G, whose elements are of the form (ih, jh), where i and j are integers. For each point $x \in S$, we denote by $N(x)$ the set of neighbors of x defined by

$$N(x) = \{x + h\alpha_i e_i \in S \cup B \mid i \in \{1, ..., m\}, \alpha_i \in \{-1, 1\}\}, \tag{9}$$

where e_i is the i-th unit vector of $\Re^m$.

We assume that we have two functions $g : S \to (0, \infty)$ and $f : B \to (0, \infty)$, which represent the discretizations of the traveling and terminal cost functions r and q, resopectively. The function g can usually be defined by $g(x) = r(x)$ for every $x \in S$. The choice of f may depend on the nature of θG because B can be disjoint from θG even if B is a good approximation to θG. We also introduce a function $V : S \cup B \to \Re$, which represents an approximation of the optimal cost-to-go function V^*.

The discretized Hamilton-Jacobi equation (8) is given by

$$V(x) = \min_{\alpha \in \{-1,1\}^m} \min_{\theta \in \Theta} \left[hg(x)\|\theta\| + \sum_{i=1}^{m} \theta_i V(x + h\alpha_i e_i) \right], \qquad x \in S, \tag{10}$$

$$V(x) = f(x), \qquad x \in B, \tag{11}$$

where:

$$\|\theta\| = \sqrt{\sum_{i=1}^{m} \theta_i^2}, \qquad \theta \in \Theta, \tag{12}$$

and Θ is the unit simplex in $\Re^m$,

$$\Theta = \left\{ \theta \in \Re^m \mid \sum_{i=1}^{m} \theta_i = 1,\ \theta_i \geq 0 \right\}. \tag{13}$$

The manner in which Eqs. (10) and (11) approximate the Hamilton-Jacobi equation is explained in [1]. In particular, consider a vehicle that starts at some point $x \in S$ and moves with a unit speed along a direction d, until the point $x + h\sum_{i=1}^{m} \theta_i \alpha_i e_i$ is reached. The direction d is determined by α, which specifies the quadrant within which d lies, and by the choice of θ, which specifies the direction of motion within that quadrant. The total time required to reach the final point is $h\|\theta\|$. The traveling cost is $hg(x)\|\theta\|$, since $g(x)$

is the traveling cost per unit time. With these approximations, it is seen that the optimal cost-to-go function $V^*(x)$ satisfies

$$V^*(x) \approx \min_{\alpha \in \{-1,1\}^m} \min_{\theta \in \Theta} \left[hg(x)\|\theta\| + V^* \left(x + h \sum_{i=1}^{m} \theta_i \alpha_i e_i \right) \right]. \tag{14}$$

Now if in the above equation we use the approximation

$$V^* \left(x + h \sum_{i=1}^{m} \theta_i \alpha_i e_i \right) \approx \sum_{i=1}^{m} \theta_i V^*(x + h\alpha_i e_i), \tag{15}$$

we obtain Eq. (10).

The equations (10) and (11) represent a special case of discretization based on finite elements. In [7], these equations are viewed as Dynamic Programming Equations for a Markov Decision Problem, which can be solved by using methods like value or policy iteration. For a more detailed description of these methods see [7] and [9]. However, as remarked in [1] and [2], these methods do not exploit the special structure of our problem and are relatively slow. In the next section we discuss methods that are much more efficient.

3. LABEL CORRECTING METHODS

The analysis and methodology of [1] and [2] rests on the following fundamental lemma:

Lemma 3.1 *Let $x \in S$, and let $\alpha \in \{-1,1\}^m$, $\theta \in \Theta$, be such that $V(x) = hg(x)\|\theta\| + \sum_{i=1}^m V(x + h\alpha_i e_i)$. Then, $V(x + \alpha_i e_i) < V(x)$, for all i such that $\theta_i > 0$.*

This lemma, which is proved in [1], can be used to show that the prototype label correcting algorithm to be described shortly *terminates in a finite number of iterations* (see [2]). This algorithm maintains a vector $V(x)$ of labels, where $x \in S$, and a candidate list L of states. At the start of the algorithm, the list L contains just an element x_1 of B at which $f(x)$ is minimized, that is:

$$V(x_1) \leq V(x), \qquad \forall\, x \in S \cup B. \tag{16}$$

Also the initial labels are given by

$$\bar{V}(x) = \begin{cases} V(x) & \forall x \in L \cup B, \\ \infty & \text{otherwise.} \end{cases} \tag{17}$$

The algorithm terminates when L is empty and upon termination the optimal cost-to-go of x is given by $\bar{V}(x)$. Assuming L nonempty at a typical iteration, the vector of labels and the candidate list are updated as follows:

1. Let y be a state in L.
   Remove y from L;

2. For each $x \in N(y)$, where
$N(y) = \{y + h\alpha_i e_i \mid i \in \{1, ..., m\}, \alpha_i \in \{-1, 1\}\}$, compute
$\hat{V}(x) = \min_\alpha \min_\theta \left[hg(x)\|\theta\| + \sum_{i=1}^m \theta_i \bar{V}(x + h\alpha_i e_i)\right]$.
If $\hat{V}(x) < \bar{V}(x)$, set
$\bar{V}(x) = \hat{V}(x)$,
and add x to L if x does not already belong to L.

The analog of Dijkstra's method is obtained when the state y exiting L is the state with the minimum value of $\bar{V}$. Reference [1] proves the remarkable fact that in this method, each state will enter and exit L at most once. An efficient way to implement Dijkstra's algorithm is to maintain the list L partially ordered in a binary heap.

If the state exiting the candidate list at each iteration is not a state with minimum label, the required number of iterations can be shown to be finite under our assumptions (see [2]), but some states many enter and exit the candidate list more than once. However, such a method avoids the overhead associated with finiding the node of minimum label. A particularly effective strategy for selecting the state to exit the candidate list was proposed in [3] in the context of the classical shortest path problem. This strategy, called Small Label First method (SLF for short), maintains the candidate list in a double-ended queue Q and inserts a node to the bottom or to the top of Q depending on whether the label of the node is larger than the label of the top node of Q or not. In [2], this method was adapted to the trajectory optimization problem as follows:

1. Let x be a state that enters Q.
Let y be the top state of Q.

2. If $\bar{V}(x) \leq \bar{V}(y)$ then insert x at the top of Q,
else insert x at the bottom of Q.

The state removed from L at each iteration is always the top state of Q. In [4], a more sophisticated state removal strategy, called Large Label Last strategy (LLL for short), is proposed for the classical shortest path problem. In this strategy, the top state of Q is repositioned to the bottom whenever its label is larger than the average node label in Q. In [2], this method was adapted to the trajectory optimization problem as follows:

1. Let $s = \frac{\sum_{x \in Q} \bar{V}(x)}{\|Q\|}$.
Let y be the top state of Q.

2. If $\bar{V}(y) > s$ then move y at the bottom of Q.
Repeat until a state y such that $\bar{V}(y) \leq s$ is found and is removed from Q.

The SLF and LLL strategies have also been combined to solve the trajectory optimization problem in [2]. The serial implementations of the methods of the present section, given in [2], have served as the starting point for the parallel implementations described in the next section.

4. PARALLELIZATION

In this section, we discuss our parallelization of the Dijkstra, SLF, and combined SLF/LLL methods described in the preceding section. Our implementations are similar to the corresponding ones described in [4] for the classical shortest path problem. The prototype label correcting method of the preceding section can be easily parallelized at least for shared memory machines. The basic idea is that several states can be simultaneously extracted from the candidate list and the labels of adjacent states can be updated in parallel. On a shared memory multiprocessor, the label of each state is stored in a unique memory location, shared among all processors. This means that when more processors try to modify simultaneously the label of the same state, they must lock the corresponding memory location to guarantee that only one processor at a time modifies the label of that state. We assume the availability of p processors. For the SLF/LLL method, we have p queues shared among the processors. Each processor i uses only the i-th queue when the LLL state removal strategy is applied and one of the p queues when a state has to be inserted according to the SLF state insertion strategy. In particular, each processor extracts the state x from the top of its queue (or moves it to the bottom of the queue, following the LLL procedure), updates the labels for the adjacent states, and uses a heuristic procedure for choosing the queue to insert a state that enters L. This queue is chosen on the basis of the minimum current value for the sum of states assigned to the queues. As remarked in [4], this heuristic is very easy to implement and ensures a good load balancing among the processors.

In order to parallelize the Dijkstra version of the label correcting method, we maintain a separate binary heap for each processor. Each processor extracts the state at the top of its own binary heap, and whenever a state must be inserted in L, the appropriate binary heap is chosen according to the same heuristic procedure used for the SLF/LLL method. Furthermore, when a processor updates the label of a state already present in L, the same processor reorganizes the corresponding binary heap, in order to keep it ordered.

Even though all the p binary heaps are ordered, the entire list L is not fully sorted. This means that in our parallel implementation of Dijkstra's method a state may exit and reenter L several times in the course of the algorithm. Nonetheless, we refer to this parallel method as the parallel Dijkstra's method, even though the corresponding parallel version is not trully a label setting algorithm. It is worth observing that the overhead for inserting and deleting a state in a binary heap strongly depends on the number of states in the binary heap. This means that by using multiple binary heaps in the parallelization scheme, this overhead is substantially reduced.

In our parallel implementations, both Dijkstra's and the SLF/LLL methods are executed asynchronously, in the sense that a new state may be removed from the list L by some processor while other processors are still updating the labels of other states. For a more detailed discussion on parallel asynchronous iterative methods see [6]. More formally, let $\bar{V}(x,t)$, $t = 0, 1, \ldots$ be the value of the label of state x at time t. This is the value of $\bar{V}(x)$ which is kept in the shared memory location. The label $\bar{V}(x,t)$ is updated at a subset of times $T(x) \subset \{0, 1, ...\}$ by some processor by using the following formula:

$$\bar{V}(x,t+1) = \min_{\alpha} \min_{\theta} \left[hg(x)\|\theta\| + \sum_{i=1}^{m} \theta_i \bar{V}(x + h\alpha_i e_i, \tau_i(x,t)) \right], \quad \text{if } t \in T(x), \qquad (18)$$

$$\bar{V}(x, t+1) = \qquad \bar{V}(x, t), \qquad \text{otherwise.} \qquad (19)$$

In this formula $\tau_i(x,t))$ represents the time at which the label $\bar{V}(x + h\alpha_i e_i)$ has been read from shared memory by the processor updating $\bar{V}(x)$ at time t. The asynchronism is due to the fact that we may have $\tau_i(x,t) < t$ and $\bar{V}(x + h\alpha_i e_i, \tau_i(x,t)) \neq \bar{V}(x + h\alpha_i e_i, t)$.

The convergence of the algorithms can be shown under very weak assumptions. The proof closely resembles the proofs given in [6], Section 6.4, and it will not be given here.

Dijkstra's and the SLF/LLL methods and its parallel asynchronous versions have been implemented and tested on an Alliant FX/80, a vector-parallel computer with 8 processors, each with 23 Mflops of peak performance, having a core memory of 32 MBytes. All the codes are written in Fortran and compiled with the FX/Fortran 4.2 compiler.

We have considered two different sets of randomly generated test problems, which are 2-D and 3-D grids obtained from discretization of a square and of a cube with sides of length that is a multiple of the discretization step h. This ensures that the distance between any two adjacent states in the same direction is always h. We have used a similar approach for generating our test problem to that used in [2]. G is the set of the interior points of the square or the cube, whereas ∂G is the set of the points on the border. S and B are the states on the grids. The values of g in all points in S are randomly generated, according to a uniform distribution in the range [1,1000]. The cost f of all border states is assumed to be infinity, except for two adjacent states in a corner of the square or of the cube. In order to consider test problems that are more realistic, we add some *obstacles*, that is, we assume that $g(x) = \infty$ for some $x \in S$. It is easy to show that these types of problems allow the use of Dijkstra's and the SLF/LLL methods to find the optimal cost-to-go from all interior points to a point on the border B.

The full list of all test problems is reported in Table 1. The percentage of obstacles listed in this table is the fraction of the number of states x in S for which $g(x) = \infty$.

Table 1. List of test problems

Problem	states	Percentage of Obstacles
2-D.1	500x500	0
2-D.2	500x500	0.05
2-D.3	500x500	0.10
2-D.4	500x500	0.15
2-D.5	500x500	0.20
2-D.6	750x750	0
2-D.7	750x750	0.05
2-D.8	750x750	0.10
2-D.9	750x750	0.15
2-D.10	750x750	0.20
2-D.11	1000x1000	0
2-D.12	1000x1000	0.05
2-D.13	1000x1000	0.10
2-D.14	1000x1000	0.15
2-D.15	1000x1000	0.20
3-D.1	25x25x25	0
3-D.2	25x25x25	0.05
3-D.3	25x25x25	0.10
3-D.4	25x25x25	0.15
3-D.5	25x25x25	0.20
3-D.6	50x50x50	0
3-D.7	50x50x50	0.05
3-D.8	50x50x50	0.10
3-D.9	50x50x50	0.15
3-D.10	50x50x50	0.20
3-D.11	75x75x75	0
3-D.12	75x75x75	0.05
3-D.13	75x75x75	0.10
3-D.14	75x75x75	0.15
3-D.15	75x75x75	0.20

The numerical results are collected in 4 different tables, reported below, one for each algorithm and category of test problems (2-D or 3-D grid problems). In these tables, time in secs, and number of iterations required to solve the test problems is reported for each algorithm. For the parallel methods we report also the speed-up values, defined by the ratio between the sequential and parallel execution time.

Table 2. Results of Dijkstra's method for 2-D grid problems

Problem	Sequential	Parallel	Speed-up
2-D.1	248004 / 97.03	254300 / 18.79	5.16
2-D.2	235603 / 88.08	243308 / 17.45	5.05
2-D.3	223169 / 79.68	226118 / 19.76	4.03
2-D.4	210649 / 71.53	213609 / 18.09	3.95
2-D.5	197959 / 63.75	200781 / 16.42	3.88
2-D.6	559504 / 225.71	565726 / 61.08	3.70
2-D.7	531524 / 205.08	537100 / 50.42	4.07
2-D.8	223169 / 79.55	234376 / 16.29	4.88
2-D.9	475314 / 166.91	480871 / 42.65	3.91
2-D.10	446669 / 149.23	451594 / 39.06	3.82
2-D.11	996004 / 410.33	1011577 / 80.56	5.09
2-D.12	946201 / 372.91	966309 / 75.67	4.93
2-D.13	896302 / 337.44	917883 / 69.55	4.85
2-D.14	846122 / 304.09	873970 / 65.07	4.67
2-D.15	795138 / 272.09	876824 / 62.13	4.38

Table 3. Results of SLF/LLL method for 2-D grid problems

Problem	Sequential	Parallel	Speed-up
2-D.1	283062 / 71.59	273319 / 14.35	4.99
2-D.2	266881 / 63.37	259438 / 13.12	4.83
2-D.3	249373 / 55.52	244863 / 15.24	3.64
2-D.4	232731 / 48.50	213609 / 13.81	3.51
2-D.5	216077 / 42.01	213939 / 12.47	3.37
2-D.6	640159 / 162.43	617794 / 32.63	4.98
2-D.7	601712 / 143.15	584866 / 37.87	3.78
2-D.8	249373 / 55.40	245057 / 11.88	4.66
2-D.9	526395 / 109.89	537960 / 26.44	4.16
2-D.10	489954 / 95.70	483180 / 29.79	3.21
2-D.11	1144104 / 291.05	1102015 / 62.95	4.62
2-D.12	1072272 / 255.82	1045384 / 58.12	4.40
2-D.13	1004776 / 224.80	985675 / 54.12	4.15
2-D.14	936121 / 196.13	946082 / 53.09	3.69
2-D.15	869957 / 170.54	861412 / 44.82	3.80

Table 4. Results of Dijkstra's method for 3-D grid problems

Problem	Sequential	Parallel	Speed-up
3-D.1	12168 / 119.92	14664 / 17.64	6.80
3-D.2	11560 / 108.26	14086 / 16.00	6.77
3-D.3	10951 / 92.56	13224 / 13.13	7.05
3-D.4	10342 / 76.58	12604 / 11.67	6.56
3-D.5	9733 / 64.37	11672 / 9.38	6.86
3-D.6	110593 / 1186.40	130801 / 167.56	7.08
3-D.7	105063 / 1003.41	124102 / 149.17	6.73
3-D.8	99534 / 874.61	118458 / 122.81	7.12
3-D.9	94002 / 725.35	111061 / 102.86	7.05
3-D.10	88464 / 626.08	104073 / 85.33	7.34
3-D.11	389018 / 4283.12	445048 / 590.98	7.25
3-D.12	369567 / 3649.76	420662 / 506.28	7.21
3-D.13	350116 / 3121.77	393070 / 437.26	7.14
3-D.14	330657 / 2619.21	386584 / 371.81	7.04
3-D.15	311176 / 2201.13	357460 / 307.05	7.17

Table 5. Results of SLF/LLL method for 3-D grid problems

Problem	Sequential	Parallel	Speed-up
3-D.1	20007 / 232.70	18635 / 19.87	11.71
3-D.2	18654 / 204.66	18390 / 17.16	11.93
3-D.3	17929 / 178.14	16881 / 13.76	12.95
3-D.4	17219 / 154.58	16132 / 9.94	15.55
3-D.5	14714 / 115.20	14673 / 9.58	12.02
3-D.6	180871 / 2194.08	173213 / 159.84	13.73
3-D.7	173189 / 1930.34	159966 / 138.56	13.93
3-D.8	152481 / 1505.92	151203 / 123.76	12.17
3-D.9	143647 / 1278.71	141053 / 89.55	14.28
3-D.10	130514 / 1030.12	126106 / 93.76	10.99
3-D.11	630325 / 7852.97	551956 / 624.99	12.56
3-D.12	592540 / 6639.18	528597 / 523.56	12.68
3-D.13	525829 / 5265.43	503412 / 473.25	11.13
3-D.14	499068 / 4513.38	497054 / 303.18	14.86
3-D.15	444179 / 3506.93	457312 / 292.57	11.99

In Tables 6 and 7 we aim to summarize the performance of the various methods. In particular, we compare the methods following an approach that is similar to the one proposed in [15], by giving to each method and for each test problem, a score that is equal to the ratio of the execution time of this method over the execution time of the fastest method for the given problem. Thus, for each method, we obtain an average score, which is the ratio of the sum of the scores of the method over the number of test problem. This average score, given in Tables 6 and 7, indicates how much a particular method has been slower on the average than the most successful method.

Table 6. Ranking on 2-D grid problems

Algorithm	Number of Processors	Performance Index
DIJKSTRA	1	6.02
	8	1.37
SLF/LLL	1	4.12
	8	1.00

Table 7. Ranking on 3-D grid problems

Algorithm	Number of Processors	Performance Index
DIJKSTRA	1	7.34
	8	1.05
SLF/LLL	1	13.26
	8	1.04

Note that for 2-dimensional problems, the SLF/LLL method is faster than Dijkstra's method in a sequential environment, despite the smaller number of iterations of Dijkstra's method. This is due to the extra overhead for finding a state with minimal label in Dijkstra's method. For 3-dimensional problems, however, each iteration is much more costly than for 2-dimensional problems, and as a result the sequential Dijkstra's method is faster than the sequential SLF/LLL method. Nonetheless the parallel versions of the two methods are competitive for 3-dimensional problems because the SLF/LLL method exhibits greater speedup in our experiments. Furthermore, the parallel version of Dijkstra's method requires substantially more iterations than its serial counterpart. The superlinear speedup of the parallel SLF/LLL method is somewhat unexpected. It is due to the fact that the computation of the cost-to-go function in the 3-dimensional case often simplifies to a 2-dimensional computation. We have experimentally observed that this simplification occurs much more frequently in the parallel SLF/LLL method than in its sequential version.

REFERENCES

1. J. N. Tsitsiklis, Efficient Algorithms for Globally Optimal Trajectories, Tech. Rep. LIDS-P-2210, Laboratory for Information and Decision Systems, M.I.T., October 1993.
2. L. C. Polymenakos, D. P. Bertsekas, and J. N. Tsitsiklis, Extensions and Implementations of Efficient Algorithms for Globally Optimal Trajectories, unpublished report, Laboratory for Information and Decision Systems, M.I.T., Cambridge, Ma, U.S.A., 1994.
3. D. P. Bertsekas, A Simple and Fast Label Correcting Algorithm for Shortest Paths, Networks, 23, 703-709, 1993.
4. D. P. Bertsekas, F. Guerriero, and R. Musmanno, Parallel Asynchronous Label Correcting Methods for Shortest Paths, Journal of Optimization Theory and Applications, 1995 (*toappear*).
5. D. P. Bertsekas, and J. N. Tsitsiklis, An Analysis of Stochastic Shortest Path Algorithms, Math. Operations Res., Vol. 16, pp. 580-595, 1991.
6. D. P. Bertsekas, and J. N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, N.J., 1989.
7. H. J. Kushner, and P. G. Dupuis, Numerical Methods for Stochastic Control Problems in Continuous Time, Springer-Verlag, New-York, 1992.
8. M. Falcone, Numerical approach to the infinite horizon problem of deterministic control theory, Applied Mathematics and Optimization, Vol. 15 (1), pp. 1-13, 1987.
9. R. Gonzales, and E. Rofman, On Deterministic Control Problems: an Approximation Procedure for the Optimal Cost, I, Stationary Problem, SIAM Journal on Control and Optimization, 23, pp. 242-266, 1985.
10. H. J. Kushner, Numerical methods for stochastic control problems in continuous time, SIAM Journal on Control and Optimization, Vol. 28 (5), pp. 999-1048, 1990.
11. W. Fleming, and R. Rishel, Deterministic and Stochastic Optimal Control, Springer-Verlag, New York, 1975.
12. R. Bellman, Dynamic Programming, Princeton University Press, Princeton, N.J., 1957.
13. F. Glover, R. Glover, and D. Klingman, The Threshold Shortest Path Algorithm, Networks, 14, 1986.
14. U. Pape, Implementation and Efficiency of Moore-Algorithms for the Shortest Path Problem, Mathematical Programming, 7, 212-222, 1974.
15. A. A. Brown, and M. C. Bartholomew-Biggs, Some Effective Methods for Unconstrained Optimization Based on the Solution of Systems of Ordinary Differential Equations, Tech. Rep. 178, Numerical Optimization Centre, The Hatfield Polytechnic, Hatfield, UK, 1987.
16. D. P. Bertsekas, Distributed Dynamic Programming, IEEE Transactions on Aut. Control, Vol. AC-27, pp. 610-616, 1982.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Financial Applications and HPF

Adriaan Joubert

The London Parallel Applications Centre, QMW, Mile End Road, London E1 4NS, UK

1. Introduction

This paper discusses the requirements for high-performance computing in finance, and the need for a standard parallel computing language. Some application areas and the specific requirements of the finance industry are considered initially in section 2. It is then proposed that HPF may be a (partial) answer to the industry's need for a standard and portable parallel language.

To illustrate the use of HPF in practice, and especially the issues arising in writing portable applications, two examples are investigated in section 7 and 8. These are American option pricing and the real-time statistical analysis of trading data, is analysed in some detail. These two examples were implemented in High Performance Fortran (HPF) on both an array processor, as well as a coarse-grained multi-processor. The disparity of algorithms on different architectures and the need for libraries of high level routines receives special attention.

Finally conclusions are drawn on the suitability of HPF for applications in finance.

2. Financial applications

In September 1992 *The Economist* already estimated that global turn-over in the foreign exchange markets was around 1 trillion dollars per day (more than the foreign exchange reserves of all IMF members), about $303bn of which in London alone [6]. Millions of pounds can be at stake in a single transaction. Traders must have up to date information on the state of the market; this is especially important with the move away from trading on the stock exchange to telephone and electronic trading. Whereas excitement in the pit would immediately convey information to a trader, he now relies completely on the information appearing on the screens in front of him.

Pension funds move billions of pounds around the globe to wherever the best return on an investment is possible. They have to optimise their asset allocation, keep track of the risk of their portfolios and hedge this risk appropriately.

Applications in the financial sector fall broadly into two categories: those which are very large, such as simulations and optimisation problems, and those which are subject to severe time constraints. The selection of portfolios or the long-term assessment of the risk in a portfolio fall into the first category, while tools to support traders tend to fall into the second.

2.1. Statistical analysis

One of the tools for traders is the statistical analysis of trading data. This provides estimates of the volatility and return of shares as well as the correlation between shares.

With advances in technology it is now possible to do this type of analysis on real-time trading (or tick) data, and this application is discussed in section 8. It has also become possible to calculate the behaviour of whole indices with respect to one another in real-time. This enables traders to see how their book of shares, or a position they wish to take, has behaved over recent time.

Traders can be notified of changes in the behaviour of instruments as they happen. Typical questions are: when does a stock trade outside its expected band, when does the correlation between stocks change, and when does a stock's volatility change. Often an instrument's behaviour deviates from the normal for only a very short period of time, which offers trading or arbitrage opportunities if spotted quickly enough.

It is also interesting to see how much the short-term behaviour of shares deviates from their long-term behaviour. The much higher volatility seen over short periods of time must be taken into account when assessing the risk of holding large numbers of shares for short periods between transactions or when writing options over short time periods.

Other applications for which statistical estimates are needed include option valuation and optimal portfolio selection [14]. Estimates of volatilities and correlations are also important for *risk* and *capital requirements* assessment. The last issue has become very important due to the European Capital Adequacy Directive, which requires trading houses to set capital aside against default by counterparties or unfavourable market movements. The amount of capital that must be set aside is dependent on the volatility of the instruments, and can be reduced if instruments are negatively correlated.

2.2. Pricing and risk assessment of a book

Lately there have been some spectacular losses because companies have not assessed the risk they were running correctly. Among these are the staggering losses made by the US subsidiary of Metallgesellschaft ($1.4bn) on oil futures [8] and the Japanese subsidiary of Shell, Showa Shell, as well as Kashima Oil which lost $1.05 bn [7] and $1.5bn [8] respectively with foreign exchange forward contracts. Therefore managers want to keep a closer eye on the development of a portfolio they own. For large banks this can mean recalculating the value of many thousands of instruments.

Keeping track of the riskiness of a portfolio is equally important. Apart from the volatility of every instrument, such calculations must also take account of cross-correlations, currency risk and hedging with, as well as exposure to, derivatives. While the riskiness of a portfolio at the end of a day is usually within pre-set bounds, the risk run during the day can be much larger.

J.P. Morgan recently published its risk assessment methodology [17]. In order to assess risk the volatilities of and correlation between more than 300 different financial instruments are needed. With novel high performance computers it is possible to calculate this information quickly *and* look at the way these parameters develop over the short as well as the long term.

2.3. Option pricing

Derivatives have become very important instruments for reducing risk as well as for speculation. Derivatives depend on the value of an underlying asset, for example, an *option* gives the right to buy or sell the underlying asset at a specified price by a certain date (the *expiration date* or *maturity*). *American options* can be exercised at any time

until the expiration date and *European options* can only be exercised on the expiration date itself. Many derivatives are traded on exchanges. Most options traded on exchanges are American options [12, p.5]. Exchange traded options are written (sold) on many underlying instruments such as exchange rates, interest rates, equities, indices such as the FTSE, hard (e.g. gold) and soft (e.g. cocoa) commodities and energy (mostly oil).

Lately there has been a large increase in the number of OTC (Over The Counter) options. These are options written for a customer to hedge a specific risk, e.g. a company's exposure to exchange rate risk. These options often have conditions attached to them which make them very difficult to price.

Options are modelled with stochastic differential equations. European options can be priced in closed form with the celebrated Black-Scholes formula [3]. If the strike price depends on more than one asset, e.g. the minimum price of a number of assets, Black-Scholes cannot be used and a multi-dimensional integral must be solved. Monte-Carlo methods can solve very high-dimensional integrals in seconds on parallel computers [1].

Pricing American options is much more difficult, as it requires the solution of a free boundary problem. Lattice methods are used for the solution of such problems, and with these, options written on a single asset can be priced [12]. Although this is a very compute-intensive process, it is easy to parallelise and such an implementation is discussed in detail in section 7. For larger numbers of underlying assets the problem quickly becomes computationally intractable. Approaches which allow the use of Monte-Carlo methods [2] and linear programming [5] for the valuation of American (or more general path-dependent) options were recently described.

2.4. Portfolio optimisation

The goal of portfolio optimisation is to select a portfolio of assets which maximise some given reward criterion. As prices and volatilities change, the portfolio must be adjusted to maintain the level of return or the acceptable level of risk. These calculations involve the solution of large optimisation problems (usually quadratic programming problems). In general portfolio optimisation is a stochastic control problem which may require the solution of hundreds of individual optimisation problems. Standard portfolio optimisation does not take transaction costs into account, but assumes that the portfolio can be adjusted smoothly and continuously. Attempts have been made to model the problem including transaction costs, but this becomes computationally intractable for more than two or three assets. An examples of a large-scale parallel portfolio optimisation problem is that developed by King [15] for the Allstate Insurance Company. Dempster's group at Essex University has also implemented large-scale, parallel portfolio optimisation programs.

2.5. Securitisation

Since the early eighties there has been a move towards the securitisation of many assets, i.e. bundling loans or investments and selling off shares. This has happened extensively in the US where large numbers of mortgages from individual homeowners are pooled. Investors purchase interest in the pool (so-called mortgage-backed securities) and receive pro-rated shares of the pool's cash-flow [20]. The motivation for banks is that it frees up much of their capital. To investors this seemed a low-risk investment, as the number of house-owners defaulting on their loans can be assessed. Unfortunately many house-owners

re-mortgaged their houses when interest rates were low and investors lost millions.

The valuation of mortgage backed securities contains aspects of both options and bonds. This is a very compute-intensive task which requires the simulation of many different interest-rate scenarios over a period of thirty years. This was one of the first widely publicised uses of supercomputers on Wall Street. For details on the parallel implementations of these algorithms see [21].

2.6. Further applications

These are just some areas in finance which require a lot of computing power. Transaction processing, the calculation of insurance premiums, the simulation of the impact of a change in interest rates on a portfolio of loans and mortgages, or bond pricing have not been discussed. The whole area of data mining has not been mentioned either. All these applications require significant amounts of compute power.

3. Parallel servers

Some of the problems discussed in the previous section are so large that they outstrip the speed of even the fastest sequential processors. The availability of relatively cheap parallel compute and database servers suggests that these requirements can be fulfilled cost-effectively with parallel machines. Although a few parallel machines have been used on Wall Street and in some of the banks, their wide-spread use in the financial industry would mark a significant digression from the traditional scientific and engineering markets. The requirements in the financial market place are however very different from those in science and engineering.

Users' machines tend to be PCs or workstations, leading to the usual Client/Server problems, which are currently being addressed within the Windows NT Advanced Server, Unix and OpenVMS worlds. Client-Server issues are not further addressed in this paper. The big advantage of client-server computing is flexibility and cost-effectiveness: if a small calculation is needed this can run on a small machine, while the large machines are reserved for the calculations that really require their compute power. Such a system is also much more robust — if one server goes down another can take its place.

On the software side this flexibility has one serious consequence: the same software must run on a variety of machines and offer reasonably good performance on all machines. It is not an option to rewrite software for every new architecture in the network, as this would make maintenance a problem and would drive software production costs up.

4. Software requirements

In the scientific and engineering community the primary reason for using parallel computers is speed, and all other considerations take second place. In finance the priorities centre more on robustness and availability than speed. The primary software requirements are listed briefly below.

Robustness In real-time applications such as those in finance, the correctness and availability of results is an over-riding concern.

Maintainability To guarantee availability the software must be easily maintainable.

Portability To avoid dependency on a single machine or a single vendor the software must be portable. There is a large emphasis on the adherence to (de-facto) standards.

Performance Many financial applications are very compute intensive or are subject to strict time constraints.

Investment protection The investment made in staff training and in software must be protected. This restricts the languages that can be used to Fortran and C, as well as various spread-sheet and database languages.

Modularity Many functions re-appear in different calculations. One of our clients put the cost of rewriting functions that are available, but hidden inside other applications, at several hundred-thousand pounds a year for his site alone. There is increasingly an insistence on re-usable software.

Programmer productivity The costs of implementing new applications and integrating them with existing applications must be acceptable. This can be a large problem on parallel computers.

From these requirements it follows that a *standard* parallel language, based on Fortran or C, is needed. Such a language must be *portable* between different operating systems and architectures, and deliver acceptable performance on all of them.

One of the first decisions that must be made when choosing a language is which programming paradigm to use. The majority of parallel systems are MIMD systems and the paradigm of choice is message passing. Practically all MIMD programs depend on data flow, i.e. there are distinct programs running on every processor which perform calculations triggered by messages sent to them. Quite apart from the danger of dead-lock, problems can occur when the data flow changes for different data sets.

A case we came across was the following: a sparse matrix application worked correctly for over six months on many different data sets, and then suddenly produced wrong answers on a new data set. When the problem was finally tracked down, it proved to be the result of two messages arriving at a node in a different order.

The increased complexity of message passing programs also means that programmer productivity is lower and maintainability is a serious issue. Furthermore such programs are still not very portable (the new MPI standard [9] may help in this respect), and require extensive retraining of staff. Clearly message passing is unsuitable for main-stream financial applications.

The data parallel programming paradigm has a single-threaded control structure and a global name space. This makes it very similar to the traditional Von Neumann model, which makes programs easier to understand and maintain, and simplifies the migration from single-processor to parallel machines. But sparse and irregular problems are difficult to implement efficiently within this model.

The Single Program Multiple Data (SPMD) model retains the single control flow, but offers more flexibility for irregular problems. It is often described as *data parallel programming with loosely synchronous parallel execution* [16].

5. Fortran 90 and High Performance Fortran

For numerical applications Fortran 90 is the language of choice. The reasons are primarily the large number of existing software libraries and the many intrinsics. Array statements greatly simplify the writing and understanding of code, as well as providing a natural way of describing data parallelism. Further enhancements that are important from a software maintenance point of view, are *modules*, which give access to global data in a way that can be checked by the compiler, and *allocatable/automatic arrays* which provide a clean way of allocating data dynamically. Within Fortran 90 most of the criteria mentioned in the previous section are satisfied. The migration from Fortran 77 to Fortran 90 is often harder than expected, but the advantage of having a more modern language is well worth it in terms of maintainability and reliability. There is also thirty years of experience in maintaining Fortran programs.

The main problem with Fortran 90 is that it only provides portability over a very limited range of parallel machines, these being vector machines and array processors. To address the large number of MIMD multi-processors, especially those offered by many workstation vendors, Fortran 90 is not sufficient.

High performance Fortran (HPF) was developed specifically to address the modern parallel architectures. It is the *only* widely accepted (de-facto) standard for a parallel programming language. The HPF working group [16] defined its mission as defining language extensions to Fortran supporting:

- A data parallel programming paradigm.
- Top performance on MIMD and SIMD computers with non-uniform memory access costs.
- Code tuning for various architectures.

Some of the new features included in HPF are the following.

- Directives and statements for data distribution:
  - `TEMPLATE` and `PROCESSORS` statements to describe a (virtual) machine.
  - `DISTRIBUTE` and `ALIGN` to specify how to map data on to a (virtual) machine.
  - `REDISTRIBUTE` and `REALIGN` to change the data mapping.
  - `INHERIT` to inherit a data mapping in a subroutine from an argument.
  - `SEQUENCE` for compatibility with code requiring a linear memory model.
- Added features for Data Parallel Programming:
  - `FORALL` (Fortran 95).
  - Scan functions `SUM_PREFIX`, `SUM_SUFFIX` etc.
  - Scatter functions `SUM_SCATTER`, `PRODUCT_SCATTER` etc.
  - `PURE` functions (Fortran 95).

The importance of HPF lies in the fact that it includes all of the Fortran 90 standard, so that old libraries can still be used, but provides a high-level method of describing parallel programs for a wide range of architectures. HPF is still limited in its ability to handle irregular problems, which is a serious problem for optimisation codes required for e.g. portfolio optimisation. These issues are addressed in the follow-up project to HPF.

To assess whether HPF satisfies the industry's requirements, and especially whether it achieves the goal of portability in practice, two applications are described in detail. The first is the valuation of American options (section 7) and the second is the statistical analysis of trading data (section 8). Both are fairly regular problems, and are in that respect not representative of financial applications in general. These are real-life applications.

As HPF compilers are still in their infancy, the emphasis here is on investigating whether these applications can be implemented efficiently in a portable manner, rather than on performance on specific machines. As mentioned earlier the financial community is prepared to accept a limited performance deterioration for increased portability.

First we present the architectures onto which the programs are mapped.

6. Architectures

Two architectures are considered in the applications discussed below. The first is a data parallel architecture (MasPar) and the second a coarse-grained MIMD architecture (Digital AlphaFarm). The MasPar consists of 8192 processors; a virtual machine to map data onto can be either a rectangular processor grid

```
!HPF$ PROCESSORS :: MPPGRID(128,64)
```

which corresponds to the physical layout of the machine, or

```
!HPF$ PROCESSORS :: MPP(8192)
```

which considers the machine to be a linear array of processors. As MasPars are built with varying numbers of processors it is convenient to use `NUMBER_OF_PROCESSORS()`, an HPF intrinsic, to determine the number of processors:

```
!HPF$ PROCESSORS :: MPPGRID(NUMBER_OF_PROCESSORS(1), &
!HPF$                                NUMBER_OF_PROCESSORS(2))
!HPF$ PROCESSORS :: MPP(NUMBER_OF_PROCESSORS())
```

The coarse-grained AlphaFarm with relatively few processors (4-12) is viewed as a linear array.

```
!HPF$ PROCESSORS FARM(NUMBER_OF_PROCESSORS())
```

The big advantage of these definitions is that the programmer can work on a convenient virtual machine and leave the mapping of the virtual on to the physical machine to the compiler.

MasPar does not yet support HPF. A translation script was developed which converts Fortran 90/HPF programs (with some limitations) to MasPar Fortran programs. A large part of the HPF library was also implemented on the MasPar. The MasPar data mapping directives are given in footnotes to the HPF directives where appropriate.

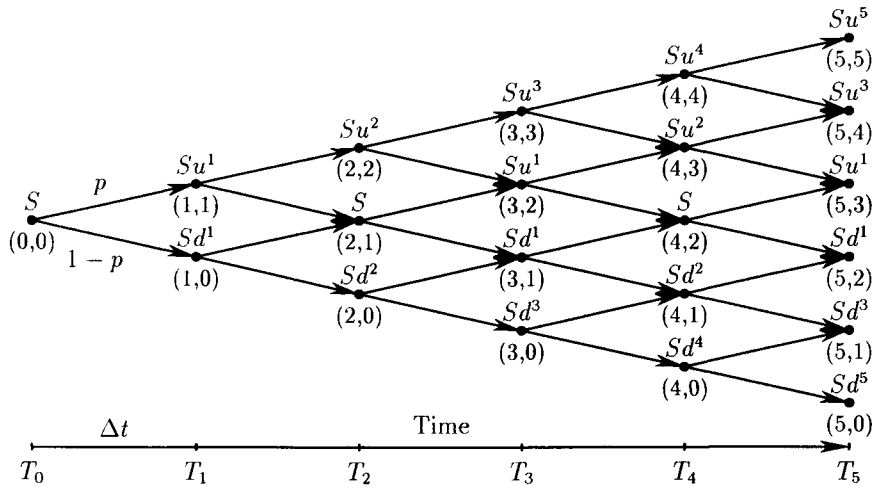


Figure 1. A tree of stock prices used to value a stock option. [12, p.338].

Digital's new Fortran 90 compiler for AlphaFarms supports (nearly) full HPF. LPAC is a β-test site for this compiler, as this compiler has not yet been released no performance details can be given.

7. Application: Valuation of American options

In this section the valuation of *American options* is investigated. An American option gives the bearer the right to buy (call option) or sell (put option) an underlying asset for a pre-determined price (the *strike price*) at any time up to the option's expiration date. Options are used to hedge positions and for speculation. The algorithms for American option pricing are fundamental for the pricing of many path-dependent options, such as caps and collars.

In the simplest case the parameters that determine the value of an option on a stock are the current *stock price* (S), the *strike price* (K), the *risk-free rate of interest* (r), the *time until expiry* (T) and the *volatility* (σ). The difficulty in valuing American options is that they can be exercised at any time up to maturity.

The first step in the valuation process is to create a tree of stock prices. From the parameters a probability p for a stock price moving up or down at every node is computed, as well as the change in stock price when moving up (u) and when moving down (d). For details on how to calculate p, u and d see [12, p. 336]. This leads to a tree of stock prices as shown in figure 1 (that tree corresponds to $N = 5$ time-steps). Denote the stock price in node (i, j) as S_{ij}.

Consider the valuation of a put option. The value of the option is known exactly at the expiry date: it is $(K - S)^+$, that is the difference between the strike and the stock price if the strike price is higher than the stock price, otherwise zero. Without going into details it is possible to calculate the expectation of the value of the option in the previous time-step by taking the risk-neutral probabilities into account: if the option price at time

T is $f_{Nj} \equiv (K - S_{Nj})^+$, the option price at time-step $N-1$ is

$$f_{N-1,j} = e^{-r\Delta t}[pf_{N,j+1} + (1-p)f_{N,j}] , \qquad j = 0, \ldots, N-1, \tag{1}$$

where the factor $e^{-r\Delta t}$ is the discount rate.

This may not be the correct value of the option at time-step $N-1$, as American options can be exercised early. Therefore, a decision must be made whether it is more profitable to retain the option or to sell it. If the stock price in the node of the stock price tree is $S_{N-1,j}$, the value of the option, if it is exercised, is $(K - S_{N-1,j})^+$. If $(K - S_{N-1,j})^+ > f_{N-1,j}$ the option would be exercised, i.e. the correct price of the option is

$$f_{N-1,j} = \max\left\{(K - S_{N-1,j})^+,\ e^{-r\Delta t}[pf_{N,j+1} + (1-p)f_{N,j}]\right\} , \qquad j = 0, \ldots, N-1. \tag{2}$$

The value f_{00} obtained recursively at the root of the tree is the value of the put option.

For the assumptions made in this model to be valid, the time-step Δt must be chosen very short. Consequently N is very large and a calculation with $N = 7500$ takes nearly 5 minutes on a SUN SparcStation. Practitioners tend to use only around 500 timesteps due to this time-constraint, and consequently get quite inaccurate answers. On an 8000 processor MasPar the calculation time for $N = 7500$ is reduced to 7.4 seconds. Implementations of American option valuation on the MasPar were also reported by Mills et. al. [18]. It should also be pointed out that, especially for short maturities, it is preferable to use trinomial trees (i.e. stock price trees where the price can go up, down or remain constant) as this avoids 0-1 cycling effects. This increases the required calculation time by half.

On both array processors and workstation farms the calculation can be parallelised trivially by calculating all the function values f_{ij} for $j = 0, \ldots, i$ in step i in parallel. As the values of $f_{i+1,j}$ are not needed any more, they can be overwritten in the calculation. The calculation of the option value in time-step i can be expressed (in Fortran 90) as

```
F(0:I) = EXP(-R*DeltaT)*(P*F(1:I+1) + (1-P)*F(0:I))
F(0:I) = MAX( F(0:I), K-S(0:I) )
```

where S are the stock prices (which can be easily pre-calculated) and K the strike price.

On an array processor the most efficient way to parallelise this calculation is to place the successive elements in F in adjacent processors. The HPF directive[1]

```
!HPF$ DISTRIBUTE (CYCLIC) ONTO MPP :: F, S
```

achieves this. In every step of the calculation every processor communicates one element of F with its neighbour, which is very fast on array processors, and the maximum available parallelism is exploited.

On a processor farm the situation is quite different: a cyclic distribution may keep the maximum number of processors busy, but the high communication costs would quickly eradicate any gains made through the increased parallelism. The aim is to exploit the available parallelism at the start of the calculation, but reduce the number of processors that are used towards the end of the calculation. This can be achieved as shown in figure 2 with the HPF directive

[1]On the MasPar this is equivalent to the directive `CMPF MAP F(allbits), S(allbits)`.

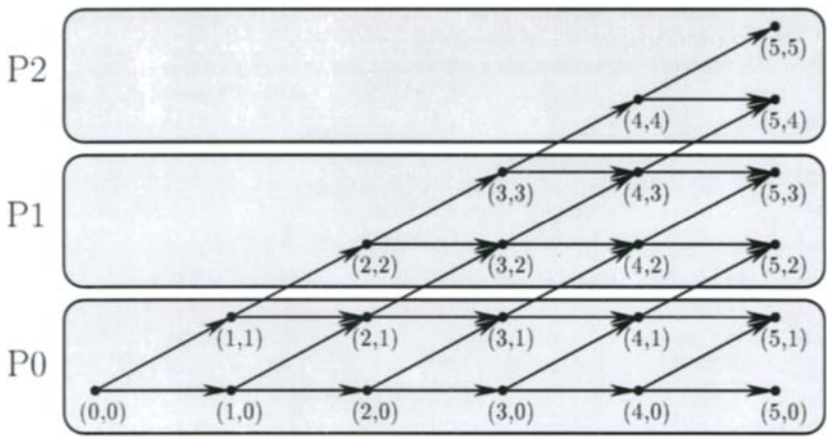


Figure 2. The block mapping of a stock price tree onto an MIMD machine. Towards the end of the calculation fewer and fewer processors are used to minimise communication.

```
!HPF$ DISTRIBUTE (BLOCK) ONTO FARM :: F, S
```

If there are only very few processors and the tree is very large this may not be the optimal distribution. Processors may have finished their block of the tree, but there may still be enough work left to keep all processors busy without communication costs nullifying any gains. In this case it is better to use a block-cyclic distribution

```
!HPF$ DISTRIBUTE (CYCLIC(L)) ONTO FARM :: F, S
```

This increases the amount of communication that is needed during the calculation, but inproves the load-balance. How `L` should be chosen depends on the ratio between the computation and communication speed.

The benefit derived from HPF in this application is that exactly the same code is used for efficient implementations on two totally different architectures by only changing a directive.

Porting applications to different architectures is not always this straightforward and a more difficult application is discussed next.

8. Application: Statistical Analysis of trading data

LPAC has developed a novel application for the statistical analysis of tick (real-time) data. The objective was to obtain quote and trade data for the UK equity markets from a data feed, to compute real-time estimates of volatilities, returns and correlations and to use these values in other calculations. As mentioned in section 2.1 these calculations are of great practical importance. This is a real-life application, in which timeliness of information is of utmost importance: late information is useless information.

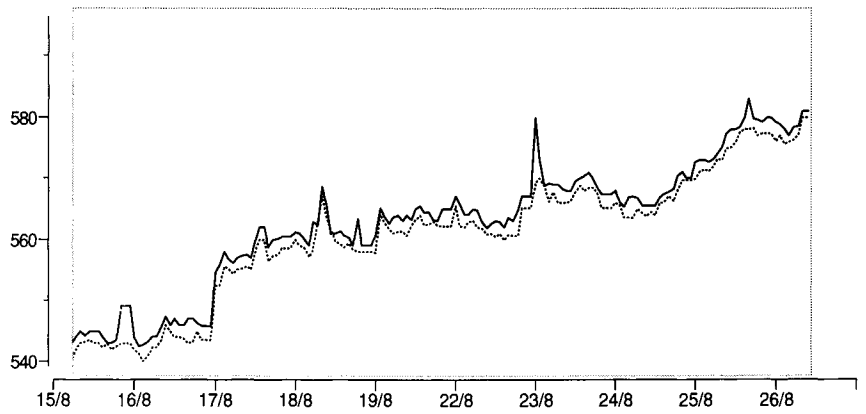


Figure 3. The bid (dashed line) and ask (solid line) share price of Barclays over 10 days, sampled in 30 minute time intervals. The samples were taken from 07h30 until 16h30 GMT on the 15th August up to and including the 26th of August 1994.

Before discussing the implementation, the data that is used and the algorithms are described. Three aspects of the implementation are detailed, each showing different uses of HPF and problems with porting efficient programs between architectures.

8.1. Pre-processing the data

A *trade record* is a notification that a stock was traded. It includes the *time* the trade took place, the *price* at which the stock was traded and the *volume* that was traded. This data is pre-processed to obtain discrete time-sequences amenable to simple statistical calculations.

For quote driven markets, such as London, the first step in this process is to separate trade records reporting *bid* and *ask* trades, i.e. whether shares were sold to or bought from a market maker. The separation of bid and ask trades is important as there is always a price difference between them (the market maker's spread), as shown in figure 3. Otherwise many non-existent movements in the share price are introduced, giving the impression that the share price is much more volatile than it is in reality.

Trades are not reported in temporal order. Especially *delayed* trades, that is trades with much larger than average volume, only have to be reported within 90 minutes of being executed.

A characteristic of dealing prices is that small trades often have widely fluctuating prices while large trades are close to the mid-price of a share. The main reason is that large trades generally command sharper prices than small trades. To iron out the price-differences for different trade-sizes the volume must be taken into account.

To obtain a discrete time-sequence of prices the volume-weighted average of the trade prices, over constant time intervals, is computed. For a sequence of trades of a single share at times t_i with prices P_i and volumes V_i, the average price in time interval $I_k = [T_k, T_k+T)$

is

$$\bar{p}_k = \sum_{t_i \in I_k} P_i V_i \Big/ \sum_{t_i \in I_k} V_i \ . \tag{3}$$

To obtain a continuous sequence over many days only time intervals during trading hours are considered, so that nights, weekends and holidays are excluded.

The statistical calculations are computed from the sequence of changes in the log prices, i.e. from a sequence

$$x_k = \log \bar{p}_k - \log \bar{p}_{k-1} \ . \tag{4}$$

The reason for using log-price changes is that these can be modelled with a Wiener process (Brownian motion). This is important for the pricing of derivatives.

8.2. Formation of estimates

General methods for the calculation of statistical estimates are discussed in this section. For a sequence updated in real-time the simplest approach is to use geometric means over the sampling time.

Let $\{X_n\}$ be a sequence of independent random variables with common mean μ and common variance σ^2. To obtain a rolling estimate of the mean at time interval n define

$$\hat{\mu}_n = \sum_{r \geq 0} \theta^r (1 - \theta) X_{n-r} \qquad X_i = X_0 \text{ for } i < 0 \tag{5}$$

with $\theta \in (0, 1)$. The parameter θ is related to the mean look-back time T_L by

$$T_L = \frac{\theta}{1 - \theta}. \tag{6}$$

For $\theta = 0.95$ the mean look-back time is $T_L = 19$ (time intervals), for $\theta = 0.90$ it is $T_L = 9$ and for $\theta = 0.8$, T_L is only 4. With a smaller value for θ the look-back time thus decreases very quickly and the process becomes much more sensitive to changes in $\{X_n\}$.

It is also worth noting that this sequence can be updated easily as new data comes in:

$$\hat{\mu}_{n+1} = (1 - \theta) X_{n+1} + \theta \hat{\mu}_n \tag{7}$$

This equation shows clearly that if θ is close to 1, new values X_{n+1} only have a small effect so that the estimate $\hat{\mu}$ looks back over a large time scale, but that if θ is close to zero X_{n+1} has a large effect and the look-back time is short.

We could attempt to estimate the variance σ^2 similarly by defining

$$v_n = \sum_{r \geq 0} \lambda^r (1 - \lambda)(X_{n-r} - \hat{\mu}_{n-r})^2 \tag{8}$$

However v_n is biased! An unbiased estimator of σ^2 is

$$\hat{\sigma}_n^2 = \frac{1 + \theta}{2\theta^2} v_n \ . \tag{9}$$

The volatilities of two indices, one of which is user-defined, and two individual banks are shown in figure **4**. Similar formulas can be derived for the covariance and the correlation. For details see [14].

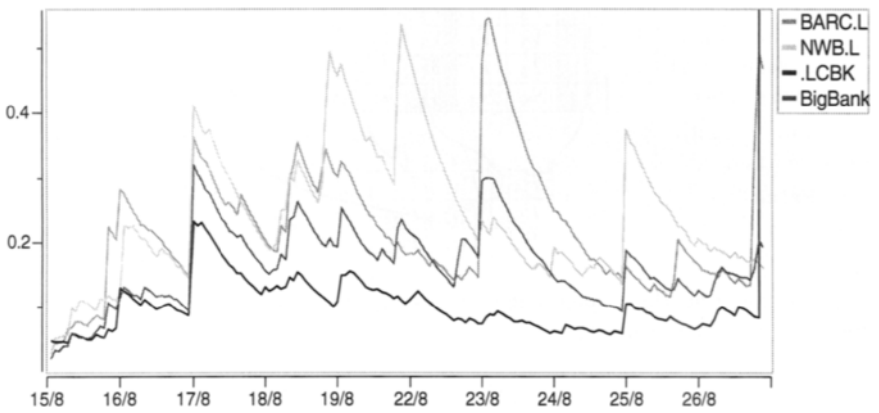


Figure 4. The volatility of a user defined index BigBank consisting of Barclays, NatWest and Lloyds shares, weighted according to market capitalisation, and the Barclays and NatWest share prices, as well as the banking sector .LCBK over 10 days, sampled in 30 minute time intervals with lag parameters $\theta = \lambda = 0.90$. The samples were taken from 07h30 until 16h30 on the 15th August up to and including the 26th of August 1994.

8.3. Aspects of the implementation

In this section difficulties that were encountered in the efficient, portable implementation of a statistical analysis program, with the functionality described above, are discussed. Three problem areas are considered:

1. mapping share identifiers onto array indices used in the calculation,
2. the computation of the volume-weighted average prices defined in equation (3), and,
3. the calculation of the linear recurrences required for the statistical estimators (equations (5) and (9)).

8.3.1. Parallel hash-table look-ups

Consider a time series of 5 minute time intervals for 10 hours of trading per day over 30 days (3600 time intervals in total). On the London Stock Exchange alone there are between 15 and 25 thousand trades a day. Assume 18000 trades per day, so that there are approximately 540000 trades in a 30-day period. To do the calculations the share numbers must be mapped onto an array index, as shown in figure 5. If share A in this example is e.g. Barclays, with share number 78201, all occurences of sharenumber 78201 in the 540000 trades must be mapped on to 1. As there are thousands of share numbers in practical applications, this is a very time-consuming operation.

The array-indices are looked up in a hash-table, and these look-ups can be done independently for every reported trade. In HPF the look-up function can be declared `PURE`, as it is without side-effects. `HashTable` contains the share numbers in a hash-table and

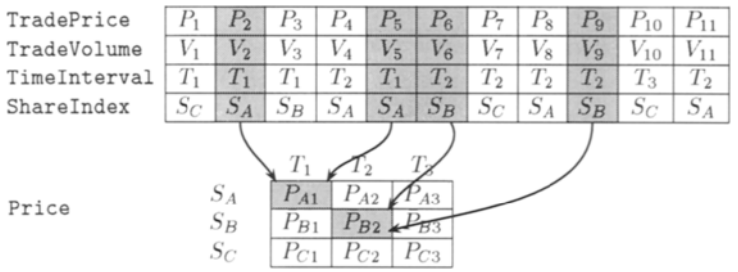


Figure 5. Computing the volume-weighted average of a share price in every time interval requires a many-to-one mapping.

HashData contains the array indices associated with the share numbers. As HASHLOOKUP is a pure function, the FORALL loop below allows the compiler to parallelise all the look-ups.

```
INTERFACE
  PURE INTEGER FUNCTION HASHLOOKUP (HashTable, HashData, Key)
     INTEGER, INTENT(IN) :: HashTable(:,:), HashData(:,:), Key
  END INTEGER FUNCTION HASHLOOKUP
END INTERFACE
...
FORALL (i=1,Number_of_Trades)
   ShareIndex(i) = HASHLOOKUP (HashTable, HashData, ShareNumber(i))
END FORALL
```

It is not possible to parallelise the look-ups on an data parallel machine in this way, as the instructions will not be identical on every processor. In fact on the MasPar, after translation to MasPar Fortran, the loop would be executed sequentially! For a data parallel machine it is better to determine the required row in the hash table first (with HashFunction) and then step through the hash table until all keys have been found.

```
IDXS(:) = HashFunction(ShareNumber(:))
DO I = 1, SIZE(HashTable,2)
  WHERE (HashTable(IDXS(:),I)==ShareNumber(:))
     ShareIndex(:) = HashData(IDXS(:),I)
  END WHERE
END DO
```

Even though the parallelisation of a loop may seem straightforward in HPF, it does not always lead to an efficient implementation on both array and loosely-coupled processors. For highest efficiency it will always be necessary to replace some operations completely for different architectures.

8.3.2. Pre-processing the data

After identifying the array indices for every share, the average price in every time interval must be calculated, as defined in equation (3) and shown diagrammatically in

figure 5. If only the FT-350 is considered, this involves mapping 540000 trade records onto four arrays (for bid and ask price and volume) of size 356×3600.

Ignoring the separation of *ask* and *bid* trades for the time being, this operation is easily solved on a sequential processor with a loop with indirect addressing as shown below. The variable names should be self-explanatory.

```
Price = 0.0
Volume = 0
DO I = 1, Number_Of_Trades
   IS = ShareIndex(I)
   IT = TimeInterval(I)
   Price(IS,IT) = Price(IS,IT) + TradePrice(I)*TradeVolume(I)
   Volume(IS,IT) = Volume(IS,IT) + TradeVolume(I)
END DO
WHERE (Volume>0) Price = Price/Volume
```

As it is not known *a priori* whether several values will be assigned to the same memory location this loop cannot be parallelised easily.

With HPF a library is defined which contains among other things a set of scatter functions which are designed to handle exactly this situation. The SUM_SCATTER function is defined as follows:

```
RESULT = SUM_SCATTER(ARRAY, BASE, INDX1, ..., INDXn [, MASK])
```

This function maps the elements of ARRAY on to BASE. For every dimension of BASE there is an index vector IDXi, of the same size and shape as ARRAY; these specify the position the corresponding element in ARRAY is mapped to. An optional MASK argument selects the elements of ARRAY that are mapped. The loop above can be rewritten in HPF as follows.

```
Price = SUM_SCATTER(TradePrice*TradeVolume, Price, ShareIndex, TimeInterval)
Volume = SUM_SCATTER(TradeVolume, Volume, ShareIndex, TimeInterval)
```

In this case the target arrays Price and Volume are 2-dimensional and two index vectors ShareIndex and TimeInterval are provided to determine the position in the target to which to map the source arrays (TradePrice*TradeVolume and TradeVolume respectively). To separate bid and ask trades the optional mask is used.

On a coarse-grained machine SUM_SCATTER can be implemented by separating source elements that are mapped onto target array positions on the same processor, from those that are mapped onto positions on a different processor. The second set of values is sent to the appropriate processor, while the first is mapped exactly as in the sequential case. Subsequently all values received from other processors are mapped with an identical loop.

On an array processor this approach does not work. An efficient algorithm should send all elements to their target processor simultaneously, but this is not a well-defined operation due to the possibility of many-to-one mappings.

The method we have chosen to implement a data-parallel SUM_SCATTER algorithm is described below (see also [13, 10]). An example is given in figure 6 for 1-dimensional ARRAY and BASE.

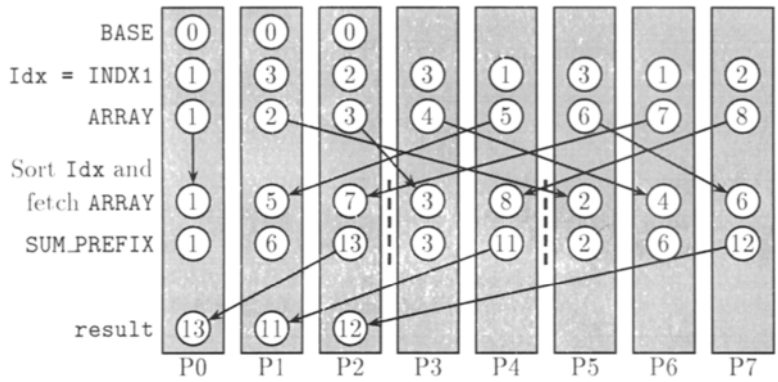


Figure 6. A SUM_SCATTER algorithm as implemented on a data parallel machine.

1. Map index vectors `IDXi` on to a single index vector.
   `Idx(I) = INDX1(I)+INDX2(I)*SIZE(BASE,1)+...`

2. Order the source array using `Idx` as sort-key (see HPF library routine `GRADE_UP`). We do this with a radix-sort algorithm which sorts an array of length n in $O(\log n)$ time on a data parallel machine.

3. Create a segment mask such that every segment contains only elements which are mapped to the same position.

4. Do a *segmented prefix scan* (`SUM_PREFIX` in HPF) on the sorted array. A prefix scan computes all the partial sums of the elements in an array. A *segmented* prefix-scan does a prefix-scan on every segment of an array independently. This can be computed in $O(\log m)$ time, with m the size of the largest segment.

5. Send the last element in every segment to the target array.

This algorithm produces the result of the scatter function in $O(\log n)$ time on an SIMD machine.

This algorithm works very well on general array processors, but on the MasPar specifically there is an even faster way to implement `SUM_SCATTER`. In the MasPar there are two communication networks, one for communication with neighbouring processors (*X-net*) and a second for communication with distant processors (*router*). If the send operation from the source array to the target array is done via the router, the data is sequenced automatically as the router can only transmit one data item at a time to a processor. This functionality is not available from Fortran, but an **mpl** (the C-like language on the MasPar) routine can be written to do a `SUM_SCATTER`. This is significantly faster than the equivalent Fortran routine, but totally machine-specific and thus unportable. On a typical problem, the mapping of 125000 trades on to a 1400×380 element array (7 days trading on to 3 minute time intervals for 380 shares) the Fortran version required 2.73 seconds and the **mpl** version called from Fortran only 0.49 seconds.

$$
\begin{array}{c}
\\
\mathbf{P0} \\
\\
\\
\mathbf{P1} \\
\\
\\
\mathbf{P2} \\
\\
\end{array}
\left(\begin{array}{cccccccc}
1 & & & & & & & \\
-a_1 & 1 & & & & & & \\
 & -a_2 & 1 & & & & & \\
\hline
 & & -a_3 & 1 & & & & \\
 & & & -a_4 & 1 & & & \\
 & & & & -a_5 & 1 & & \\
\hline
 & & & & & -a_6 & 1 & \\
 & & & & & & -a_7 & 1 \\
 & & & & & & & -a_8 \; 1
\end{array}\right)
\mathbf{x} =
\left(\begin{array}{c}
b_0 \\ b_1 \\ b_2 \\ \hline b_3 \\ b_4 \\ b_5 \\ \hline b_6 \\ b_7 \\ b_8
\end{array}\right)
$$

Figure 7. A linear recurrence rewritten as the solution of a bi-diagonal system of equations

$$
\begin{array}{c}
\\
\mathbf{P0} \\
\\
\\
\mathbf{P1} \\
\\
\\
\mathbf{P2} \\
\\
\end{array}
\left(\begin{array}{cccccccc}
1 & & & & & & & \\
 & 1 & & & & & & \\
 & & 1 & & & & & \\
\hline
 & & -a_3 & 1 & & & & \\
 & & f_4 & & 1 & & & \\
 & & f_5 & & & 1 & & \\
\hline
 & & & & & -a_6 & 1 & \\
 & & & & & f_7 & & 1 \\
 & & & & & f_8 & & & 1
\end{array}\right)
\cdot \mathbf{x} =
\left(\begin{array}{c}
b_0 \\ b'_1 \\ b'_2 \\ \hline b_3 \\ b'_4 \\ b'_5 \\ \hline b_6 \\ b'_7 \\ b'_8
\end{array}\right)
$$

Figure 8. The bi-diagonal system after the parallel elimination of the sub-diagonal in every partition.

This section demonstrated how a practical problem requires the SUM_SCATTER routine from the HPF library and how different the implementation of this routine can be on different machines. If hardware support for some operations is available these fundamental algorithms can be speeded up significantly. These library routines are decisive in maintaining efficiency on different architectures.

8.3.3. Calculating statistical estimates

In order to calculate statistical estimates a linear recurrence function is needed. Linear recurrences are defined as a sequence

$$
\begin{aligned}
x_0 &= b_0 \\
x_i &= a_i x_{i-1} + b_i, \qquad i > 0.
\end{aligned}
$$

From equation (7) it follows that the calculation of the mean estimator is a linear recurrence with $a_i = \theta$ and $b_i = (1-\theta)X_i$. For the estimation of the variance a linear recurrence with $a_i = \lambda$ and $b_i = (1-\lambda)(X_i - \hat{\mu}_i)$ must be solved.

There are three general approaches to parallelising linear recurrences[2] [11, 4]: recursive doubling, cyclic reduction and partitioning. Recursive doubling is the preferred algorithm on data parallel machines and cyclic reduction on vector machines. These algorithms have

[2]Linear recurrences cannot be rewritten with the scan routines in the HPF library. If only the last element in a recurrence is needed, it can be calculated with
`result = SUM(EOSHIFT(PRODUCT_SUFFIX (A),1,1.0)*B)`.

been investigated since the early sixties. We use a recursive doubling algorithm on the MasPar.

On a coarse-grained parallel machine, which often have few processors and for which computation is much faster than communication, recursive doubling or cyclic reduction cause too much communication. On such machines some variant of a partitioning algorithm (see e.g. Wang [19]) is preferable. For an efficient implementation of linear recurrences on parallel vector-processors see Blelloch et. al. [4].

A linear recurrence can be rewritten as the solution of a bi-diagonal system as shown in figure 7. If this system is distributed over processors as indicated, it can be transformed to the linear system shown in figure 8 in parallel. If every processor contains m equations, the elimination requires $m-1$ (parallel) multiplications. The back substitution requires $(m-1)$ multiplications as well, so that there is no advantage in using this algorithm on 2 or 3 processors. There is an advantage if there are more processors though: once x_2 has been computed on P0, it can be sent to P1, which computes x_5 and sends it to $P2$ etc. The remaining x values can then be computed in parallel.

It is possible to do this propagation of offsets with a recursive doubling algorithm. If this approach is used on a system partitioned into 2×2 blocks the resulting algorithm is cyclic reduction. In most practical cases the most efficient algorithm will be a mix between the different approaches mentioned above.

This is another example of an algorithm that is implemented in completely different fashion on different architectures. If a program is moved from one machine to another the linear recurrence routines will have to be replaced to retain efficient code. It is thus desirable to have an expanded library of routines that are implemented by the manufacturers for different architectures.

8.4. Summary of algorithmic examples

The first application, the valuation of American options, is an example of an HPF program that can migrate from one architecture to another with the change of just a few directives. The second application, the statistical analysis of trading data, high-lighted the importance of libraries on different machines.

The computation of average trading prices in fixed time-intervals requires an HPF library function, `SUM_SCATTER`, for which an implementation is provided by the manufacturer. The calculation of the estimators requires the solution of linear recurrences — a routine that is not in the HPF library. For an efficient implementation of these routines a good understanding of the hardware is required, and it may take weeks of work to balance communication, and computation (and vectorisation) against each other. Such fundamental algorithms must be available in libraries as no application programmer can be expected to know several architectures so well that all implementations will be efficient. Other routines that should be available are the BLAS and LAPACK-routines, which have already been ported to some parallel computers.

Another example was the parallel look-up in hash tables. This algorithm had to be implemented specially for array processors. It is a specialised application and such an implementation will not be available in general libraries – especially as the `FORALL` construct provides a good way of expressing the operation on many parallel computers. Unfortunately it is unavoidable that some routines will always have to be re-implemented to get

the best performance on a specific architecture.

9. Conclusions

Several application areas in finance have been discussed and the need for high performance computing established. When examining the software requirements in the financial world, it became clear that the lack of standard parallel languages, which can be integrated easily with existing software packages and are portable over a wide range of architectures, is one of the main reasons for the slow introduction of parallel computers.

High Performance Fortran seems to offer a solution to this problem: it can easily be integrated with existing Fortran libraries and looks familiar to developers of financial modelling software. Due to a single-threaded control-flow and the use of data mapping directives it seems to be both simple to understand and portable over a wide range of parallel computers.

To test this assumption two applications, option pricing and real-time statistical analysis, were implemented on both a coarse-grained and fine-grained machine. The option pricing application proved to be very easy to port between machines, and no major code changes were necessary. The statistical analysis program proved to be more difficult to port and three problem areas were discussed.

The conclusion is that HPF does go a long way towards satisfying the stated requirements, but only when used in conjunction with library routines customised to the different architectures. For really efficient programs some routines will always have to be ported by hand, as different algorithms are required. Currently HPF does not provide support to easily implement irregular problems such as large, sparse optimisation problems, which is a serious drawback.

10. Acknowledgements

This work was sponsored in part by Digital Equipment Corporation and NAG Ltd under LPAC gran CTC07 and their support is gratefully acknowledged. I also wish to thank Digital for making their HPF compiler available. And finally I wish to thank Chris Rogers, Jon Hill and Carlos Falcó Korn for many discussions.

REFERENCES

1. J. Barraquand. Numerical valuation of high dimensional multivariate european securities. Technical Report 26, Digital, Paris Research Laboratory, 1993.
2. J. Barraquand and T. Pudet. Pricing of American path-dependent contingent claims. Technical Report 37, Digital, Paris Research Laboratory, 1994.
3. F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–54, 1973.
4. G.E. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. In *Proceedings Sixth International Parallel Processing Symposium*, pages 416–424, March 1992.

5. M.A.H. Dempster. Fast numerical valuation of American, exotic and complex options. Technical Report 94-23, Institute for Studies in Finance, University of Essex, Colchester CO4 3SQ, UK, 1994.
6. The Economist. Realignment merchants. 26 September 1992.
7. The Economist. Shell game: Shell's foreign exchange disaster. 27 February 1993.
8. The Economist. Derivatives: The beauty in the beast. 14 May 1994.
9. MPI Forum. MPI: A message-passing interface, MPI forum. Technical Report CS/E 94-013, Department of Computer Science, Oregon Graduate Institute, March 1994.
10. J.M.D. Hill. The HPF library in Fortran 90: sorting. Technical Report LPA7/TR.02.9408, LPAC, QMW, Mile End Road, London E1 4NS, 1994.
11. R.W. Hockney and C.R.Jesshope. *Parallel Computers.* Adam Hilger Ltd., 1981.
12. J.C. Hull. *Options, Futures and Other Derivative Securities.* Prentice Hall, Englewood Cliffs, New Jersey 07635, 2nd edition, 1993.
13. A.W. Joubert. The HPF library in Fortran 90: prefix and suffix scans. Technical Report LPA7/TR.01.9408, LPAC, QMW, Mile End Road, London E1 4NS, 1994.
14. A.W. Joubert and L.C.G. Rogers. Ticking over in real-time. In *Quantitative and Computational Finance*, pages 199–211. UNICOM, November 1994.
15. A. King. Stochastic programming at IBM research. In *Quantitative and Computational Finance*, pages 81-90. UNICOM, November 1994.
16. C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele, and M.E. Zosel. *The High Performance Fortran Handbook.* Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA., London, 1994.
17. Jacques Longerstaey. Introduction to riskmetrics. Morgan Guaranty Trust Company, Market Risk Research, New York., October 1994.
18. K. Mills, G. Cheng, M. Vinson, S. Ranka, and G. Fox. Software issues and performance of a stock option pricing model on the Connection Machine-2 and DECmpp-12000. In *Proc. of Fifth Australian Supercomputing Conference*, Melbourne, December 1992.
19. H.H. Wang. A parallel method for tridiagonal equations. *ACM Trans. on Math. Softw.*, 7(2):170–183, June 1981.
20. S.A. Zenios. *Parallel Monte-Carlo simulation of mortgage-backed securities*, chapter 14, pages 325–343. Cambridge University Press, 1993.
21. S.A. Zenios and R.A. McKendall. Computing price paths of mortgage-backed securities using massively paralllel computing. Technical report, The Wharton School, University of Pennsylvania, Philadelphia, PA 19104-6366, 10 1992.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Virtual Reality and High Performance Computing

Henry A. Sowizral

Boeing Computer Services, P.O. Box 24346, MS 7L-48, Seattle, WA 98124-0346, U.S.A., email: sowizral@atc.boeing.com

Virtual reality (VR) has become the hot new buzzword. It has captured everyone's imagination, businessmen and marketers, poets and scientists. All this interest has generated as many potential uses (and definitions) as the number of people who talk about it. Having evolved seemingly overnight, VR is still in its infancy. Unfortunately, VR will have a difficult time delivering on all the expectations—at least within the near term "implied" by its enthusiasts. Despite the disappointment that not meeting inflated expectations will cause in the short term, VR will fundamentally change the way we view and interact with computers.

High performance computing serves a pivotal role in VR, both as a source of the data that user will want to interact with using VR technology and as a source of computing technology for enabling users to interact with large virtual environments—their data—in realtime.

1. WHAT IS VIRTUAL REALITY?

Virtual reality is a paradigm shift, a new way of using and interacting with computers. A virtual reality system generates a synthetic environment where a user can operate as if physically present. Important elements of such environments include egocentricity or a first person experience, and an appropriate coupling between the user's senses and the synthetic environment.

VR is an interaction methodology that focuses on using the entire human body. VR's enabling technologies focus on the human body, how it moves, and how we use it to interact with the real world. We turn our head left and right to see what is to our left and right. VR mimics this. We reach out with our hand and we see our hand reaching. VR mimics this. A good VR system will create an adequate sense of presence for its users.

1.1. Improving the interface between people and computers

Virtual reality is the latest in a serious of attempts at matching a computer's I/O capability with those of a human. The earliest human-computer interaction paradigm used the computer's front panel; the user would read the lights and either press or flip switches on the computer's front panel to cause changes. That interaction technology was less than satisfying. Almost immediately, the front panel gave way to the teletype and scrolling CRT. They significantly improved

our ability to interact with the computer and remained the main interaction technology for multiple decades. The introduction of the mouse and overlapping windows allowed us to interact with the computer in a much more intimate manner: we could point to various places on the computer's screen and make selections. Multimedia technologies introduced even more sensory dimensions, video and stereo sound.

Virtual Reality takes another step along the path of sensory integration. Where we look becomes just as important as what we type, where we point, and how we select things. No longer do we sit in front of a computer monitor and stereo speakers. With VR, we wear the displays and earphones. We expect the visual scene to match our gaze direction and we expect fixed sources of sound to remain rooted in the virtual environment when we turn our head or move about the environment.

1.2. Different kinds of VR

Where does VR start? What does it include? VR has a few salient defining features: a first person perspective, images that change with changes in head position and orientation, and a sense of immersion. Using this definition a first-person window-based game would not qualify nor would a stereo image-display without head tracking. Still, various kinds of virtual reality environments do exist.

We can classify VR into three subcategories depending on how much of the image is generated by computer: immersive VR, if the computer generates the entire image; augmented reality, if the computer generates only part of the image; and, telepresence, if the computer acts only as the control element, generating none of the image.

1.2.1. Immersive virtual reality

An immersive virtual reality user sees an image entirely generated by the computer. The computer tracks the user's head, determines the user's gaze direction, and generates the first person view from that gaze direction. The display medium further classifies the kinds of immersive VR systems.

When most people think about immersive VR, they think of head-mounted or head-coupled VR systems. In such systems, VR users either wear a helmet or place a boom[1] against their face.

Fish-tank VR couples a head tracker, a computer monitor, and an LCD-based shuttering system to alternately occlude the left and right eye. The VR system synchronously draws a left and right eye view on the computer monitor.

Caves or portals require a dedicated room. The cave's front and side walls consist of projection screens. Behind those walls, projectors fill the screens with

---

[1] A boom is an articulated arm, similar to an articulated lamp or a robot arm,that supports a moveable binocular display at one end.

computer generated images. Shutter goggles synchronized with the rear screen projectors present the user with a stereo, head-tracked, 270-degree, computer-generated view of the virtual environment.

1.2.2. Augmented reality

Augmented realty (AR) relaxes the requirement that the computer generate the entire image presented to the user. The AR user wears a headset that mixes a real-world image with images generated by computer.

We can mix the real and computer-generated images in one of two ways depending on whether the real-world image comes directly from the real world or from a video camera attached to the user's helmet. In the first case, the user wears a head-mounted display with an optical combiner. The user looks through the combiner's angled, half-silvered mirror to see the real world and simultaneously, by way of the mirror, at the computer-generated, augmenting image. In the second case, the user wears an immersive display with an attached video camera. The computer electronically mixes the camera's video signal with the computer-generated augmenting information.

1.2.3. Telepresence

Telepresence fully removes the computer from the task of generating images. Instead, the computer controls the position and orientation of a remote platform from the user's head position and orientation. The remote platform can include servo-controlled stereo cameras, microphones, and even physical manipulators.

Telepresence, as the word itself implies, allows its user to work with an environment remotely. Working with a remote system usually refers to geographic remoteness but it can also include the more general spatial remoteness. A telepresence user can just as easily inhabit the microscopic world of a scanning tunneling microscope as well as the undersea world of a submersible, remotely manipulated robot.

1.3. A VR system overview

VR systems are real time systems. They accept inputs, compute state changes based on those inputs and generate output. The more rapidly, the VR system can execute this loop; the more accurately the VR system can monitor the user's actions and mimic corresponding sensory changes.

The main loop gathers input from two kinds of sources: from trackers attached to various parts of the body; and, from other inputs such as gloves or switches. These inputs change the system's internal state. Using the freshly updated state information, the main loop computes which objects fall into the user's field-of-regard, and draw the current view of the world.

Figure 1 shows a typical VR hardware arrangement. It provides partial hardware support for three senses. The head mounted display supports our visual sense. The 3D sound generator, earphones, and microphone support our auditory sense. The tracker and dataglove, support our proprioceptive sense.

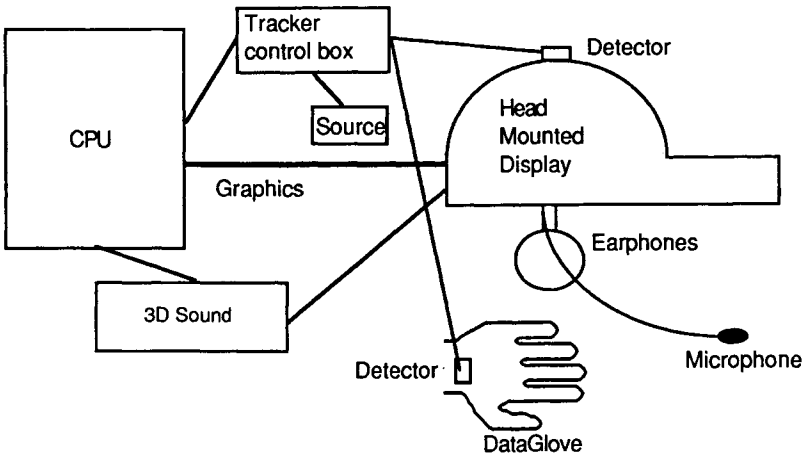


Figure 1. A typical virtual reality hardware configuration that consists of a central processing unit, a head-mounted display, a glove, a 3D sound generator, and a tracker. The tracker consists of a source and two detectors. Each detector receives the source's signal and repetitively computes its current position and orientation (pose). One detector provides the head's pose, the other that of the dataglove. Also attached to the head-mounted display are earphones and a microphone. The 3D sound generator feeds the headphones with spatially located sound. The microphone feeds the CPU's speech recognition software. The hardware arrangement in this figure provides partial support for three senses: vision, sound, and positioning-touch.

1.4. Hardware requirements

A VR system can stress computer hardware quite significantly depending on the type of VR system and the complexity of the application. Immersive VR generally places the largest burden, AR a moderate burden, and telepresence the smallest burden on the computer and I/O hardware.

2. A CAUTIONARY TALE

VR's general premise, *letting people see and do things as they would in the real world,* is seductive. Describing a task-specific VR application makes it sound similar to how we would perform the same task in the real world. This apparent similarity in form and function invites listeners to draw from their own experiences. Inevitably, listeners add unintended details and expectations.

A few application descriptions show how unintended details or expectations can creep into our perceptions.

2.1. Descriptions of VR applications

Put on some wrap-around glasses, add earphones and gloves, and step into another world. You can be a geologist standing on Mars; an architect touring an as-yet-unbuilt building; a scientist standing inside a wind tunnel studying how air flows over the space shuttle's wing; an oceanographer controlling an undersea

robot and seeing what the robot "sees;" or, even a chemist looking at and "touching" the surface of a molecule as imaged by a scanning tunneling microscope.

Alternatively, put on those same wrap-around glasses and let the computer provide you with visual hints that help you interpret what you see. You can be a quality control expert looking at a physical workpiece and, superimposed on that workpiece, you would also see a computer-generated image of how that workpiece looks when assembled correctly; or, you can be a manufacturing worker seeing a computer-generated image of the next part and an arrow pointing to the point on the physical assembly where you should attach that part.

2.2. The "reality" behind the descriptions

VR researchers have demonstrated every one of these applications. Each application is both heartening and disappointing. Heartening, because each application manages to provide its user with some sense of being there and of performing the associated task, some sense of presence, some confidence that VR can fulfill at least part of its promise. Disappointing, because the "sense of reality" each application generates with today's rather crude technology does not measure up to the expectations generated by that application's description.

We reason with and set our expectation from our own experiences. Our relationship with reality is very intimate: we experience reality day in and day out. Compared to our day-to-day experience with reality, virtual reality just cannot measure up, nor is it clear that it ever will. Much work remains before VR can fulfill even these much lower expectations of providing workable and workaday interfaces. Despite these rather sobering statements, VR has great potential for improving the way we use and interact with our data and our computers.

2.3. A tool–not a reality replacement

VR provides us with an important new perceptual tool. It helps us view data in three-space the same way we do in the real world. By rapidly tracking our head position, the VR system can redraw the world as seen from those various head positions. In the real world we use stereopsis[2] and motion parallax[3] to understand three dimensional interrelationships. In essence we "look around" to get a sense of how the various objects in the environment interrelate. Our body is an integral part of our understanding. We integrate our head position with what we see from that position to build our mental of our environment.

We can enhance our ability to understand object placement within the virtual environment by including a virtual image of our arms within the virtual environment. From our earliest experiences in the crib, we learned to reach out

---

[2]Stereo vision.

[3]The fact that objects close to our viewpoint move in the opposite direction from our head and far away objects move in the same direction as our head.

touch and interact with our environment. Our body was our most important measurement device. It remains so today.

3. THE NEED FOR HIGH PERFORMANCE COMPUTING—TWO EXAMPLES

A VR system is a realtime system. It reads the user's current head pose, updates and computes its internal state, and then draws the new view.

More importantly, the system must repeat these three step quickly enough to ensure that the images flow coherently. Television refreshes its picture 30 times per second, films refresh 24 times per second. Refresh rates that drop below approximately 8 frames-per-second start appearing choppy, below about 4 frames-per-second the images start appearing disconnected.

For short term work inside a virtual environment, frame rates near 20 frames-per-second seem adequate. Achieving a rate of 20 frames-per-second require the computer to complete a full circuit through its main loop in less than 50 milliseconds.

3.1. Two ways VR can overload a computer

Though a budget of 50 milliseconds may seem generous to execute a VR system's main loop, we can overload a high performance computer with even a medium-complexity virtual environment in two ways, geometric or computational complexity. The virtual environment may contain a large number of faceted objects or, it may perform an involved computation before it draws an image.

Geometric complexity, world models with high polygon counts, can easily overload a computer's rendering hardware. All rendering pipelines have an upper limit on how many polygons they can draw in one second. A fairly capable rendering pipeline, the Silicon Graphic Reality Engine 2, is rated at two million polygons per second[4]. To achieve a 20 Hz update rate, the VR system can draw no more than 100,000 polygons per frame. Realistically, we must draw far fewer polygons (50,000 polygons) to maintain a 20 Hz update rate.

Computational complexity, the amount of computation performed by the system before it draws an image, can easily overload a computer's cpu. CPU's, too, have an upper limit on how many instructions they can execute in one second. If a computation requires too much computation, the VR system cannot maintain a 20HZ update rate.

Two VR applications, one a design visualization application and the other a scientific visualization application, how geometric and computational complexity can swamp the capabilities of high performance computers.

---

[4] A rendering rate achieved only under ideal conditions.

3.2. Designing complex structures

Large design efforts such as skyscrapers, airplanes, or ships, involve many designers working over a long period of time. The resulting design involves large numbers of parts, many with complex geometries. Such designs require gigabytes of storage[5]. By visualizing these digital designs, designers can find and fix design flaws long before final assembly, long before workers must fit parts together. Airplanes, ships, and skyscrapers all contain millions of parts. Most parts average approximately 200-500 triangles per part. All together a digital design will require gigabytes of storage. Unfortunately, no known visualization system can manage to display designs of these magnitudes in real time.

The RealEyes VR system developed at Boeing has had some success in visualizing large scale (but significantly smaller amounts of) geometries in real-time. Without access to high performance computers, RealEyes could not have happened. The shear scale of complex mechanical systems makes high performance computing the only possible recourse for their visualization.

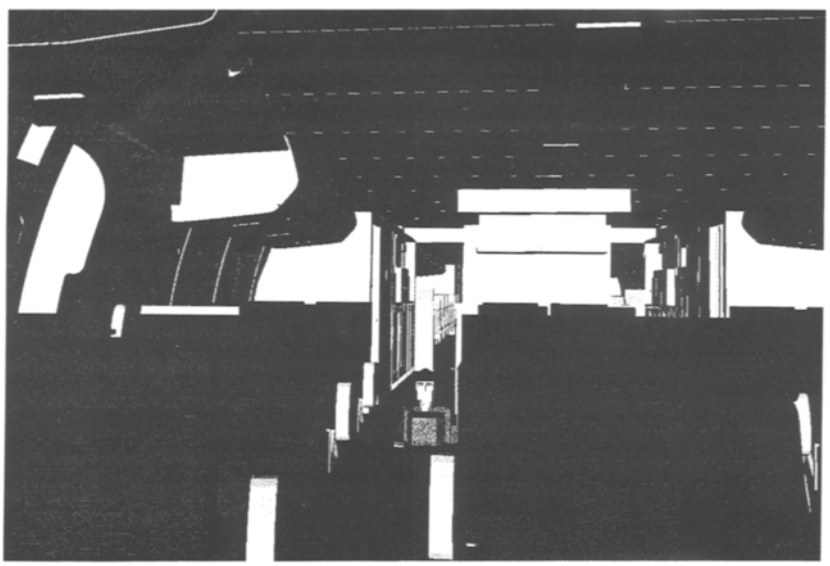


Figure 2. The interior of a non-production airplane. This interior model consists solely of geometric data–no texture maps. It graphically underscores one problem with CAD-generated worlds, large polygon counts that bring commercially available VR systems to their knees.

Figures 2 and 3, illustrate the types of world models RealEyes targets and handles relatively well. In both cases, we import CAD-generated data into RealEyes. The data contains pure geometry, no texture maps. Figure 2 shows a RealEyes view of a nonexistent airplane's interior. The human model sticking out from the floor is standing on a ladder leading out from a downstairs cocktail lounge. Figure 3 shows a RealEyes view of Space Station Freedom's never-to-be-

[5] Millions of parts, hundreds of polygons per parts, order of 100 bytes per polygon.

realized lab module. It shows the module's exterior assemblies and their interconnections.

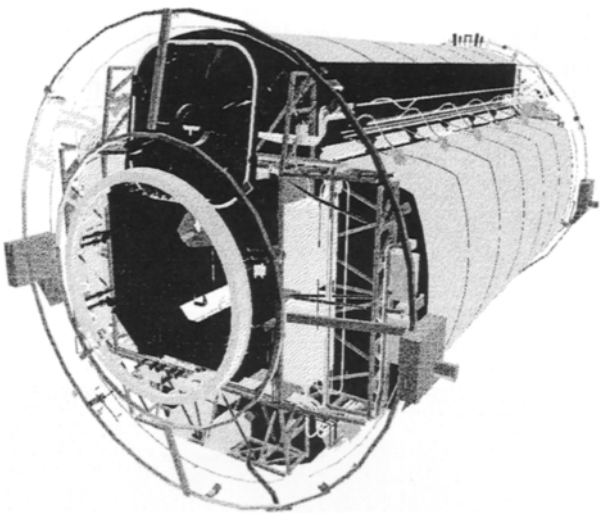


Figure 3. The exterior of Space Station Freedom's never-to-be-built lab module. This model contains no texture maps, consisting entirely of geometric data. It demonstrates how useful visualization can be in the design process by showing how some of the external piping goes through structural members.

3.3. The RealEyes virtual reality system

Visualize complex geometry such as airplanes, ships, and skyscrapers using commercially available hardware requires careful algorithm development. RealEyes, Boeing's internally developed VR system, can render images like those in Figures 2 and 3 at reasonable frame rates.

Using a Silicon Graphics (SGI) onyx class computer with eight MIPS R4400 processors each running at 150 Mhz and with two Reality Engine 2 (RE2) hardware rendering pipelines, we can barely handle virtual environments containing 30,000-50,000 polygons with straightforward rendering techniques. However, using concurrent processing techniques and carefully designed rendering algorithms, such hardware can adequately handle virtual environments with polygon counts almost two orders of magnitude larger.

RealEyes achieves good realtime performance because it tries never to draw more triangles than necessary. If our world model contains 50,000 triangles we can draw the entire world and the hardware alone will guarantee us a 20 Hz frame rate. At 1,000,000 triangles, we can no longer use a brute force approach. Relying on the hardware alone would ensure a one Hz frame rate. At 1,000,000 triangles, we must pick and choose carefully what we draw. At 10,000,000

triangles, the brute force approach shows how wasteful it really is. A computer monitor cannot display more than one million pixels at any one time. If we draw more than one million triangles, we guarantee that some of the triangles will not be seen. So why draw them? The drawing algorithm must very carefully budget what it draws and when. A high performance, muli-processor machine provides the ideal platform for developing such algorithms.

3.4. Viewing and interacting with scientific data

The visualization of scientific data presents a different set of challenges from those of visualizing large geometric databases. An analyst might spend many hours studying and interpreting the results of a single computation. That same analyst might need to execute and analyze many more variations on that same computation before acquiring the necessary insight into a problem's structure.

Instead of staring at a large pile of numbers or at a single perspective rendering of the data, analysts can surround themselves with the data. Analysts can use an interactive VR environments to move around within the data and view it from multiple perspectives. This provides a different route for our analysts to use in acquiring an intuition into their problems.

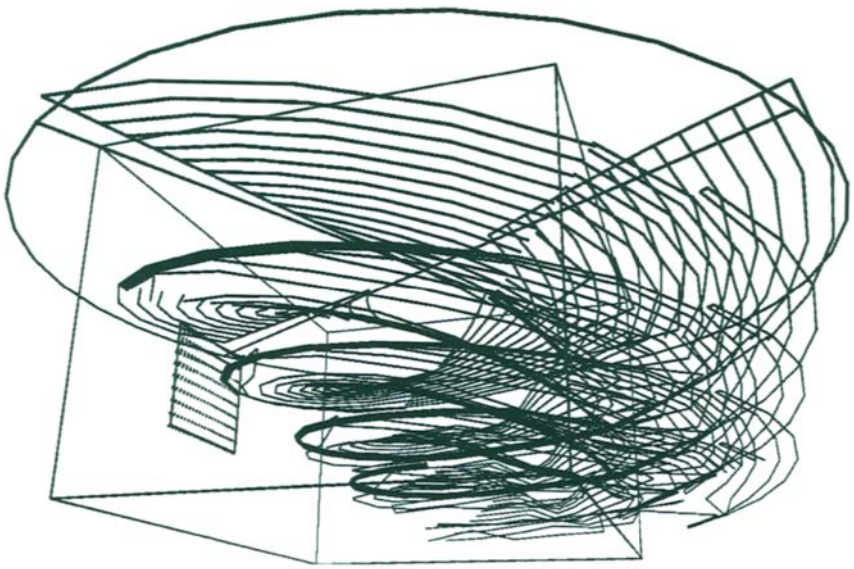


Figure 4. A model of a helicopter rotor and its associated rotor wakes. The helicopter is in straight and level flight coming towards us and about to pass overhead. Each of the four rotor blades generates a distinct rotor wake. A static views restricts an analysts ability to see wake interactions, the use of animation or an ability to examine vorticity locally could help analysts gain understanding more rapidly.

Figure 4 shows an example of a helicopter rotor wake. We are in front and below the helicopter looking up. The largest and closest circle represents the outer perimeter of the rotors. Within the circle, we can see four rotor blades.

Emanating from each rotor blade are vortex tubes. Because the helicopter is traveling towards us, the vortex tubes recede into the background.

Static view such as the one in Figure 4 make understanding the phenomena rather difficult. Just the ability to more around and look at the vortex tubes from multiple directions would allows us to find interesting phenomena that deserve a more detailed examination and maybe even a more detailed re-computation.

Tools that could allow analysts to reach into the virtual environment and inserting stream lines or alternatively move a localized vorticity indicator could significantly aid analysts in the studies.

3.4.1. The need for interaction

A VR system must provide more than just visualization support. As designers wander around within the digital design, they will want to ask questions and interact with their environment. Similarly, analysts will want to examine subareas of their data in detail. They will want to:

- Navigate through the environment, *e.g.*, move themselves forward; put themselves in the basement; locate themselves at the start of a flow stream.
- Find and manipulate particular objects, *e.g.*, find the electric distribution panel; locate the maximal point of curvature along a stream line; move the light switch up and to the right; change the chair's fabric to a different texture; follow an iso-surface.
- Access, create, or change "meta," environmental, or system parameters, *e.g.*, access the name of a part's designer; ask where they are within the environment; rerun the computation with a different parameter; change lighting conditions; make a notation; change their height; change their interocular offset; load the electrical models; save the current state of the environment.

These and other questions and commands require that the designer or analyst have some means for interacting with the VR system and with the objects in the virtual environment. A good interaction technique should blend naturally into the environment. It should allow a user to interact with the "meta-world" without forcing the user to psychologically "leave" the virtual environment. The interaction technique should provide a natural metaphor for invoking, dismissing, manipulating, and controlling the resolution of the interaction device and its associated information. It should additionally engage the user's propreoceptive[6] and haptic[7] senses. Satisfying all these conditions is not easy.

Figure 5 shows one possible interaction tool, a moveable paddle. The paddle has an array of indicators that follow the field's vorticity lines, we can specify that

---

[6]The propreoceptive sense refers to our sense of body. For instance, it provides us with all the information we need to touch our nose even when we close our eyes.

[7]The haptic sense refers to our entire sense of touch. A haptic display can provide tactile and force feedback.

their color indicates any one of a number of other state variables. The analyst controls the paddle through a six-degree-of-freedom tracking device. This allows the analyst to move the paddle within the space and to rotate it into whatever orientation provides the most information. The use of our body to interact with the virtual environment adds substantially to our understanding of the space.

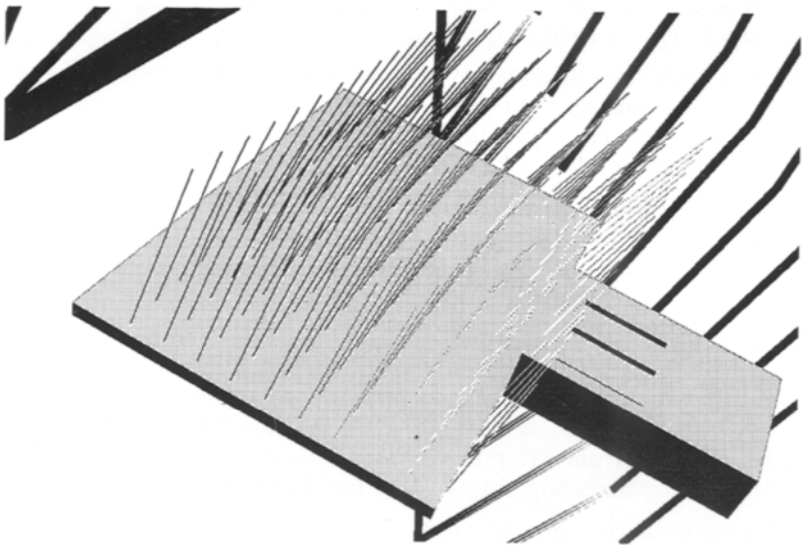


Figure 5. A close up of one part of the rotor wake with informational paddles and attached vorticity hedgehog. By moving a six-degree-of-freedom tracker in the real world, an analyst controls the paddle's position and orientation within the virtual environment. The hedgehog reflects the vorticity at the paddle's position in the virtual world

Proprioception, or our ability to know how our various limbs are oriented by feeling the strain in our muscles, is integral and fundamental to how we interpret our environment. Our body is our fundamental measurement tool, one we started using the first time we reached for an object while in our cribs. When we reach out and touch an object, we not only know it is within arm's length, we also know how far away it is. This act of reaching out and knowing you have touched an object makes the virtual environment seem less foreign, less distant, more real, but it also

4. RESEARCH ISSUES

Virtual reality is an emerging field. Much research still remains to be done. High performance computers provide an ideal environment for developing these compute intensive applications. Appropriate application of high performance computing to VR problems can hasten their use in solving real problems.

Rendering remains the most limiting factor where complex geometries dominate. We need more research into techniques for carefully selecting which objects to draw and at what level of detail to draw them.

Simple culling (not drawing objects that are not in a user's current field of regard) can significantly cut down the number of object we send to the hardware for rendering. Other, more aggressive techniques could further reduce the number of objects we send to the hardware renderer.

We also need to understand how to take best advantage of the rendering hardware. We can use many optimization techniques developed to extract performance from vector pipelines and, almost directly, apply them to extract performance from rendering pipelines.

Interaction techniques in geometric systems require collision detection. Codes that detect when two geometric objects intersect require fairly involved computation. We need to developing collision detection algorithms that perform the same task in a less computationally intense manner.

Managing the great amount of data whether geometric or spatial-state variables will dominate much thinking in future VR applications. Accessing and retrieving megabytes of information in less than a second from files containing gigabytes of information will require considerable thought. As an example, taking an elevator from the ground floor of a skyscraper to the 89th floor can potentially require accessing almost all of the skyscraper's geometric data.

High performance computers will have their biggest impact when we develop parallel VR codes. Whether the parallelism derives from functional decomposition or for inherently-parallel algorithm design does not matter. The fact that we no longer need to serially execute some computations will improve the system's overall performance.

5. SUMMARY

Virtual reality improves the impedance match between people and computers. It provides a more complete method for using our body to interact with the computer. This improvement will make VR more and more useful to a wide range of users, especailly as costs drop. Currently, high performance computers are barely able to provide the capabilities needed to support high end VR applications. We can make up for some hardware inadequacies by developing new software algorithms.

6.. ACKNOWLEDGEMENTS

I thank Karel Zikan who worked with me in co-invented the core RealEyes algorithms. I thank Ian Angus who worked with me to develop fully the interaction device. I thank David Kerlick who provided the helicopter data. I also thank the Boeing virtual reality team, Sean Callahan, Chris Esposito, Jeff

Heisserman, Adam Janin, David Mizell, and Bruce Wilson, for their support and constant willingness to discussion VR issues.

REFERENCES

Burdea, Grigore and Philippe Coiffet, *Virtual Reality Technology,* Wiley Interscience, 1994.

Kalawsky, Roy S., *The Science of Virtual Reality and Virtual Environment,* Addison-Wesley, 1993.

Sowizral, Henry A. "Interacting With Virtual Environments Using Augmented Virtual Tools," in SPIE proceedings, San Jose, CA, February 1994.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
1995 Elsevier Science B.V.

Collaborative Computing

Kish J. Sharma

Boeing Computer Services, P.O. Box 24346, Seattle, WA 98124, U.S.A.

1. COLLABORATIVE COMPUTING: AN EMERGING PARADIGM

Collaborative Computing is an emerging discipline in computer science [1.2]. It addresses the relationship between technology and human collaboration. Its three key elements are: computing and communication technology, people or workgroups and organizational tasks they perform. Collaborative computing focuses on group rather than individual problem solving and decision making tasks necessary to accomplish business and scientific objectives. It provides an environment in which people can share information without the constraints of space and time. Tasks supported by such an environment involve dynamic organizational processes that characterize interactions between workgroups, and between workgroups and computing and communication technologies. Such processes also specify task dependencies as well as what organizational resources are required for performing them. Tasks can execute specific problem solving applications; e.g., complex system simulation using high performance computing; or, supporting geographically distributed workgroups who collaborate on new product development activities.

2. GROWING IMPORTANCE OF COLLABORATIVE COMPUTING

Many organizations like Boeing are becoming collaborative enterprises. Rapid changes in market forces and increasing competition at the global level are redefining the business context for many companies, especially those with trans-national interests. Such redefinition is requiring new and different ways for product development, manufacturing and support. It is also forcing companies to reevaluate their business practices and restructure their organization design. Increasingly, information is becoming a significant factor that influences product design, manufacturing and support. Indeed, clever utilization of information technologies is being recognized as the distinguishing feature of innovative and successful organizations. Progressive companies are thinking about their products in terms of their information intensity, thus viewing information as a distinct factor of production.

An especially interesting practice being used by many multinationals is to develop, produce, and support their products from geographically distributed work centers. For example, it is common

practice in the aircraft industry to have suppliers distributed globally. Indeed, the product content of a modern commercial airplane is being influenced significantly by suppliers in many different countries. This mode of product development is being pursued so that the total product cost is minimized while product quality is improved. However, this practice of geographically distributing organizational activities creates significantly different types of business problems. In particular, such practices substantially change the ways in which organizations function. For example, success of distributed organizations depends on accurate and timely availability of most relevant information and tighter coordination and integration of organizational and human activities.

Similar trends are also occurring in the organization of scientific and technical problem solving endeavors. As scientific and technological problems become larger and more complex, their solution is requiring new innovations. Collaboration among scientists and technology experts is becoming more and more important for exploring and developing rapid solutions. Computer and communication technologies are already providing innovative ways to support scientific collaboration.

An important issue is emerging: what is the best way to effectively manage the coordination of distributed organizational problem solving and decision making activities. Manual means of coordination and collaboration are proving to have limited effectiveness because they are sluggish and become unmanageable as scale of collaboration increases. Development of standard organizational procedures that rely on computational techniques are also insufficient because their development and execution require rather precise understanding of how workgroups collaborate. Collaborative computing is an area that provides innovative ways to address this issue.

3. THREE LAYERS OF COLLABORATIVE COMPUTING

Growth in collaborative computing is being driven by rapid developments in desktop processing power and the availability of high speed networks. Networked computers is changing the way information is created and can be shared by those who need it. It is breaking down geographic separation of workgroups, making virtual computing and virtual workplace almost a reality.

Three appropriately interfaced technology layers form the essence of collaborative computing: communication, cooperation, and coordination. *Communication* is the most basic layer, expressed in terms of "sender-receiver" context. For human collaboration, this means, for example, technologies for conversation; for collaboration between humans and computing machinery, this means user interfaces, transmission protocols, etc. *Cooperation* layer provides information space that can be shared by teams of problem solvers and decision makers working on common tasks; e.g., shared databases that contain product definition. *Coordination* layer addresses the dependencies, for example between tasks and organizational resources. Tasks may be designed and executed by humans and/or computers. More importantly, coordination layer manages dependencies [3, 4]. Management of dependencies, in turn, requires: (a) developing usable, computable representation of the dependency structures; and, (b) indicating what computing and/or organizational mechanisms are needed to manage the dependencies. The three

layers are highly inter-dependent: without the communication layer, cooperation layer can not effectively function; without the cooperation layer, coordination layer will be unworkable. In essence, collaborative computing is a rendering of each and all of the three layers.

4. REQUIRED CAPABILITIES FOR COLLABORATIVE COMPUTING

Development and deployment of collaborative computing require several capabilities. These fall into at least the following categories: (a) characteristics required of the enabling technologies; (b) availability of models of collaboration - i.e., models that express how workgroup members collaborate; and, (c) availability of requisite task models - i.e., computable representation of tasks to be performed by the workgroups.

4.1 Enabling Technology Characteristics

(a) *Architecture to Support Collaborative Computing* : An architecture delineates technology components and their relationships that are required to accomplish specific objectives. Collaborative computing architecture functions in a distributed computing environment, with appropriate levels of security, concurrency control, configuration management, and version control so that workgroup members can work transparently. The user interface for collaborative technologies must address the nature of group-based applications, allowing multi-user access and manipulation of user interface objects [5]. The architecture should also allow building applications from component application models by using proper integration techniques. For example, an user should be able to produce a collaborative design application by combining relevant application models that specify how: design engineers produce and exchange design information; design workflows are created and coordinated; and, design decisions are made and recorded. The application models, in turn, rely on common services such as groupware services, database management services, and authentication and security services. Common services also provide application development tools for software engineering. Collaborative computing architecture is designed to separate the users from the complexities of computing environment.

(b) *Attributes of Enabling Technologies*: Key attributes of technologies that enable collaborative computing include the ability to: provide facilities for sharing information; operate ubiquitously in networked computing environment; eliminate human labor whenever possible; capture and reuse knowledge; provide tracking and monitoring of the collaborative processes; permit use of analytical resources that can supply needed data and information to workgroups; provide an environment in which workgroup members can share information without constraints of space and time; and, be able to integrate and execute various components of problem solving and decision making processes from an user's desktop.

4.2 Models of Collaboration

To properly apply collaborative computing, one must have some idea of how people work together to accomplish specific tasks. Situational models that describe collaborative behavior of teams are relatively scarce. Models based on concepts such as "communities of practice" , "coordination theory", "speech acts", and "cooperative processing" are reasonable candidates

[6-8]. However, reference models for deriving specific models of collaboration are not yet widely available

An often-used approach in computer-supported cooperative work is to focus instead on modes of workgroup collaboration [9]. Modes of collaboration are expressed in two dimensions: time and space. Interactions by workgroups are then specified in terms of when and where they occur, yielding four possibilities: same time-same place; same time-different place; different time-same place; and, different time-different place. Face-to-face interactions occur at the same time and same place, requiring synchronous modes of collaborative support. Interactions occurring at different time and different place (e.g., electronic mail, collaborative authoring) require asynchronous distributed support. Asynchronous interactions occur at one place at different times (e.g., visibility rooms, work shifts), while synchronous distributed collaboration occurs at same time but at different places (e.g., video or desktop conferencing). Although this typology is useful in thinking about technologies required for team support, its usefulness is limited for specifying analytical models of collaboration.

4.3 Task Models

Recent interest in process management, business process reengineering, and workflow management is providing considerable impetus for developing reasonably good task models. In addition, task models can be developed using concepts from operations and project management.

Task models for use in collaborative computing need to contain information about at least four major aspects. First, it must specify the task structure by showing and typing the dependencies among tasks at the required levels of abstraction.. Second, task model should clearly indicate mechanisms that are needed to manage these dependencies; e.g., allocation of organizational resources or the organizational policies and controls needed for supporting task execution. Third, task model must clearly provide time-based view of the task structure. This view is necessary not only for tracking and monitoring of tasks as they are executed by the assigned organizational resources, but it also reveals ways to shorten flow times associated with the task structure. Finally, task model should indicate measures to be used for performance assessment. Such measures address the effectiveness of workgroup collaboration, execution speed of algorithms , ability to handle scalability, efficiency of assigned organizational resources, and so on.

5. ENABLING TECHNOLOGIES

Enabling technologies for collaborative computing support the human-centered architecture described earlier. The human is at the center with access to three mutually dependent systems: workgroup support system, intelligent information system and group decision support system. Enabling technologies provide the foundation necessary for building the three systems.

5.1 Workgroup Support System

As mentioned earlier, products and services are increasingly provided by workgroups or teams. Such teams have been labeled "virtual teams', "concurrent product development teams", and "integrated product development teams". Indeed, great deal of work is being devoted to understanding how to organize and deploy successful teams [10]. Technologies for team support

address needs for workgroup communication, cooperation, and coordination. Technologies of interest here include video and desktop conferencing, group calendering and scheduling, shared data bases, electronic meeting facilities, and workflow management systems.

5.2 Intelligent Information System

Workgroup support system requires technologies that can provide means for creating, storing, updating, and retrieving information so that workgroup members can use the information simultaneously, without the constraints of time and space, and within the deadlines allocated to their tasks. Intelligent information system provides right information to the right user at the right time. It provides information for maximum understanding. Technologies of interest here include intelligent agents, intelligent data bases, information filters, collaborative authoring systems, and search and retrieval mechanisms.

5.3 Group Decision Support System

Group decision support system enables workgroup members to effectively use the information provided by intelligent information system, to maximize the benefits of using collaborative computing. Decisions can be structured or ill-structured depending on the quality of available information. When information available is uncertain and facts are controversial, it may be necessary to use computer-supported negotiations and conflict resolution techniques, and to capture decision rationale and history. Collaborative technologies of interest here include techniques from distributed artificial intelligence, blackboard technology, and computer-supported group dynamics [11].

6. AN ILLUSTRATIVE APPLICATION

To illustrate how collaborative computing is utilized, a real application is described in this section. The application is required to support several geographically distributed project teams responsible for rapidly designing new products and specifying the associated manufacturing plans and processes. These groups include product design, manufacturing, and marketing. The application needs to support exchange of large amount of information and data and maintain shared understanding of the project through out its life cycle. Group calendering and scheduling tools are used to track project and team performance. Further, collaborative process models are developed. These models represent responsibilities and interdependencies of the three groups in terms of their functions and allocated organizational resources, and constraints affecting their work.

Collaborative models formally describe tasks to be performed and also capture decisions as well as the reasoning systems for developing and selecting product decisions. The models also describe data and information dependencies. For example, engineering designers' models explicitly describe how design and associated manufacturing processes will affect the product costs. Thus, product designers need access to the parameters that characterize manufacturing costs by linking design activities with resultant manufacturing processes. Similarly, manufacturing engineers' representation of production processes considers the impacts of design decisions on the manufacturability of new products.

The product design group develops representations of new products in mathematical and qualitative terms. The group is concerned with issues such as selection of materials, component relationships, and performance of alternative designs. The group also assesses how difficult it would be to produce particular functions specified for the new products. The group therefore relies on models of manufacturing processes developed by manufacturing engineers. Both the design and manufacturing groups invoke applications from their desktop to perform design analysis and develop process plans that specify sequence of manufacturing operations to build the products.

Similarly, the marketing group is concerned with satisfying the customers' needs, Their model is concerned primarily with representing the customers' perspective that must be clearly understood by design and manufacturing groups.

In this application, groups use various technologies and tools. The product data are stored in a shared database accessible by any team member with proper authorization. Workflows among the three groups are formally delineated to manage data and information flows as well as for tracking and monitoring progress of the project activities. Group scheduling is used to understand how contributions of various team members affect the project schedule and performance. Video and desktop conferencing are used to maintain on-going dialog among the groups. Electronic notebooks are kept for project documentation as well as for referencing by any project team member. In this application, it's the collection of such tools, rather than individual technologies, that make the difference in reducing product-market time and product cost, and improving product quality.

This application vividly illustrates how collaborative computing can be used to support rapid development of new products and how it enables capturing, formalizing, and communicating the concerns and considerations of the various groups. Currently, many organizations use the data flow approach; i.e., a group performs its activities, produces data for other groups, and communicates these data either manually or automatically. However, merely providing data to each group does not ensure effective coordination of activities and dependencies. The use of collaborative computing takes these groups far beyond their need for data from each other. Rather, collaborative computing sets up an environment in which groups are able to function indeed as a team.

7. CONCLUDING REMARKS

This paper has provided an overview of collaborative computing, an emerging paradigm for supporting distributed workplaces and organizations. Considerable work is still needed to more fully understand the potential of this paradigm and to determine the best ways to provide computer support to workgroups. However, it seems certain that we are moving in an era of computing that is characterized by the need to support collaborative enterprises using ubiquitous computing, high speed networks, cooperative processing, and application development tools that significantly improve the productivity of software engineers. The real challenges lies :in (a) determining the best way to integrate rapidly improving, commercially available technological capabilities for producing collaborative solutions; (b) building necessary computing infrastructures

that make it possible to implement these solutions; and, (c) developing appropriate descriptions for formally expressing the computer-supported human collaboration.

REFERENCES

1. J.D. Palmer and N.A. Fields (eds.), Computer-Supported Cooperative Work, Special Issue of Computer, May 1994.
2. Special Section devoted to Computer-Supported Cooperative Work, Communications of the ACM, December 1991.
3. T.W. Malone and K.G. Crowston, "Toward an Interdisciplinary Theory of Coordination", ACM Computing Surveys, March, 1994.
4. M. Jarke, et al, "Sharing Processes: Team Coordination in Design Repositories", International Journal of Intelligent and Cooperative Information Systems, 1992.
5. R. Bentley, et al, "Architectural Support for Cooperative Multi-user Interfaces", Computer, May 1994.
6. J.S. Brown and P. Duguid, "Organizational Learning and Communities of Practice", Organization Science, February 1991.
7. R. Medina-Mora, T. Winograd, R. Flores and F. Flores, "The Action Workflow Approach to Workflow Management Technology", Proceedings of ACM 1992 Conference on Computer-Supported Cooperative Work, 1992.
8. T. Winograd and F. Flores, Understanding Computers and Cognition, Addison-Wesley, 1987.
9. R. Johansen, Groupware: Computer Support for Business Teams, The Free Press, 1989.
10. J. Katzenbach and D. Smith, The Wisdom of Teams, Harvard Business School Press, 1993.
11. L.M. Jessup and J.S. Velacich (eds.), Group Support Systems, New Perspectives, MacMillan, 1993.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Sparse block and cyclic data distributions for matrix computations*

R. Asenjo, L.F. Romero, M. Ujaldón and E.L. Zapata

Dept. Arquitectura de Computadores, Universidad de Málaga, Spain
Plaza El Ejido S/N, 29013 Málaga. Spain - (ezapata.atc.ctima.uma.es)

A significant part of scientific codes consist of sparse matrix computations. In this work we propose two new pseudoregular data distributions for sparse matrices. The Multiple Recursive Decomposition (MRD) partitions the data using the prime factors of the dimensions of a multiprocessor network with mesh topology. Furthermore, we introduce a new storage scheme, storage-by-row-of-blocks, that significantly increases the efficiency of the Scatter distribution. We will name Block Row Scatter (BRS) distribution this new variant. The MRD and BRS methods achieve results that improve those obtained by other analyzed methods, being their implementation easier. In fact, the data distributions resulting from the MRD and BRS methods are a generalization of the Block and Cyclic distributions used in dense matrices.

1. INTRODUCTION

One of the most challenging problems in distributed memory multiprocessors is to find good data distributions for irregular problems [19]. The main characteristics of the irregular problems are: Low spatial locality (irregular data access and multiple levels of indirection); low temporal locality (limited data reusability) and the sparse matrices need to be represented in a compact way so that the storage requirements and computational time are kept to reasonable levels.

One solution to this problem is the one proposed by Saltz et al [21] that consists in endowing the compiler with a run-time library (PARTI) that facilitates the search and capture of data located in the distributed memory. The most important drawback of this approach is the large number of messages that are generated as a consequence of accessing a distributed data addressing table, and its associated overhead of memory (value based distributions [13]). In fact, the communications have a dominant impact on the performance of massively parallel processors [6]. Besides, this table occupies a relevant amount of memory. In order to enable the compiler to apply more optimizations and simplify the task of the programmer, Bick and Wijshoff [4] have implemented a restructuring compiler which automatically converts programs operating on dense matrices into sparse code. This method postpones the selection

* This work was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC92-0942-C03 and by the Human Capital and Mobility programme of the European Union (ERB4050P1921660)

of a data structure until the compile phase.

Another alternative, the one we will follow in this work, consists in defining heuristics that perform an efficient mapping of the data and can be incorporated to the data parallel languages in the same way as the popular block and cyclic distributions [23]. The idea is to define pseudoregular data distributions for efficient addressing of sparse matrices without expensive global tables. A pseudoregular data distribution must preserve a compact representation of matrices and vectors, exploit the locality of data and computations, obtain an efficient load balance, and minimize the communications.

Sparse matrix vector multiplication (SpMxV) constitutes one of the most important basic operations of numerical algebra and scientific computation [9], [18]: solution of equation systems by means of iterative methods; Sparse neural networks [14]; EM reconstruction on tomography [7]; etc. The computation of the SpMxV product is completely different from the general case (dense matrices). Sparse methods are mainly based on compacting the data in order to reduce the memory needs and optimize the arithmetic operations. As the SpMxV algorithm is highly computation intensive, there has been great interest in developing parallel formulations for it and test its performance in different parallel architectures [1] and VLSI mesh implementations [15]. In this paper we will concentrate on distributed memory mesh multiprocessors.

Multiprocessor systems with mesh topology present a simple interconnection network that makes them attractive for massively parallel computation. An important number of real machines based on this architecture are currently available. The multiprocessors with mesh topology are made up by a network of processing elements (PEs) arranged as a d-dimensional matrix $A(p_{d-1}, p_{d-2}, ..., p_0)$, where p_i is the size in dimension i. A PE located in $A(i_{d-1}, i_{d-2}, ..., i_0)$ is connected with the PEs located in $A(i_{d-1}, ..., i_j \pm 1, ..., i_0)$ with $0 \leq j < d$ (if they exist). In this work we will consider two dimensional meshes of size $p \times q$. In what follows, we will call the processor located in row r and column s of the mesh PE[r,s]. Using simple indexing, PE[t] represents the t-th PE, $0 \leq t < p \times q$.

In designing parallel sparse algorithms, a key issue is the distribution of the workload among the PEs. Usually we have to solve the tradeoff between a balanced distribution of workload and a minimal communication and synchronization overhead. The complexity of the parallel algorithm for the SpMxV product is strongly conditioned by the distribution of the data. Choosing of a good partitioning for the sparse matrix is crucial in order to balance the load and minimize communications. In this work we present a new distribution method we call Multiple Recursive Decomposition (MRD). The MRD method performs the data partitioning using the prime factor decomposition of the dimensions of the multiprocessor. Furthermore, we introduce a new variant of the Scatter distribution (we will name Block Row Scatter (BRS)), which organizes the storage of data using a storage-by-row-of-blocks. We will analyze and compare the performance of the MRD and BRS methods with several alternative methods for obtaining the SpMxV product in 2D meshes.

The organization of this work is as follows. In section 2 we define the basic steps of an iterative algorithm that includes the SpMxV product as a basic core. The MRD method is described in section 3. A new storage-by-row-of-blocks for the Scatter method it is presented in section 4. Finally, in section 5 we will carry out a performance analysis of the two new data distributions and their specification on HPF (High Performance Fortran).

2. SPARSE MATRIX VECTOR MULTIPLICATION

Given a matrix M of dimensions mxn, and a vector a, the Sparse Matrix-Vector product (SpMxV) $c=M\,a$ is mathematically characterized by expression

$$c_i = \sum_{j=0}^{n-1} M_{ij} \cdot a_j \qquad i=0,1,\ldots,m-1 \qquad \textbf{(1)}$$

In most cases, as in the iterative methods for solving equation systems, the resulting vector c of a SpMxV product is converted into an operand vector a, either directly or modified, of a new product (this requires that m=n). When considering matrix M as sparse, a significant savings both in computation time and in local storage requirements will be achieved. We will call *Sparsity rate ß* the ratio between the non null elements (*entries*) of M (we will denote this quantity as α) end the total number of elements (m·n). Vectors a and c, will be taken from now on as dense (there is no special treatment of the null elements). The complete algorithm for the SpMxV product is

Preprocessing (M, a^1)
for v=1 *to* number of iterations *do*
    Product (M, a^v)
    Collection (c^v)
    Redistribution (c^v, a^{v+1})
end for

Once matrix M has been distributed (in the *Preprocessing* stage), each elements of the operand vector a is distributed to each column of matrix M (specifically, to the PEs where its elements are stored). The distribution of vector a is equivalent to the creation of a matrix A consisting in m copies of vector a^T (T means transpose), which contains the same number of rows and columns as M. This process is known as array expansion [10]. Mathematically, matrix A is obtained

$$\begin{pmatrix} 1 \\ 1 \\ .. \end{pmatrix} \cdot \begin{pmatrix} a_1 & a_2 & \ldots \end{pmatrix} = \begin{pmatrix} a_1 & a_2 & \ldots \\ a_1 & a_2 & \ldots \\ \ldots & \ldots & \ldots \end{pmatrix} \qquad \textbf{(2)}$$

Each element A_{ij} will be stored in the PE that contains elements M_{ij}.

During the second phase, the *Product* of each element of matrix M with the corresponding element of vector a is obtained. Using the previous analogy, it consists in the product of

components $R=M\otimes a$, where $R_{ij} = M_{ij} \cdot A_{ij}$

$$\begin{pmatrix} M_{11} & M_{12} & \dots \\ M_{21} & M_{22} & \dots \\ \dots & \dots & \dots \end{pmatrix} \otimes \begin{pmatrix} a_1 & a_2 & \dots \\ a_1 & a_2 & \dots \\ \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} M_{11}\cdot a_1 & M_{12}\cdot a_2 & \dots \\ M_{21}\cdot a_1 & M_{22}\cdot a_2 & \dots \\ \dots & \dots & \dots \end{pmatrix} \quad (3)$$

The computation of the product is carried out within each PE without communications.

In the *Collection* stage, the elements of vector c are obtained by collecting and adding the products of the second phases. Each element c_i is obtained as the sum of the i-th row of matrix $R=M\otimes a$. Finally, in the *Redistribution* phase, the elements c^v (result of the v-th iteration of the product) are transferred from the place where they are stored to the places where the a^{v+1} are required for the *Product* stage of the v+1 iteration. Note that in most cases, the *Redistribution* could partially overlap the *Collection* phase. Figure 1.a shows an example of sparse matrix (m=n=8, α=13, ß=20.3); figure 1.b represents the sequential SpMxV product for a storage by row of matrix M (see figure 1.c, where vectors Data, Column and Row store entries, columns and number of entries per row, respectively).

The distribution of matrix M is the operation with the greatest impact on the complexity and efficiency of the SpMxV product, being the one that requires a more precise analysis. However, sparse matrices appear with very different patterns, and so it is very difficult to establish comparative parameters a priori. Nevertheless, as we will see now, statistics is a tool that can be of help, although with certains precautions.

When a population, as the elements of a sparse matrix M, with two kinds of individuals ("entries" and "zeroes"), is randomly distributed in any number of equally sized groups (p.e., p·q groups), it can be easily shown that, with a probability of being right of Π, the maximum number of individuals of the kind "entries" in a group is

$$w = \frac{\alpha}{p\cdot q} + z_\pi \left(\frac{\alpha}{p\cdot q}\right)^{1/2} = \left\langle \frac{\alpha}{p\cdot q} \right\rangle^{z_\pi}, \qquad (\pi = \sqrt[p\cdot q]{\Pi}) \quad (4)$$

being α the number of entries, and z_π the variable corresponding to the area π under the Gaussian curve. From now on, we will follow the notation $\langle V\rangle^z = V + z\cdot V^{1/2}$, where z is a parameter that modifies the mean value in all PEs of any expression (the expected for a perfect balance) to obtain the maximum expected value between all PEs.

3. MULTIPLE RECURSIVE DECOMPOSITION

Recently, Berger and Bokhari [3] have proposed the *Binary Recursive Decomposition (BRD)*, a well-known distribution algorithm where the matrix M is recursively bisected, alternating vertical and horizontal partitions until we have as many submatrices as PEs. Other possibilities for performing these divisions consist in altering the order of the partitions so that horizontal and vertical partitions are not alternated, introducing other arrangements [22]. This distribution method, apart from achieving a good load balance, permits a simple assignment

of the submatrices to a PE network with hypercube or binary tree topology. However, there are serious problems in communications, as adjacent elements in the matrix may be projected onto PEs that are not directly communicated. Besides, the BRD method is only applicable to PE networks with a number of PEs that is a power of two.

In this section we present a new method for the distribution of matrix *M*. We call it *Multiple Recursive Decomposition (MRD).* It can be considered a generalization of the BRD method for an arbitrary number of PEs.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 & 10 \\ 0 & 11 & 0 & 0 & 0 & 12 & 13 & 0 \end{pmatrix}$$

1.a. $\alpha = 13$, $\beta = 20.3\%$

```
DO I=1, N
  DO J=Row(I), Row(I+1)-1
    Y(I)= Y(I)+Data(J)* X(Column(J))
  ENDDO
ENDDO
```

1.b. Sequential SpMxV Code

Data	*Column*
1	*1*
2	*7*
3	*3*
4	*7*
5	*8*
6	*5*
7	*4*
8	*2*
9	*5*
10	*8*
11	*2*
12	*6*
13	*7*

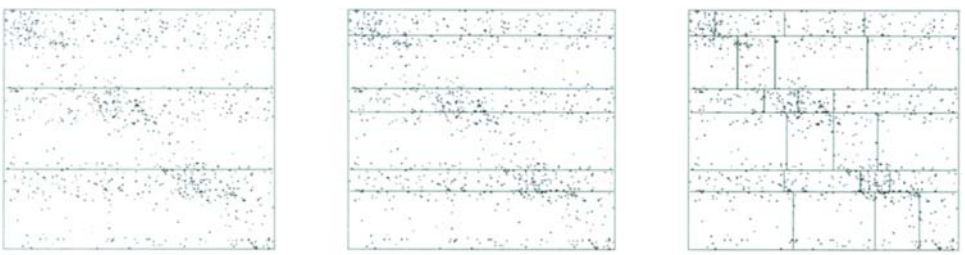
Row
1
3
5
6
7
8
9
11
14

1.c. Data Storage

1. Example of a Sparse Matrix

Let us assume a p processor network and let $P_1 \cdot P_2 \cdot ... P_k$ be the prime factor decomposition of p. The MRD distribution method performs p partitions of matrix *M* in k levels, operating in a very similar way to the BRD method. During the level 1 partition, matrix *M* will be divided into P_1 submatrices with the same load (with the same number of entries) by means of divisions in the horizontal (or vertical) direction. Each submatrix is divided (in a perpendicular direction to the one used in the previous level) into P_2 submatrices during the level 2 partition. This process continues until the level k partition is reached, alternating horizontal and vertical divisions.

Even though the MRD method still presents the communications problems pointed out for the BRD, it is possible to adapt it to the requirements of the SpMxV product in a PE mesh. Let us assume a mesh of size pxq, where the decomposition in prime factors of p and q is $p=P_1 \cdot P_2 \cdot .. \cdot P_a$ and $q=Q_1 \cdot Q_2 \cdot .. \cdot Q_b$, respectively. The MRD decomposition of matrix *M* into p submatrices will be executed as indicated for the BRD, but without alternating the direction of the partitions, always taking the horizontal direction. Figure 2.b shows the resulting p submatrices for the p=6 (P_1=3, P_2=2) case. In a second phase, we will perform a partition into q submatrices of each of the p submatrices generated in the horizontal division using the same technique with the prime factors of q but in the vertical direction (see figure 2.c, where $q=Q_1 \cdot Q_2$=4).



2.a. (P_1=3) **2.b.** (P_2=2) **2.c.** ($Q_1 \cdot Q_2$=2·2)

2. Multiple Recursive Decomposition (p×q= 6×4 PEs)

As it will now be shown, the MRD distribution method is enough and efficient for the requirements of the SpMxV product. During the *Preprocessing* stage, matrix *M* is divided and distributed as we have indicated. Also, this phase can be carried out in parallel as follows. Initially, every PE is owner of the matrix *M*. Then, 1) in the i-th horizontal partitions ($i \in \{1,..,a\}$), each PE[r,s] divides the matrix it owns after the level i-1 partition into P_i equal submatrices, keeping submatrix $\lfloor r/(P_{i+1} \cdot P_{i+2} \cdot .. \cdot P_a) \rfloor$,and rejecting the rest. Next, 2) in the i-th vertical partitions ($i \in \{1,..,b\}$), each PE[r,s] divides the matrix it owns after the previous partition into Q_i equal submatrices, keeping submatrix $\lfloor s/(Q_{i+1} \cdot Q_{i+2} \cdot .. \cdot Q_b) \rfloor$, and rejecting the rest. Even though this procedure seems to require large amounts of local memory for storing *M* in each PE, in practice this decomposition can be carried out over the symbolic matrix. The algorithmic complexity is m·n.

During the horizontal division, each one of the p submatrices generated contains, at most, $\alpha/p+m/2$ entries of the matrix (m/2 is the maximum possible unbalance, as the division does not affect the contents of a row). The maximum number of rows in the submatrices, $\langle m/p \rangle^{Z_r}$

(where z_r, is a "shape factor" depends on the concentration of entries in certains rows). After the vertical division, this unbalance is partially corrected, resulting the maximum number of entries in a PE

$$w=\frac{\alpha}{p\cdot q}+\frac{m}{2\cdot q}+\frac{1}{2}\cdot\left\langle\frac{m}{p}\right\rangle^{z_r} \tag{5}$$

In the *Preprocessing* stage each PE receives as many elements of vector *a* as columns it has in its local submatrix $M_{r,s}$. Thus, the MRD method induces q partitions of vector *a* for each PE row, assigning $\langle n/q\rangle^{Z_c}$ elements to each PE (where z_c, as "second level shape factor" depends on the concentration of entries in certains columns inside of certains rows). Figure 3.a shows the MRD partitioning of the matrix example of figure 1.a, considering a mesh of 2×2 PEs. Note that the MRD distribution splits *M* into four sparse rectangular blocks. Each rectangular block is stored in its corresponding PE as shows figure 3.b. The local storage by row of figure 3.b preserve the structure of the sequential algorithm (see figure 1.c). The only difference between figures 1.c and 3.b is which this last stores in vector *Column* the local column of each rectangular block.

The *Product* stage is carried out in each PE, that will contain a subvector c^r of length given by the p partitions of the rows of *M*. In this case, the MRD method induces p partitions of vector *c*, assigning $\langle m/p\rangle^{Z_r}$ elements to each PE. Figure 2.c shows the result of applying the MRD method to matrix *M* and vectors *a* and *c*, considering a mesh with 24 PEs (p=6,q=4). Note that the horizontal partition only affects vector *c*, whereas the vertical partition decomposes vector *a*.

In the *Collection* stage, all of the elements R_{ij} of the same row are added in each PE. These partial results are collected and added (using the cascade addition algorithm) in each of the rows of the mesh without any need for moving data between PE rows. q message exchanges of size $\langle m/p\rangle^{Z_r}$ are required. After this stage, all of the PEs in a row of the mesh have a copy of the corresponding *c* subvector.This data redundance is useful in iterative applications requiring the *Redistribution* stage.

The *Redistribution* of vector c^v can be carried out by means of an exchange of data between rows so that each PE obtains this way all the elements it requires from c^v, in order to transform them into elements of a^{v+1}. q messages of size $\langle n/q\rangle^{Z_c}$ are needed.

We now summarize some of the most significant parameters of the iterative SpMxV multiplication based on the MRD distribution method. From now on, and without any loss of generality we will consider that the mesh and matrix *M* are square and of dimensions p×p and m×m, respectively. Equation (5) specifies the number of entries assigned to each PE. This expression allows us to state that the MRD method generates a good computational load balance, performing w multiplications and $w+\langle m/p\rangle^{Z_r}\cdot\log(p/2)$ additions. The size of the local memory for each PE is given by the number of entries of local submatrix *M* (w), its addressing, and local subvectors *a* and *c*: $w+\langle m/p\rangle^{Z_r}+\langle m/p\rangle^{Z_c}$ floating point data and $w+\langle m/p\rangle^{Z_r}$ integers, necessary for addressing the entries using a storage-by-row of submatrix *M*. The communications are concentrated in the *Collection* and *Redistribution* stages, generating a total of 2·p messages of size $\langle m/p\rangle^{Z_r}$ and $\langle m/p\rangle^{Z_c}$.

It can be easily seen that the MRD distribution scheme encompasses, as particular cases, the BRD [3], [22] and One Way Strip Partitioning (OSP) [2], [5], [8],[20] methods. When the

number of PEs of the mesh is a power of two, the MRD method coincides with the BRD method. The OSP method coincides with the MRD method when it is just applied to the rows (Row OSP method) or columns (Column OSP method) of matrix *M*. In section 5 we will compare the MRD method to other alternative methods for the SpMxV product in multiprocessors with mesh topology.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 & 10 \\ 0 & 11 & 0 & 0 & 0 & 12 & 13 & 0 \end{pmatrix}$$

3.a. Matrix Decomposition

#0

Data	*Column*
1	*1*
3	*3*
6	*5*
-	-

Row
1
2
3
3
4

#1

Data	*Column*
2	*2*
4	*2*
5	*3*
-	-

Row
1
2
3
4
4

#2

Data	*Column*
7	*4*
8	*2*
11	*2*
-	-

Row
1
2
3
3
4

#3

Data	*Column*
9	*1*
10	*4*
12	*2*
13	*3*

Row
1
1
1
3
5

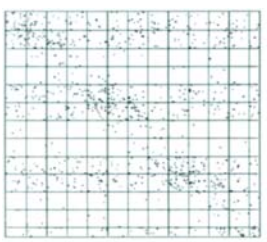
3.b. Local Data Storage_by_Row

3. Multiple Recursive Decomposition (p×q=2×2, N×N=8×8)

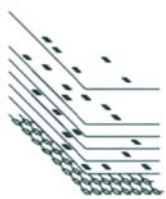
4. BLOCK ROW SCATTER METHOD

The Scatter distribution techniques are based on the division of any computation domain (as the case of a sparse matrix) into several blocks, all of the same spatial shape and size. Each of these blocks is uniformly distributed over the p×p PE network, so that each PE contains a fraction of each block.

Let us consider a sparse matrix M of size m×m partitioned into a set of submatrices $B(k,l)$ of size p×p, so that $M_{ij}={}^{kl}B_{rs}$ where $i=p\cdot k+r$, $j=p\cdot l+s$ $(0\le i,j<m)$; $k=\lfloor i/p \rfloor$, $l=\lfloor j/p \rfloor$ $(0\le k,l\le m/p)$; $r=i-k\cdot p$, $s=j-l\cdot p$ $(0\le r,s<p)$. The pairs (i,j), (r,s) and (k,l) are the global indices, local indices and block indices, respectively. For matrix M to be divisible into submatrices $B(k,l)$, it may be necessary to add rows or columns with null elements to it. The distribution of the elements of matrix M among the PEs is by projecting each one of the blocks of size p×p onto the PE mesh. The scatter method admits larger block sizes [12], but this increase in the grain causes a bigger load unbalance. Figure 4 shows the distribution of the blocks of M onto a mesh.



4.a. Matrix Partitioning



4.b. Matrix Distribution

4.c. Local Memories

4. Scatter Decomposition (16×16 PEs)

Once the partition of matrix M among the PEs has been carried out as indicated, the entries distributed to each of the PEs can be stored in different ways in order to optimize the number of calculations and the memory requirements. In [1] we find a review of the storage technique proposed by Morjaria and Makinson [16] (MM method) and two new schemes are proposed by Andersen et al, the Extended Stacking Scheme (ESS) and the Block Banded Scheme (BBS). The BBS scheme contains the other two as particular cases. The more significant differences between this three schemes appears in the collection stage of a SpMxV product. In the MM scheme all the elements of the k-th block row are added in each PE and the corresponding element of c is obtained in each row before increasing the k index. So, the number of additions is considerably increased. To reduce it, the ESS scheme [1] does not compact in the local memory the elements coming from all the blocks, only those belonging to the same block row (k index). In this case, the local memories are organized into $\lfloor m/p \rfloor$ different local bands. However, the ESS scheme can produce an excessive increase in the requirements of memory when the matrices are very sparse.

The BBS scheme permits grouping the elements of different consecutive row blocks, which, according to ESS, are in different local bands of memory, into a single band (*block band*). A

set of local bands are united into a block band only if the number of entries of that group of row blocks is smaller than a parameter Φ that was predetermined. The local memory required by a BBS scheme is a function of parameter Φ. Given the broad variability interval of Φ we must perform a statistical estimation of it. In general we will consider equation

$$w(\Phi)=\frac{\alpha}{p^2}+z(\Phi)\cdot\left(\frac{\alpha}{p^2}\right)^{1/2} \tag{6}$$

where $z(\Phi)$ varies between z_π (when $\Phi=\alpha$) and $z_\pi \cdot \lfloor m/p \rfloor^{1/2}$ (if $\Phi=0$). It can be easily seen that equation (6) is a very approximated estimation of the memory requirements experimentally reported in [1]. This way, if $\Phi=0$, there is no grouping of local bands (the method coincides with ESS); if $\Phi=\alpha$, the grouping is complete, that is, all of the elements are in a single memory area, as suggested by the MM method. An additional problem of the BBS scheme is that in [1] no technique is described for determining the most adequate Φ parameter, although it is known that it depends, to a great extent, on the sparsity rate of the matrices, being small in matrices that are not very sparse and large in very sparse matrices.

We introduce in this section a new scheme, we will name Block Row Scatter Method (BRS), that significantly increases the efficiency of the Scatter distribution of sparse matrices. We call u plane the set of elements located in the u-th position of the local memory of each PE. During the *Preprocessing* stage of the BRS method, the entries of M will be stored in row major order in consecutives locations of the corresponding PE memory. It is easy to see that in each plane we may find entries coming from different blocks (k,l). The maximum number of entries in a PE is

$$w=\frac{\alpha}{p^2}+z_\pi\cdot\left(\frac{\alpha}{p^2}\right)^{1/2}=\left\langle\frac{\alpha}{p^2}\right\rangle^{z_\pi} \tag{7}$$

which coincides with the number of planes.

The distribution of vector a is carried out in block row major order. That is, vector a is divided into $\lfloor m/p \rfloor$ blocks of size p, where each block a^l multiplies the l-th block column of matrix M. Each element of block a^l is only stored in those planes that reference elements of the l-th block column of matrix M. Consequently, in order to carry out the distribution we have to store with each element the index l corresponding to the block column to which it belongs (w integers). It is also necessary to have w floating point memory positions in each PE initially for storing the elements of a and afterwards for storing R_{ij}.

In the *Product* stage of the BRS method, each PE performs as many multiplications as memory planes there are in its local memory. It is possible to define a Scatter method which drastically reduces the number of additions in the *Collection* stage. A first approach consists in storing with each entry of matrix M the indices (k,l) of the block to which it belong. This makes the organization of data as block bands unnecessary, although we are introducing redundance in the storage of data. A more efficient alternative is to organize the storage of the data using a storage-by-row-of-blocks. The Block Row Scatter method allows us to consider matrix M as a matrix of blocks of size p×p. Besides, each PE can only have a single element of each block of matrix M, defined by the position of the PE in the mesh, assigned

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 & 10 \\ 0 & 11 & 0 & 0 & 0 & 12 & 13 & 0 \end{pmatrix}$$

5.a. Matrix Decomposition

#0

Data	*Column*
1	*1*
2	*4*
9	*3*
-	-

Row
1
3
3
3
4

#1

Data	*Column*
5	*4*
7	*2*
10	*4*
-	-

Row
1
1
2
3
4

#2

Data	*Column*
3	*2*
4	*4*
6	*3*
13	*4*

Row
1
3
4
4
5

#3

Data	*Column*
8	*1*
11	*1*
12	*3*
-	-

Row
1
1
1
2
4

5.b. Local Data Storage_by_Row_of_Blocks

5. Block Row Scatter (p×q=2×2, N×N=8×8)

to it. Consequently, the entries of M assigned to a PE are identified if we store them in a vector and if in two auxiliary vectors we store the number of entries in each block row and the index of the block column. The first auxiliary vector will have m/p elements and the second's length is equal to the number of entries assigned to the PE (see equation (7)).

Figure 5.a shows the BRS partitioning of the matrix example of figure 1.a, considering a mesh of 2×2 PEs. Note that the BRS distribution organizes M into a sparse block matrix (16 blocks). Each entry of a block is automatically assigned to a specific PE as shows figure 5.b. The local storage by row of blocks of figure 5.b preserve the structure of the sequential algorithm (see figure 1.c). The only difference between figures 1.c and 5.b is which this last stores in vector *Column* the local block column of each entry.

Using the storage-by-row-of-blocks the PEs know the value of block row index (k) associated with each entry. This allows us to compute the *Collection* stage in two phases. In the first phase each PE calculates all $\lfloor m/p \rfloor$ partial additions without communications. Next, using the cascade addition algorithm all the PEs of the r-th row obtain the $\lfloor m/p \rfloor$ elements of vector c. This phase requires p messages of size $\lfloor m/p \rfloor$. Summarizing, in the *Collection* stage of the BRS method $w+\lfloor m/p \rfloor \lfloor \log_2(p/2) \rfloor$ additions are carried out.

In the *Redistribution* stage ($a^{v+1}=f(c^v)$) the $\lfloor m/p \rfloor$ values of c^v stored in the r-th row of the PE mesh have to be distributed to the PEs in the r-th column. This distribution can be efficiently carried out if the PE of indices (r,r) is the one providing the data to each column. Therefore, only p messages of size $\lfloor m/p \rfloor$ are required for this operation.

5. EVALUATION

In this section, we will carry out a comparative analysis of both methods proposed in the two previous sections and other methods recently appeared in the literature. We also include a High Performance Fortran specification of the iterative SpMxV multiplication and some experimental results. In order to facilitate the comparison of these, we will substitute the variables until they are reduced to three: α, p, and m (the mesh and matrix M are square). A stricter measure for the calculation of the comparison parameters consists in the determination of the worst possible case. However, we have discarded this option as it does not reflect the characteristics of most of the methods. On the other hand, an analysis of the pattern of the sparse matrix can be carried out using only m^2 boolean steps in order to precisely determine the exact value of each one of the parameters. This makes sense when we are interested in a specific type of sparsity. Finally, we consider that a simultaneous exchange of messages between two adjacent PEs is possible. In this case, the additions performed in the *Collection* stage can be simultaneously executed in all the PEs that supply the terms to be added.

The result of the analysis performed is specified in table I. In the first column of the table we have expressed the difficulty of implementation for each distribu-

tion method. The next three columns of the table show the complexities of the *Preprocessing* and *Product* stages and the number of additions performed in the *Collection* stage. The values found in the table for each parameter are a estimation to their real values. In those cases in which it has been impossible to establish an estimation of the parameter a priori we have indicated the worst cases. This is what happens, for instance, with the number of additions calculated for the BBS Scatter method when $\Phi=\alpha$, whose value is obtained from the number of planes that reference the k-th block row. The optimum value for $\Phi=\alpha$ is the expected value for $\Phi=0$. The real number of additions for $\Phi=\alpha$ (and, in general, for any value of Φ) is a quantity between the two extreme cases presented in the table. We have also made use of the worst possible case in the OSP and MRD methods, although with less influence on the approximations. Due to the "border effect" we have added m/2 entries in the OSP method and <m/p>/2 entries in the MRD method.

The Snake method [14], [15] is the one that presents a smaller number of products and sums due to the optimal load balance it obtains, but it requires a disproportionate number of messages (see fifth and sixth columns of table I) and more local memory in each PE (see seventh column). The MRD method exhibits the smallest arithmetic complexity of the three. The difference between the MRD and Row OSP is due to the border effect we have mentioned before, which is more pronounced in partitions with elongated shapes and which can be critical in very sparse matrices with dense rows.

The BBS Scatter(α) method presents an optimal behavior in the number of products but generates a disproportionate number of additions. This problem was already pointed out in [1]. When Φ decreases, the number of additions is significantly reduced at the price of an increase in the number of products. The BRS method drastically reduces the number of additions and yields a smaller arithmetic complexity than the MRD method, although it slightly increases the size of the local memory (see seventh column). It also simplifies programming of the Scatter distributions.

The number of messages, together with their sizes, of the *Collection* (r) and *Redistribution* (d) stages are reflected in the fifth and sixth columns of table I. The Snake method is the one presenting a larger number of messages as a consequence of the change of storage from Snake-like Column Major Order to Snake-like Row Major Order. The rest of the methods require a much smaller number of messages ($2 \cdot p$), and the Row OSP method is the one that exhibits a smaller message size in inter-row communications. We must also point out the regularity in the size of the messages of the Scatter methods.

The seventh column of table I shows the local memory requirements. The memory needed for the MRD and Row OSP methods is much smaller than for the rest as a consequence of the simplicity of the local storage scheme (storage-by-row). Again, the MRD method obtains smaller values as a consequence of the minimization of the lateral effects. Note that the BRS method needs an intermediate amount of memory between the two extremes of the BBS Scatter method.

Figure 6 shows the normalized number of arithmetic computations in a logarithmic scale (fig. 6.a) and the normalized memory size (fig. 6.b) for a PE as a function of the number of PEs in the mesh. We have considered a sparsity rate

of ß=0.1 and m=10000. The Snake method exhibits a constant behavior that is independent from p^2. The Scatter and MRD methods present a smooth increase in the computations and memory requirements, being the Row OSP method the most sensitive to the mesh size. The bigger the mesh, the more difficult it will be to obtain a good load balance with it. Besides, as the number of PEs is increased, the impact of vector a in the size of the local memory grows. If we decreases the sparsity rate, the slope of all the plots in figure 6 is accentuated.

The last column of table I represents the number of evaluations of f (function that transforms the elements of vector c^v into a^{v+1}). The Row (Column) OSP method is the one that performs the smallest number of operations of this type. Nonetheless, it is possible to modify the other methods so that they perform this operation in approximately the same number of cycles ($<m/p^2>$). For example, in the MRD method it is necessary for each region to know the dimensions of the other regions, in which case the size of the messages can also be reduced.

In Figure 7 we include the HPF code [23] for the parallel SpMxV multiplication based on the BRS data distribution. We have used a new pair of directives (highlighted with !!!!!) in order to indicate to the compiler that this data distribution must be generated. The HPF code for the parallel SpMxV multiplication based on the MRD data distribution is exactly the same of figure 7, but changing these two directives by the following ones:

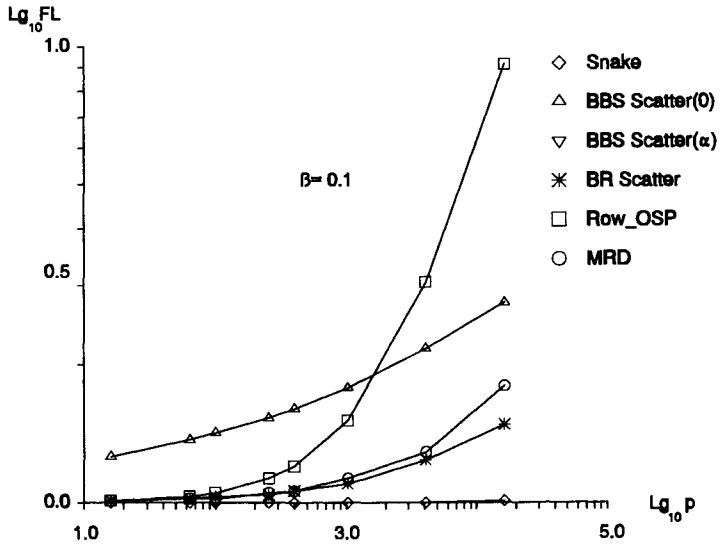
```
!!!!! DISTRIBUTE A(SPARSE BLOCK, SPARSE BLOCK) ONTO MESH
!!!!! DISTRIBUTE Data,Column,Row SPARSE A
```

Looking at the code, we can separate it into two parts: First, we find the declaration part in which we capture the hardware topology we are using, and declare, align and distribute the variables we need to compute the SpMxV; then we place a interface block where we declare the extrinsic subroutine we are going to use. Second, in the execution part, we begin calling the code that performs the local product using SPMD model with local index onto each distributed memory. Then, a global loop will be computed to collect all the partial sums we have performed locally before. This last stage needs communications to obtain the partial sum from processors in the same mesh row. Finally, we realign the solution vector in order to use it as input in the following iteration.

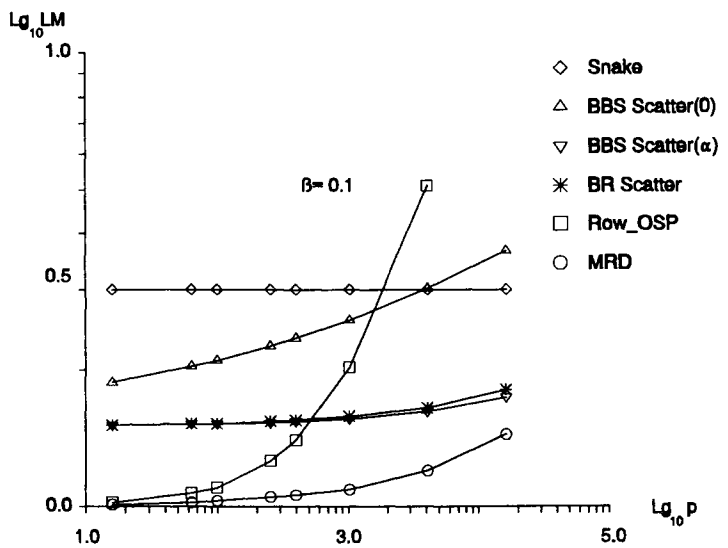
In figure 8 we present the isoefficiency [11] for differents squared meshes performed on a PARAMID 16/i860SYS machine with 16 i860 nodes. It can be observed that the good scalability which presents the parallel SpMxV multiplication is a direct consequence of exploit the spatial locality exhibed by MRD and BRS sparse data distributions.

	PROGRAMMING	PREPROCESS	PRODUCTS	ADDITIONS	N^{os} MESSAGES*	SIZE OF MESSAGES*	LOCAL MEMORY: INTEGERS FLOATS	$a^{v+1}=f(c^v)$ EVALUATIONS
Snake	Complex	$m^2+\alpha\cdot\log\ \alpha$	$\frac{\alpha}{p^2}$	$\frac{\alpha}{p^2}+\log(3\cdot p)$	$(r)\ \frac{3\cdot\alpha}{p}+3\cdot p$ $(d)\ 2\cdot p$	$(r)\ 1$ $(d)\ \frac{\alpha}{p^2}$	$\frac{4\cdot\alpha}{p^2}$ $\frac{2\cdot\alpha}{p^2}$	$\frac{\alpha}{p^2}$
Scatter (BBS,Φ=0)	Half	$m^2\cdot\left\langle\frac{\alpha}{m\cdot p}\right\rangle^{z_a}$	$\left\langle\frac{\alpha}{p^2}\right\rangle^{z_r\cdot(m/p)^a}$	$\frac{m}{p}\cdot\log\frac{p}{2}+\left\langle\frac{\alpha}{p^2}\right\rangle^{z_r\cdot(m/p)^a}$	$(r_r)\ p$ $(d_c)\ p$	$(r_r)\ \frac{m}{p}$ $(d_c)\ \frac{m}{p}$	$\left\langle\frac{\alpha}{p^2}\right\rangle^{z_r\cdot(m/p)^a}$ $2\cdot\left\langle\frac{\alpha}{p^2}\right\rangle^{z_r\cdot(m/p)^a}$	$\left\langle\frac{\alpha}{p^2}\right\rangle^{z_r\cdot(m/p)^a}$
Scatter (BBS,Φ=α)	Half	$m^2\cdot\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$	$\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$	$\frac{m}{p}\cdot\left(\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}+\log\frac{p}{2}\right)$	$(r_r)\ p$ $(d_c)\ p$	$(r_r)\ \frac{m}{p}$ $(d_c)\ \frac{m}{p}$	$\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$ $2\cdot\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$	$\left\langle\frac{m}{p}\right\rangle^{z_a}$
Block Row Scatter	Easy	m^2	$\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$	$\frac{m}{p}\cdot\log\frac{p}{2}+\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$	$(r_r)\ p$ $(d_c)\ p$	$(r_r)\ \frac{m}{p}$ $(d_c)\ \frac{m}{p}$	$2\cdot\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$ $2\cdot\left\langle\frac{\alpha}{p^2}\right\rangle^{z_a}$	$\left\langle\frac{m}{p}\right\rangle^{z_a}$
1_Way Str. Row	Easy	m^2	$\frac{\alpha}{p^2}+\frac{m}{2}$	$\frac{\alpha}{p^2}+\frac{m}{2}\cdot\left\langle\frac{m}{p^2}\right\rangle^{z_a}$	$(d_r)\ p$ $(d_c)\ p$	$(d_r)\ \left\langle\frac{m}{p^2}\right\rangle^{z_a}$ $(d_c)\ \left\langle\frac{m}{p}\right\rangle^{z_a}$	$\frac{\alpha}{p^2}+\frac{m}{2}+\left\langle\frac{m}{p^2}\right\rangle^{z_a}$ $\frac{\alpha}{p^2}+\frac{3\cdot m}{2}+\left\langle\frac{m}{p^2}\right\rangle^{z_a}$	$\left\langle\frac{m}{p^2}\right\rangle^{z_a}$
Multiple Recurs. Decomp.	Easy	m^2	$\frac{\alpha}{p^2}+\frac{m}{2\cdot p}+\frac{1}{2}\cdot\left\langle\frac{m}{p}\right\rangle^{z_a}$	$\frac{\alpha}{p^2}+\left\langle\frac{m}{p}\right\rangle^{z_a}\cdot(\log p-\frac{1}{2})+\frac{m}{2\cdot p}$	$(r_r)\ p$ $(d_c)\ p$	$(r_r)\ \left\langle\frac{m}{p}\right\rangle^{z_a}$ $(d_c)\ \left\langle\frac{m}{p}\right\rangle^{z_a}$	$\frac{\alpha}{p^2}+\frac{m}{2\cdot p}+\frac{3}{2}\cdot\left\langle\frac{m}{p}\right\rangle^{z_a}$ $\frac{\alpha}{p^2}+\frac{m}{2\cdot p}+\frac{3}{2}\cdot\left\langle\frac{m}{p}\right\rangle^{z_a}+\left\langle\frac{m}{p}\right\rangle^{z_a}$	$\left\langle\frac{m}{p}\right\rangle^{z_a}$

(*) (r) recollection of c^v. (d) redistribution from c^v to a^{v+1}. (the r and c subindices represents row and column data movements)



6.a. Floating Points Operations (FL= $f(p^2)$)

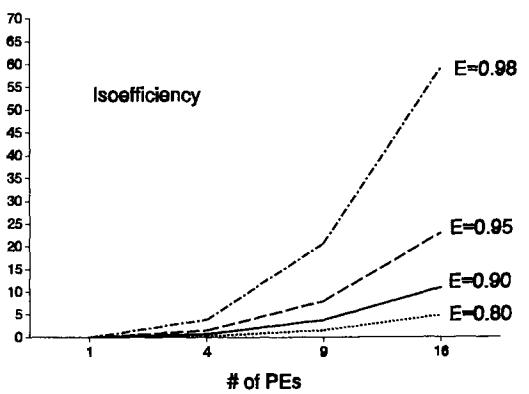


6.b. Local Memories (LM=$f(p^2)$)

6. Scalability (m= 10^4)

```
      PARAMETER(P=NUMBER_OF_PROCESSORS(DIM=1))
      PARAMETER(Q=NUMBER_OF_PROCESSORS(DIM=2))
      PARAMETER(ALPHA=8000,N=1000,K=10)
      REAL X(N), Y(N), P_Y(N,Q), Data(ALPHA)
      INTEGER Column(ALPHA), Row(N+1), V, I, IP
      CHAR DIST_P_Y(9)

      USE HPF_LIBRARY


!HPF$   PROCESSORS MESH(P,Q)
!HPF$   TEMPLATE A(N,N)
!HPF$   ALIGN WITH A(*,:) :: X(:)
!HPF$   DYNAMIC, ALIGN WITH A(:,*) :: Y(:)

!!!!! DISTRIBUTE A(SPARSE CYCLIC,SPARSE CYCLIC) ONTO MESH
!!!!! DISTRIBUTE Data,Column,Row SPARSE A

      CALL HPF_DISTRIBUTION(A,AXIS_TYPE(1)=DIST_P_Y)

!HPF$   DISTRIBUTE P_Y(DIST_P_Y,I) ONTO MESH(I)

      INTERFACE
        EXTRINSIC(HPF_LOCAL) SUBROUTINE SPMXV(X,Y,D,C,R)
            REAL, DIMENSION(:,:), INTENT(OUT) :: Y
            REAL, DIMENSION(:), INTENT(IN) :: X, D
            INTEGER, DIMENSION(:), INTENT(IN) :: C, R
        END SUBROUTINE SPMXV
      END INTERFACE

        DO V=1, K
C --- Perform product and partial sum locally
          CALL SPMXV(X,P_Y,Data,Column,Row)
C --- Collection into each row of PEs
          DO IP = 1, Q
            DO I = LBOUND(Row,1), UBOUND(Row,1)-1
              Y(I) = Y(I) + P_Y(I,IP)
            END DO
          END DO
C --- Redistribution
!HPF$       REALIGN Y(:) WITH X(:)
          X = Y
        END DO
      END


      --- EXTRINSIC SUBROUTINE USING SPMD MODEL ---
      EXTRINSIC(HPF_LOCAL) SUBROUTINE SPMXV(X,Y,D,C,R)
        REAL, DIMENSION(:,:), INTENT(OUT) :: Y
        REAL, DIMENSION(:), INTENT(IN) :: X, D
        INTEGER, DIMENSION(:), INTENT(IN) :: C,R
        INTEGER I,J
C --- Local Product
        DO I = LBOUND(R,1), UBOUND(R,1)-1
          DO J = R(I), R(I+1)-1
            Y(I) = Y(I) + D(J)*X(C(J))
          END DO
        END DO
      END
```

7. HPF Code for SpMxV Multiplication based on BRS Distribution



8. Isoefficiency for squared meshes (BRS and MRD data distributions)

6. CONCLUSIONS

From the analysis of the different techniques for executing the SpMxV product in multiprocessors with mesh topologies we deduce that it is difficult to obtain a perfect equilibrium of the number of entries stored in each PE with an optimum communications cost (as happens in the Snake method). However, values that are very close to this equilibrium can be very satisfactory. Also, the simplicity in the storage of the elements positively influences both the size of the local memory and the number of computations. This causes potentially good distribution methods to loose their efficiency because of their storage complexity (as is the case of MM, ESS and BBS Scatter methods). There is a close relationship between the geometry of the computations and the geometry of the distributions. The computation of the SpMxV product requires a larger number of calculations by rows than by columns and, therefore, it is a good solution to keep the rows together in a single PE, or, at least, to find regularity in communications by rows. For this reason the results obtained with the Row OSP method were better than those of the Column OSP. This idea has contributed to making us opt in the MRD method for performing the first subdivision by rows and then by columns, facilitating this way horizontal communications and reducing the number of additions. Other applications of this method may use variations in the order of the partitions, even using new dimensions.

The Scatter methods are specially effectives when the elements of the computation domain that require a larger number of calculations are grouped in certain zones of the domain. However, this can produce strong load imbalance when matrix M contains structures that are periodically repeated with a period that is an integer multiple of the dimensions of the network [17]. We can prevent the coupling between the matrix and the network by means of the reduction of the size of the latter. On the other hand, a dense row or column can also lead to an inefficient storage in a PE row or column.

The MRD method we propose in this work is presented as a general method in which the OSP and BRD methods are some of its particular cases. It has a good load balance (similar to the BRD method) and a good disposition of message exchange (characteristic of the OSP methods). Consequently, this method achieves results that improve those obtained by the rest of the methods analyzed.

Finally, we want to point out the analogy of the MRD and BRS distributions to the Block and Cyclic distributions used in parallel algorithms with dense matrices. In fact, these last can be considered particular cases that do not require additional storage for addressing the data. The MRD and BRS distributions can also be used to determine good data distributions for irregular problems avoiding expensive phases of communication for addressing of nonlocal data because the PEs have sufficient local information to know where are allocated the data. Moreover, their incorporation to a data parallel language is immediate.

REFERENCES

1. J. Andersen, G. Mitra, D. Parkinson, "The Scheduling of Sparse Matrix-Vector Multiplication on a Massively Parallel DAP Computer", Par. Computing, vol.18, pp.675-697, 1992.
2. C.Aykanat, F.Ozgüner, F.Ercal, P.Sadayappan, "Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes", IEEE Trans. Comput., vol.37, no.12,

pp.1554-1568, 1988.
3. M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors", IEEE Trans. Comput., vol. 36, no. 5, pp. 570-580, 1987.
4. A.J.C. Bik and H.A.G. Wijshoff, "Compilation Techniques for Sparse Matrix Computations", ACM Int'l Conf. on Supercomputing (Tokyo), pp. 416-424, july 1993.
5. J.E. Boillat, "Load Balancing and Poisson Equation in a Graph", Concurrency, Practice and Experience, vol. 2, no. 4, pp. 289-313, 1990.
6. E.L. Boyd, J.D. Wellman, S.G. Abraham and E.S. Davidson, "Evaluating the Communication Performance of MPPs Using Synthetic Sparse Matrix Multiplication Workloads", ACM Int'l Conf. on Supercomputing, (Tokyo), July 1993.
7. C.M. Chen, S.Y. Lee and Z.H. Cho, "Parallelization of the EM Algorithm for 3-D PET Image Reconstruction", IEEE Trans. on Medical Imaging, vol.10, no.4, pp.513-522, 1991.
8. Y.C. Chung and S. Ranka, "Mapping Finite Element Graphs on Hypercubes", J. of Supercomputing, vol. 6, no. 3/4, pp. 257-282, 1992.
9. I.S. Duff, A.M.Erisman, J.K.Reid, "Direct Methods for Sparse Matrices", Clarendon Press, Oxford, 1986.
10. P. Feautrier, "Array expansion". ACM Int'l Conf. on Supercomputing, July, 1988.
11. A. Gupta, V. Kumar, "The Scalability of FFT on Parallel Computers", IEEE Trans. on Parallel and Distr. Systems, vol. 4, no. 8, pp. 922-932, August 1993.
12. M. Gupta, P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers", IEEE Trans. on Parallel and Distr. Systems, vol. 3, no. 2, pp. 179-193, 1992.
13. R.V. Hanxleden, K. Kennedy, J. Saltz, "Value-Based Distributions in Fortran D - A Preliminary Report", Tech. Rep. TR93365-S, CRPC, December 1993.
14. M. Misra, P. Kumar, "Implementation of Sparse Neural Networks on Fixed Size Arrays", in Parallel Algorithms and Architectures for DSP Applications, M.A. Bayoumi (Ed.), pp. 255-279, 1991.
15. M. Misra, D. Nassimi, V.K. Prasanna, "Efficient VLSI implementation of iterative solutions to sparse linear systems", Parallel Computing, vol. 19, no. 5, pp. 525-544, 1993.
16. M. Morjaria, G.J. Makinson, "Unstructured Sparse Matrix Vector Multiplication on the DAP". Super Computers and Parallel Computation. pp.157-166, Clarendon Press, Oxford 1984.
17. D.M. Nicol and J.H. Saltz "An Analysis of Scatter Decomposition", IEEE Trans Comput., vol. 39, no. 11, pp. 1337-1345, November 1990.
18. A.T. Ogielski, W. Aiello, "Sparse Matrix Computations on Parallel Processor Array", SIAM J. Sci. Comput., vol. 14, no. 3, pp. 519-530, 1993
19. R. Ponnusamy, J. Saltz, R. Das, C. Koelbel, A. Choudhary. Embedding Data Mappers with Distributed Memory Machine Compilers. ACM SIGPLAN Notices, vol.28, no.1, pp.52-55, 1993.
20. P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes", IEEE Trans Comput., vol. 36, no. 12, pp. 1408-1424, 1987.
21. J. Saltz, R. Das, R. Ponnusamy, D. Mavriplis, H. Berryman and J. Wu. PARTI Procedures for Realistic Loops. 6th Int'l Conf. on Distributed Memory Computing, 1991.
22. D.W. Walker, "Characterizing the Parallel Performance of a Large-Scale, Particle-in-Cell Plasma Simulation Code", Concurrency, Pract. and Exper., vol.2, no.4, pp.257-288, 1990.
23. "High Performance Fortran Language Specification". Version 1.0, Technical Report TR92-225, CRPS, Rice University, May 1993.

This Page Intentionally Left Blank

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

Parallel Image Compression with JPEG for Multimedia Applications

Guido Falkemeier and Gerhard R. Joubert [a]

[a] Department for Computer Science,Technical University of Clausthal,
Leibnizstraße 19, 38678 Clausthal, Germany

1. Introduction

Modern high speed computers make it possible to solve large and complex problems. Due to the continuing increase in compute speeds the complexity and size of problems which can be treated increases. This trend will persist for the foreseeable future. The result is that the implemented application software and its use becomes increasingly more complex. Often large and complex sets of input data and parameters must be prepared for each run. Subsequently the results must be interpreted and evaluated. Data preparation and result interpretation often take weeks or even months.

Techniques, such as graphical user interfaces and data visualisation methods, can greatly facilitate user interaction with such complex information processing systems. Interactive systems allow users to assess intermediate results and, if need be, to adjust parameters while a system is running. A disadvantage of such systems is that they usually have considerably higher processing demands than similar, but non-interactive, batch programs. In spite of this their use can in many instances be justified as they often reduce the overall time needed to obtain solutions. The advantages offered by interactive systems are especially important in the case of time constrained problems, such as control systems in complex manufacturing environments or in medical applications.

One of the main difficulties with the development of interactive systems is the definition and realisation of suitable human-computer interfaces. It has become common practice to use graphics and colours in addition to text and numbers to support user communication with complex systems. Such interfaces commonly use static representations of text, numbers, graphics and images. They are not well suited to support user interaction with systems simulating dynamic situations.

The ideal is to develop interactive systems allowing user interaction more closely resembling that of humans with the real world. In order to develop such systems additional modes of communication between users and computer systems must be exploited. Thus, for example, animated graphics and moving images or video clips can in many instances improve the interaction with systems and the interpretation of results.

The primary aim with multimedia information processing is to achieve improved human-computer communication through the use of a wider spectrum of media than is the case with traditional systems. Multimedia systems integrate the processing of time independent media, such as text, numbers, graphics and still images, with that of time dependent media, such as sound and animations. Using the inherent facilities supplied with such

systems results can be represented as still or animated 2D- or 3D-graphics, as still images or video-clips with or without sounds.

The processing of images, both still and moving, forms one of the fundamental aspects of multimedia systems. Even with moderate resolution requirements the volumes of data to be processed are huge. A standard colour image with 640 x 480 pixels and 24 bits resolution allowing for 256 shades in each colour component requires approximately 1 MByte. A video-clip of 25 images per second (PAL standard) thus needs more than 22 MBytes of storage. Quite apart from the storage requirements this data must also be transferred from a disk or CD-ROM into the processor memory before it can be displayed. In the case of distributed multimedia systems this data must be communicated over the network.

Two possible solutions for this problem exist:

- Storage capacities, processing and communication speeds can be increased. In view of the large storage and speed requirements, the development of sufficiently fast hardware solutions will still take years. The realisation of the required communication infrastructures on a global scale will take even longer.

- The volume of data can be reduced by compression techniques. A large number of image compression methods have been developed or are under investigation.

The main difficulties associated with image compression methods are:

- The compression, and often also the decompression, process is usually very time consuming. This fact usually makes real-time compression and decompression unfeasible, except if specially designed hardware is used.

- Depending on the compression method used, the quality loss in the decompressed image may be unacceptable.

- Some compression methods are sensitive to noise in the original images, such as are for example obtained with medical ultrasound equipment.

- Standardisation of compression methods is essential if these are to be used on a universal scale.

One of the first image compression methods to be standardised is the JPEG (Joint Photographic Experts Group) method. This method was designed primarily for the compression of still images. The speed of compression is not affected by noise in the image. In practical applications compression ratios of approximately 20:1 can be achieved before image quality degrades. The history of the development of the method is described in [3].

Motion-JPEG is used to describe specially designed hardware methods which implement JPEG compression and decompression in real time, i.e. at 25 images per second (PAL) or 30 images per second (NTSC).

A disadvantage of the JPEG algorithm is the effort required to compute the compressed and decompressed code.

In this paper a parallel variant of the JPEG-algorithm is proposed. Performance results obtained by using PVM (Parallel Virtual Machine) on a network of workstations are given.

It is shown that, by using parallel methods, Motion-JPEG can be achieved on standard off-the-shelf processors. This obviates the need for the development of special hardware and can thus reduce costs.

An important aspect of Parallel-JPEG is that images compressed with this method can be decompressed on single processor systems. Thus the compression of images can be executed on a parallel system with retention of the full portability of the compressed images accross different platforms.

A brief introduction to the JPEG standard is given and different approaches to parallelise the algorithm are outlined. Subsequently the test platform and the rules for the Parallel-JPEG are described. In the final section results obtained are shown and commented.

2. The JPEG algorithm

The following description of the JPEG-method gives a brief introduction to the algorithm. Readers interested in more detailed information are referred to [1], [2] and [6]. Figure 1 gives an overview of the encoding process.

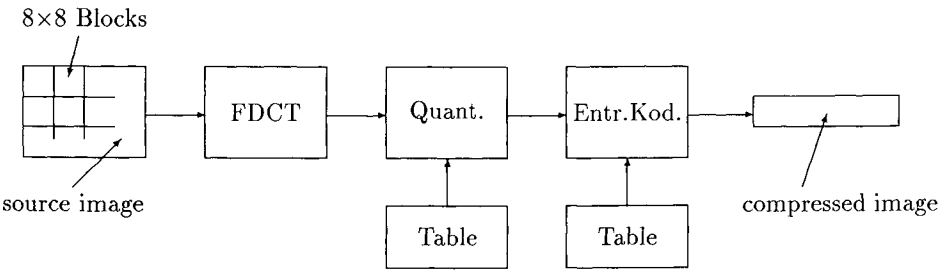


Figure 1. Encoding process

At the start of the compression the source image is grouped into 8×8 pixel (picture element) blocks. The resulting 64 pixels are called a search range. These are mathematically described relative to the characteristic of the pixel in the top left hand corner of the particular search range considered. The integer pixel values lie in the range 0 - 255 if a resolution of 8 bits per pixel is used. In general, if p bits are used, the pixel values fall in the range of $[0, 2^p - 1]$. These values are shifted to signed integers within the range $[-2^{p-1}, 2^{p-1} - 1]$.

These shifted values form the input for the forward discrete cosine transform (Forward-DCT or FDCT). The transformation is executed separately for each block or search range.

The FDCT is given by the following formula:

$$F(u,v) = \frac{C(u)C(v)}{4}\left[\sum_{m=0}^{7}\sum_{n=0}^{7} f(m,n) * \cos\frac{(2m+1)u\pi}{16}\cos\frac{(2n+1)v\pi}{16}\right] \quad (1)$$

where $u = 0, ..., 7$, $v = 0, ..., 7$ *and*
$C(u), C(v) = \frac{1}{\sqrt{2}}$ *for* $u, v = 0$, $C(u), C(v) = 1$ *otherwise.*

After the calculation of the FDCT, all 64 DCT coefficients are quantized in conjuction with a 64-element Quantization Table $F^Q(u,v)$ $(u,v = 0..7)$. The table must be specified before the start of the compression. If 8 bits were used to represent each pixel in the source image each element of the table must be in the range of 0 to 255. The quantization is given by dividing each DCT coefficient by the corresponding element of the table and rounding to the next integer:

$$F^Q(u,v) = IntegerRound\left(\frac{F(u,v)}{Q(u,v)}\right) \tag{2}$$

where $u,v = 0,...,7$

In order to reconstruct the compressed image the inverse discrete cosine transform (Inverse-DCT or IDCT) must be used:

$$f(m,n) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v) * \cos\frac{(2m+1)u\pi}{16}\cos\frac{(2n+1)v\pi}{16}\right] \tag{3}$$

where $m = 0,...,7$, $n = 0,...,7$ *and*
$C(u), C(v) = \frac{1}{\sqrt{2}}$ *for* $u,v = 0$, $C(u), C(v) = 1$ *otherwise.*

The dequantization is given by the inverse function:

$$F^{Q'}(u,v) = F^Q(u,v) * Q(u,v) \tag{4}$$

where $u,v = 0,...,7$

After the quantization the elements of the block are rearranged in zig-zag sequence (see figure 2). Every square in the figure is one integer of the particular search range or block considered. The element in the top left corner of the search range is called the DC- (or direct) component. All other elements are the so-called AC- (or alternating) components.

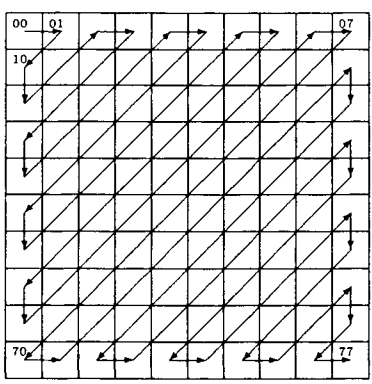


Figure 2. Zig-Zag sequence

The last part of the encoding process is the entropy coding. The JPEG method specifies two methods which can be used: the Huffman coding and the arithmetic coding method. The arithmetic coding method is patented and is hardly used in practice. In this paper the Huffman coding method is used.

An understanding of the encoding used for the DC- and AC-components respectively is essential for the parallelisation of the JPEG algorithm. Note that the DC- and AC-components are encoded differently.

2.1. DC-Components

The DC-Components are coded using a difference method. This means that not the explicit value of each component is stored, but rather the difference to the DC-component of the previous search range. The DC-component of the first search range is encoded by assigning it the difference to zero. Mathematically this can be described as follows, where $PRED$ is the value of the DC-component of the previous search range and $ZZ(0)$ the value of this component in the search range considered:

$$DIFF = ZZ(0) - PRED \tag{5}$$

In order to encode the calculated differences the following method, which generates two different values, is used:

The first gives the number of bits needed to store the calculated difference. The calculated value is not stored as such, but is encoded using a Huffman table. The table can, for example, be as follows (see [1]):

Table 1
Huffman table for DC-components, Value 1

Type	Code length	Code
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Type indicates the number of bits to be stored. If, for example, eight bits are needed to store a particular difference, the code according to the table is 111110.

The second value contains the calculated difference, which is stored in a form derived from Table 2. The first column gives the number of bits used and the second the possible

differences which can be represented by these. The length of the table is determined by the size of the DCT. In the case of an 8×8 DCT it is shown that with input values from the interval $[-2^7, 2^7 - 1]$ the output will not exceed $[-2^{10}, 2^{10} - 1]$.

Table 2
Number of bits needed to represent DC-difference

Number of bits	Size of the difference
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023
11	-2047..-1024,1024..2047

The difference value zero is encoded by the bit string 00. Positive and negative values are indicated as follows:

- If the value is positive, the difference is stored as is, and
- if the value is negative one is subtracted from the value of the difference.

The result is that, in the case of a negative difference the encoded value has a leading zero and in the case of a positive value the leading bit is one.

Consider the example of a search range which is the first block of an image considered: $PRED$ is equal to zero, $ZZ(0) = -22$ and the encoded value is -22. Five bits are needed to encode this difference. The sequence of bits generated consist of the Huffman code of Type 5 (the first 5 bits) and the value for -22 (the second five bits):

11110 01001

2.2. AC-components

In order to encode the AC-components again two values are used. The first value consists of two numbers: RUNLENGHT and SIZE, each of which is stored in four bits. RUNLENGTH gives the number of leading zero's before the first non-zero element in the sequence of AC-components (zig-zag ordering). If this number is greater than 15, the value (15,0) is encoded. Thus a sequence of 16 zero's is encoded. If all AC-components up to the end of the block are zero the special value (0,0) for 'end of Block (EOB)' is used.

The variable SIZE indicates the number of bits needed to store the next non-zero AC-component value. The different values of SIZE and the associated amplitudes of the

AC-components are given in Table 3. The length of the table is again not arbitrary. The amplitudes can only lie in the interval $[-2^{10}, 2^{10} - 1]$. This is, as in the case of the DC-components, derived from the input values of the DCT ($[-2^7, 2^{-7} - 1]$) and the possible output values (maximum increase by three bits).

The second value used to encode the AC-components is analogous to the second value used in the case of the DC-components. The only difference is that not the differences, but the values of the amplitudes are used (see Table 3).

Table 3
SIZE and the associated amplitudes

SIZE	Amplitude
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023

Consider, for example, an image in which the first five AC-components are:

-11	-6	-4	-18	-16

In this case the following values of RUNLENGHT and SIZE are obtained:

(0,4) ; (0,3) ; (0,3) ; (0,5) ; (0,5)

The pairs of numbers are encoded using a Huffman table which, in view of its length, is not given. A table which can be used is given in [1].

In order to show the beginning of the bit stream generated in the case of the example, the code for the number pairs (0,3), (0,4) and (0,5) are given:

(0,3):100
(0,4):1011
(0,5):11010

Thus the following code is generated, where the code of the components is separated by the symbol '|'. The bits before the space give the Huffman code for the particular number pair, the following bits the encoding of the amplitude of the component.

values: (0,4) -11 | (0,3) -6 | (0,3) -4 | (0,5) -18 | (0,5) -16
encoded bits: 1011 0100 | 100 001 | 100 011 | 11010 01101 | 11010 01111

In order to decode the image the encoding process is executed in reverse order (see Figure 3). In order to be able to decode the image all information about the image must be stored in the header of the compressed image.

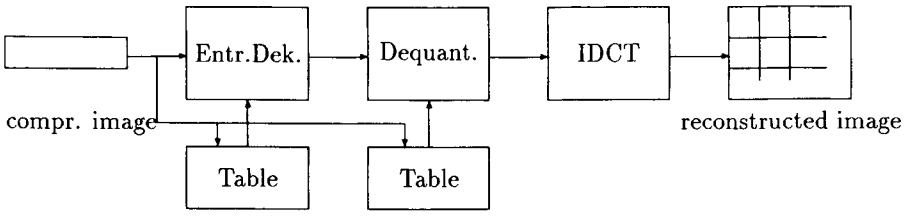


Figure 3. Decoding process

3. Approaches to parallelise the algorithm

The JPEG-Algorithm can be parallelised in three different ways:

- parallelise the algorithm,
- divide the image in blocks,
- divide the image in strips.

All these methods have advantages and disadvantages.

3.1. Parallelising the algorithm

The first method will be described for the case of a JPEG-encoder. The encoding process for an 8×8 search range can be divided into four steps:

1. Shift the source image pixel values of the range $[0, 2^p - 1]$ into signed integers of the range $[-2^{p-1}, 2^{p-1} - 1]$,
2. Calculate the FDCT,
3. Arrange the samples in zig-zag order (see figure 2),
4. Entropy-coding.

The processing of the steps is pipelined in that each part, from the second onwards, uses the results of the previous step as inputs. Separate programs can be written for the different steps. These form the constituent processes of the pipeline which can execute the parallel JPEG-compression for a complete image (see figure 4).

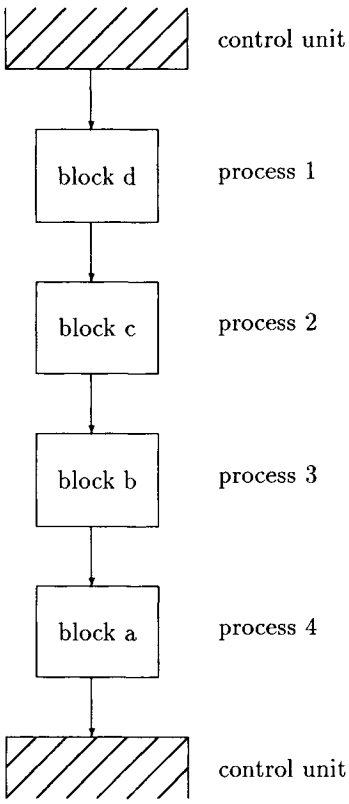


Figure 4. Processes in the Pipeline

At the beginning and the end of the pipeline control units are needed. The first unit inputs a new search range into the pipeline. When the first program has finished its calculation, it passes the data on to the second, and can start processing the next search range.

The results of each process are passed down the pipeline. Finally a control unit writes the coded (compressed) data to a file.

This approach has two major disadvantages:

- The effort required to manage the pipeline is very high compared to the total processing effort. It must be checked when the last process has completed processing each search range in order that the data can be stored and a new search range can be fed into the pipeline.

- Only four search ranges can be calculated simultaneously.

One possibility tu reduce this problems is to start two or more parallel pipelines. This, however, increases the relative amount of time spent on controlling the activities in the pipes. The reason for this is that the computation times for different search ranges are not fixed. Thus a buffering mechanism must be provided at the control unit end of the pipes. This requires an additional control unit to manage the buffers.

The time wasted in controlling the pipeline(s) reduces the potential advantages offered by the parallelisation of the algorithm.

3.2. Subdividing the image

The next two methods are based on data parallelism, in that the image is subdivided into a number of smaller images. These are compressed in parallel by different processors. After compressing the image sections the code is combined and the compressed image is obtained.

3.2.1. Subdividing into blocks

The first approach at data parallelism is to divide the image into blocks (see figure 5).

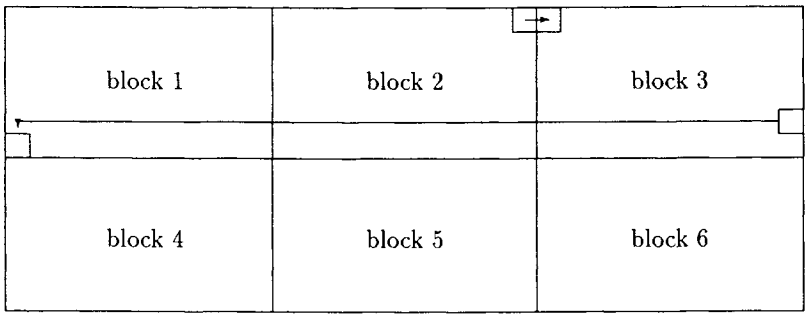


Figure 5. Dividing a source image into blocks

This seems to be a good way to subdivide the image as it is very easy to manage. Each section of the image can be compressed by using the normal JPEG algorithm directly. A natural way is to divide the image such that one processor can be used for compressing each sub-image. This should give good performance as the number of processes can be made to match the number of processors (within limits of course).

Although this seems to be a good approach, it soon appears that the JPEG-algorithm cannot be applied to the sub-images created. The reason for this is the chained calculation of the DC-components. If the source image is divided into sub-images which are processed in parallel, the calculation of the DC-components of the leftmost 8×8 search ranges of each sub-image is impossible as the required values of the preceeding DC-components are not available.

3.2.2. Subdividing into strips

An alternate approach is to subdivide the image into strips (see figure 6). As is the case with the block method described in 3.2.1 the potential speed-up which can be attained will be maximised if the number of strips are chosen to match or be a multiple of the number of processors available.

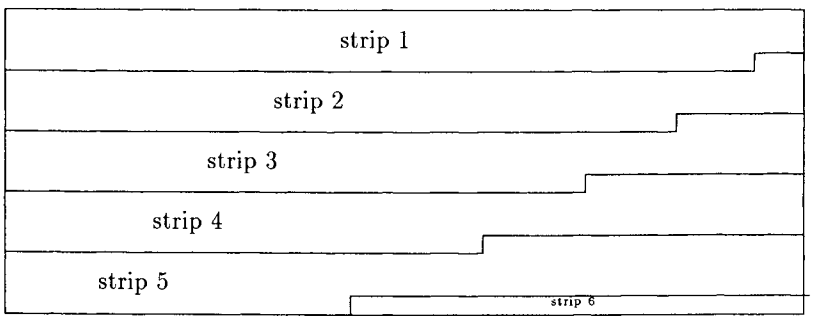


Figure 6. Dividing the image into strips

By using this method, the problem with the difference coded DC-component arises only for the first 8×8 search image of each strip. The remaining search images in each strip can be encoded in the normal way. To solve this problem for the first search image of each strip a feature of the JPEG-standard can be used.

In the standard so-called *Restart-Markers* [1] are defined. These markers are normally used for error-correction. When a Restart-Marker is encountered in the compressed code, the value for the difference coding of the DC-components is reset. This means that the difference is calculated relative to the value zero. This feature can be used for marking the beginning of a new strip.

The only disadvantage of this method is that the number of strips to be used must be decided at the beginning of the encoding process. As was mentioned before this number should be equal to or be a multiple of the number of available processors in order to obtain the best performance. If the compressed image is later to be processed on a parallel system on which the number of strips do not match the number of processors, performance will not be optimal.

An advantage of the method is that the JPEG-Standard is not changed and the compressed image can equally well be decompressed on any single processor system or any system using specially designed chips. A requirement is, of course, that specially designed chips follow the standard.

Of the three possibilities for parallelisation, this latter approach appears to offer the greatest advantages. This method is to be further investigated in the following sections.

4. Parallel JPEG format

For the implementation of the method described in 3.2.2 some rules for the *parallel JPEG format* to be used must be formulated.

Every encoded JPEG-image is stored with a header which contains the parameters needed for decoding the image. In order for a JPEG-image to be processed in parallel, the header of the parallel format of the file must contain the following application segment:

FF	EF	00	0A	50	41	52	4A	50	45	47	ns

Figure 7. The application segment of the Parallel-JPEG file format

The fields of the application segment have the following meaning:

- The marker for the application segment is stored in the first two bytes;
- The next two bytes give the length of the segment;
- The following seven bytes give the string 'PARJPEG', in hex '50 41 52 4A 50 45 47';
- The last byte denoted by *ns* gives the number of strips in the compressed image.

Using one byte for *ns* allows for a maximum of 256 strips. For the purpose of the images considered in this paper this number is large enough. Even 256 processors cannot be used efficiently for compressing images which can be displayed on the presently available display screens. If too many processors are used the time needed for communication and control will surpass the total computation time. This break even point is also hardware dependent and must be established for each platform separately. In the special case of very large images where a larger number of processors can be used efficiently, two or even more bytes may be reserved for *ns*.

In addition to the application segment, the header of a Parallel-JPEG file must also contain a segment giving the restart markers (see [1]).

Finally the restart markers must be inserted at the correct positions in the compressed image.

5. Test environment

The implementation and performance tests were done on a network of workstations. The main reason for this choice of platform is the conviction that the resources available to users of heterogeneous networks will in future increasingly be used for parallel computing purposes. Advantages offered by this form of parallel computing are, for example:

- Application software running on one station can use additional resources available on the network for compute intensive tasks. As an example consider a multimedia system running on a PC linked to a network. A PC can usually cope with the major portion of the processing involved with multimedia applications, but would be inadequate for the real-time compression of image data. The PC should be able to call on a number of more powerful workstations connected to the network to use parallel compression techniques for the real-time execution of this task.

- Standard software development environments on PC's and workstations are often superior to those available on parallel computers. It is thus often easier to develop application software for a workstation environment, and later port it to a parallel machine.

- It is relatively easy to adapt software developed to run on one workstation to run in parallel on a number of similar workstations in a network. This effort is less than the effort involved in porting the software to a parallel machine with its own and often unique architecture.

- Manufacturers of workstations have a large pool of existing software to protect. Thus they usually are very concerned to maintain upwards compatibility of applications software to new releases of hardware and/or system software. Parallel software developed for a network of workstations benefit from this. Slower stations can be replaced by faster machines usually without any changes needed to the application software. Thus parallel software running on a network of workstations can immediately utilise the higher performance of newer hardware.

In order to implement and test the Parallel-JPEG algorithm PVM *Parallel Virtual Machine* was used. The heteregenous network used comprised three DECstation 5000/125 (25 Mips, 3,7 Megaflops) and one DEC 3000 with alpha processor (4 times faster than the DECstation). The computers were connected with ethernet.

The Parallel-JPEG program was written to process grey scale images. The method can easily be extended to also handle colour images. A master/slave paradigm was used, i.e. the program consists of a master module which can generate many slave processes.

The master sends one strip of image data to each slave. The slave processes this data and returns the results to the master. The master then assembles the various pieces of data.

The main advantage of this approach is that the whole process can easily be controlled. A major disadvantage is that all data communication is between the master and the slaves. This reduces perfomance.

6. Testing results

In order to test the performance of the algorithm an image with 1200×960 pixels was used.

To measure speed-up it is immaterial whether compression or decompression is considered, as both require approximately the same amount of processing. In practice real time decompression is usually more important as this allows video clips to be played at normal

speed. Here measurements obtained for decompression with 1,2,3 and 4 computers are given.

The image was divided in five different ways, with 1,2,3,4 and 20 strips.

The results are given in table 1. The times given are averages of up to 10 runs using the same configuration.

Table 4
Testing results in seconds

No. strips	1 CPU	2 CPU	Speed-up	3 CPU	Speed-up	4 CPU	Speed-up
1	24.4						
2	24.4	15.6	1.56				
3	25.0	18.5	1.35	12.4	2.01		
4	25.2	15.5	1.62	15.5	1.62	11.1	2.27
20	32.4	22.1	1.47	16.4	1.97	13.3	2.43

Consider the results for a single processor given in the second column (1 CPU). Processing time increases with the number of strips used. The reason for this is that more and more processes must be started, one for each strip. Starting up additional processes and their administration costs time, which in the single processor case is not compensated by the use of additional processors.

The 2 CPU column gives the times for two connected computers, i.e. a parallel computer with two processors. As can be seen the calculation is executed much faster than on one computer. An anomaly in these figures is that the computation for the image with three strips takes longer than for the image with two strips. The reason for this lies in the imbalance in the scheduling of the three processes to run on two processors. The result is that for half of the elapsed time one processor is idling. Furthermore, the two strips executed in sequence represent two thirds of the pixels to be processed.

Due to the time wasted in starting up and managing the processes the processing time for an image divided into 20 strips is the longest. Due to the two processors used the elapsed time is, however, shorter than that required to process one strip, i.e. the whole image, on a single processor.

The results obtained with three processors are given in the 3 CPU column. Similar comments apply as in the case of two parallel processors.

In the last column results obtained with four processors are given. Here the highest performance of all configurations is achieved.

The speed-ups achieved are shown in figures 8 and 9. Even with the limitations of PVM and the relatively slow communication speeds of the network used, an appreciable speed-up is obtained. It is reasonable to expect that these will be much better on a parallel machine offering fast communication between processors.

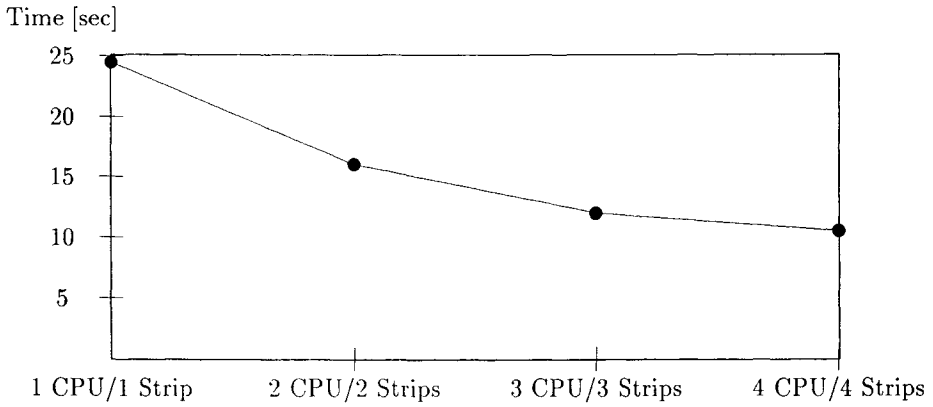


Figure 8: Performance for optimal balance of strips and CPU's

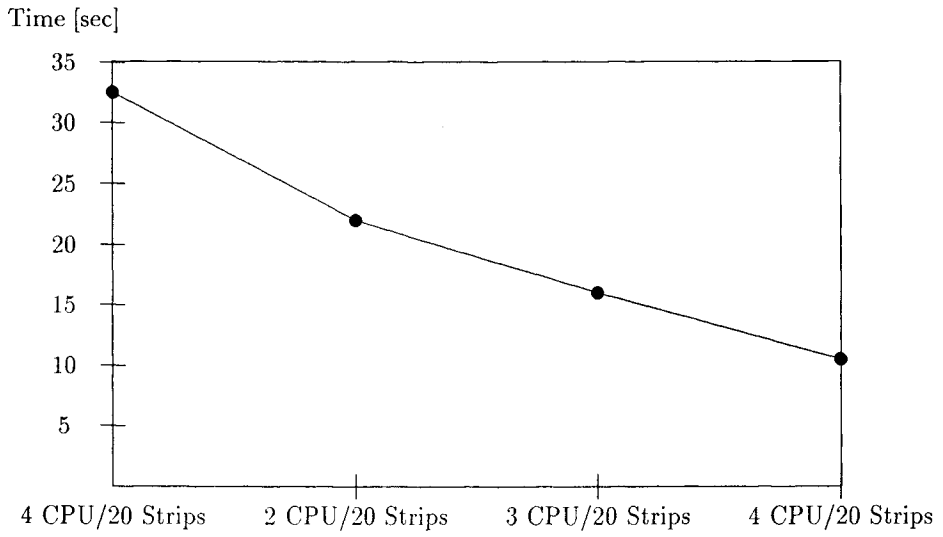


Figure 9: Performance for 20 strips and different numbers of CPU's

7. Conclusions

The tests show that the proposed parallel approach can reduce processing times for image compression and decompression according to the JPEG standard. These results were obtained using PVM on a standard network of workstations. Apart from the relatively slow communication speeds, results are also negatively affected by the way PVM handles the communication of data between processes.

Using a parallel computer system which allows for much higher communication speeds between processors, the speed-up of the Parallel-JPEG method described here can be expected to be significantly better. On the other hand a network of workstations such

as was considered here offers great flexibility and easy upgradability to newer and faster processors.

The only disadvantage of the parallel format of JPEG described, is that information about the number of strips used - and parallel processes created - must be supplied in the compressed image. This is not a disadvantage for decoding the compressed images, as any normal JPEG encoder/decoder can read the Parallel-JPEG format. The reason for this is that this format still conforms to the standard. Any standard software JPEG encoder/decoder could easily be adapted to also write the parallel format.

The approach can equally well be extended to apply to colour images.

REFERENCES

1. Digital Compression and Coding of Continous-Tone Still Images, Part 1, Requirements and Guidelines. ISO/IEC JTC1 Committee Draft 10918-1, Feb. 1992.
2. Digital Compression and Coding of Continous-Tone Still Images, Part 2, Compliance Testing. ISO/IEC JTC1 Committee Draft 10918-2, March 1993.
3. Hudson, G., et. al.,The International Standardisation of a Still picture Compression Technique, Globecom '88.
4. Huffman, D.A., A method for the construction of minimum redundancy codes, Proceedings IRE, vol. 40, 1962, pp 1098-1101.
5. Wallace, G.K., Overview of the JPEG (ISO/CCITT) still image compression standard. Image Processing Algorithms and Techniques, Proceeding of the SPIE, vol. 1244 (Feb 1990), pp. 220-233.
6. Wallace, G.K., The JPEG Still Picture Compression Standard, Communication of the ACM, April 1991, Vol.34, No.4.

High Performance Computing: Technology, Methods and Applications
J.J. Dongarra, L. Grandinetti, G.R. Joubert and J. Kowalik (Editors)
© 1995 Elsevier Science B.V. All rights reserved.

High Performance Computational Chemistry; NWChem and Fully Distributed Parallel Applications

M. F. Guest, E. Aprà, D. E. Bernholdt, H. A. Früchtl, R. J. Harrison, R. A. Kendall, R. A. Kutteh, X. Long, J. B. Nicholas, J. A. Nichols, H. L. Taylor, and A. T. Wong
High Performance Computational Chemistry Group

and

G. I. Fann, R. J. Littlefield and J. Nieplocha
High Performance Computing Software Support

Environmental Molecular Sciences Laboratory
Pacific Northwest Laboratory
PO Box 999, Mail Stop K1-96
Richland, WA. 99352

Abstract

An account is given of work in progress within the High Performance Computational Chemistry Group (HPCC) at the Pacific Northwest Laboratory (PNL) to develop the molecular modeling software application, NWChem, for massively parallel processors (MPPs). A discussion of the issues in developing scalable parallel algorithms is presented, with a particular focus on the distribution, as opposed to the replication, of key data structures. Replication of large data structures limits the maximum calculation size by imposing a low ratio of processors to memory. Only applications that distribute both data and computation across processors are truly scalable. The use of shared data structures, which may be independently accessed by each process even in a distributed-memory environment, greatly simplifies development and provides a significant performance enhancement. In describing tools to support this programming paradigm, an outline is given of the implementation and performance of a variety of the modules comprising NWChem; particular focus is given to a highly efficient and scalable algorithm to perform quadratically convergent, self-consistent field calculations on molecular systems. A brief account is also given of the development of corresponding MPP capabilities in the areas of periodic Hartree Fock, Möller-Plesset perturbation theory (MP2), density functional theory, and molecular dynamics. Performance figures are presented using both the Intel Touchstone Delta and Kendall Square Research KSR-2 supercomputers.

I. Introduction

The mission of the HPCC group at PNL is to develop molecular modeling software applications that provide 10-100 times more computing capability than has been available with conventional supercomputers. While increases in raw computing power alone will greatly expand the range of problems that can be treated by theoretical chemistry methods, a significant investment in new algorithms is needed to fully exploit the potential of Massively Parallel Processors (MPPs). Merely porting presently available software to these parallel computers does not provide the efficiency required to exploit their full potential. Most existing parallel applications show a significant deterioration in performance as greater numbers of processors are used. In some cases, the efficiency is so poor that the use of additional processors decreases, rather than increases, the performance. Thus, new algorithms must be developed that exhibit *parallel scalability* (i.e., show a near linear increase in performance with the number of processors). Although perfect scalability is very difficult to achieve, we have demonstrated our ability to approach this level of performance with our new self-consistent field (SCF) codes (see below).

Another important consideration in using MPPs is how data are stored. So-called *replicated-data* schemes require that a copy of each data item in the program be stored on each processor, so that the size of the problem that can be handled is limited by the memory of a single processor. In *distributed-data* applications each processor holds only a part of the total data; in such cases, the problem size is limited only by the total memory of the machine, allowing much larger problems to be treated. Our efforts focus on distributed-data applications. They span virtually the whole range of computational chemistry methods, with significant efforts in Hartree-Fock and correlated techniques, multi-configuration self-consistent field methods, density functional theory, semi-empirical methods, and classical Monte Carlo and molecular dynamics.

In addition to the creation of new algorithms for computational chemistry on parallel processors, we are also creating the high-level data and control structures needed to make parallel programs easier to write, maintain, and extend. We have developed portable tools for memory allocation, message passing, distributed global arrays, performance analysis, decomposition strategies, prototyping of new kernels, refinement of performance models, code architecture determination, and teraflops projections. Those tools crucial to our parallel chemistry applications are considered below in Section II.

Our applications feature major design improvements over traditional codes, are modular at all levels, and have culminated in the **N**orth**W**est **Chem**istry (NWChem) package; we consider issues central to the design, implementation and performance of many of the component modules of NWChem in Section III below.

In concert with the PNL Extensible Computational Chemistry Environment effort, others associated with our project are also creating graphical user interfaces and molecular visualization modules, and are linking all of our computational codes to an object-oriented database. This allows for easy and accurate input, visualization, storage, analysis, and retrieval of molecular geometries and properties.

II. Software Development Tools

We describe below those tools crucial to our parallel chemistry applications, providing an outline of the role, status and capabilities of the TCGMSG message passing toolkit in Section II.1, and corresponding attributes of the tools for memory allocation and distributed global arrays in Sections II.2 and II.3 respectively. The central role and development of parallel linear algebra routines is considered in Section II.4, while Section II.5 introduces the concept of the runtime database, a repository of parameters and information for all subsequent application modules.

II.1 Message Passing

TCGMSG [1] is a toolkit for writing portable parallel programs using a message passing model. Supported are a variety of common UNIX workstations, mini-super and super computers and heterogeneous networks of the same, along with true parallel computers such as the Intel iPSC, Delta and Paragon, the Kendall Square Research KSR-1/2, SGI Power Challenge, the IBM SP1/2 and Cray T3D. Applications port between all of these environments without modification to the parallel constructs. TCGMSG has distinguished itself from other toolkits in its simplicity, high performance and robustness. The limited functionality provided includes point-to-point communication, global operations and a simple load-balancing facility, all designed with chemical applications in mind. This toolkit is available in the public domain (from ftp.tcg.anl.gov), and is distributed with a set of example chemical applications.

The TCGMSG programming model and interface is directly modeled after the Argonne PARMACS [2]. The message-passing interface (MPI) standard [3] will soon obsolete the TCGMSG interface. However, TCGMSG has superior robustness and performance when compared with current portable MPI implementations. In addition, the simple TCGMSG interface is readily implemented using MPI.

II.2 Memory Allocation

One of the key problems on high performance parallel supercomputers is managing the memory utilization on each node. Many computational chemistry and other FORTRAN based applications use statically allocated arrays. These are either multiple arrays with hard wired dimensions that commonly require recompilation to change the problem size or a static array that is used as a simple

stack throughout the entire application. The Memory Allocator (MA) is a library of routines that comprises a dynamic memory allocator for use by C, FORTRAN, or mixed-language applications. C applications can benefit from using MA instead of the ordinary malloc() and free() routines because of the extra features MA provides: both heap and stack memory management disciplines, debugging and verification support (e.g., detecting memory leaks), usage statistics, and quantitative memory availability information. FORTRAN applications can take advantage of the same features, and may in fact require a library such as MA because dynamic memory allocation is not supported by all implementations of the language. The coexistence of heap and stack operations can benefit application design by tuning the memory allocation based on the specific application requirements. Once FORTRAN-90 is widely available, library tools such as MA may not be required but it is currently essential for memory management in complex applications.

MA is designed to be portable across a variety of platforms and is currently supported on nearly a dozen different workstation and high performance parallel computers. The current implementation utilizes a segment of memory that is obtained from the OS upon initialization. The low end of the segment is managed as a heap and the high end of the segment as a stack. The C language interface utilizes standard C include mechanisms and the FORTRAN interface needs only a simple "include file" preprocessing system such as that provided by cpp. The allocated memory segment of each block can explicitly be typed as any formal type available in FORTRAN or C.

II.3 Global Arrays

No emerging standards for parallel programming languages (notably just High Performance FORTRAN (HPF-1)) provide extensive support for MIMD programming [4]. The only truly portable multiple instruction multiple data (MIMD) programming model is message passing, for which a standard interface has been recently proposed [5]. It is, however, very hard to develop applications with fully distributed data structures using the message-passing model [6, 7]. The shared-memory programming model offers increased flexibility and programming ease but is less portable and provides less control over the inter processor transfer cost. What is needed is support for one-sided asynchronous access to data structures (here limited to one- and two-dimensional arrays) in the spirit of shared memory. With some effort this can be done portably [8]; in return for this investment, we gain a much easier programming environment, speeding code development and improving extensibility and maintainability. We also gain a significant performance enhancement from increased asynchrony of execution of processes [9]. Message passing forces processes to cooperate (e.g., by responding to requests for a particular datum). Inevitably, this involves waiting for a collaborating process to reach the same point in the algorithm, which is only partially reduced by the use of complex buffering and asynchronous communication strategies. With a one-sided communication mechanism, where each process can access what it needs without explicit participation of another

process, all processes can operate independently. This approach eliminates unnecessary synchronization and naturally leads to interleaving of computation and communication. Most programs contain multiple algorithms, some of which may naturally be task-parallel (e.g., Fock matrix construction), and others that may be efficiently and compactly expressed as data-parallel operations (e.g., evaluating the trace of a matrix product). Both types of parallelism must be efficiently supported. Consideration of the requirements of the self-consistent field algorithm discussed later in this chapter, and also the parallel COLUMBUS configuration interaction program [10], second order Møller-Plesset perturbation theory [11] and parallel coupled-cluster methods [12] led to the design and implementation of the Global Array toolkit [8] to support one-sided asynchronous access to globally-addressable distributed one- and two-dimensional arrays.

II.3.1 Global Array Toolkit

The Global Array (GA) toolkit provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed matrices, without need for explicit cooperation by other processes. Unlike other shared-memory environments, the GA model exposes the programmer to the Non-Uniform Memory Access (NUMA) timing characteristics of the parallel computers and acknowledges that access to remote data is slower than to local data. From the user perspective, a global array can be used as if it were stored in shared memory, except that explicit library calls are required to access it. Details of actual data distribution and addressing are encapsulated in the global array objects. Matrices are physically distributed blockwise, either regularly or as the Cartesian product of irregular distributions on each axis. The information on the actual data distribution can be obtained and exploited whenever data locality is important. Each process is assumed to have fast access to some "local" portion of each distributed matrix, and slower access to the remaining "remote" portion. A very efficient, direct access to the local data is supported. Remote data can be accessed through operations like "get", "put" or "accumulate" (floating point sum-reduction) that involve copying of the globally accessible data to/from process-private buffer space. Processes can communicate with each other by creating and accessing GA distributed matrices, and also (if desired) by conventional message-passing. Each GA operation can be categorized as a primitive operation, constructed from primitive operations, or providing access to third party software. Operations also differ in their implied synchronization and atomicity. Elementary operations, with architecture-specific implementation but a portable user interface, that are invoked simultaneously by all processes, include capabilities to:

- create an array, controlling alignment and distribution;
- create an array following a provided template (an existing array);
- destroy an array; and
- synchronize all processes.

A set of primitive operations may be invoked in true MIMD style by any process with no implied synchronization with other processes and, unless otherwise stated, with no guaranteed atomicity. These include:

- fetch, store and atomic accumulate into a rectangular patch of a two-dimensional array;
- gather and scatter array elements;
- atomic read and increment of an array element;
- inquire about the location and distribution of the data; and
- direct access to local elements of an array to support and/or improve performance of application specific data-parallel operations.

A set of BLAS-like data-parallel operations have been developed on top of the primitive operations (synchronization is included as a user convenience). These include:

- vector operations (e.g., dot-product or scale) optimized to avoid communication by direct access to local data;
- matrix operations (e.g., symmetrize, transpose or multiplication) optimized to reduce communication and data copying by direct access to local data.

The vector, matrix multiplication, copy, and print operations exist in two versions that operate on either entire array(s) or specified sections of array(s). The array sections in operations that involve multiple arrays do not have to be conforming - the only requirements are that they must be of the same type and contain the same number of elements. The data-parallel operations must be invoked simultaneously by all processes. Data consistency is only guaranteed for multiple read, or multiple accumulate, or multiple disjoint write operations.

Additional functionality is provided through a variety of third party libraries made available by using the GA primitives to perform the necessary data rearrangement. These include:

- standard and generalized real symmetric eigensolvers; and
- linear equation solver (interface to SCALAPACK).

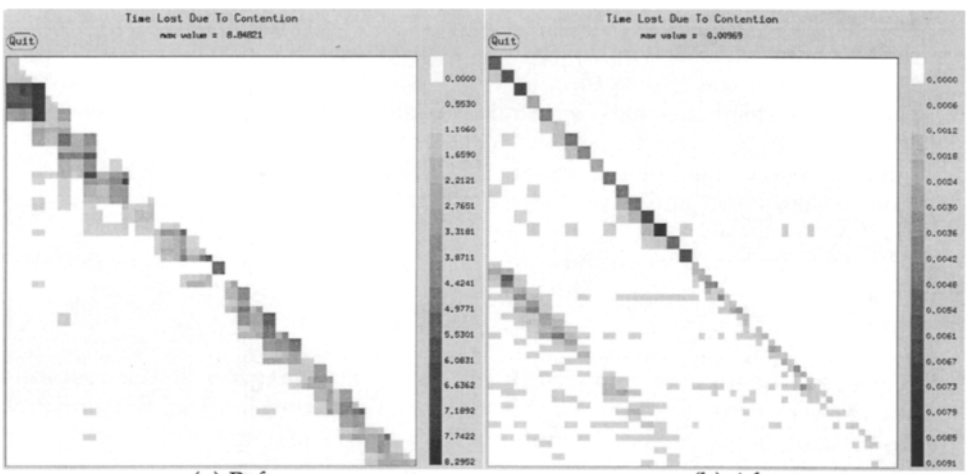
The $O(N^2)$ cost of data rearrangement is observed to be negligible in comparison to that of $O(N^3)$ linear-algebra operations. These libraries may internally use any form of parallelism appropriate to the host computer system, such as cooperative message passing or shared memory:

The Global Array interface has been designed in the light of emerging standards. In particular HPF-1 and HPF-2 will certainly provide the basis for future standards definition of distributed arrays in FORTRAN. A long term goal must be to migrate to full language support, and to eliminate as much as possible the practice of parallel programming through subroutine libraries. The basic

functionality described above (create, fetch, store, accumulate, gather, scatter, data-parallel operations) may be expressed as single statements using FORTRAN-90 array notation and the data-distribution directives of HPF. However, HPF currently precludes the use of such operations on shared data in MIMD parallel code. The Global Array tools have been implemented on the Intel parallel computers, the IBM SP1 (under the EUIH message-passing library), the Cray T3D, UNIX workstation networks, multiprocessor UNIX computers using shared memory, and the Kendall Square Research KSR-2 parallel computer.

II.3.2 Global Array Visualizer

In order to aid in tuning the performance of applications using global arrays, a visualization and animation tool has been developed. This tool helps the programmer design efficient task scheduling strategies for MIMD algorithms that operate on the distributed two-dimensional data. The Global Array Visualizer is used to animate access patterns to sections of 2-D arrays, through trace data that is gathered in a file during the program execution.



(a) Before (b) After

Figure 1. The GA visualization tool as applied to the distributed data self-consistent field method (see Section III.2); (a) significant data contention problem, (b) reordering of tasks, introduction of caching, and incorporation of a stratified randomizing scheme demonstrates both load-balance and almost contention-free data access.

A color coding is used to differentiate levels of access contention for the particular data blocks. After the animation of events recorded in a tracefile is completed, a composite access contention index is displayed using a different color coding for the entire distributed array. The tool was applied to our distributed self-consistent field program (see Section III.2). Our first scheduling of the tasks realized poor parallel efficiency, but the reason was not apparent from simple

timing data. The performance tool showed that significant contention for data was the problem, as shown in Figure 1a.

This problem was readily addressed by reordering the tasks so as to more uniformly spread out references to the GA matrices, and by the introduction of caching (in the application code) to eliminate redundant data references. This eliminated most of the time lost due to contention, but the parallel speedup was still not as good as expected. The dynamic visualization (animation) then showed that some large tasks were being scheduled too near the end of the computation, causing a load-balance problem. This was resolved by incorporating a stratified randomizing scheme that approximately preserved the large-to-small order of tasks, necessary for load-balancing, while still reordering tasks of similar size so as to spread out the GA references and avoid contention. The final scheduling, depicted in Figure 1b, is both load-balanced and almost contention-free. The performance of the main computational kernel of the self-consistent field program was improved approximately four-fold by the above tuning.

II.4 Distributed Matrices and Linear Algebra

Many electronic structure computations are formulated in terms of dense or nearly dense matrices of size roughly N by N, where N is the number of basis functions. Two distinct classes of operations are performed on these matrices:

- random access to small blocks, for the purpose of constructing matrices as a function of many one- and two-electron integrals;

- linear algebra operations on the entire matrix, such as eigensolving, Cholesky decomposition, linear system solution, inversion, and matrix-matrix multiplication.

Both types of operations must work on distributed matrices if the resulting application is to be truly scalable. This imposes a stringent limit on functionality; if even a single linear algebra operation requires that the entire matrix be present on one processor, then the application cannot be scalable.

The limitation on performance is not so severe, however. In many applications, linear algebra operations on large matrices comprise only a small fraction of the total number of floating point operations. For example, in an SCF application, it is common for 80% or more of the arithmetic to be consumed by two-electron integral evaluation, with the remaining 20% or less being spent on diagonalization and other linear algebra operations. This imbalance allows the use of distributed linear algebra routines that have less than perfect scaling, with little impact on the application's overall time-to-solution and scalability. In addition, as applications evolve to work with progressively larger matrices, it becomes increasingly feasible to reorganize the data so as to optimize the performance of expensive linear algebra operations. This is because data

reorganization costs (typified by matrix transpose) require only $O(N^2/P)$ time, while expensive linear algebra operations are typically $O(N^3/P)$.

The development and acquisition of distributed linear algebra routines has been a gradual process, driven by the needs of our applications. Much of our early work used replicated data and conventional sequential linear algebra routines. This approach was adequate with small numbers of processors and relatively slow integral evaluation routines. However, as more processors and faster integrals permitted larger basis sets to be used, linear algebra became a bottleneck for both speed and problem size, and we were forced to convert to fully distributed linear algebra routines.

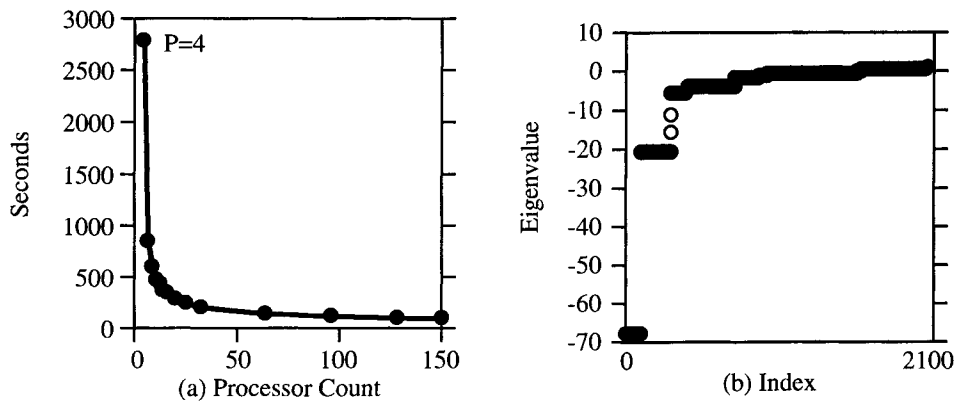


Figure 2. Scalability (a) for varying numbers of processors for the determination of all eigenpairs of a 2053 by 2053 matrix whose spectrum is displayed in (b).

A particular focus of our distributed linear algebra work has been the development of a scalable, fully parallel eigensolver whose numerical properties satisfy the needs of the chemistry applications. This package, called PeIGS, solves dense real symmetric standard (Ax=lx) and generalized (Ax=lBx) eigenproblems. The numerical method used by PeIGS is multisection for eigenvalues [13, 14] and repeated inverse iteration and orthogonalization for eigenvectors [15]. Accuracy and orthogonality are similar to LAPACK's DSPGV and DSPEV [16]. Unlike other parallel inverse iteration eigensolvers in the current literature, PeIGS guarantees orthogonality of eigenvectors even for arbitrarily large clusters that span processors.

PeIGS is both fast and scalable -- on a single processor it is competitive with LAPACK, and parallel efficiency remains high even for large processor counts. For example, in one of our SCF applications, the standard eigenproblem Ax=lx was solved for all eigenpairs of a 2053 by 2053 matrix. This computation required only 110 seconds on 150 processors of an Intel Paragon computer, a time-to-solution estimated as 87 times faster than LAPACK for the same problem on 1

processor (The full 2053 by 2053 problem is too large to fit on a single processor, so the 1-processor time is extrapolated from smaller problems and confirmed from larger processor counts). Scalability for varying numbers of processors is shown in Figure 2a.

The performance of PeIGS depends to some extent on how eigenvalues cluster within the spectrum. Generally speaking, systems with many eigenvalues close together take longer to solve due to increased computation and communication costs for reorthogonalization. This is an important issue because actual applications with large matrices often have highly clustered spectra. For example, Figure 2b shows the spectrum of the 2053 by 2053 matrix referred to above.

Internally, PeIGS uses a conventional message passing programming model and column-distributed matrices. However, it is more commonly accessed through an interface provided by the Global Array toolkit, which supports one-sided data transfers and block-distributed matrices. The necessary data reorganization is handled by the interface. As noted above, such reorganization is very fast compared to the $O(N^3/P)$ linear algebra times.

PeIGS runs on a variety of parallel computer systems and can be ported easily to any machine that supports message passing. The current version of PeIGS utilizes either the TCGMSG portable communication package or Intel NX intrinsics; an MPI port is planned. It is being used at PNL as a production-grade library package in the construction of NWChem modules, including parallel SCF, MCSCF, DFT, and MP2 methods. Due to the speed and scalability of PeIGS, the time for eigensolving and associated linear algebra operations has been reduced to a small fraction of the total time-to-solution for these applications.

II.5 Run Time Data Base

The run time data base (RTDB) is the parameter and information repository for all application modules comprising NWChem. This is similar in spirit to the GAMESS [17-20] dump file or GAUSSIAN [21] checkpoint file. An input parsing module(s) stores the input and each application module obtains the appropriate parameters for execution and communicates to other modules via the RTDB. The RTDB library is accessible from both FORTRAN and C, is based on simple UNIX data base tools, and is widely portable. The storage is based on a string to data (of a single type) mapping, with complex data structures stored as separate data blocks of the appropriate type. There is also contextual levels built into this tool; for example, the gradient convergence threshold or print level would be a parameter applicable to various modules and would thus have a different value for each method/module. The appropriate context can be set to arbitrate between the possible values available.

III. Parallel Computational Chemistry Applications

We present in this section an outline of the development, implementation and performance of a variety of the modules comprising NWChem. Following an outline in Section III.1 of the common components of the chemistry applications, we consider in Section III.2 the attributes of a highly efficient and scalable algorithm to perform quadratically convergent, self-consistent field (SCF) Hartree Fock calculations on molecular systems. Section III.2.1 provides a brief account of the development of corresponding capabilities using the so-called "resolution of the identity" (RI) integral approximation [22], while Section III.2.1 describes parallel periodic Hartree Fock (III.2.2) capabilities Algorithmic developments involved in performing the so-called integral transformation, a necessary step in moving from the SCF to more quantitative methods are considered in Section III.3, while Section III.4 presents some initial thoughts on performing "direct" multiconfiguration SCF calculations. Two implementations of Möller-Plesset perturbation theory (MP2) designed to be optimum in the treatment of both spatially extended and compact molecules are considered in Section III.5, "mixed basis" MP2 (III.5.1) and RI-MP2 (III.5.2). Section III.6 turns to consider the parallel implementation of density functional theory (DFT), an increasingly popular and truly cost-effective alternative to more conventional correlated ab initio methods.

Moving away from the electronic structure modules of NWChem, we consider in Section III.7 the implementation of molecular dynamics (MD) capabilities, and consider the algorithmic issues involved in treating long range interactions. Finally, in Section III.8, we describe our activities in developing techniques for combining quantum mechanics and molecular mechanics.

The performance figures presented throughout this section were derived from using both the Intel Touchstone Delta and Kendall Square Research KSR-2 supercomputers.

III.1 Common Application Components

The software tools outlined above are examples of libraries with a standard application programmer interface (API). The designing of a standard API for each functionality that is required by many of our codes promotes code reuse, decreases the effort to develop new applications and also hides complexity, which further decreases development and maintenance costs. For example, when accessing an element of a global-array the application module does not require the specific knowledge of where the data resides or whether the data was retrieved via a message-passing call or some shared-memory mechanism. If it is subsequently necessary to change the implementation details of a GA then the application is insulated from this. There are many chemistry-specific components that can be similarly defined, and with proper design and implementation we may realize similar benefits. Figure 3 illustrates the architecture of our applications in a two dimensional block diagram although the actual interplay is multidimensional.

This design may be applied at all levels within the code. For instance, our *ab initio* molecular dynamics module invokes the self-consistent field and self-consistent field gradient modules as simple components through a standard API. To solve the self-consistent field equations requires neither access to the complete details of the basis set of the chemical system nor the exact position of the atomic centers. The self-consistent field module may need to know the total numbers of centers and contracted functions, but only the integral evaluation package will require actual coefficients and exponents of the primitive Gaussian basis. To illustrate this further we outline the basis set and geometry objects and the integral API used in the NWChem software.

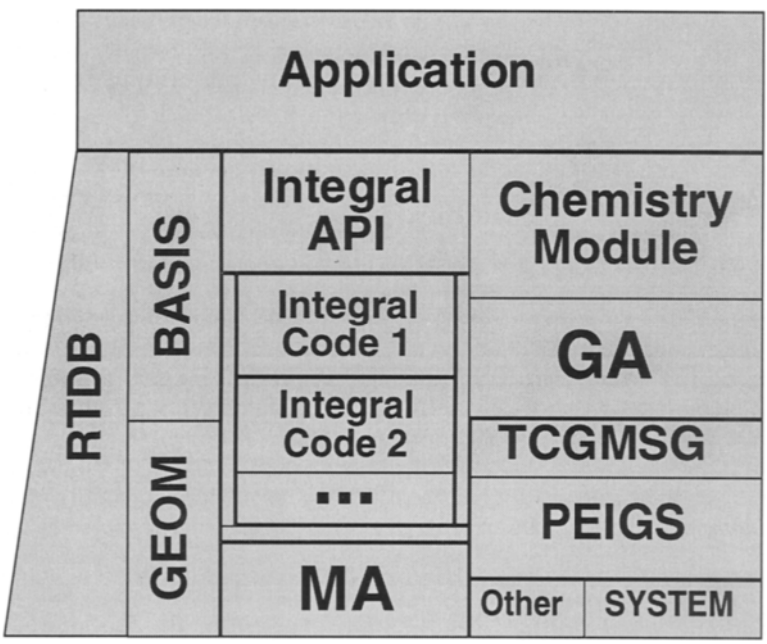


Figure 3. The interplay of software tools, common application components and specific module software.

The geometry object specifies the physical makeup of the chemical system by defining atomic centers, their respective position in space and nuclear charge as well as an associated name for the center. It also includes a possible applied electric field, the symmetry (point group or periodic) and other characteristics of the system. The basis set object handles the details of the gaussian basis set i.e., the approximate atomic orbitals, and stores only the unique information, for each center. The combination of a basis set object and a geometry object provides all non-state-specific information required by an *ab initio* calculation. These are not true "object-oriented" objects, since we do not support inheritance, but are well defined API's that enforce strict data hiding. Such a design allows the utilization of multiple basis sets and geometries throughout an application module. This

has reduced the burden of implementing techniques that require multiple basis sets, such as the density functional theory, RI-self-consistent field, RI-Møller-Plesset perturbation, etc., by allowing simple access to different "instances" of these objects.

All of the computational chemistry applications described below require, at some level, one- and two-electron integrals and integral derivatives over the Gaussian basis set(s) of the system. The Integral API is a layer between the base integral code and the application module, and allows the application programmer to essentially ignore the details of how the integrals are computed. This facilitates the incorporation of different integral technology into the applications that use the code; currently, the API requests integrals of various types based on computing shell blocks of integrals, with two different base integral codes used for production. The in-house developed code is a straightforward, open ended implementation of the McMurchie-Davidson [23, 24] algorithm.

III.2 Self-Consistent Field Hartree Fock

The essential core functionality in an electronic structure program suite is the direct self consistent field (SCF) module. It is increasingly evident that the application of direct SCF to large molecules is best performed on MPPs due to the enormous computational requirements. However, targeting systems in excess of 1,000 atoms and 10,000 basis functions requires a re-examination of the conventional algorithm and assumed memory capacities. The majority of previous parallel implementations use a replicated-data approach [4] which is limited in scope as the size of these arrays will eventually exhaust the available single processor memory. Our implementation of parallel direct SCF distributes these arrays across the aggregate memory using the Global Array tools. This ensures that the size of systems treated scales with the size of the MPP and is not constrained by single processor memory capacities.

The computationally dominant step, the construction of the Fock matrix, is readily parallelized as the integrals can be computed concurrently. With distributed matrices, a strip-mined approach is used where the integral contributions to small blocks of the Fock matrix are computed locally and accumulated asynchronously into the distributed matrix. By choosing blocking over atoms, the falloff in interaction between distant atoms can be exploited, while simultaneously satisfying local memory constraints. A simple performance model indicates that the ratio of computation to communication remains high, as shown in Figure 4 which demonstrates the scalability of this algorithm for a medium-sized system.

The conventional SCF solution scheme is based on repeated diagonalization of the Fock matrix. In the quadratically convergent SCF (QCSCF) approach [25], the SCF equations are recast as a non-linear minimization which bypasses the diagonalization step. This scheme consists of only data parallel operations and matrix multiplications which guarantees high efficiency on parallel machines.

Perhaps more significantly, QCSCF is amenable to several performance enhancements that are not possible in conventional approaches. For instance, orbital-hessian vector products may be computed approximately (e.g., using the resolution of the identity approximation discussed below) which significantly reduces the computation expense with no effect on the final accuracy. Figure 4 shows the speedup for the distributed Fock matrix construction and the complete QCSCF algorithm for a modest 461 basis function calculation of disilasesquioxane on both the Intel Touchstone Delta (Figure 4a) and the KSR-2 (Figure 4b). When the number of processors is of order $N_{atom}^2/2$, a degradation in the speedup for Fock matrix construction is observed due to load-balancing. The overall QCSCF speedup for small molecular calculations is also affected by the finer distribution of the global data structures as the number of nodes increases. In the disilasesquioxane case, neither effect is evident; the scalability of the QCSCF algorithm using distributed data has been demonstrated up to 512 nodes with systems of N ~3500.

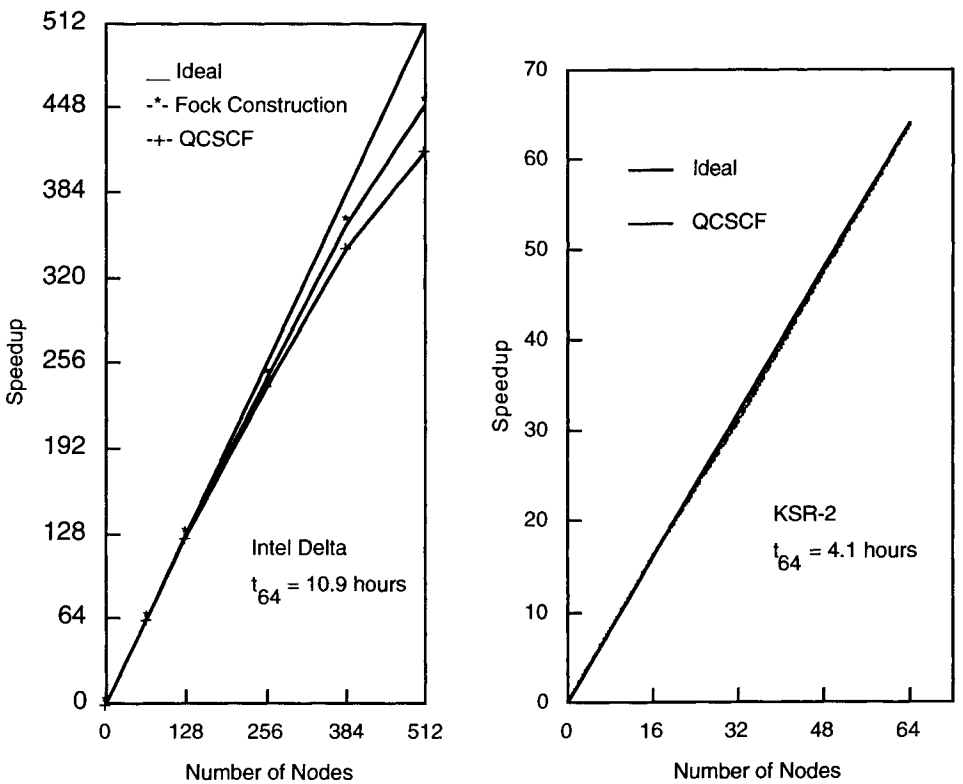


Figure 4. Quadratically Convergent SCF (QCSCF) performance (a) on the Intel Touchstone Delta and (b) on the KSR-2.

III.2.1 Resolution Of The Identity - Self-Consistent Field

One way to reduce the computational expense associated with the calculation of the four-center two-electron repulsion integrals is to approximate these integrals as linear combinations of three center integrals, using the so-called "resolution of the identity" (RI) integral approximation [22]. The idea is that products of Gaussian functions of the atomic orbital (AO) basis are approximated by linear combinations of Gaussians in a separate "fitting" basis. If the fitting or expansion basis were chosen to be the full Gaussian Product Theorem (GPT) result of all possible combinations of the AO basis functions, the RI result would be exact [26]. Vahtras et al [27] have investigated several ways of expanding the four-center integrals in terms of three-center integrals, with the so-called V-approximation best suited to reproduce total and dissociation energies.

$$(ij|kl) = \sum_{tu} (ij|t) V^{-1}_{tu} (u|kl) \text{ , where } V_{tu} = (t|u)$$

The expansion basis, which approximates products (or the potential arising from products) of basis functions, has to contain more functions and higher angular momenta than the atomic orbital basis. In our calculations we used atom-centered, even-tempered, uncontracted basis sets with maximum angular momenta two higher than in the AO basis.

The three center electron repulsion integrals are calculated only once and transformed with $V^{-1/2}$, which simplifies the subsequent calculation of the four center contributions to

$$(ij|kl) = \sum_{stu} (ij|s) V^{-1/2}_{st} V^{-1/2}_{tu} (u|kl) = \sum_{\bar{t}} (ij|\bar{t})(\bar{t}|kl)$$

Only these transformed integrals are stored as a global array, utilizing the (possibly distributed) memory of the parallel computer.

The calculation of the Coulomb contribution to the Fock matrix

$$J_{ij} = \sum_{\bar{t}kl} (ij|\bar{t})(\bar{t}|kl) D_{kl}$$

is most efficiently formulated as two sparse matrix-vector products

$$W_{\bar{t}} = \sum_{kl} (\bar{t}|kl) D_{kl} \quad \text{and} \quad J_{ij} = \sum_{\bar{t}} (ij|\bar{t}) W_{\bar{t}}$$

which is a computational effort of order $O(N^3)$, assuming that the number of expansion functions is proportional to the number of atomic orbitals.

The exchange component of the Fock matrix elements

$$K_{ij} = \sum_{\bar{t}kl} (ik|\bar{t})(\bar{t}|jl) D_{kl}$$

is not so easily partitioned and has to be done as two matrix multiplications

$$T^{\bar{t}}_{jk} = \sum_{l} (\bar{t}|jl) D_{kl} \text{ , } W^{\bar{t}}_{ij} = \sum_{k} (ik|\bar{t}) T^{\bar{t}}_{jk} \text{ , and } K_{ij} = \sum_{\bar{t}} W^{\bar{t}}_{ij}$$

Under the above assumptions this part of the calculations scales as $O(N^4)$, but as it consists only of linear algebra and does not include the calculation of integrals, it is executed much faster than the corresponding step in direct SCF.

Figure 5 shows the speedup curve on our KSR-2 computer for a calculation on propane using a cc-pVTZ basis set. The expansion basis was even-tempered and uncontracted with 5s4p3d2f1g for hydrogen and 6s5p4d3f2g1h for carbon .

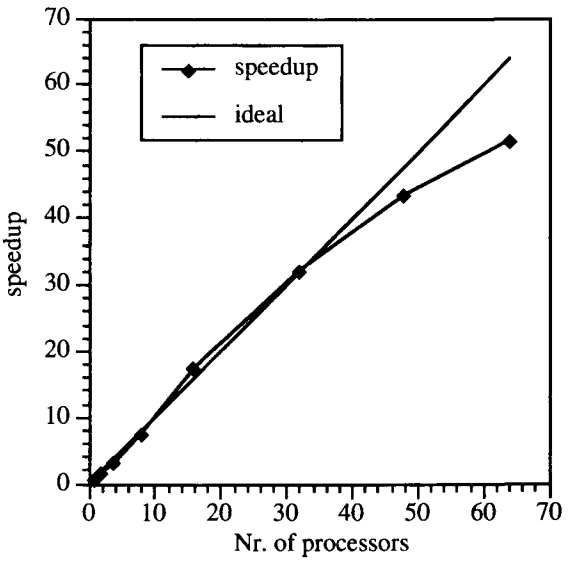


Figure 5. RI-SCF speedup curve on the KSR-2 for a calculation on propane using a cc-pVTZ basis set.

The method scales well with an efficiency of more than 80% up to 64 nodes, with the total 64 node run time of 396 seconds to be compared to 4200 seconds for direct SCF with four center integrals. The total energy calculated with this RI method, is within 0.5 mHartree of the full SCF energy with the same atomic orbital basis.

These results show that the RI-SCF method is well suited as a fast and efficient way to calculate approximate SCF energies, especially for compact systems with expensive basis sets. Approaches for larger systems ($> \approx 400$ basis functions), where an in-core storage of the transformed integrals is no longer practicable, are currently under investigation. Also a more systematic approach to the choice of expansion basis sets, especially for spatially extended systems with diffuse functions, has to be found, as in this case atom-centered functions give only a poor approximation to the product functions.

III.2.2 Periodic Hartree Fock; Crystal 92

The periodic Hartree Fock code, Crystal [28-30], has been parallelized initially using message passing with a replicated data approach. Both the PVM3.2 [31] and TCGMSG message-passing libraries can be used. The parallelization of the post-SCF computation of crystalline properties, for example properties connected with the eigenvalues e(k): density of states and band structure, has been accomplished, as has parallelization of the X-ray structure factors, electron momentum maps, and Compton profiles.

Improved parallel performance of Crystal has been obtained by both reducing the amount of I/O and introducing the GA tools. Reduced I/O is achieved by using the direct SCF version of the code and by introducing the dynamic MA to store quantities such as eigenvectors and mono-electronic integrals. Introduction of the GA tools provides for functionality such as parallel matrix eigensolvers and multipliers, the aim being to distribute among multiple nodes the diagonalization process at a single point instead of distributing work over k-points. This should be exploited in the case of systems were the Fock matrix has large dimensionality and where computational accuracy can be achieved using few k-points.

Number of Nodes	Wall-time	Efficiency
1	19426.4	1.000
4	4971.1	0.977
8	2622.3	0.926
16	1482.5	0.819
24	1115.8	0.725
32	884.8	0.686
40	785.7	0.618
48	798.7	0.507

Table 1. KSR-2 wall times and efficiencies for periodic direct SCF calculations on the molecular crystal of urea.

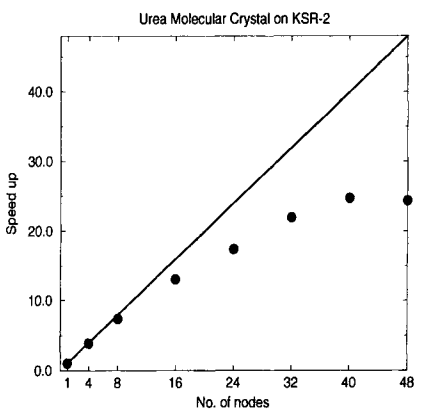


Figure 6. Periodic direct SCF calculations on the molecular crystal of urea.

Future developments include moving to a completely "distributed data" version where the largest data structures (Fock and density matrices) will be distributed among the nodes, instead of each node having its own copies. This approach will dramatically decrease the amount of memory needed and, therefore, increase the size of the problem amenable to treatment. The most important steps to consider are the calculation of the Fock matrix in direct space $F_{\mu\nu;g}$ using the density

matrix $P_{\mu\nu;g}$ and the 2-electron integrals, plus the Fourier transform of the Fock matrix from direct to reciprocal space: $F_{\mu\nu;g} \rightarrow F_{\mu\nu}(k)$.

Preliminary results on the KSR-2 are tabulated in Table 1 and shown in Figure 6 for a direct-SCF calculation of the Hartree-Fock wave function of a molecular crystal of urea (solid line represents theoretical linear speed-up). In this calculation there were 160 atomic orbitals, 58 k-points, and 5.5×10^6 mono-electronic integrals. For the same test on a CRAY-90, the CPU time spent was 7932 seconds, with an associated wall time of 21655 seconds.

III.3 Direct Four Index Transformation

The four index transformation of atomic orbital (AO) integrals to the molecular orbital (MO) representation is the intermediate step in moving from SCF calculations to more sophisticated correlated methods. This procedure would be difficult to implement efficiently on parallel architectures using traditional disk-based approaches. However, by computing the AO integrals as required, analogous to direct SCF, the four index transformation can be very efficiently performed.

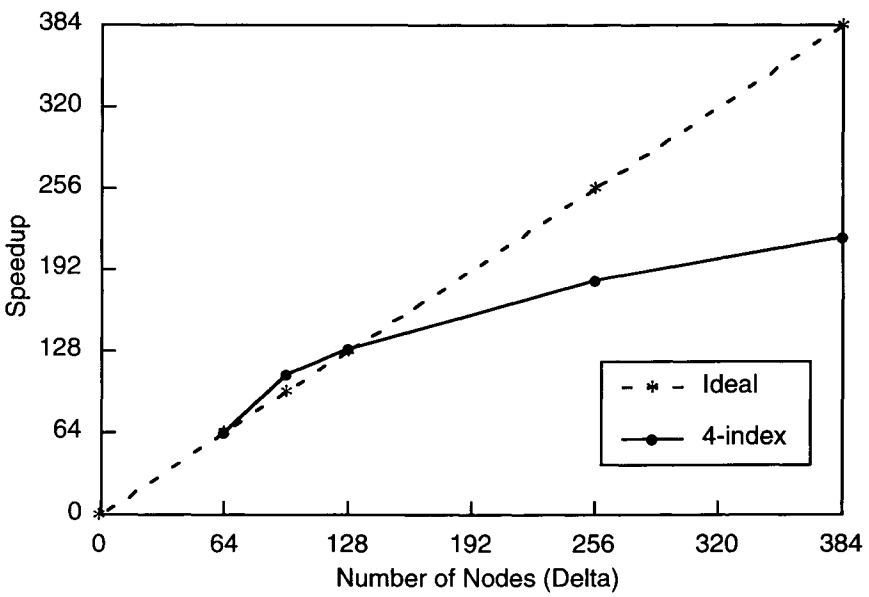


Figure 7. Speedup obtained for a 4 index transformation on 18-crown-6 ether (114 basis functions) using the Intel Touchstone Delta.

The computation of integrals combined with the transformation can readily extract almost peak computational rates from each processor given sufficient parallelism and low communication overhead (for example, 3.5 Gflops were

recently obtained from 128 nodes of the Intel Touchstone Delta). As in the SCF method, the judicious use of the memory is the overriding issue in the design of an algorithm capable of treating large systems. Our current four index transformation routine has $O(N^2)$ local memory requirements, a significant improvement over available programs which typically assume $O(N^3)$ memory. Figure 7 plots the speed up obtained for a prototype version which exhibits acceptable scalability although the performance model for our latest version indicates that a much higher level of parallelism can be achieved by removing synchronization points.

III.4 Direct Multiconfiguration Self Consistent Field

Multiconfiguration self consistent field (MCSCF) is a generalization of the SCF method and is an indispensable tool in studies of intermediate and reactive species. The traditional MCSCF procedure requires the transformation of integrals and solution of the secular problem which scales as $O(n_a!)$, where n_a is the number of active orbitals. As a consequence of the high computational demand of these two components, MCSCF is currently limited to treating relatively small systems. As recognized by Almlöf and Taylor [32] the number of transformed integrals computed and stored externally can be significantly reduced in a *direct*[1] approach equivalent to QCSCF. Although such a technique is not economic for sequential computers, on MPPs the recomputation of integrals at each optimization step is an acceptable cost compared to external storage of these integrals. There is ongoing work within the HPCC group to produce a direct MCSCF program which should significantly expand the capability of this method.

The ability to treat an MCSCF wavefunction with a moderately-sized active space is problematic because of the factorial scaling in the secular component. Even taking into account the large aggregate memory capacity envisioned in future MPPs, the size of the active space is only expected to increase two-fold. For cases which necessitate large active spaces, alternatives, such as the restricted active space (RAS) [33] method, will be necessary.

III.5 Møller-Plesset Perturbation Theory

The Hartree-Fock SCF method can be improved upon by methods that incorporate the effects of dynamic electron correlation. The use of correlated methods is often necessary in order to obtain predictive, or "chemical" accuracy. The simplest such method is second-order many-body perturbation theory (MBPT(2), also known as Møller-Plesset perturbation theory, MP2), which has proven useful in a wide range of problems for which the Hartree-Fock method is inadequate.

---

[1]This term is also used to describe MCSCF solution schemes that do not explicitly store the Hessian, a technique that is also used in our current implementation.

The MBPT(2) method involves $O(N^5)$ computational work and $O(N^4)$ data, as compared with $O(N^4)$ and $O(N^2)$ respectively for the SCF. While MBPT(2) presents many problems characteristic of more sophisticated correlated methods (most notably the large volume of data to be dealt with), it is also worth noting that like the SCF method, most other correlated methods involve a factor of N^2 more computation than data. Consequently MBPT(2) is one of the tougher problems for parallel performance in computational chemistry.

As computational and algorithmic advances have made calculations on progressively larger chemical systems possible, it has become clear that despite the formal $O(N^5)$ cost of the method, characteristics of many chemical systems will allow for lower cost calculations if "non-traditional" algorithms are employed. Alternative methods involve new ways of representing the interactions within the system exactly, using new mathematical formulations, controllable approximations, or a combination of both. In our work on large-scale correlated calculations on MPPs we have chosen to pursue two of these alternative approaches that are particularly well suited to different problem domains, the "mixed-basis" approach and the RI-MP2 method.

III.5.1 Mixed basis Møller-Plesset Perturbation Theory

The traditional MBPT(2) algorithm uses molecular orbitals (MOs) as the basis for computing interactions in the system. Unfortunately, these MOs are usually distributed across many atoms in the molecule. Consequently the interactions, which depend on distance, are rarely small enough to neglect. By recasting the problem so that a more spatially localized basis can be used, however, it should be possible to introduce significant sparsity into the interactions. The new representation involves localizing the occupied MOs and representing the unoccupied MOs, which are not easily localized, in the atomic orbital basis. This is the origin of the term "mixed basis" which we use to describe this method. The mixed basis formulation should be well suited to spatially extended chemical systems with moderately sized basis sets.

We have a prototype fully-distributed parallel mixed-basis MBPT(2) code that, for some relatively small systems, shows good scalability on the Intel Touchstone Delta. Figure 8 illustrates the speedup on a calculation on C_4H_{10} with an augmented pVDZ basis. The two dominant computational steps are the construction of the exchange operators and the determination of the excitation amplitudes. Exceptional speedup is shown in the first step, while degradation is shown for the second due to the smallness of the system and the relatively high ratio of communication-to-computation time. This code will be used as a testbed to determine the most efficient way to take advantage of sparsity in this approach. Without the use of very high performance parallel computers, it has not been possible to routinely perform calculations large enough to exhibit sufficient sparsity to be interesting in this context. We believe that once sparsity is incorporated in the parallel code we will be able to treat systems on the order of

1000-3000 basis functions, or 100-300 atoms, nearly an order of magnitude larger than the largest reported MBPT(2) calculations to date. The much larger amount of data involved in an MBPT(2) calculation will limit the size of calculation unless scalable I/O facilities are available to make disk storage of some of the data practical.

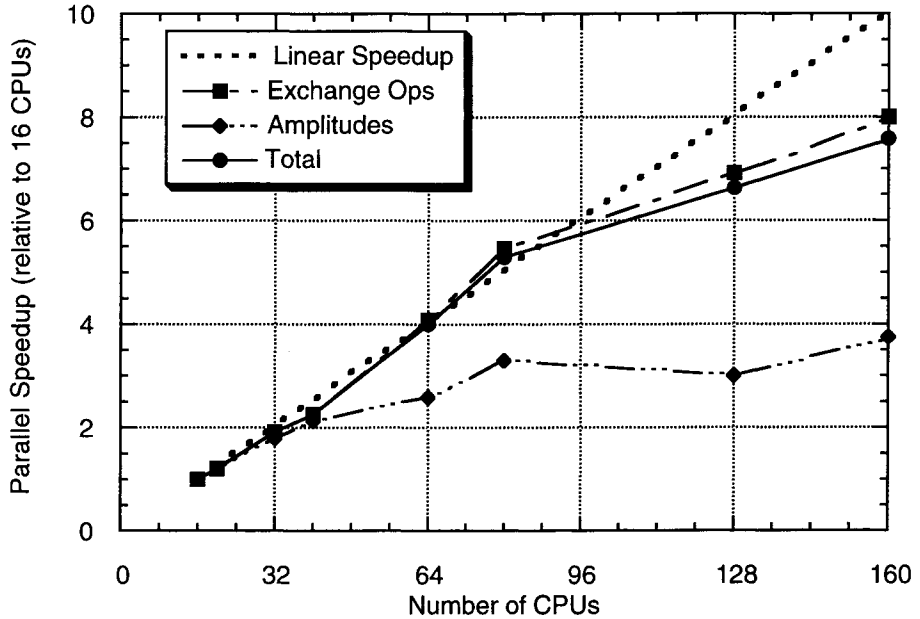


Figure 8. Mixed-basis MBPT(2) speedup obtained on the Intel Touchstone Delta for a calculation on C_4H_{10} with an augmented pVDZ basis.

III.5.2 Resolution Of The Identity - Møller-Plesset Perturbation Theory

Another type of system which is of particular interest is that which is relatively compact in spatial terms but for which large basis sets are used. In this case physical proximity of the atoms and characteristics common to large basis sets result in very few of the interactions being negligibly small.

For problems of this type, we have implemented a parallel RI-MP2 code which uses the so-called "resolution of the identity" (RI) integral approximation [22]. As mentioned in section III.2.1, if the associated fitting basis were chosen to be the full GPT result of all possible combinations of the AO basis functions, the RI result would be exact [26]. However experience has shown that the GPT basis is generally very nearly linearly dependent, so it should be possible to reduce the fitting basis and still retain most of the accuracy of the calculation. In general, the larger the AO basis set, the more near-redundancy in the GPT basis which is why we feel the RI-MP2 method is particularly suited to compact systems with relatively large basis sets.

Use of the RI approximation changes the amount of data that must be dealt with in most of the MBPT(2) procedure from $O(N^4)$ to $O(N^3)$ – significant because this is the amount of data that is stored on disk if there is not enough memory to do the whole calculation in core. The computational dependence of the RI-MP2 remains $O(N^5)$, but it may be formulated as a large distributed matrix multiplication which performs well on MPPs. The RI-MP2 procedure can be expressed very straightforwardly in terms of the GA toolkit and is implemented in such a way as to require only $O(N^2)$ memory provided that $O(N^3)$ disk space is available, making it possible to perform RI-MP2 calculations on most systems for which an SCF is possible.

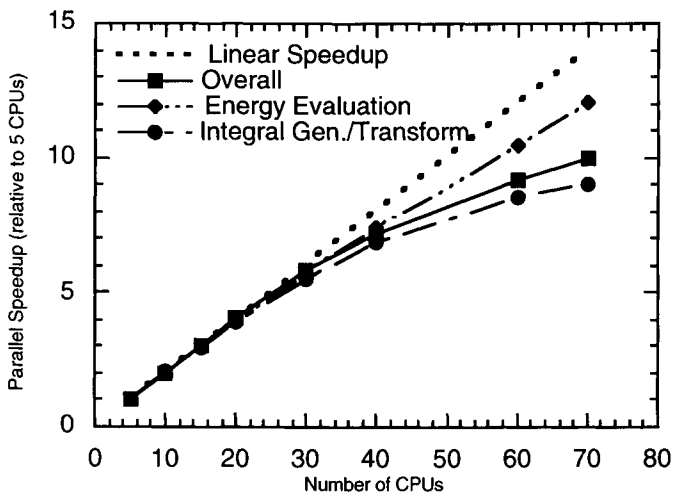


Figure 9. RI - MBPT(2) speedup obtained on the KSR-2 for calculations on 2,2'-di(trifluoromethyl)biphenyl using 204 AO basis functions and 966 fitting functions.

In Figure 9 we show the speedup obtained on the KSR-2 for calculations on 2,2'-di(trifluoromethyl)biphenyl using 204 AO basis functions and 966 fitting functions. The primary components of the overall timing are the integral transformation and the energy evaluation. The energy evaluation, which is dominated by a large distributed matrix multiplication displays excellent speedup from 5 to 70 processors. The transformation also scales well, but suffers from a degree of load imbalance in one step which reduces the performance slightly compared to the energy evaluation. Although for this case the transformation and energy components require comparable times, they have different computational complexities; for larger problems the energy evaluation component will dominate. Although these runs were disk-based, I/O time on the KSR is a negligible part of the run time on this problem because of the small volume of data (compared to an exact MBPT(2) calculation) and the buffering of the KSR's I/O subsystem and operating system. With this implementation in hand we feel that the most important area for further work on the RI-MP2

method is not in parallelization but rather in development of fitting basis sets. Because this approach is very recent [22], there is little detailed information available as yet on how to construct fitting basis sets to achieve given levels of accuracy. The parallel RI-MP2 will allow a much broader range of systems to be examined rapidly as part of this development effort.

III.6 Density Functional Theory

The density functional theory (DFT) software makes extensive use of the tools described above, including MA, TCGMSG, and GA. All data objects larger than O(N) are stored in global arrays. The code we have developed is a MPP implementation of the Hohenberg-Kohn-Sham formalism [34, 35] of DFT.

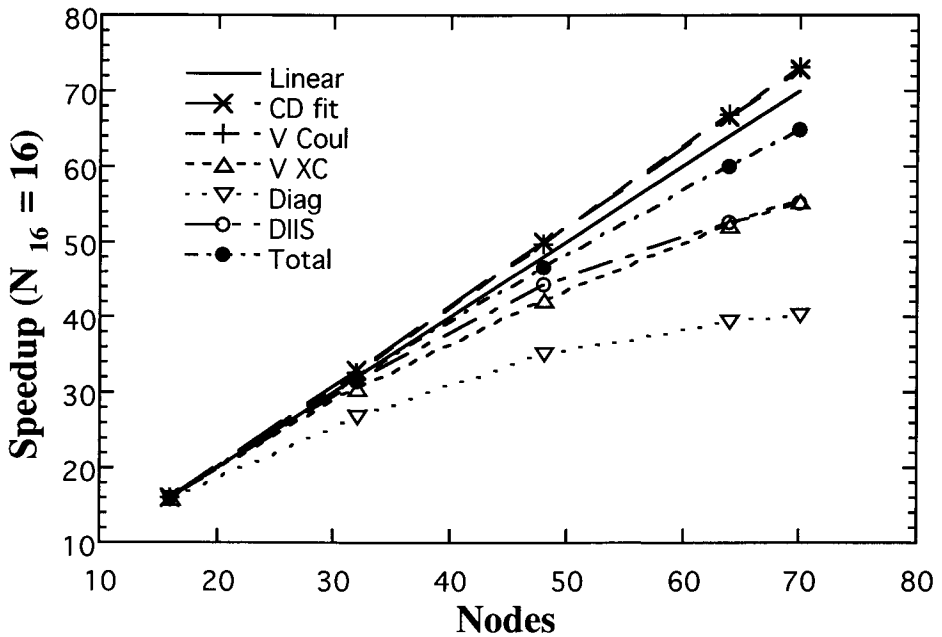


Figure 10. The speedup obtained for construction of Fock matrix components and subsequent diagonalization in DFT calculations of the zeolite fragment $Si_{26}O_{37}H_{30}$ (1199 AO basis functions and 2818 CD fitting functions) using up to 70 nodes of the KSR-2.

This method is able to give results similar to those from correlated ab initio methods at substantially reduced cost. It assumes a charge density and successively better approximates the Hamiltonian (the exchange correlation functional), in contrast with traditional ab initio molecular orbital methods (see above) that assume an exact Hamiltonian and successively better approximates the wavefunction [36]. The Gaussian basis DFT method in NWChem breaks the Hamiltonian down into the same basic one-electron and two-electron

components as traditional methods, with the two-electron component further reduced to a Coulomb term and an exchange-correlation term. The treatment of the former can be accomplished in a fashion identical to that used in traditional SCF methods, from a fitted expression similar that found in RI-SCF, or from the commonly used Dunlap fit. DFT is really distinguished from other traditional methods, however, by the treatment of the exchange-correlation term. This term is typically integrated numerically on a grid, or fit to a Gaussian basis and subsequently integrated analytically.

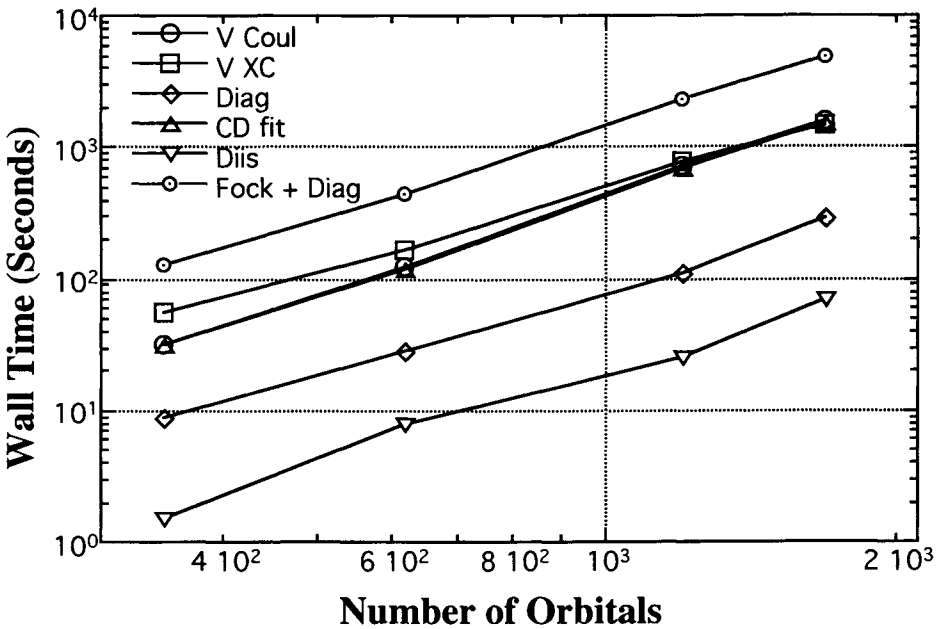


Figure 11a. The scaling of the components of the DFT code (wall time versus number of atomic orbitals) on various zeolite fragments using 48 nodes of the KSR-2. In a fit to the function $t = An^B$ for each of the components the value B obtained was 2.47, 2.50, 2.11, 2.15, 2.32, 2.33, for CD fit, V_{Coul}, V_{XC}, Diag, DIIS, and Fock+Diag respectively.

In our prototype DFT code the electron density and the exchange-correlation functional can be expanded in terms of auxiliary Gaussian basis sets. We have incorporated the exchange-correlation functionals (Dirac's exchange functional [37] and the Vosko, Wilk, Nusair's correlation functional [38]) required to obtain the Local Density Approximation (LDA) and Local Spin Density Approximation (LSD) for open shell systems. In addition, we have implemented two gradient corrected functionals for the exchange-correlation contributions; Becke's exchange functional [39] and Lee, Yang, and Parr's correlation functional [40]. All of these functionals are fully implemented for closed and open shell systems. Several techniques have been incorporated to improve convergence in the

iterative SCF procedure: damping, level shifting and DIIS. These have, of course, all been implemented so as to exploit MPP technology with maximum efficiencies.

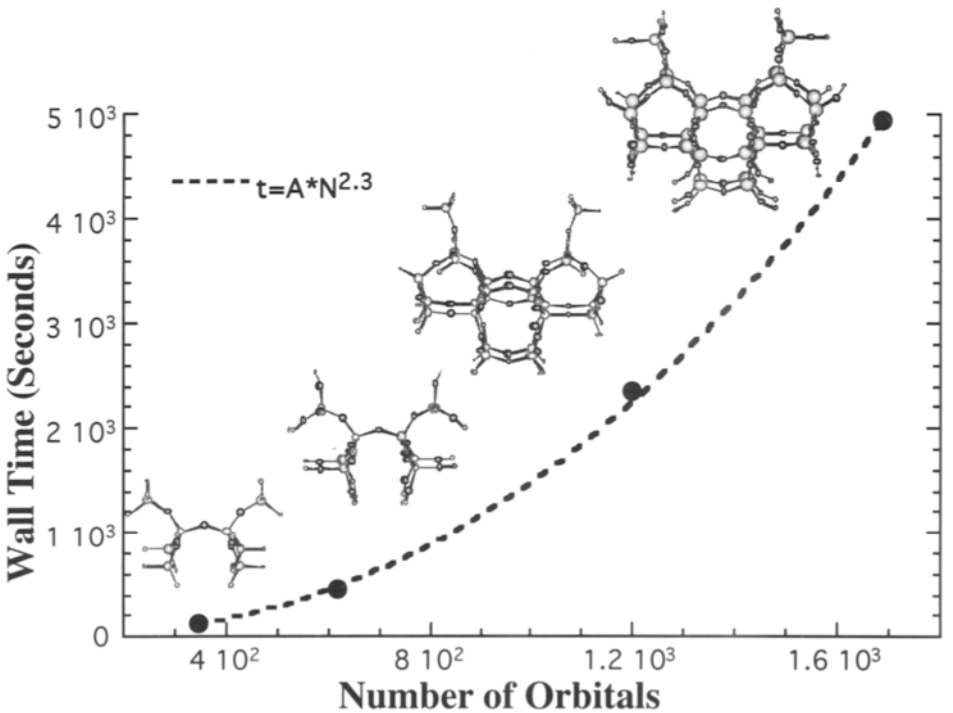


Figure 11b. The overall scaling of the DFT code (wall time versus number of atomic orbitals) on various zeolite fragments using 48 nodes of the KSR-2.

The time consuming portions of NWChem DFT calculations have already shown significant speedup on the KSR-2 (in calculations on various zeolite fragments with system sizes ranging up to 1700 atomic orbitals and 4000 fitting functions). The four time consuming steps are: fit of the charge density, calculation of the Coulomb potential, evaluation of the exchange-correlation potential, and diagonalization of the Fock matrix. Overall speedup of better than 90% using up to 70 processors for building the Fock matrix and subsequent diagonalization is seen. Figure 10 shows the speedup obtained on up to 70 nodes of the KSR-2 for $Si_{26}O_{37}H_{30}$. Figures 11a and 11b show the scaling of each of the components mentioned above as well as the overall scaling of the DFT code on zeolite fragments using 48 nodes of the KSR-2.

The Intel machines (unlike the KSR-2 which has an ALLCACHE "shared memory" architecture) are truly massively parallel distributed memory platforms. In porting to the Touchstone Delta we encountered a few algorithmic

flaws (with respect to data locality) which had to be corrected in order to obtain the excellent speedup as shown below for $Si_{26}O_{37}H_{30}$ in Figure 12.

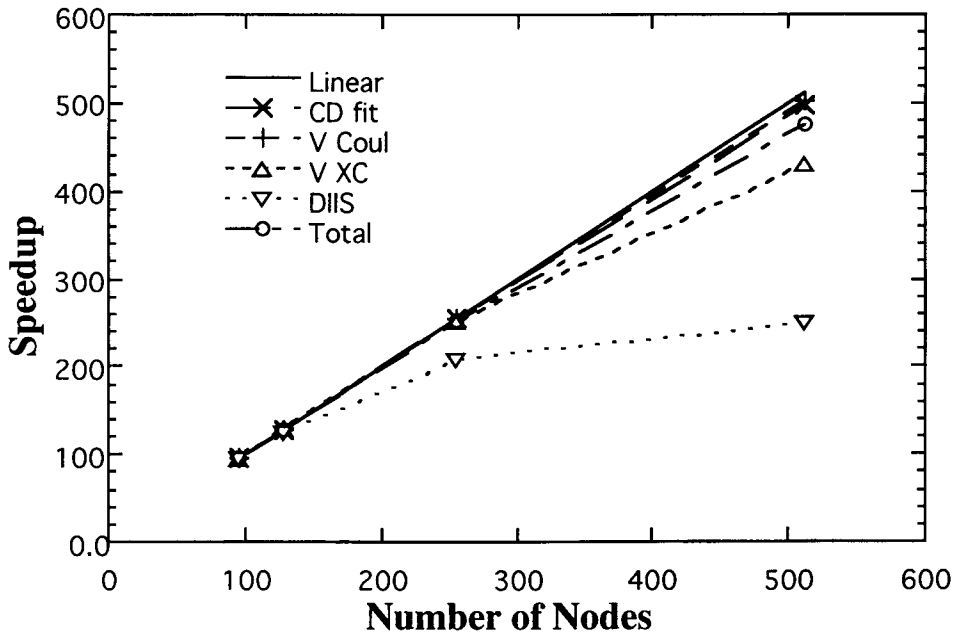


Figure 12. The speedup obtained for calculations of the Fock matrix and it's components for the zeolite fragment $Si_{26}O_{37}H_{30}$ on up to 256 nodes of the Intel Touchstone Delta.

III.7 Molecular Dynamics

We are developing a full-featured, general-purpose module within NWChem for molecular dynamics (MD) simulations on MPPs, designed to treat virtually all chemical systems, with the same functionality as many commercial programs. The code features an extensive and easily modifiable force field, efficient treatment of long range interactions via Ewald summations, and fast multipole methods, a variety of statistical mechanical ensembles, energy minimizations, and normal mode calculations. The code runs on a wide range of MPPs and gives large performance gains over scalar and vector computers. For example, on 512 processors of the Intel Delta, our code requires less than a tenth of a second to perform a step of MD on 7776 atoms. We have performed some of the largest MD simulations to date studying diffusion in zeolites to better understand separation processes in these materials. These nanosecond simulations involved ≈10,000 atoms and required ≈100 hours on the Intel Delta. We are developing a distributed data version of this code that will extend the size of systems that can be studied to beyond a million atoms.

The major long range interactions in MD simulations are Coulombic and dipolar, and are the most expensive to compute, often accounting for 90% or more of the total computation time. Typically, MD simulations are performed with periodic boundary conditions imposed on the simulation box, and the traditional methods of computing the forces involve then the use of either minimum image or Ewald techniques. For a system of N particles, minimum image, in conjunction with spherical cut-off and some book-keeping methods [41], reduces the computation time to O(N). However, minimum image is more adequate for short range forces, such as dispersion and Van der Waals, typically modeled by a Lennard-Jones potential; for the long range charge and dipole interactions, minimum image is generally inaccurate [42, 43]. These long range forces are therefore usually handled using Ewald or Ewald-Kornfeld methods [44]. While accurate, these Ewald techniques scale computationally as $O(N^{3/2})$ and are hence not practical for very large simulations. Similar difficulties arise in computing the long range forces in a finite (i.e. vacuum or free space boundary conditions) system. For a finite system of N charges or dipoles, direct calculation of electrostatic forces scales as N(N-1)/2. This approach leads to prohibitively high computational costs for systems of more than a few thousand particles and is not feasible for very large systems. On the other hand, potential truncation methods like their periodic counterparts, ignore long range tails and are generally inaccurate. In the recent past, "hierarchical codes" or "tree codes" have been developed in astrophysics [45-47] to efficiently compute Coulomb-type sums. Unlike Particle-Mesh methods [48], tree codes are gridless and instead divide the computational region into a hierarchy of cells, where cells at successive levels of refinement form a tree structure (i.e., subdivision of a cell at some level leads to children cells at the next level of refinement). The mass and center of mass of each cell are computed at a level and used in a monopole approximation to compute the interaction between every particle and its cell's interaction list (the interaction list of a cell consists of the children of the near neighbors of its parent which are not near neighbors of the cell itself). The algorithm then recursively performs the same sequence of steps on neighboring cells at the next level of refinement. These algorithms are of O(N log N) but of limited accuracy.

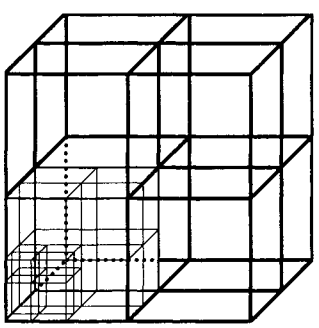


Figure 13. Tree structure used in FMM and CMM calculations.

Addition of higher terms in the expansion (asymptotic expansion) [49], introduction of Taylor expansions at observer sites, and shifting of multipole and

Taylor expansions between levels (eliminating costly recomputation at every level, as for multipole expansions in tree codes) culminated in the O(N) fast multipole method(FMM) of Greengard and Rokhlin [50, 51]. The formulation of three-dimensional FMM [51] is couched in spherical harmonics language. Recently, however, a Cartesian version of FMM, the cell multipole method(CMM), was proposed by Ding et. al. [52]. Both approaches use a tree structure where a cell at level (l) is divided into eight children cells at level (l+1), each of which is divided in turn into eight children cells at level (l+2), and so on as illustrated in Figure 13.

The algorithm consists of an upward pass (finer to coarser levels) during which the multipole moments are computed, and a downward pass (coarser to finer levels) during which the Taylor coefficients are computed. At the finest level, the near interactions are computed directly and the far interactions are obtained from the Taylor expansions, and both contributions are summed to get the total energy and forces.

CMM is physically more transparent than FMM, is easier to implement, and can be naturally incorporated into various MD calculations which are typically set in Cartesian coordinates. In addition, CMM is more amenable to extension to other than charge-charge interactions e.g., to any polarizable charged dipolar system [53-55]. In addition to the rapid computation of all of the interactions in such a system, the algorithm performs sufficient self-consistent iteration of the induced dipoles. In FMM, a desirable accuracy can be achieved by retaining a high enough number of terms in the expansions. In CMM, increased accuracy requires explicit addition of higher terms in the expansions. The relative complexity of Cartesian expansion coefficients and the considerable effort required to program them impose a practical limit on the attainable accuracy of CMM. These drawbacks, however, do not seriously limit the use of CMM, since in practice very good accuracy is obtained with only the first few orders [51, 54, 55]. Fast multipole methods such as FMM and CMM, exploit the slow variation of the far part of long range interactions over particle coordinates. They consequently approximate the direct computation of the far part with more efficient multipole and Taylor expansions. Figure 14 shows the speed-up obtained on the Intel Touchstone Delta for the Coulomb energy and forces computation, using our second order CMM implementation (single level), in a simulation on a random collection of 23,375 charged particles, with 4 particles per cell on average.

In contrast to these spatial approximations, multiple time step methods(MTS) [56, 57] exploit the slow variation of far interactions over time in MD simulations. In MTS the force on a particle is divided into a rapidly changing primary component due to nearest neighbor particles, and a slowly changing secondary component due to more remote neighbor particles. The idea then is to compute the primary forces every time-step δt and the secondary forces every $n\delta t$. Between $t + \delta t$ and $t + n\delta t$, the secondary forces are propagated through a Taylor expansion. The method then effectively uses two time-steps: δt for primary forces and $n\delta t$ for

secondary forces (it is also possible to use more than two time-steps). The same idea can be applied to interactions computed with the fast multipole methods in MD simulations. The directly calculated near part is computed every time-step. The expansion calculated far part is computed less frequently. Effectively, a larger time-step is used to compute the Taylor coefficients of the potential expansion. Between their successive computations, the Taylor coefficients are propagated through a Taylor expansion in time. When incorporated into FMM or CMM, this MTS algorithm should lead to a substantial increase in efficiency. Vector and parallel implementations for tree codes and fast multipole methods have been examined by several authors [58-60].

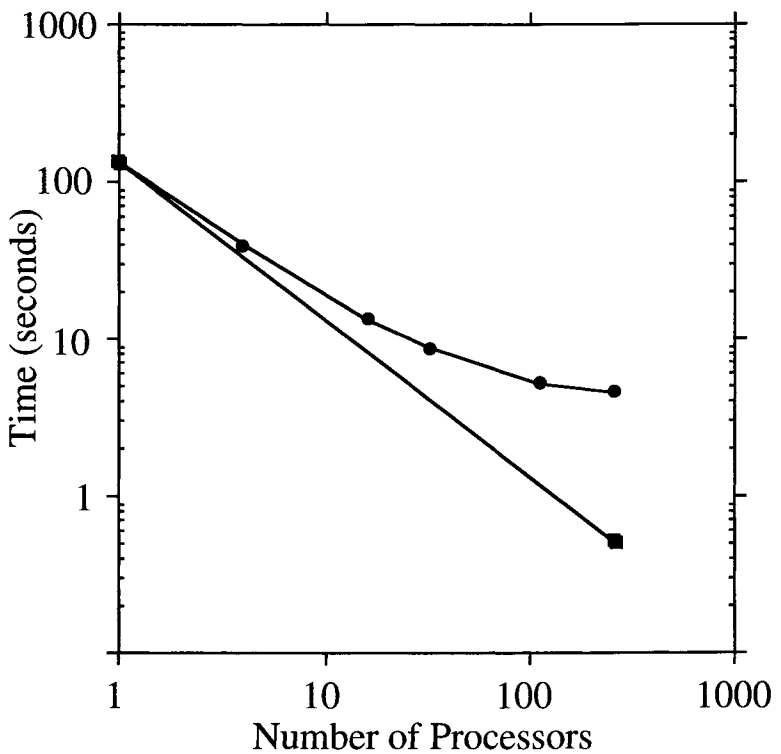


Figure 14. Speed-up on the Intel Touchstone Delta for the Coulomb energy and forces computation, using second order CMM (single level), on a random collection of 23,375 charged particles, with 4 particles per cell on average.

III.8 Combined Quantum Mechanics/Molecular Mechanics Theory

Ab Initio quantum mechanics (QM) calculations potentially can give highly accurate results. However, due to their high computational cost, they are customarily applied to only small molecules and clusters (less than 50 atoms). In contrast, molecular mechanics (MM) calculations are much cheaper and are readily applied to very large systems (≈100,000 atoms) such as hydrated proteins.

MM methods are based on a classical force field, which represents inter- and intramolecular interactions by simple functions, with the required force field parameters derived by comparison to experimental data. The low cost of MM methods allows the study of the time evolution of these large systems by integration of the equations of motion in MD simulations. While MM methods are computationally efficient, they are not able to adequately describe chemical phenomena, such as the breaking and forming of bonds, required to model reactions. In order to make tractable the study of the chemical behavior of large systems, it is possible to combine features of both the QM and MM methods [61-69]. Thus, small parts of the systems that require a rigorous description are treated quantum mechanically, while the majority of the system is described classically.

The basic idea is to partition the system into a QM region and a MM region. Under this partition, the effective Hamiltonian (H_{Eff}) of a system can be written as

$$H_{Eff} = H_{QM} + H_{MM} + H_{QM/MM}$$

in which H_{QM} describes the QM atoms, H_{MM} gives the molecular mechanical force field for the MM atoms, and $H_{QM/MM}$ is the Hamiltonian that couples the QM and MM atoms.

$$H_{QM/MM} = -\sum_{iM} \frac{q_M}{r_{iM}} + \sum_{\alpha M} \frac{Z_\alpha q_M}{R_{\alpha M}} + \sum_{\alpha M} \frac{B_{\alpha M}}{R_{\alpha M}^{12}} - \frac{A_{\alpha M}}{R_{\alpha M}^{6}}$$

The terms in $H_{QM/MM}$ represent electron-MM partial charge interactions, QM/MM nuclear-nuclear interactions, and long range dispersion effects. The total energy of the system is the expectation value of the wavefunction for H_{Eff}: $E_{Tot} = \langle Y | H_{Eff} | Y \rangle$. The forces on QM and MM atoms, needed for geometry optimizations and dynamic simulations, are $F_{QM} = - dE_{Tot} / dR_{QM}$ and $F_{MM} = - dE_{Tot} / dR_{MM}$. It is obvious from the above that the only additional quantum calculations involved are the one electron integrals between the MM partial charges and the electronic wavefunction of QM atoms. Many variations of the combined QM/MM method are possible, the key differences being the manner in which the QM and MM regions are described. For example, the QM region can be described with a semiempirical (SE) method, a Hartree-Fock (HF) or correlated method, or density functional theory (DFT). The MM region can be described by the empirical force field approach described above, continuum models, frozen density methods, or local orbital methods.

We have developed prototypes of both the SE/MM and DFT/MM formalisms. The prototypes integrate our parallel MD program with the semiempirical code MOPAC6 and the density functional code DeFT. We are currently using serial versions of both MOPAC [70] and DeFT [71]. The prototype codes already allows us to apply our extensive MD simulation capabilities to a wide range of problems, such as the study of catalysis by enzymes and zeolites, solvation dynamics, and reactions in solution. The use of the parallel, hierarchical CMM (described above) for the rapid evaluation of long range classical electrostatics allows us to treat systems containing as many as a million classical atoms with the QM/MM

method. We are currently integrating the SCF and DFT modules of NWChem with the MD code. The use of the parallel SCF and DFT codes will greatly expand the possible size of the QM region in the QM/MM method.

IV. Conclusions

We have described work in progress to develop the molecular modeling software application, NWChem, for massively parallel processors (MPPs). Consideration has been given to the issues in developing scalable parallel algorithms with emphasis on the distribution, as opposed to the replication, of key data structures. We have described tools to support this programming paradigm and have provided an outline of the implementation and performance of a variety of the modules comprising NWChem, with particular focus on a highly efficient and scalable algorithm to perform quadratically-convergent self-consistent field calculations on molecular systems. Accounts have also been given of the development of corresponding MPP capabilities in the areas of periodic Hartree Fock, Möller-Plesset perturbation theory (MP2), density functional theory and molecular dynamics. Performance figures using both the Intel Touchstone Delta and Kendall Square Research KSR-2 supercomputers demonstrate the potential of MPPs in the area of computational chemistry.

Acknowledgements

This work was performed under the auspices of the Office of Scientific Computing and under the Office of Health and Environmental Research, which funds the Environmental and Molecular Sciences Laboratory Project, D-384. This work was performed under Contract DE-AC06-76RLO 1830 with Battelle Memorial Institute, which operates the Pacific Northwest Laboratory for the U. S. Department of Energy

References

1. R. J. Harrison, *Int. J. Quant. Chem.* **40**, 337-347 (1991).
2. J. Boyle, et al., *Portable Programs for Parallel Processors* (Holt, Rinehart and Winston, Inc., 1987).
3. MPI, University of Tennessee, MPI: A Message-Passing Interface Standard (1994).
4. R. A. Kendall, R. J. Harrison, R. J. Littlefield, M. F. Guest, in *Reviews in Computational Chemistry* K. B. Lipkowitz, D. B. Boyd, Eds. (VCH Publishers, Inc.., New York, 1994).
5. The MPI Forum: A Message Passing Interface, Supercomputing '93 (IEEE computer Society Press, Los Alamitos, California, Portland, OR, 1993), pp. 878-883.
6. M. E. Colvin, C. L. Janssen, R. A. Whiteside, C. H. Tong, *Theoretica Chimica Acta* **84**, 301-314 (1993).
7. T. R. Furlani, H. F. King, *J. of Comp. Chem. (in press)* (1995).
8. J. Nieplocha, R. J. Harrison, R. J. Littlefield, Global Arrays; A Portable Shared Memory Programming Model for Distributed Memory Computers, Supercomputing '94 (IEEE Computer Society Press, Washington, D.C., 1994).

9. M. Arango, D. Berndt, N. Carriero, D. Gelernter, D. Gilmore, *Supercomputer Review* **3(10)** (1990).
10. M. Schuler, T. Kovar, H. Lischka, R. Shepard, R. J. Harrison, *Theoretica Chimica Acta* **84**, 489-509 (1993).
11. D. E. Bernholdt, R. J. Harrison, *Int. J. Quant. Chem. Symposium (in press)* (1994).
12. A. P. Rendell, M. F. Guest, R. A. Kendall, *J. Comp. Chem.* **14**, 1429-1439 (1993).
13. E. R. Jessup, Ph. D. Thesis, Yale University (1989).
14. S. S. Lo, B. Phillipps, A. Sameh, *SIAM J. Sci. Stat. Comput.* **8(2)** (1987).
15. G. Fann, R. Littlefield, Parallel Inverse Iteration with Reorthogonalization, Sixth SIAM Conference on Parallel Processing for Scientific Computing (SIAM, 1993), pp. 409-413.
16. Anderson, et. al., *SIAM* (1992).
17. M. Dupuis, D. Spangler, J. J. Wendolowski, University of California, Berkeley, NRCC Software Catalog (1980).
18. M. W. Schmidt, et al., *QCPE Bull.* **7**, 115 (1987).
19. M. F. Guest, R. J. Harrison, J. H. v. Lenthe, L. C. H. v. Corler, *Theo. Chim. Acta* **71**, 117 (1987).
20. M. F. Guest, et al., Computing for Science Ltd, SERC Daresbury Laboratory, Daresbury, Warrington WA4 4AD, UK.
21. M. Frisch, et al., Gaussian, Inc., 4415 Fifth Avenue, Pittsburgh, PA 15213, USA, (1992).
22. M. Feyereisen, G. Fitzgerald, A. Komornicki, *Chem. Phys. Lett.* **208(5-6)**, 359-363 (1993).
23. L. E. McMurchie, Ph. D. Thesis, University of Washington (1977).
24. L. E. McMurchie, E. R. Davidson, *J. Comp. Phys.* **26**, 218 (1978).
25. G. B. Bacskay, *Chem. Phys.* **61**, 385 (1982).
26. J. Almlof, "Auxiliary Basis Expansions - The End of Four-Center Integrals?", West Coast Theoretical Chemistry Conference Sandia National Laboratory, Livermore, CA, (1994).
27. O. Vahtras, J. Almlof, M. Feyereisen, *Chem. Phys. Lett.* **213**, 514 (1993).
28. R. Dovesi, C. Pisani, C. Roetti, M. Causa, V. R. Saunders, Quantum Chemistry Programs Exchange, Publication 577, University of Indiana (1988).
29. R. Dovesi, V. R. Saunders, C. Roetti, University of Torino, CRYSTAL92 user's manual (1992).
30. N. M. Harrison, V. R. Saunders, S.E.R.C Daresbury Laboratory, Daresbury, CRYSTAL92 user's manual (1994).
31. A. Geist, et al., ORNL, PVM 3 User's Guide and Reference Manual; PVM is available from netlib (1994).
32. J. Almlof, P. R. Taylor, in *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* C. Dykstra, Eds. (D. Reidel, Dordrecht, 1984).
33. P. A. Malmqvist, A. Rendell, B. O. Roos, *J. Phys. Chem.* **94**, 5477 (1990).
34. P. Hohenberg, W. Kohn, *Phys. Rev. B.* **136**, 864-871 (1964).
35. W. Kohn, L. J. Sham, *Phys. Rev. A.* **140**, 1133-1138 (1965).
36. E. Wimmer, in *Density Functional Methods in Chemistry* J. K. Labanowski, J. W. Andzelm, Eds. (Springer-Verlag, 1991) pp. 7-31.

37. P. A. M. Dirac, *Cambridge Philos. Soc.* **26**, 376 (1930).
38. S. H. Vosko, L. Wilk, M. Nusair, *Can. J. Phys.* **58**, 1200-1211 (1980).
39. A. D. Becke, *Phys. Rev. A.* **38(6)**, 3098-3100 (1988).
40. C. Lee, W. Yang, R. G. Parr, *Phys. Rev. B* **37(2)**, 785-789 (1988).
41. M. P. Allen, D. J. Tildesley, *Computer Simulation of Liquids* (Oxford University Press, 1990).
42. J. Guenot, P. A. Kollman, *J. Comp. Chem.* **14**, 295 (1993).
43. K. Tasaki, S. McDonald, J. W. Brady, *J. Comp. Chem.* **14**, 278 (1993).
44. S. W. DeLeeuw, J. W. Perram, E. R. Smith, *Proc. R. Soc. Lond.* **A373** (1980).
45. A. W. Appel, *SIAM J. Sci. Stat. Comput.* **6**, 85 (1985).
46. J. E. Barnes, P. Hut, *Nature* **324**, 446 (1986).
47. L. Hernquist, *Comp. Phys. Comm.* **48**, 107 (1988).
48. R. W. Hockney, J. W. Eastwood, *Computer Simulation Using Particles.* (Adam Hilger, Bristol and New York, 1989).
49. L. V. Dommelen, E. A. Rundensteiner, *J. Comp. Phys.* **83**, 126 (1989).
50. L. Greengard, V. Rokhlin, *J. Comp. Phys.* **73**, 325 (1987).
51. L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems* (MIT Press, Cambridge, 1987).
52. H.-Q. Ding, N. Karasawa, W. A. G. III, *J. Chem. Phys.* **97**, 4309-4315 (1992).
53. R. Kutteh, *J. Comp. Phys.* (1994).
54. R. Kutteh, J. B. Nicholas, *Comp. Phys. Comm.* (1994).
55. R. Kutteh, J. B. Nicholas, *Comp. Phys. Comm.* (1994).
56. H. Grubmuller, H. Heller, A. Windemuth, K. Schulten, *Mol. Sim.* **6**, 121 (1991).
57. W. B. Streett, D. J. Tildesley, G. Saville, *Mol. Phys.* **35**, 639 (1978).
58. L. Greengard, W. D. Gropp, Eds., *Parallel Processing for Scientific Computing* (1989).
59. J. F. Leathrum Jr., J. A. Board Jr., Department of Electrical Engineering, Duke University, The Parallel Fast Multipole Algorithm in Three Dimensions (1992).
60. F. Zhao, MIT, Report 995, An O(N) Algorithm for Three Dimensional N-body Simulations (1987).
61. A. Warshel, M. Levitt, *J. Mol. Biol.* **103**, 227 (1976).
62. M. J. Field, P. A. Bash, M. Karplus, *J. Comp. Chem.* **11(6)**, 700-733 (1990).
63. R. V. Stanton, D. S. Hartsough, K. M. Merz, *J. Phys. Chem.* **97**, 11868-11870 (1993).
64. T. Wesolowski, A. Warshel, *J. Phys. Chem.* **98**, 5183-5187 (1994).
65. C. J. Cramer, D. G. Truhlar, *Science* **256**, 213-217 (1992).
66. M. W. Wong, M. J. Frisch, K. B. Wiberg, *J. Am. Chem. Soc.* **113(13)**, 4776-4782 (1991).
67. J. L. Gao, X. F. Xia, *Science* **258**, 631-635 (1992).
68. K. A. Sharp, B. Honig, *J. Phys. Chem.* **94**, 7684-7692 (1990).
69. J. L. Chen, L. Noodleman, D. A. Case, D. Bashford, *J. Phys. Chem.* **98**, 11059-11068 (1994).
70. J. J. P. Stewart, *J. Computer-aided Molecular Design* **4**, 1-105 (1990).
71. A. St-Amant, Ph. D. Thesis, Universite de Montreal (1992).

This Page Intentionally Left Blank