This is to certify that the

dissertation entitled

Effects of Communication Costs on the Design and Implementation of Parallel Numerical Algorithms

presented by

Christian Trefftz

has been accepted towards fulfillment
of the requirements for

PhD degree in Computer Science

Major professor

Date 6/6/94

Effects of Communication Costs on the Design and Implementation of Parallel Numerical Algorithms

By

Christian Trefftz

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science Department

1994

ABSTRACT

Effects of Communication Costs on the Design and Implementation of Parallel Numerical Algorithms

By

Christian Trefftz

An insatiable demand for more computational capacity characterizes many computer applications that model various physical phenomena, forecast the behavior of natural and artificial systems, and explore different design options in manufacturing. Solving large numerical applications has traditionally been one of the tasks best suited for supercomputers. Over the last few years, supercomputer architectures have migrated from large vector mainframes to *scalable* parallel architectures, which are designed to offer corresponding increases in performance as the number of processors is increased. Such systems encompass both *massively parallel computers* (MPCs) and *clusters* of high-performance workstations interconnected by high-speed networks.

Both MPC and clusters are characterized by the distribution of memory among processor nodes. In such systems, the performance of applications depends on the efficiency of communication, which in turn depends on the underlying communications architecture. The wide variety of communications platforms for scalable systems has led to an increasing interest in the development of communications libraries, which can increase portability and usually offer better performance than communication

routines embedded in the applications software. The operations included in typical communications libraries include not only *point-to-point* primitives, but also *collective* communication primitives, which involve more than two processes. Collective communication is receiving increasing attention due a better understanding of its wide applicability in parallel processing.

Thus has arisen the need for the research described in this dissertation: to study the effects of interprocess communication costs, particularly that of collective operations, on the design, implementation, and performance of parallel numerical algorithms. The thesis can be stated as: The use of communication operations, designed to exploit properties of new generation communications architectures for distributed-memory platforms, can significantly improve the performance of parallel numerical algorithms; moreover, the redesign of such algorithms to account for point-to-point and collective communication costs explicitly, can result in further performance improvement. The dissertation makes several specific contributions, including: comparisons of cluster and MPC environments for numerical algorithm design; study of the effects on performance of communication operations that are optimized for new generation parallel architectures; and case studies for specific highly parallel numerical linear algebra algorithms that have recently been designed.

To my wife Ana Cristina

ACKNOWLEDGMENTS

I am extremely grateful to Dr. Philip McKinley, my advisor, for his guidance, generosity, and patience. His dedication and commitment to research are remarkable. A good advisor can motivate his/her students to try to do their best and that was certainly the case with Dr. McKinley. I also would like to express my appreciation to the members of my committee, Dr. Li, Dr. Ni and Dr. Mutka, for their comments and suggestions. It was a privilege to be in their classes. Dr. Zeng, from the math department, deserves a special mention for all his help with mathematical aspects of the programs.

I am also indebted to Dr. Carl Page for encouraging me to come to Michigan State in the first place, to Dr. Diane Rover for her class on visualization which was instrumental for this research and to Dr. W.J. Hsu who guided my first steps in research in parallel systems.

My dissertation has benefited from the interaction with many of my fellow students at the department of Computer Science at Michigan State: Edgar Kalns, Ron Sass, Hong Xu, Jon Engelsma, Steve Walsh, Joe Sharnowski, Bill Ryan, Isaac Tsai, Chengchang Huang, Eric Kasten, Dan Judd, Marwan Krunz.

I am also very grateful to Ms. Lora Mae Higbee for all her opportune help regarding departmental and university procedures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Computers have become an essential tool for researchers in many different sciences. Programs are being used to model various physical phenomena, forecast the behavior of natural and artificial systems, and explore different design options in manufacturing. An insatiable demand for more computational capacity characterizes many of these applications. Many of the most demanding of such applications are numerical algorithms from fields of study that include computational chemistry, structural biology, fluid and combustion dynamics, petroleum reservoir and groundwater modeling, and high-energy physics.

Solving large numerical applications has traditionally been one of the tasks best suited for supercomputers. Supercomputing has undergone many changes over the years, however. Pushing aside traditional vector supercomputer architectures, parallel processing, which can greatly decrease the time required to solve many problems, has begun to dominate the supercomputer industry [1].

Scalable parallel architectures are designed to offer corresponding increases in performance as the number of processors is increased. Scalable computing platforms include both *massively parallel computers* (MPCs) and collections or *clusters* of high-performance workstations interconnected by high-speed networks. Although they are different architectures, both platforms are scalable because they share the "distributed

memory" property. In MPCs, memory is distributed among an ensemble of *nodes*, which are often interconnected by a point-to-point, or *direct* network; nodes communicate by passing messages though the network [2]. In clusters, memory is naturally distributed among the autonomous workstations. As new networking technologies mature, the difference in communication latency between distributed-memory multiprocessors and networks of workstations is diminishing. Hence, clusters are considered promising platforms to compete with MPCs in solving computationally intensive applications [3].

In distributed-memory computing environments, the efficiency of communication is critical to performance. Even if the computational workload is distributed evenly among the processors, communication overhead can severely limit performance. Minimizing communication overhead, in turn, requires efficient transfer of data among memory modules and efficient coordination of processors. In numerical scientific computing, this problem typically involves the decomposition and alignment of arrays among local memories, sending of partial results from one node to several other nodes, implementation of synchronization points (barriers), and communication needed in methods used to balance the computation load on processors.

Efficient implementation of such communication operations depends on the underlying communications architecture. However, a wide variety of communications platforms are currently in use for parallel computing. Not only is there little consensus among vendors regarding MPC interconnections, but cluster computing, by its very nature, must accommodate heterogeneous networking components. In the past, such architectural diversity has implied that a new version of a particular algorithm had to be developed for each new architecture. Hence, an increasing amount of recent research is addressing communications libraries for parallel computing. The advantages of libraries include increased portability, better software modularity, and less redundancy among different projects. In addition, use of libraries can lead to

improved performance because the systems programmer who implements the library is usually more familiar with the architecture and therefore more likely to exploit its characteristics than is the applications programmer.

The operations found in typical communications libraries include not only *point-to-point* primitives, which involve a single source and single destination, but also *collective* communication primitives, which involve more than two processes. Collective communication, also known as *group communication*, is receiving increasing attention due a better understanding of its wide applicability in parallel processing. Examples of collective communication primitives include *multicast*, in which the same data is sent to multiple destinations; *scatter*, in which different data is sent to multiple destinations; and *reduction*, in which a commutative and associative operation, such as **max** or **sum**, is performed on data that resides at different nodes. Barrier synchronization, the most popular coordination mechanism in parallel programming, is also an example of collective communication.

The development and evolution of numerical libraries has been ongoing for many years. Beginning nearly 20 years ago, numerical libraries have been written for specific problem domains, such as eigenvalue problems and linear systems of equations. In fact, numerical libraries have become useful in many different application fields. Some libraries are produced by computer manufacturers and are optimized for specific architectures; others are in the public domain and are designed to be portable across multiple computing platforms. In fact, new versions of libraries are designed to execute on different distributed-memory architectures. Since the primary goal of many such efforts is portability, the software uses de facto standard communication libraries that are not optimized for specific architectures. While these libraries sometimes offer interfaces to collective operations, those operations may actually be implemented inefficiently with simple unicast communication primitives.

Thus has arisen the need for the research described in this dissertation: to study the effects of interprocess communication costs, particularly that of collective operations, on the design, implementation, and performance of parallel numerical algorithms. The thesis can be stated as: The use of collective communication operations, designed to exploit properties of new generation communications architectures for distributed-memory platforms, can significantly improve the performance of parallel numerical algorithms; moreover, the redesign of such algorithms to account for point-to-point and collective communication costs explicitly, can result in further performance improvement. This dissertation makes the following contributions:

1. Compares the performance of MPC and cluster environments, the effects of communication costs in each, and methods to accommodate them in numerical algorithm design.

2. Measures and models the relative effects of optimized collective operations, particularly multicast and broadcast, versus unicast implementations, with particular emphasis on the scale of the system.

3. Studies the use of point-to-point and collective communication designed specifically for new generation communications architectures, namely, wormhole-routed networks and switch-based LANs.

4. Produces case studies for particular highly parallel numerical linear algebra algorithms that have recently been designed, which will result not only in an understanding of how collective operations affect their performance, but also in efficient implementations of those algorithms for use in numerical libraries.

The remainder of this dissertation is structured as follows. Chapter 2 presents background material on the areas that motivate the proposed research. Chapter 3 describes the implementation of a parallel eigenvalue algorithm on an MPC. The

experiences and results obtained in porting the same program to a cluster of workstations are described in chapter 4. Chapter 5 contains the results of experiments with the same program conducted on switch-based clusters and with different broadcast alternatives. A model of the performance of this program is presented in chapter 6. In chapter 7, a second study case is presented: A new algorithm for singular value decomposition (SVD) was also implemented on an MPC and a cluster platform. Chapter 8 summarizes the dissertation.

CHAPTER 2

Motivations and Related Work

This chapter is intended to describe the state of parallel computing on distributed-memory platforms as it relates to the issues addressed in the dissertation. Six topics are covered: the wide variety of communication architectures for MPCs, emerging designs for workstation clusters, collective communication algorithms, programming support for parallel computing on distributed memory platforms, the evolution of numerical libraries, and the analysis of the effects of collective communications on the performance of different algorithms.

2.1 Communication Architectures for MPCs

Communication architectures of MPCs are characterized by several parameters, including topology, routing, switching, port model, and startup latency [2]. The *topology* of a network defines how the nodes are interconnected by channels. *Routing* determines the path selected by a packet in order to reach its destination. *Flow control* deals with the allocation of channels and buffers to a packet as it travels along a path through the network, while *switching* is the mechanism that removes data from an one channel and places it on another channel along the path. In some systems, communication-related tasks are handled by a separate *router* at each node; the *port*

model of a system refers the number of communication channels connecting the local processor to the router. *Start-up latency* is the system call time required for handling of the message at both the source and destination nodes.

Several researchers have concentrated on the problem of determining the most appropriate topology given certain communication parameters and under certain assumptions about data traffic. Agarwal [4] presents a mathematical model of the performance of interconnection networks and validates his model with simulations. He concludes that a 3D-mesh is the best topology. Dally analyzes the performance of k-ary n-cube networks that use wormhole routing in [5] and concludes that low-dimensional networks offer better performance.

While there has been some consensus on the solutions to some of these design issues, there has been little or no consensus on others. The large number of feasible combinations of these factors has led to a wide variety of parallel communication architectures, which in turn has hindered the progress towards portable parallel programming and motivated the use of communication libraries.

Many MPC systems use *direct network* architectures, in which each node has a point-to-point, or *direct*, connection to some number of other nodes, called neighboring nodes. The popularity of direct networks stems from their ability to scale well, that is, as the number of nodes in the system increases, the total communication bandwidth of the network also increases [2].

Although the topologies of commercial MPCs vary, many of them are special cases of either n-dimensional meshes or k-ary n-cubes. These classes of topologies, which include hypercubes, meshes, and tori as special cases, are popular in part because they lend themselves to very simple routing algorithms. In recent years, low-dimensional meshes and tori have attracted larger followings than hypercubes, in part due to their simpler physical layouts and better scalability [2].

A notable exception to mesh-based direct networks is an indirect network called a *fat tree* [6], which is also claimed to scale well with simple routing. Figure 2.1. depicts several MPC topologies.

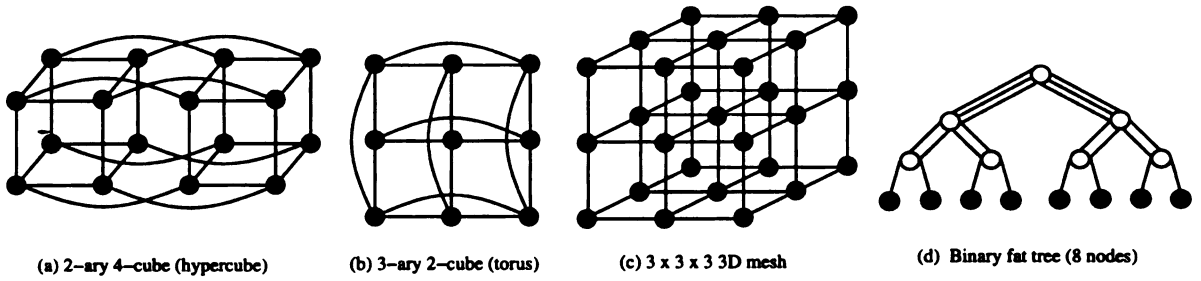


Figure 2.1. Examples of MPC topologies (Taken with permission from[2]).

At the present time, routing in most commercial MPCs is deterministic, that is, the path followed by a message is determined solely by the the source and destination addresses. In meshes and tori, the routing method most commonly used is *dimension-ordered routing*, in which a message is forwarded through dimensions of the topology in strictly ascending (alternatively, descending) order. Research in adaptive routing algorithms, which improve performance by accounting for current network conditions, is very active [7, 8]. Generally, however, those results are too recent to have infiltrated commercial designs at this time.

The switching techniques employed in distributed-memory computers have evolved over the years. Early systems used *store-and-forward* switching, whereby each intermediate node completely received and stored a packet before forwarding it along the path to its destination. Subsequent machines used circuit switching [9], where a physical circuit is established between the source and destination nodes; the circuit is reserved exclusively for traffic between the source and the destination.

Many new MPCs use the wormhole routing switching strategy [10], in which a packet is divided into *flits* (flow control digits). The route is determined by the header flit, and the remaining flits follow the head flit in pipelined fashion. Using wormhole routing, communication latency is nearly distance-insensitive in lightly-loaded networks [2]. Furthermore, since blocked messages remain *in the network*, only very small buffers are required at the routers. Virtual channels [11] have been proposed to improve the performance of wormhole routing. In this approach, multiple channels, each with its own buffer, are multiplexed on each physical channel. The use of virtual channels reduces a situation that occurs in regular wormhole routing, where a channel is available for use by a message but the corresponding buffer is occupied by a second message that is waiting on a channel not needed by the first message [11].

Another parameter that affects the performance of a communication subsystem is the port model. Figure 2.2. depicts a typical node in a wormhole-routed network. The external channels interconnect routers in order to establish the network topology. The internal channels, or *ports*, provide the node with access to the network. The port model determines how many messages a node can send/receive simultaneously. This number may be of little consequence when a node is sending and receiving infrequent unicast messages. However, when a node participates in a collective operation, it is often required to send/receive several messages at nearly the same time. In these situations, the port model is critical to performance.

Finally, as communication speeds have increased and wormhole routing has minimized the effect of distance in the total communication cost, startup latency has become an important factor in communication cost. In some current systems, startup latency actually dominates the time needed to send a message across the network. Claims of much lower startup latencies in recently announced systems illustrate the importance placed on this parameter by manufacturers [12, 13].

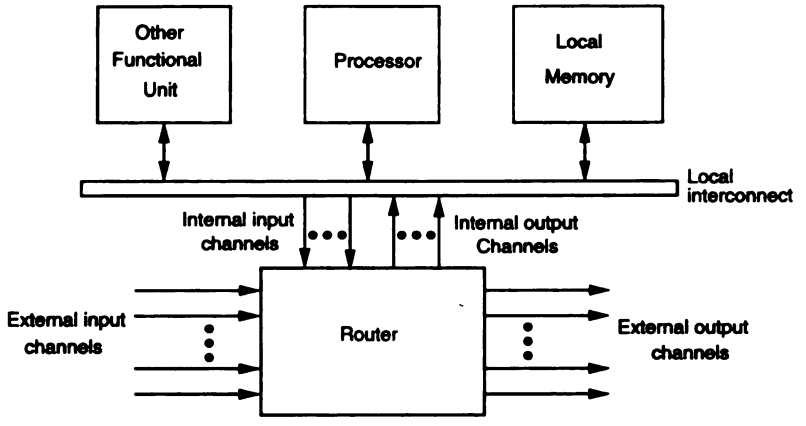


Figure 2.2. A node on a MPC (Taken with permission from[2]).

Several architectures have either found their way to the commercial market or have made significant contributions as research prototypes. Hypercubes are still being offered by nCUBE [14, 12], 2D-meshes are produced by Intel [15] and were produced by Symultek [16], 3D-meshes are used in the Cray T3D [17] and on the experimental J-machine [13], and fat trees are used on the CM-5 [6]. Some manufacturers are offering machines that provide hardware support for distributed shared memory, as the Kendall Square's KSR-1 and Convex Exemplar system. Most recent machines use microprocessors designed by workstation manufacturers.

The wide variety of MPC architectures illustrates the lack of a single solution to providing efficient communication primitives. This research explores the effects of new collective communication implementations, optimized to exploit specific architectural properties, on the design of numerical algorithms.

2.2 Clusters of Workstations

As an alternative to MPCs, there has recently emerged increasing interest in using clusters of workstations for parallel scientific computing. The reasons for the popularity of these so-called "distributed supercomputers" are several [18]. First, clusters are often more economical than either traditional vector-based supercomputers or MPCs. Second, the memory capacity of each workstation is typically much greater than that of an MPC node, allowing large problem instances to be addressed using simpler programming methods. Third, the I/O capacity of the system is larger than that of an MPC because each workstation has its own disks and monitor. Fourth, a cluster is more flexible than an MPC: additional computing and communication capacity, in the form of new workstation models and faster networks, can be easily configured into the system. Finally, clusters can be used for parallel computing at the same time that they are used as workstations to meet the computing needs of individuals. The IBM SP1 [19] is an example of this kind of systems. IBM RS/6000's are stacked in frames and interconnected by one or more high-speed networks.

However, such sharing of resources (both processors and communication links) can also be considered a drawback of clusters, since it is likely to reduce the performance of a particular application. Another disadvantage of clusters is that communication latency is often much longer than in an MPC, particularly if conventional networking technology is employed. New high-speed local area networks (LANs) are being used to address this problem.

A major issue in the design of cluster systems software, such as communication libraries, concerns portability versus performance. Communication latency depends on the physical network architecture, network protocols and their implementations, and the architecture and software of the network interface. Implementing workstation clusters on top of Ethernet LANs with communication primitives based on $UNIX^{TM}$

sockets and TCP/IP is quite common due to the ubiquity of such environments. However, shared physical media and inefficient protocol implementations can produce interprocess communication latencies that preclude effective execution of many applications that are communication-intensive.

In order to reduce the difference in communication latency between MPCs and clusters, higher-speed networks, more efficient protocol implementations and streamlined communications interfaces are needed. Many sites have already installed 100 Mbps FDDI (Fiber Distributed Data Interface) rings. Recently, several switch-based interconnects have been proposed. These include the High Performance Parallel Interface (HiPPI) standard and the Fibre Channel Standard (FCS) to interconnect supercomputers and workstations, respectively. The nodes on an IBM SP1 can be connected using a high performance switch that uses a MIN design. In addition, fiber-optic LANs using Asynchronous Transfer Mode (ATM) protocols [20] are now commercially available [21]. Although ATM was primarily designed for future Broadband Integrated Services Digital Networks (B-ISDN), the high data transfer rates and interoperability of standardized protocols makes it attractive for use in high-speed LANs. A major advantage of switch-based interconnects over shared media is that they can provide an enormous aggregate bandwidth because multiple packets can simultaneously be passed through the switch at the full channel rate.

Figure 2.3 shows a typical cluster configuration; the workstations are interconnected by both a switch (for parallel computing) and a regular LAN (for traditional networking).

Several tools that are employed to use a cluster of workstations as a platform for parallel computing are based on TCP/IP protocols. Experience has shown that the traditional implementations of those protocols introduce a significant amount of overhead and that when the physical medium is replaced by a faster one, the increase in performance is not proportional to the increase in capacity at the physical

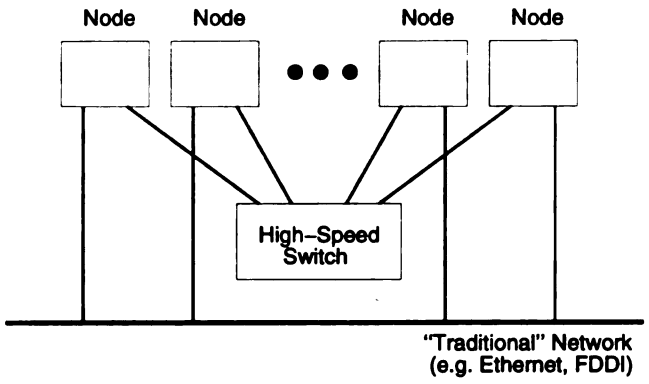


Figure 2.3. Diagram of a cluster of workstations.

level [22]. This problem is being attacked in several ways: better implementations of TCP/IP [23]; new network interfaces that avoid copying between the user and the kernel and then again to the network interface, by instead copying directly from the user space to the network interface [22]; and for the case of ATM networks, new libraries that bypass completely TCP/IP and use directly the ATM protocols while providing enhanced functionality [24]. The Nectar project [25], is pursuing several of these approaches simultaneously: new network boards and switches have been developed, as well as software for both the interfaces and the host workstations.

Several computer vendors have either recently brought cluster-based products to the market or have announced such systems for release in the near future. The commercial offerings came after several research projects produced tools that allowed existing groups of workstations, interconnected through some network, to be used as a single computer. That is, the research projects created software tools that stirred interest in the user community, which in turn led the manufacturers to offer new products. Many manufacturers include such software tools with their products. Clusters of workstations are being offered by DEC, IBM and Convex/Hewlett Packard. The clusters from these manufacturers can be interconnected via regular Ethernet or using proprietary high-speed switches.

Clearly, cluster platforms are being increasingly used for parallel scientific computing. The research so far has focused on taking advantage of the computing power of the machines involved and on taking advantage of faster networks. However, with the exception of multicast in Nectar, collective communication on clusters has not yet received significant attention. Many of these cluster environments can potentially support efficient collective operations, particularly data distribution operations such as multicast and scatter. For example, Ethernet-based systems support IP-multicast [26], and most switch-based LANs support some type of hardware multicast. We report the results of using both broadcasting alternatives.

2.3 Collective Communication

The advent and popularity of distributed-memory parallel platforms has produced increasing attention from the research community on efficient implementations of collective communication operations. This section discusses several of the most important collective operations and describes research efforts into their implementation on particular communication architectures. Since collective communication operations are included in the *Message Passing Interface* (MPI) standardization effort, which is discussed later.

Collective operations are often defined in terms of a group of processes. The group may constitute all or a subset of the processes in the parallel application. While the abstract quality of such definitions is useful for studying the semantics of operations, it may be difficult to study their performance, which depends on the physical relationships between the group and the system. For example, the number of processes per node and the distribution of processes in the network affect the performance of collective operations on the group. Therefore, in this dissertation, we will generally discuss a particular collective operation in terms of the physical

network architecture and the specific messages that constitute that operation. The distinction will become more clear in the following, which describes the functionality and possible uses of various collective communications operations.

Broadcast is perhaps the most fundamental collective communication primitive. In this operation, the same data is sent from one source node to all other nodes in the network. Broadcast is used in many parallel numerical algorithms, including matrix multiplication, Gaussian elimination, LU-factorization, and Householder transformations [27]. An even more common use of broadcast is at the initiation of most parallel algorithms; broadcast is used to send to all processes those input data and other parameters that are needed for the execution of the algorithm. As will be discussed in Chapter 6, broadcasts, if implemented inefficiently, can severely degrade scalability and performance.

Broadcast is actually a special case of *multicast*, in which the same data is sent from a source node to a subset of nodes in the network. Some confusion may arise here, because the MPI standard does not explicitly refer to multicast; rather, MPI would describe multicast as a *broadcast* to a set of processes that happen to reside on only a subset of the nodes in the network. We will discuss multicast in terms of sending to nodes, rather than processes. The applications of multicast are similar to those of broadcast; in fact, sending to a subset of nodes (a parallel application may only execute on a subset of nodes) is more common than sending to *all* the nodes in the network. Henceforth, we will use the term multicast, except for cases when the message is actually transmitted to all nodes in the network.

Other collective communication operations include: *multinode broadcast/multicast*, in which every node sends a piece of data to all members of the group, *scatter*, in which the source node distributes different data to each member of the group, *gather*, in which different data from all members of the group are collected by one member, *global exchange*, where every member of a group sends different data

to every other node in the group, *global* operations, which include both *reduction* and *scan* (also known as parallel prefix). In reduction, an associative and commutative operation is applied across data items from each member of the group. Examples include sum, max, min, bitwise operations, and so on. In a scan operation, a parallel prefix with respect to an associative and commutative operator is performed. Every process has a rank i, and the result of this operation returns to the calling process the value of the reduction of the data of processes 0 through i. Formally, given processes $p_1, p_2, \ldots, p_n$ and data items $d_1, d_2, \ldots, d_n$, an associative operator $\odot$ is applied such that the result at process p_i is $d_1 \odot d_2 \odot \ldots \odot d_i$.

The operations described thus far can be classified as *data movement* operations. Collective communication may also be used to implement *control* operations. An important example is *barrier synchronization.* A synchronization barrier is a logical point in the control flow of an algorithm at which all the members of a subset of the processes must arrive before any of the processes in the subset are allowed to proceed further. Barrier synchronization occurs frequently in programs for parallel applications, especially for those problems that can be solved by using iterative methods because it is useful in supporting parallel loop synchronization. In a distributed-memory environment, barrier synchronization must be implemented by a gather operation followed by a multicast operation [28]. Each node involved in the barrier sends a message to the *barrier processor.* Upon receiving all reduction messages, the barrier process must instruct all processes waiting on the barrier that they may proceed by multicasting a *distribution message* to them.

The communication characteristics of each combination of topology, port model, and switching technique are different, and collective communication service must be implemented differently in order to exploit those particular characteristics. Research has been conducted on different architectures. Most of the research to date on collective communication has addressed store-and-forward architectures; much of that

work is not relevant to new generation distributed-memory architectures, specifically, wormhole-routed systems and switch-based clusters. In the following, we concentrate primarily on multicast and broadcast operations.

Saad and Schultz [29] studied hypercube communications and developed algorithms for broadcasting, multinode broadcasting, scattering, and gathering. Although their work assumed a store-and-forward switching model, since they considered only full subcubes and nearest neighbor communication, much of these results pertain to wormhole-routed systems as well. Hillis and Steel [30] survey several algorithms that are used in the CM-2. In their survey, there are versions of both global operations and scan operations based on recursive doubling. Johnsson and Ho [31, 32, 27] have worked on the particular problems of broadcasting and "personalized communications"(one-to-all and all-to-all) in hypercubes, in both one-port and n-port models. They have presented optimal algorithms in the one-port model using only nearest neighbor communication. Their approach uses n edge-disjoint spanning trees of the hypercube; the message is partitioned into n segments, each of which is transmitted along a different spanning tree. The spanning trees are defined in such a way that there are no edge (channel) conflicts. Again, these algorithms can also be used effectively in wormhole-routed networks. Barnett, Payne, and Van de Geijn [33, 34] have studied both the broadcast and reduction operations in one-port wormhole-routed 2D meshes. The broadcast primitive embeds a minimum spanning tree in a mesh and uses it for broadcasting.

The more general problem of multicast (to an arbitrary subset of nodes) was originally studied for hypercubes and 2D-meshes by Lan, Esfahanian and Ni [35]. Their proposed distributed greedy heuristic algorithm, called LEN, was intended to be used to support multicast communication in hardware in virtual cut-through switched systems; it could also be used in software in store-and-forward networks. McKinley, Xu, Esfahanian, and Ni [36] developed software (unicast-based) multicast algorithms

for one-port wormhole-routed meshes and hypercubes. Their *U-cube* and *U-mesh* algorithms exploit the distance-insensitivity of wormhole routing, and are optimal for one-port architectures. Unicast-based collective communication for multiport systems has come under study recently. We [37] have proposed the *double-tree* (DT) algorithm a broadcast algorithm for hypercubes that conform to the all-port model. In the first step, the sender sends the message to the node that is farthest away, as well as to the neighboring nodes except one. By exploiting the distance insensitivity of wormhole routing, the DT algorithm requires only half the number of message-passing steps of that of the usual SBT algorithm. Robinson *et al* [38] have recently proposed a multicast algorithm for all-port wormhole-routed hypercubes.

Most of the approaches described above are designed for implementation in software; collective operations can also be implemented in hardware. For example, the CM-5 [6] control network supports broadcast and reduction, but only for small messages. Hardware-supported wormhole multicast can be based on trees similar to those described above. Tree-based routing is used to support broadcast and a restricted form of multicast in the nCUBE-2 [14], but is susceptible to deadlock. Lin *et al* [39] have developed a deadlock-free approach to hardware wormhole multicast, called *path-based* routing, in which each of multiple worms distributes the message to certain subsets of destinations.

Collective communication has been studied in networks as well, thereby affecting parallel computing on clusters. A significant amount of work has been done at the network layer of the IP protocol to support multicast routing of packets [26], and a multicast service is currently available on the IP protocol. Reliable multicast protocols built atop an underlying unreliable multicast service have been the subject of much research in the last several years. For example, Tanenbaum *et al* [40] have proposed a simple reliable multicast protocol to be used in local area networks. For message-passing programming, however, ordering may not be needed because either only one

message is multicast at a time or because messages can be distinguished by some type checking [14]. With the growing popularity of cluster-based parallel computing, distributed implementations of other collective operations are receiving attention. For example, Huang and McKinley [24] are presently studying implementations of collective operations on switch-based LANs.

We have investigated the effects on performance and scalability of different implementations of broadcast on the nCUBE-2. On a cluster environment, we have explored the effects of implementing multicast operations using IP-Multicast, a broadcast primitive available on the ATM environment [41], and an enhanced version of PVM.

2.4 Communication Libraries for Distributed-Memory Environments

In the past, the programmer of a distributed-memory system has invoked system primitives to send messages among processes executing on different nodes. While such low-level control over communication allows the user to exploit characteristics of the architecture, message-passing programming is usually tedious and error-prone. Furthermore, the increasing variety of distributed-memory parallel computing platforms, and particularly the emergence and rapid growth of cluster-based computing, can potentially require that a new version of a particular algorithm had to be developed for each new architecture.

One method that has been used to address these problems is to construct communication libraries. Libraries hide the details of the underlying architecture and vendor-specific interfaces from the user but provide a common interface across multiple platforms, permitting user code to be more easily ported among machines. The correctness of the library routines can be verified independently of the applications

using them, thus shortening the application development cycle. In addition, use of libraries can lead to improved performance because the systems programmer who implements the library is generally more familiar with the architecture than is the applications programmer.

A variety of research groups and commercial companies offer communication libraries for distributed-memory environments. Some of them started as programming facilities for multicomputers and have been ported later to cluster environments. Others have migrated in the opposite direction. PVM [42] and P4 [43] are widely used public domain tools available from national laboratories. Another library for distributed-memory platforms, being developed locally at Michigan State University, is ComPaSS [44]. Many communication libraries other than those mentioned above have been developed, including the CHIMP project [45], Zipcode [46], Express [47, 48], nCUBE's Vertex [14], and ORNL's PICL [49] and Linda [50].

The use of data parallel languages offers many advantages over message-passing programming, and may represent the long-term direction for parallel programming. However, despite its drawbacks, message passing *is* used for parallel programming, and will likely continue to be used for the foreseeable future. As shown above, a wide variety of communications libraries have been produced. Many of the organizations involved in these efforts have recognized the need for portability, and several of the libraries have been ported to multiple platforms. These events indicate that the time has come for standardization of message passing operations.

Recently, a volunteer group has been organized to develop the *Message Passing Interface (MPI)* [51], which represents the first effort to bring about such a standard. The MPI forum seeks to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI has been strongly influenced by PVM, Express, Vertex, P4, and other libraries.

The research described in this document complements ongoing research and development of communications libraries. We use some of the existing tools and evaluate the performance of two implementations of numerical algorithms on top of those tools. Furthermore, we investigate alternative implementations of multicast operations and their effect on performance of the implementations of the algorithms.

2.5 Libraries for Numerical Analysis

Problems in numerical analysis are, usually, clearly defined, and the specific circumstances in which those problems are solved are relatively standard. That is, there exists a relatively small set of variations on a given problem that are commonly used, and those variations can be accounted for in standard methods. These properties, along with the advantages of modular software, have led to the development of libraries of methods for solving various numerical problems. In fact, such numerical libraries have become useful in many different fields.

Numerical libraries are available from several sources. For example, computer manufacturers often provide libraries that are optimized for their products. Software houses that specialize in libraries for specific domains have been founded; examples include Numerical Algorithms Group Limited (NAG) and International Mathematical and Statistical Libraries (IMSL). Other libraries are in the public domain and are designed to be to be portable across multiple computing platforms. The collected algorithms of the ACM are an early example of this approach.

Numerical libraries have been written for specific domains. For example, EISPACK [52] was written to solve eigenvalue problems. Routines in EISPACK find the eigenvalues and the eigenvectors for several classes of matrices: complex general, complex hermitian, real general, real symmetric, real symmetric banded, and real symmetric tridiagonal. Other routines can solve the generalized eigenproblem of

symmetric and non-symmetric real matrices and the singular value decomposition of an arbitrary matrix. LINPACK [53] is intended to solve systems of linear equations and to perform matrix factorizations. It was developed, in Fortran, from 1976 to 1979. The routines in this library are used to solve linear systems for several classes of matrices: general, banded, symmetric indefinite, symmetric positive definite, and triangular. Two of the LINPACK routines (one that factors a matrix and one that solves a system of equations) are used for benchmarking high performance computers [54]. LAPACK [55] integrates and refines most of the procedures available in EISPACK and LINPACK. The routines in LAPACK solve systems of linear equations, linear least squares, eigenvalue and singular value problems. There are also routines for estimating condition numbers and matrix factorizations. The routines are provided for banded and dense matrices and for real and complex matrices. For nonlinear equations and nonlinear least squares problems, MINPACK [56] is used. QUADPACK [57] is utilized in the domain of numerical quadrature (numerical integration). Several other packages are designed for other specific problem domains: *fishpack*, for separable elliptic partial differential equations; *fitpack*, for splines under tension; *fftpack*, for fast fourier transforms; *hompack*, for nonlinear equations by homotopy method; *itpack*, for iterative linear system solutions; and *odepack*, for ordinary differential equations.

Developers of these libraries, LINPACK and LAPACK in particular, have found that many primitives are used repeatedly while implementing a library. This in turn has led to the creation of lower level libraries, which contain primitive operations for particular domains. For the specific case of linear algebra, a set of low level routines called BLAS (Basic Linear Algebra Subroutines), has been developed. Three different levels (1, 2, and 3) of BLAS routines have been written, containing functions that perform vector-vector, vector-matrix and matrix-matrix operations, respectively [58, 59, 60]. Efficient implementation of the BLAS routines helps to

obtain good performance on specific machines. For example, the BLAS routines have been specially compiled for the Intel i860 and are available for programs that run on the Intel Paragon [15].

As massively parallel computers and networks of workstations become more widely used for solving numerical problems, it becomes necessary to port the existing libraries to new machines. A project called ScaLAPACK [61] is targeted to producing a distributed-memory version of LAPACK. Currently, a subset of the routines in LAPACK (Cholesky decomposition, QR, LU, and Hessenberg and tridiagonal reduction for the algebraic eigenvalue problem) has been ported to distributed-memory parallel computers [62]. In the current version of ScaLAPACK, square block-scattered decomposition of the matrices over the processors is used. The block size and the size of the rectangle sides are parameters that the user can adjust. There is a beta version of ScaLAPACK available which can run on Paragon and CM-5 machines as well as on clusters of workstations that use PVM. This beta version is implemented in Fortran 77. The different architectures and programming paradigms found in parallel computers create the need for other low-level libraries or sets of macros that facilitate the porting of the existing libraries to the new environments. In ScaLAPACK, simplicity is maintained by using distributed versions of the BLAS routines, contained in a library called PB-BLAS. PB-BLAS uses the communications services offered by a communication library called BLACS and calls BLAS routines for the local operations. BLACS (Basic Linear Algebra Communication Subprograms) [63], is designed to provide basic matrix-related communications operations for MIMD message-passing machines. The routines offer communication primitives and global operators for general rectangular matrices and trapezoidal matrices. The four communication operations are: send a message, receive a message, broadcast a message, and receive a broadcast message. Three global operators are provided: SUM, MAX,

and MIN. BLACS has been implemented atop at least three lower level communication packages: the Intel communication primitives [64], PVM [65], and the CM-5 communication primitives [6]. An object-oriented programming approach is being pursued in ScaLAPACK++ [66], a version of ScaLAPACK designed to be callable from C++. Matrices are treated as objects in ScaLAPACK++. Figure 2.5 depicts graphically the evolution and relationships among the numerical libraries mentioned above.

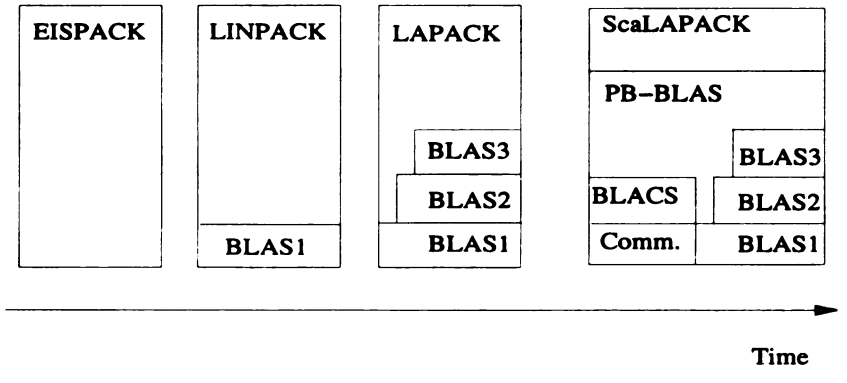


Figure 2.4. Libraries and their structure.

With the migration from traditional supercomputers to distributed-memory platforms comes the need to port existing libraries (or variations of them) to the new environments. The conversion should be made with portability and efficiency as objectives, and this in turn suggests the use of communication libraries with a well-defined set of primitives, carefully optimized for each environment. Our experiences in implementing different numerical algorithms show that performance depends heavily on efficient implementation of the communication primitives.

The new algorithms that were parallelized as part of this research might become in the future part of a numerical library. In the same vein, the parallel versions could, after further refinement, become part of a library like ScaLAPACK.

2.6 Analysis of Collective Communication Costs

Some aspects of the effect of communications on the performance of parallel algorithms have been studied previously. The research projects reviewed here fall into three categories: general analysis of families of algorithms; analysis of particular numerical algorithms; and analysis of the effect of communication on scalability. Each area is discussed in turn.

2.6.1 General Analysis

Scherson and Corbett [67] examine the effect of communications on the expected speedup of applications on multidimensional meshes. They provide expressions for the effect of communication on the maximum speedup of an application under the assumption of uniform distribution of communications. Their model assumes that communication time is directly proportional to the distance between the communicating processes, an assumption which has become less important with the advent of wormhole routing. In contrast, our analysis considers the characteristics of both wormhole-routed systems and clusters, and is based on case studies of specific numerical algorithms. Furthermore, our analysis emphasizes the effects of different implementations of collective communications, whereas their work is concerned with communications in general without distinguishing between point-to-point and collective communications.

2.6.2 Analysis of Specific Numerical Algorithms

Gannon and van Rosendale [68] examined the effects of communications on three numerical algorithms: FFTs, the solution of tridiagonal systems of equations, and the solution of PDEs. Their analysis covers both shared-memory and distributed-memory systems, specifically, multistage interconnection networks (MINs) and direct networks. They calculate limits on the performance of the algorithms as a function of the communication costs. Their model assumes store-and-forward switching, which was prevalent at the time of the research. Our analysis is based on different algorithms, and considers new generation architectures. We do not consider shared-memory systems since these are generally not considered to scale well.

Brochard [69] studied and modeled the (point-to-point) communication and synchronization costs of several basic matrix and vector operations, an iterative method for solving partial differential equations, and a multigrid method. Brochard implemented the algorithms on an iPSC hypercube and compared the experimental results with the models. Johnsson [70] has analyzed the performance of FFT and BLAS 2 routines for a library for data parallel languages and has studied the most appropriate collective communications, available on the Connection Machine, for those routines. His research was conducted in the context of a SIMD system, a CM-2, which was based on a hypercube topology. Dongarra and van de Geijn [71] implemented a parallel reduction to Hessenberg form in a 2D-mesh(Paragon). They studied the amount of time spent in communications and observed that a significant percentage of the overall execution time is spent in all-to-all broadcast operations. Although they analyzed the composition of the communication overhead, they did not explore the impact of different broadcast alternatives. In our work, we also validate our model against experimental results, but, again, we focus on different algorithms and the costs of collective communication operations.

2.6.3 Scalability Analysis

Amdahl's law [72] states that the sequential fraction of an algorithm represents a limit to its scalability. Let f be the fraction of an algorithm that has to be performed sequentially, where $0 \leq f \leq 1$. Then the maximum speedup S achievable by a machine with p processors is:

$$S \leq \frac{1}{f+(1-f)/p}$$

Over the years, various other models have been developed to address the scalability of parallel algorithms, which account for properties not handled in Amdahl's law. Gustafson, Montry and Benner [73] implemented several applications on a 1K-node nCUBE-2. They realized that on a large numbers of processors, it is possible to solve larger problems than what would have been possible on a single processor or on a small number of processors. When using a large number of processors, they scale the total size of the problem, keeping constant the size of the subproblem assigned to every processor. This alternative view of speedup is called *scaled speedup.* More recently, Sun and Ni [74] realized that memory on the nodes of an MPC can be a limiting factor on the size of the problems that are solvable. Based on their experiences in using multicomputers, they developed the *memory bounded speedup* model, which takes into account the inherent parallelism of the application, the computation power and the memory capacity of a given MPC. Worley [75], based on experiments with partial differential equations solvers, observed that if one has a time constrain, then, for certain problems, it might not be feasible just to scale the size of the problem up to use efficiently a larger number of processors, as the execution time would be larger than the given limit. The size of other problems, though, can be increased without violating the time constrain.

Kumar and Rao [76] proposed a scalability metric, called the *isoefficiency function.* This function relates the size of the problem, W, to the number of processing elements,

p, in a parallel system. They define T_0 as the total parallel overhead. If the problem size is kept fixed as p increases, T_0 grows as well, and the efficiency decreases. To maintain a constant efficiency as p increases, W must grow at a certain rate. The isoefficiency function is the rate at which W must grow, in terms of p, in order to keep the efficiency constant. The function varies from one algorithm to another and from one architecture to another. T_0 includes the communication overhead; hence, the efficiency of the communications is certain to affect the isoefficiency function. The isoefficiency models for workstation clusters have assumed that the cost of a broadcast operation is linear on the number of processors involved. If the cost becomes constant or logarithmic, the isoefficiency of a given algorithm on clusters is likely to improve. This is one of the results of this research: In the new switch-based clusters of workstations, it is feasible to use broadcast functions that are logarithmic on the number of participating nodes. The scalability of such systems improves with this result.

The work by Xu, McKinley and Ni [28] deserves a special mention as it inspired some of the research reported here. They report the design and implementation of an efficient barrier synchronization primitive on an nCUBE-2. Their experiments compare 2 different implementations of barrier synchronization, one based on separate addressing and the other one based on their U-cube tree algorithm and indicate the better scalability of the program that uses the U-cube based version. The research reported here is not based on an artificial workload but on an actual algorithm and the emphasis is not on barrier synchronization but on broadcast and load balancing. Also the work reported here includes experiments on clusters of workstations. This research, as theirs, emphasizes the importance of efficient implementations of collective communications in the scalability of given programs on actual systems.

CHAPTER 3

A Parallel Eigenvalue Solver on an MPC

As quantitative analysis becomes increasingly important in the sciences and engineering, the need grows for faster and more efficient methods to solve eigenvalue problems. Large eigenvalue problems occur in a wide variety of applications, including the dynamic analysis of large-scale structures such as aircraft and ships, prediction of structural responses in solid and soil mechanics, the study of solar convection, modal analysis of electronic circuits, and the statistical analysis of data. Solving for the eigenvalues of large systems is a computationally-intensive task that may need to be carried out many times within a particular application; reducing its execution time will improve the performance of the application.

The research described in this dissertation has its roots in a project to parallelize a new algorithm for finding the eigenvalues of real symmetric tridiagonal matrices. The *split-merge* eigenvalue algorithm, proposed by Li and Zeng [77], has a high potential for successful parallelization while preserving the accuracy and stability found in other algorithms. Nevertheless, realizing the potential parallelism of the algorithm has illuminated many issues regarding the effects of communication in parallel numerical algorithms. The split-merge algorithm has been implemented and studied in two

different environments: an nCUBE-2 hypercube multicomputer and a cluster of Sparc-10 workstations interconnected via Ethernet and via an Asynchronous Transfer Mode (ATM) switch. This chapter describes the results obtained in the implementation of the algorithm on the nCUBE-2 and describes the impact of collective communications on its efficiency and speed on this particular environment.

3.1 Split-Merge Algorithm

The problem of finding the eigenvalues of a matrix can be stated as follows: Find the values λ that satisfy the equation: $A\mathbf{x} = \lambda\mathbf{x}$ for a vector $\mathbf{x}$, which is called an eigenvector. The values λ are eigenvalues. The problem can be rewritten as follows: Given $[A - \lambda I]\mathbf{x} = 0$, solve $f(x) = det[A - \lambda I] = 0$. Symmetric tridiagonal (ST) matrices have the form shown in Figure 3.1(a). All nonzero entries occur on either the main diagonal, the superdiagonal, or the subdiagonal. Furthermore, the superdiagonal is identical to the subdiagonal.

Finding the eigenvalues of symmetric matrices is a very common problem in many different fields. A procedure frequently used to solve this problem is to reduce the original "full" matrix to a tridiagonal matrix (reduced or tridiagonal form). This is achieved by premultiplying the matrix by an orthogonal matrix U and postmultiplying it by U^T. The matrix U is chosen so that it introduces 0s in the original matrix except in the main diagonal, the superdiagonal and the subdiagonal. The eigenvalues of the original matrix and the reduced matrix are the same. Thus, finding the eigenvalues of a symmetric tridiagonal matrix is crucial in the process of finding the eigenvalues of symmetric matrices.

As one of the most fundamental problems of computational mathematics, the symmetric tridiagonal eigenvalue problem continues to receive considerable attention in the literature due to its wide applicability.

The new *split-merge algorithm* was designed by Li and Zeng [77] originally for shared-memory parallel architectures. This algorithm is inherently parallel and takes advantage of a fast iteration technique, namely, Laguerre's method [78](pp. 263-266).

$$A = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & & \\ & & & \cdots & \\ & & & & \beta_n \\ & & & \beta_n & \alpha_n \end{bmatrix} \qquad A' = \left[\begin{array}{cc|cc} & & & \\ & A_1 & & 0 \\ & & & \\ \hline & & & \\ & 0 & & A_2 \\ & & & \end{array}\right]$$

(a) symmetric tridiagonal matrix (b) use of separation property

Figure 3.1. Matrices in split-merge eigenvalue solver

The split-merge algorithm relies on the so-called *separation property* [79] in order to find the eigenvalues of ST matrices. Given an ST matrix A with nonzero subdiagonal elements, this technique uses the matrix A' produced by replacing some β_i with 0, as shown in Figure 3.1(b). Let $\lambda_1^0 \leq \lambda_2^0 \leq ... \leq \lambda_n^0$ be the eigenvalues of the submatrices A_1 and A_2, and let $\lambda_1 < \lambda_2 < ... < \lambda_n$ be the eigenvalues of A. The separation property states that $\lambda_{i-1}^0 < \lambda_i < \lambda_{i+1}^0$ and $\lambda_{i-1} < \lambda_i^0 < \lambda_{i+1}$ for all values of i.

In the split-merge algorithm, the separation property is used as follows: the eigenvalues of matrices A_1 and A_2 are found and then used as the initial approximations to the eigenvalues of matrix A. The process is applied recursively until matrices of sizes 2×2 are reached, for which eigenvalues may be found easily. Without loss of generality, assume that n, the order of the matrix, is a power of 2; specifically, let $n = 2^k$. The algorithm proceeds in a series of k stages. In the first stage, eigenvalues for each of the 2×2 arrays are found using the quadratic formula. Results from pairs

of neighboring subarrays are merged, sorted, and used in solving 4×4 arrays in the second stage. The same procedure is repeated for the following stages: At stage i, eigenvalues are found for 2^{k-i} submatrices of size 2^i, using the results of stage $i-1$ as initial approximations.

In order to find an individual eigenvalue from an initial approximation, the split-and-merge algorithm uses Laguerre's method [78](pp. 263-266) for finding the zeroes of a polynomial with real and simple zeros. This method has cubic convergence. Given an initial approximation x to a zero of the polynomial $f(x)$, a better approximation is obtained by:

$$L_{\pm}(x) = x + \frac{n}{(-\frac{f'(x)}{f(x)}) \pm \sqrt{(n-1)[(n-1)(-\frac{f'(x)}{f(x)})^2 - n(\frac{f''(x)}{f(x)})]}},$$

where f,f', and f'' are calculated using three-term recurrence equations. Li and Zeng [77] developed an alternative scheme to calculate the quotients $\frac{f'}{f}$, and $\frac{f''}{f}$ that avoids underflow-overflow problems. The $\pm$ in the $L_{\pm}(x)$ expression denotes the fact that two different values are obtained depending on which operation is performed in the expression in the denominator. The appropriate value is chosen based on the situation of the approximation with respect to the eigenvalue being calculated. Sturm's sequence evaluation [80] can be used to determine if the current approximation is smaller or bigger than the actual eigenvalue.

The split-and-merge algorithm, which is described in detail in [77], operates as shown in Figure 3.2.

The split-merge algorithm can be viewed as a "tree" of tasks, as shown in Figure 3.3. At the leaves of the tree, the eigenvalues for the 2×2 submatrices are found. These results are merge-sorted in a pairwise fashion, and in the next stage, the eigenvalues for the 4×4 submatrices are found. This process continues until the eigenvalues for the original matrix are produced at the root of the tree.

Algorithm 1: Split-Merge
Input: Symmetric tridiagonal matrix A of order 2^k
Output: The eigenvalues of A
Procedure:

```
begin
  find eigenvalues for the 2^{k-1} 2 × 2 submatrices of A
  for i = 2 to k
    for j = 1 to 2^{k-i}
      combine submatrices 2j − 1 and 2j of order 2^{(i-1)} into one matrix of size 2^i
        (that is, merge their eigenvalues)
      for ℓ = 1 to 2^i
        use Laguerre iteration to find λ_ℓ for the jth submatrix of order 2^i
      endfor
    endfor
  endfor
end
```

Figure 3.2. Split-merge algorithm.

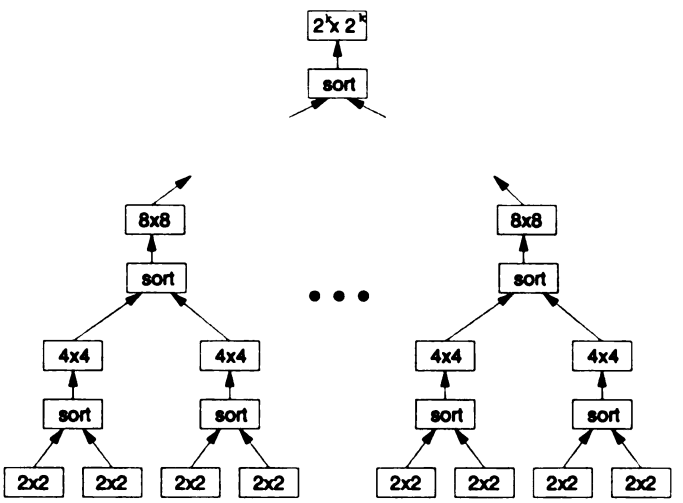
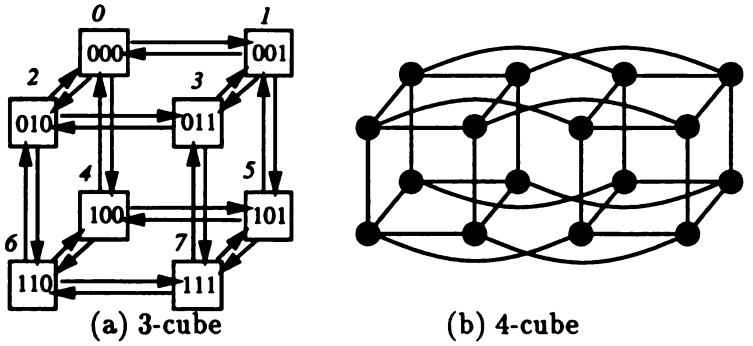


Figure 3.3. Abstract representation of the split-merge algorithm

Although the split-merge algorithm can be logically viewed as a tree, the nodes in this tree clearly should not also represent processors in a parallel implementation. If this were the case, for example, a single node would be solely responsible for the last stage of the algorithm, severely reducing the parallelism of the algorithm. In parallelizing the split-merge algorithm, a key objective is to maximize the fraction of time that each processor is busy solving for eigenvalues in each of the k stages.

The algorithm was implemented initially on an nCUBE-2, a wormhole-routed parallel system with a hypercube topology. Formally, a *hypercube* (or n-cube) consists of 2^n nodes, each of which has a unique n-bit binary address. For each node v, let v also denote its n-bit binary address and $\|v\|$ represent the number of 1s in v. A channel $c = (u, v)$ is present in an n-cube if and only if $\|u \oplus v\| = 1$, where $\oplus$ is the bitwise exclusive OR operation on binary numbers. Figure 3.4(a) depicts a 3-cube; notice that adjacent nodes are connected by two unidirectional channels in opposite directions. Figure 3.4(b) shows a more abstract representation of a 4-cube. On an nCUBE-2, a user is allowed to request a subcube of the hypercube to execute his/her program. For purposes of discussion, let 2^d be the number of processors being used for the execution of the program, and let 2^k be the size of input matrix.



(a) 3-cube (b) 4-cube

Figure 3.4. Different representations of hypercubes

3.2 Simple Parallel Implementation

In our initial parallel approach, the responsibility for solving for eigenvalues is divided evenly among processors, that is, at all stages, every processor is responsible for finding 2^{k-d} eigenvalues. In the first $k-d$ stages, no communication is necessary between processors, since each submatrix is small enough to be handled by a single processor. Specifically, each node i initially performs the first $k-d$ stages independently, thereby solving for the 2^{k-d} eigenvalues of the i^{th} (numbering from left to right) submatrix of order $k-d$. In the last d stages, nodes must communicate their results to other nodes to be merged, sorted, and the appropriate sets of eigenvalues returned to be used as input to the next stage. The processing elements (nodes) are divided in two categories: sinks and clients. Henceforth, the first $k-d$ stages of the algorithm will be referred to as the *local* stages, and the last d stages will be referred to as the *distributed* stages.

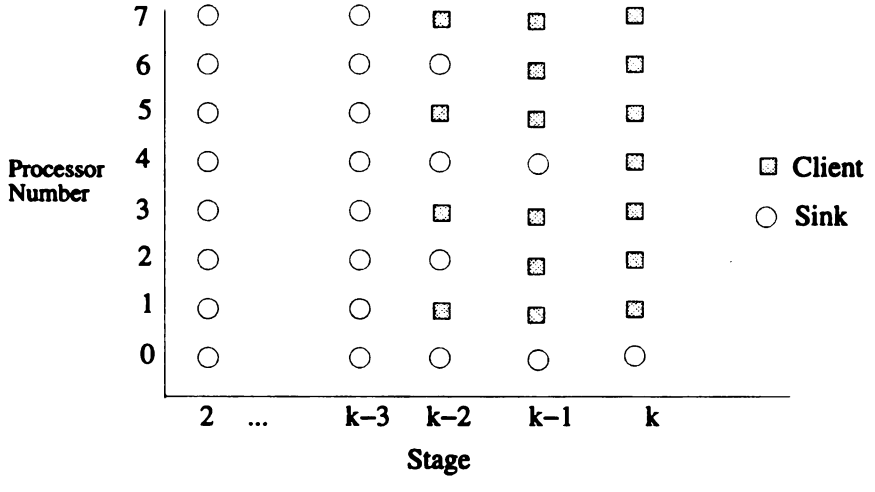


Figure 3.5. Sinks and clients in a 3-cube

The initial parallel implementation on the hypercube is illustrated in Figure 3.5, as implemented on the nCUBE-2. The 2^d processors allocated for the execution of

the program are numbered sequentially from 0 to 2^{d-1}. Consider the case of a matrix with 2^k entries being solved in a hypercube with 2^d nodes; at the beginning of the program, the original matrix is divided evenly among the participating PEs. In the example, this means that every node is working on a submatrix of 2^{k-d} entries. In stages 1 through $k-d$, all nodes work independently from the other nodes. At the end of stage $k-d$, every node has found the eigenvalues of a submatrix of size 2^{k-d}. To proceed, it becomes necessary to merge the results obtained by different nodes. That is, at stage $k-d+1$, processors need to start sharing data to continue the process. Every odd numbered node, say ℓ, sends its submatrix to its neighboring node $\ell-1$. Node $\ell-1$ uses a merge-sort to merge the two lists of eigenvalues, each of size 2^{k-d} into a single list of size 2^{k-d+1}. The even numbered nodes have become "sinks" as they are receiving the results from other nodes. The odd numbered nodes have become "clients". Each sink node merges the eigenvalues it found in stage $k-d$ with those received from its client.

In the next stage, every sink node needs to find the eigenvalues of a matrix of size 2^{k-d+1}. The clients are idle at this moment, and it would be a waste of computational resources to let the sinks find the eigenvalues of the matrices of size 2^{k-d+1} alone without using the capacity of the clients. Therefore, each sink returns the upper half of the just merged eigenvalues (which become the initial approximations to find the eigenvalues of a larger matrix) to the client from which it had previously received data. The sink solves for the lower half of the eigenvalues. Once the clients have solved their part, they send the new results to their respective sinks. After both sink and client have solved their respective halves, and the sink has received the results from the client, every sink will have the eigenvalues of a matrix of size 2^{k-d+1}.

To continue, it becomes necessary again to merge the results being kept by two sinks. In the next stage, every node with a processor id that is a multiple of 4 becomes a sink. Observe that the number of sinks decreases by half at each stage. In stage

$k-d+2$ each sink node i distributes the work evenly among itself and nodes $i+1, i+2$, and $i+3$. Each of the four nodes solves for one-fourth of the 2^{k-d+2} submatrix that resulted from the previous merge. After its clients have finished, the sink collects the results and a merge takes place. The process continues in this manner until, in the last stage, node 0 becomes the only sink and all the rest of the nodes are clients. Node 0 distributes the work evenly among all nodes and collects the results. At this point node 0 has the eigenvalues of the original matrix and the problem has been solved. Notice that the number of eigenvalues calculated by each node remains constant through all stages.

This pattern of communication and computation is very well-suited to the hypercube topology. The sinks at any stage are immediate neighbors of the other sinks of the previous stage. Each sink and its clients work in their own subcube, so their communication is disjoint from the communication of other sinks and clients. The nCUBE-2 supports broadcast operations within subcubes, which can be used by a sink to deliver results to its clients efficiently.

In spite of these advantages, the original implementation did not perform well. Performance results of this parallel algorithm indicated that speedup was limited to approximately 25 for 64 processors. In order to discover the reason behind this behavior, we used Paragraph [81], a visualization tool from Oak Ridge National Laboratory. Paragraph was used to create the graphs of the execution of the programs. Tracefiles were generated by the eigenvalue program in a manner consistent with the Paragraph format. In order to minimize any distortion of results due to I/O operations, records of the traces were maintained in main memory and copied to secondary storage only after execution of the program.

Figure 3.6 shows a trace of the execution of the the algorithm for a 2048×2048 matrix on a 16-node subcube of an nCUBE-2. Shaded areas indicate when processors are busy. In this example, the algorithm requires 11 stages. Stages 1 though 7

execute without interprocessor communication and are represented in black at the far left of the figure. Stages 8 through 11 require cooperation (and hence communication) among nodes. As shown in the Figure 3.6, processors are idle a large fraction of the time. In the figure, each unit on the x axis represents 0.15 seconds; the algorithm requires 65.9 seconds to complete.

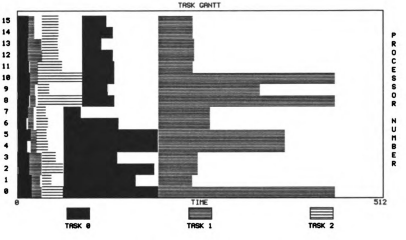


Figure 3.6. Trace of initial parallel algorithm

Further investigation revealed that the time required for processors to finish their share of eigenvalues varied greatly due to relatively high variance in the number of iterations needed to find eigenvalues. In the local stages of the algorithm, such variance of the time to solve the eigenvalues does not imply any idle time on any processing element. In the latter distributed stages of the algorithm, however, where communication between processors is necessary, this imbalance caused those processors that finished early to remain idle while others continued to work. Figure 3.7 shows the distribution of the number of iterations required by Laguerre's method to find eigenvalues in the last three stages of the algorithm for the same matrix used in Figure 3.6. Although the majority of the eigenvalues require a single iteration, a

significant number require more iterations, leading to the load imbalance. This unexpected phenomenon greatly reduced the efficiency of the algorithm, thereby limiting speedup.

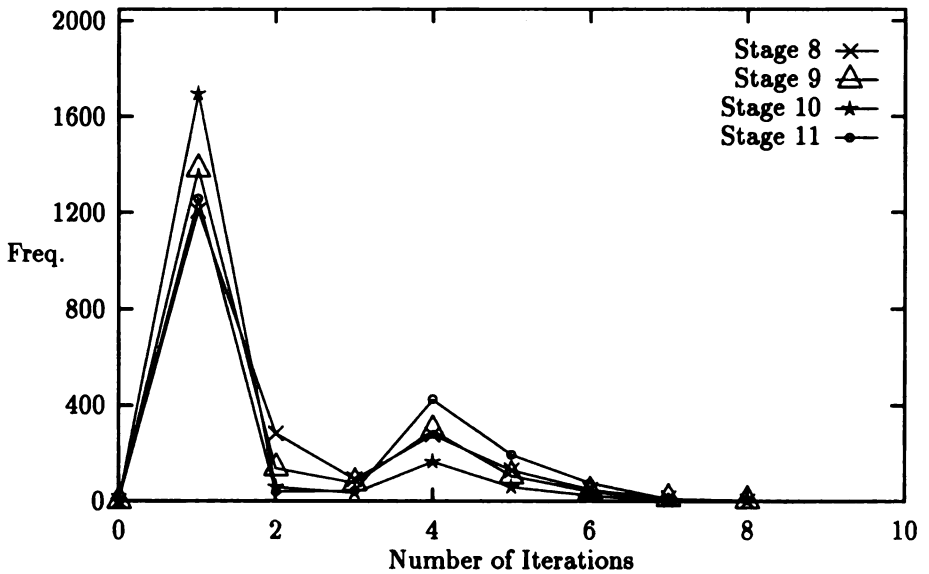


Figure 3.7. Distribution of the number of iterations needed in the Laguerre routine

3.3 Use of Dynamic Load Balancing

The previous result led to a redesign of the algorithm in an attempt to achieve better load balancing among processors.

The load balancing approach used in the eigenvalue study uses a simple client-server model, similar to those described in [82] as dynamic and centralized approaches. Many algorithms for load distribution have been proposed and they vary in different ways. Load distributing can be static, dynamic or adaptive. Static algorithms ignore completely the state of the nodes before a job is assigned. They assign the jobs to the

nodes on a deterministic fashion. Dynamic algorithms take into account, in some way, the load of the nodes in the system before assigning a job. Adaptive algorithms are a particular class of dynamic algorithms that change their parameters according to the state of the system. Another dimension is the degree of centralization: Algorithms can be centralized, hierarchical, fully decentralized, or a combination of those. Sender-initiated and receiver-initiated algorithms have been studied extensively. In a sender-initiated algorithm [83], an overloaded node looks for a receiver for some of its extra load. In a receiver-initiated algorithm, an underloaded node looks for an additional task.

This strategy is also related to the techniques employed in shared memory parallel machines to schedule the n iterations of a loop among p processors when the iterations are independent and can be executed in any order [84] (*DOALL loop scheduling*). Lilja [84] reports in a recent survey four different techniques: Chunk scheduling [85], where the iterations are divided evenly among the available processors, and which is the approach used initially in our implementation; Guided Self Scheduling (GSS) [86], where a chunk of c_i iterations is allocated to the processor making the ith request and $c_i = \lceil R_i/p \rceil$, where R_i is the number of iterations remaining, with $R_1 = n$; Factoring [87], where iterations are scheduled in batches of p chunks of the same size $c_i = \lceil R_i/2p \rceil$, once a batch of p chunks has been dispatched, the value of R_i is updated as $R_{i+1} = R_i - pc_i$; and Trapezoid Self Scheduling [88], where the programmer provides values for an initial and a final chunk size and the dispenser of the iterations gradually decreases the chunk size from the initial value to the final chunk size as the loop progresses.

Kumar *et al* [89] also encountered the need to balance the load among the processors of an nCUBE when parallelizing search algorithms. The algorithms where these techniques were used are characterized by the existence of a stack of solutions that need to be explored. The stack of unexplored alternatives can be partitioned and

distributed among several processors. They explored several alternatives, which they broadly classify in source initiated and server initiated load balancing algorithms.

In source initiated load balancing algorithm, a processor that runs out of work solicits more work from some other node. On receiving such a request, the other node splits its stack and "generates" work for the initiator. If the other node is idle, it sends a "reject" message to the initiator. If the initiator succeeds, it proceeds to work on the problem that it has received, otherwise, it tries from another processor. Different strategies can be used to determine the node from whom the receiver asks for more work. In *Asynchronous Round Robin*, each processor maintains an independent variable called *target*, whose initial value is set to $(id\_of\_the\_node + 1)$ *modulo* $number\_of\_processors$. When the processor runs out of work, it sends a request to the node indicated by *target*. The value of *target* is incremented after each request is sent. In *Global Round Robin*, there is a single global *target* variable, which is kept at processor 0. When a processor needs to request more work, it obtains the current value of *target* and then requests work from the node with that value. Processor 0 increments the value of target by 1 after every request. In *Random Polling*, the other node is selected at random. Their experiments on a nCUBE with 1024 nodes showed that *Random Polling* was the most scalable of the three approaches. For these approaches, a termination detection algorithm needs to be added for the case that the algorithm fails to find a solution.

In a server initiated load balancing algorithm, the generation of subtasks is independent of the requests for work from the idle processors. The generation of tasks can be done by a single node (single level) or it can be arranged hierarchically (multi level) as an m-ary tree of a certain depth. The root processor generates "super sub-tasks" which are assigned to its successors in the tree. The successors in turn distribute them to successors processors on request. The leaf processors perform the actual work and

they request work from their parents. When the parent of a leaf runs out of work, the leaf is assigned to a different parent.

The general strategy that was used in our implementation works as follows. Within a given stage, each node is initially required to solve only a fraction of the eigenvalues that it would have been responsible for in the original algorithm; the set of eigenvalues assigned to each node is called its *initial workload.* One node serves as the *coordinator* and is responsible for managing allocation of the remaining eigenvalues to nodes that finish early. When a node completes its initial workload, it sends its results to the coordinator, which may dispense an additional set of initial eigenvalue approximations to that node, called a *subsequent workload.* This process repeats until all the eigenvalues have been found for the current stage. Node 0 collects the results from the other nodes and then sends the respective results back to the sinks, which are still responsible for merging and sorting the eigenvalues solved at each stage.

This approach was effective in reducing the load imbalance among the nodes cooperating within each stage. Figure 3.8 shows a trace of the execution of the modified algorithm for the same 2048 $\times$ 2048 matrix as was used in the trace in Figure 3.6. The nodes finish each stage at approximately the same time, resulting in much higher efficiency. In fact, the total time required in this example is less than half that of the original algorithm. The scale used in Figure 3.8 is the same as that used in Figure 3.6; the load balancing algorithm requires only 31.8 seconds to find all 2048 eigenvalues.

Our approach falls in the category of *Server Initiated - Single Level Load Balancing* in Kumar *et al.*'s taxonomy of load balancing schemes. The effort to "generate" a task is minimal: incrementing a counter. Our algorithm is not a search algorithm and we are guaranteed that the algorithm will finish. With this approach, there is no need for a termination detection algorithm and there is no need to add overhead on the workers checking for incoming messages from other nodes.

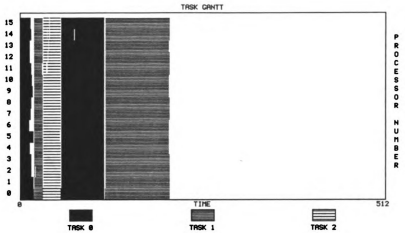
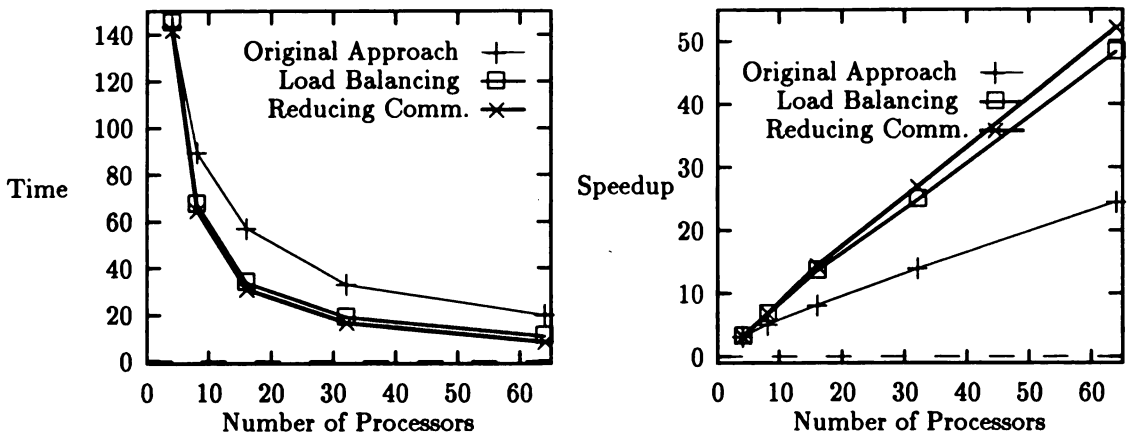


Figure 3.8. Trace of load balancing algorithm

The speedup of the algorithm is sensitive to the sizes of the initial and subsequent workloads. If the workloads are too small, then too much time is spent in communication between the clients and the coordinator. In addition, the coordinator may not be able to service all the outstanding messages immediately, delaying those nodes requesting additional work. On the other hand, if the workloads are too large, then the phenomenon of load imbalance appears again. For 2048×2048 matrices, the best overall performance was attained for an initial workload of 8 eigenvalues and subsequent workloads of 6 eigenvalues.

Figure 3.9(a) plots the average time to find all the eigenvalues in 10 random 2048×2048 matrices using different versions of our algorithm. Figure 3.9(b) displays the speedups calculated for the times in Figure 3.9(a). The average speedup for the load balancing algorithm on 64 processors was 48.3, with a maximum speedup of 53.4. This figure emphasizes the importance of load balancing. In particular, the curve for the improved version is still rising sharply at 64 processors.

It is interesting to observe that this load balancing technique, which is essentially very similar to loop distribution techniques employed in shared memory machines, is quite effective on a distributed memory machine.



(a) execution time

(b) speedup

Figure 3.9. Execution Times and Speedups for a sample of 10 matrices

One can consider load balancing a *collective communication primitive* in a broad sense, as it involves communication among all the participating nodes. It is relatively straightforward to define, as MPI has done, standards for calls to collective communications operations as broadcast and reduce. It is not clear if it would be feasible to design a "standard" call for a load balancing (loop distribution) operation as the one described above. Yet, as this case demonstrates, the performance gain can be very significant. The designers of parallel programs use collective communications as tools or primitives in their work. Somehow, it is necessary to include load balancing operations among those frequently used primitives, even if there is no single function available to perform it.

It is reasonable to expect that the centralized nature of this load balancing approach might lead to congestion problems on larger number of processors. The coordinator node can become a "hot spot" [90], a congestion point, slowing down the entire computation. It should not be difficult to implement a hierarchical approach for large numbers of processors, where a tree of coordinators would distribute the

work among the workers. In the machines where our experiments were run, though, reasonable values for the workloads seemed to avoid the problem of congestion on the coordinator.

3.4 Reducing Communication Costs

The load balancing approach described above was effective in part because it used an efficient implementation of broadcast. In distributed-memory systems, efficient communication is critical to performance. Figure 3.10 shows two traces of the load balancing algorithm using different implementations of broadcast. The shaded areas correspond to the time each node waits for the broadcast from the coordinator, node 0. The horizontal line at the bottom of each trace indicates the total execution time. In Figure 3.10(a), a naive implementation of broadcast is used, whereby a separate message is sent to each destination. In Figure 3.10(b), a more efficient *tree-based* broadcast method is used. Clearly, the tree-based primitive is much more efficient, however, a significant amount of the total execution time is still spent waiting for broadcasts. In the split-and-merge algorithm, the number of broadcasts increases with the size of the hypercube, and the time for each broadcast increases with the size of the matrix. The effect of using the naive approach is negligible with 8 or fewer nodes, but degrades performance in cubes containing more than 32 nodes.

The traditional tree-based hypercube broadcast algorithm is known as the *spanning binomial tree* (SBT) [91]. In the first step of the SBT algorithm, the source node sends the message to its neighbor whose address differs from it own in the lowest (alternatively highest) bit position, that is, in the first dimension. Next, these two nodes send to their respective neighbors in the second dimension. Following that, all four nodes holding copies of the message forward it to their neighbors in the third dimension. This process continues until, in the last step, half of the nodes in the

network forward the message to the other half through the highest dimension. This algorithm requires n message passing steps to reach all nodes in an n-cube, regardless of the number of ports between processors and their routers [37].

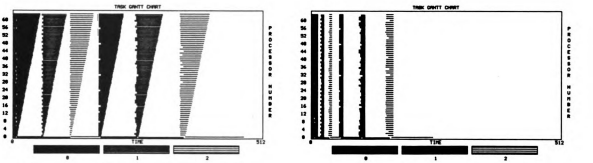
We [37] have developed a new method to reduce broadcast time in wormhole-routed hypercube systems. The method, called the *Double Tree* (DT) algorithm, is designed to take advantage of the distance insensitivity of wormhole routing and the presence of multiple ports. The DT algorithm begins with the source node s sending the message to the node whose address is the bitwise complement of s, call it $\overline{s}$. Subsequently, nodes s and $\overline{s}$ become the roots of partial spanning binomial trees. The tree rooted at s is called the *forward* tree, and the one rooted at $\overline{s}$ is called the *backward* tree. The message is distributed along the branches of both trees in parallel, reducing the number of message passing steps required to reach all nodes in an n-cube to $\lceil n/2 \rceil$. Experiments on the nCUBE-2 show that the actual latency of a broadcast operation can be significantly reduced [37], thereby improving the performance of the split-merge algorithm. The broadcast primitive available in the nCUBE programming environment takes advantage of certain characteristics of the architecture that are not available to regular programmers (DMA programming). To evaluate the performance gained by using the double tree algorithm in place of the regular SBT tree algorithm, both primitives were implemented at the user level and the program was run with both broadcast alternatives to compare the performance. The results of the execution of the program on a random matrix of size 2048 are presented in Table 3.1.

The results are slightly better with the double tree algorithm for large numbers of processors. These results were obtained with the latest release of the operating system of the nCUBE, whose optimizing compiler seems to be significantly better than the previous version.

Besides using an efficient broadcast algorithm, other communication-related alterations were made to the algorithm in order to improve performance. For example,

Processors	SBT	DT	Workload size
1	164.35	164.35	
2	135.96	135.96	16
4	53.08	53.08	16
8	24.43	24.43	16
16	12.56	12.56	16
32	5.94	5.93	8
64	3.49	3.47	8

Table 3.1. Execution times on the nCUBE with different broadcast algorithms



(a) *broadcast with separate messages* (b) *tree-based broadcast*

Figure 3.10. Traces of Broadcasts

dynamic load balancing turned out to improve performance only when used in the distributed stages of the algorithm. Although the use of dynamic load balancing was tested in the local stages as well, increased communication overhead actually reduced overall performance. Further improvements were achieved by reducing the amount of data to be broadcast through the use of redundant parallel computing. As part of the preparation for the Laguerre iteration, the values of the entries in the main diagonal, the squares of the off-diagonals and the current approximations to the eigenvalues are scaled *. The scaling was being done at the nodes in charge of the merging and sorting (the sinks), which sent those scaled values to the coordinator, which in turn broadcasted them to the nodes. By performing the scaling of the main diagonal and the off-diagonals redundantly, but in parallel, at every processor, the total broadcast time was reduced. Combined with the use dynamic load balancing in the distributed stages, these improvements resulted in an average speedup of 52.2 and a maximum speedup of 55.3 for a sample of 10 random matrices on a 64-node nCUBE-2. These results can be observed in Figure 3.9.

3.5 Performance Study

The preceding discussion was based on experiments performed using random matrices. To verify the robustness of the parallel split-merge algorithm, other types of input matrices, designed specifically to test the accuracy and speed of eigenvalue solvers, were used.

A comparison of the *sequential* version of the split-and-merge algorithm with other sequential algorithms for finding eigenvalues has been conducted previously

*The latest version of the algorithm does not perform this scaling operation anymore. We report on the previous version as it is interesting to observe that at times it is faster to perform redundant computation than expensive communications.

by Li and Zeng [77]. In that study, the other sequential algorithms included bisection/multisection(DSTEBZ in Lapack), divide-and-conquer (TREEQL [92]), RFQR (Root-free QR, DSTERF from LAPACK) and QR (DSTEQR from LAPACK). Split-and-merge achieved the best accuracy, while RFQR was the fastest algorithm.

In our study, the parallel version of split-and-merge was compared against a parallel version of bisection. Bisection is very well suited for parallelization: once the initial data has been broadcast to the participating nodes, each node can work on its part of the problem without any communication with other nodes, except for reporting the final results to the initiating node. QR methods are difficult to parallelize when only the eigenvalues are sought [93]. Ipsen and Jessup [94] report that parallel bisection is faster than divide-and-conquer, hence the decision to use bisection in comparisons.

Twelve kinds of input matrices were used in the experiments and are described in Table 3.2. The following conventions are used in the description of the matrices: $\alpha_i, i = 1, ..., n$ represent the (main) diagonal entries and $\beta_i, i = 1, ..., n-1$ denote the offdiagonal entries. The values of a and b were chosen to be 4 and 1, respectively, for all the matrices. The experiments were performed on matrices of size 128, 256, 512, 1024 and 2048 for types 1 through 7, size 128, 256 and 512 for types 8 through 12.

The following results were obtained on the nCUBE-2. The experiments were executed on subcubes of size 1, 8, 16, 32, and 64. The execution times for the bisection code and split-and-merge can be observed in Table 3.3. The times are reported in seconds. Figure 3.11 highlights the execution times for matrices of size 2048 of types 1 through 6 on 8, 16, 32, and 64 processors on the nCUBE-2. Figure 3.12 plots the execution times for matrices of size 512 of the types 7 through 12 on 8, 16, 32, and 64 processors.

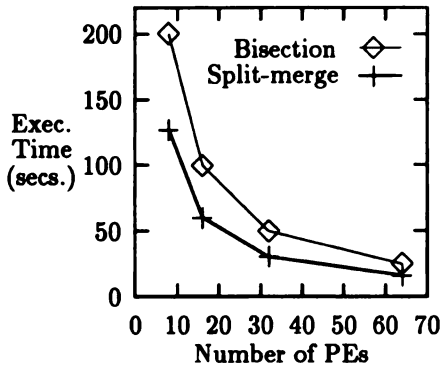
The sequential version of split-and-merge is significantly faster than the sequential version of bisection, as reported already by Li and Zeng [77]. This holds true for all

Type	Entries α_i and β_i	Eigenvalues	Comments
1	$\alpha_i = a$ $\beta_i = b$	$\{a + 2bcos\frac{k\pi}{n+1}\}_{k=1,\cdots,n}$	Toeplitz $[b, a, b]$, [95](p. 137)
2	$\alpha_1 = a - b$ $\alpha_i = a$ for $i = 2, \cdots, n-1$ $\alpha_n = a + b$. $\beta_j = b, j = 1, \cdots, n-1$.	$\{a + 2bcos\frac{(2k-1)\pi}{2n}\}_{k=1,\cdots,n}$	[95](p. 138)
3	$\alpha_i = \begin{cases} a \text{ for odd } i \\ b \text{ for even } i \end{cases}$ $\beta_i = 1$.	$\{\frac{a+b\pm[(a-b)^2+16cos^2\frac{k\pi}{n}}{2}\}_{k=1,\cdots,n/2}$ and a if n is odd	[95] (p. 139)
4	$\alpha_i = 0$ $\beta_i = \sqrt{i(n-i)}$	$\{-n + 2k + 1\}_{k=1,\cdots,n}$	[95](p. 140)
5	$\alpha_i = -[(2i-1)(n-1) - 2(i-1)^2]$ $\beta_i = i(n-i)$	$\{-k(k-1)\}_{k=1,\cdots,n}$	[95](p. 141)
6	for even n: $\alpha_i = \begin{cases} \frac{n}{2} - i + 1 & 1 \le i \le \frac{n}{2} \\ i - \frac{n}{2} & n/2 < i \le n \end{cases}$ for odd n: $\alpha_i = \begin{cases} \frac{(n+1)}{2} - i + 1 & 1 \le i \le \frac{(n+1)}{2} \\ i + 1 - \frac{(n+1)}{2} & \frac{(n+1)}{2} < i \le n \end{cases}$ $\beta_i = 1$.	Most are in pairs, consisting of two numerically indistinguishable eigenvalues	Wilkinson matrices W_n^+. [96](p. 308)
7	Generated randomly in $[0, 1]$.	Random	Random Matrices
8	Generated by LAPACK test matrix generator	Evenly distributed between the smallest and the largest eigenvalue	[55]
9	Generated by LAPACK test matrix generator	Geometrically distributed: $\{q^k\}_k = 1, \cdots, n$ for some $q \in (0, 1)$.	[55]
10	Generated by LAPACK test matrix generator	One eigenvalue 1 The rest are in $(-\epsilon, \epsilon)$.	[55]
11	Generated by LAPACK test matrix generator	Evenly distributed in the interval $(0, 1]$ except one very small one	[55]
12	Generated by LAPACK test matrix generator	Evenly distributed in the interval $[10^{-12} - \epsilon, 10^{-12} + \epsilon]$ except one with value 1	[55]

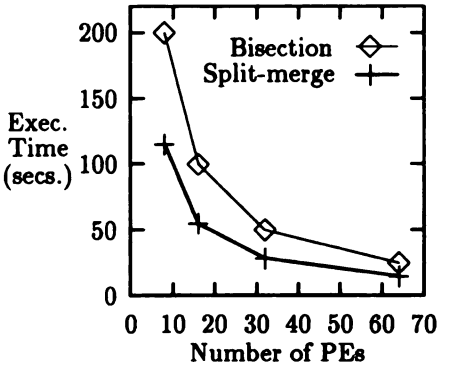
Table 3.2. Types of test matrices

		Bisection					Split-and-merge				
Type	Size	1 PE	8 PE	16 PE	32 PE	64 PE	1 PE	8 PE	16 PE	32 PE	64 PE
1	128	6.62	0.79	0.39	0.19	0.10	3.75	0.63	0.33	0.34	0.26
	256	26.39	3.14	1.56	0.78	0.39	14.60	2.19	1.15	0.68	0.57
	512	105.33	12.54	6.25	3.12	1.56	57.14	8.40	4.12	2.42	1.40
	1024	420.91	50.09	24.97	12.47	6.24	224.42	32.39	15.83	8.01	4.56
	2048	1682.80	200.23	99.78	49.82	24.91	882.94	127.06	60.35	30.80	15.92
2	128	6.58	0.78	0.39	0.19	0.10	3.40	0.58	0.31	0.32	0.26
	256	26.21	3.13	1.56	0.78	0.39	13.28	2.02	1.10	0.63	0.54
	512	104.63	12.50	6.23	3.11	1.56	51.95	7.68	3.83	2.27	1.31
	1024	418.09	49.92	24.88	12.43	6.22	203.71	29.47	14.52	7.36	4.27
	2048	1671.52	199.55	99.45	49.65	24.82	800.27	115.38	55.02	28.45	14.63
3	128	6.70	0.80	0.40	0.20	0.10	3.91	0.69	0.39	0.35	0.31
	256	26.71	3.18	1.58	0.79	0.390	15.11	2.42	1.37	0.84	0.75
	512	106.61	12.68	6.32	3.16	1.58	58.61	9.10	4.77	2.96	1.83
	1024	425.99	50.65	25.24	12.60	6.31	228.23	34.31	17.74	9.86	6.29
	2048	1730.05	202.44	100.88	50.36	25.18	891.73	132.31	65.74	36.00	21.13
4	128	6.85	0.81	0.40	0.20	0.10	3.63	0.61	0.34	0.33	0.39
	256	27.29	3.24	1.62	0.81	0.40	14.15	2.15	1.13	0.63	0.69
	512	108.93	12.96	6.46	3.23	1.62	55.25	8.24	4.03	2.47	1.33
	1024	435.27	51.72	25.78	12.87	6.44	215.48	31.33	15.34	7.67	4.55
	2048	1740.20	206.92	103.15	51.50	25.75	843.64	122.05	58.09	29.42	15.21
5	128	6.71	0.80	0.40	0.20	0.10	3.57	0.60	0.34	0.32	0.42
	256	26.74	3.18	1.58	0.79	0.40	13.76	2.09	1.12	0.61	0.74
	512	106.70	12.72	6.33	3.16	1.58	53.71	8.04	3.92	2.23	1.46
	1024	426.29	50.82	25.31	12.61	6.31	209.36	30.43	14.98	6.80	4.39
	2048	1704.13	203.19	101.20	50.44	25.19	819.34	118.62	56.51	26.04	14.82
6	128	6.71	0.80	0.40	0.20	0.10	6.03	1.04	0.54	0.51	0.58
	256	26.74	3.18	1.59	0.79	0.40	22.96	3.42	1.95	1.06	1.02
	512	106.72	12.70	6.33	3.16	1.58	87.04	12.96	6.14	3.83	2.12
	1024	426.36	50.75	25.30	12.63	6.32	322.15	47.13	22.43	11.46	7.14
	2048	1704.43	202.86	101.10	50.48	25.24	1202.37	174.70	81.74	41.53	21.62
7	128	6.73	0.80	0.40	0.20	0.10	3.04	0.54	0.37	0.39	0.46
	256	26.80	3.18	1.59	0.79	0.40	7.51	1.17	0.68	0.59	0.65
	512	106.97	12.71	6.33	3.16	1.58	33.45	5.26	2.63	1.77	1.68
	1024	427.45	50.77	25.29	12.63	6.31	124.31	18.19	9.27	4.89	3.42
	2048	1708.97	202.90	101.08	5.04	25.22	520.88	76.37	36.38	19.03	9.96
8	128	6.72	0.80	0.40	0.20	0.10	3.28	0.57	0.30	0.32	0.39
	256	26.78	3.18	1.59	0.79	0.40	12.67	1.91	1.01	0.62	0.68
	512	106.92	12.72	6.34	3.17	1.58	49.41	7.29	3.6	2.27	1.37
9	128	6.75	0.82	0.41	0.20	0.10	2.11	0.38	0.32	0.34	0.41
	256	26.80	3.19	1.59	0.79	0.40	7.92	1.22	0.76	0.72	0.79
	512	106.92	12.72	6.34	3.17	1.58	49.41	7.29	3.60	2.27	1.37
10	128	6.69	0.83	0.41	0.21	0.10	1.91	0.42	0.34	0.38	0.43
	256	26.67	3.31	1.66	0.83	0.41	7.14	1.18	0.69	0.56	0.62
	512	108.19	13.44	6.72	3.36	1.68	26.87	4.12	2.12	1.46	1.18
11	128	6.72	0.80	0.40	0.20	0.10	3.21	0.55	0.30	0.31	0.38
	256	26.79	3.19	1.59	0.79	0.40	12.20	1.85	1.00	0.65	0.67
	512	106.94	12.72	6.34	3.17	1.58	47.41	7.07	3.45	2.25	1.30
12	128	6.68	0.83	0.41	0.21	0.10	1.72	0.31	0.22	0.24	0.33
	256	26.74	3.33	1.67	0.83	0.42	6.54	1.04	0.61	0.46	0.51
	512	107.92	13.43	6.72	3.36	1.68	26.40	4.52	2.62	1.91	1.61

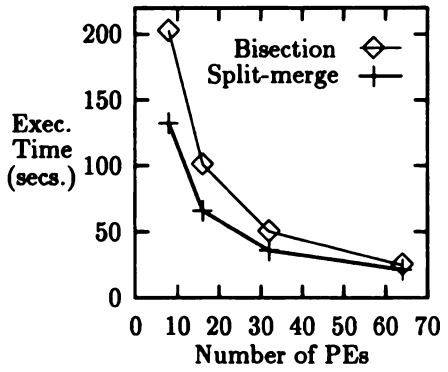
Table 3.3. Execution times (in seconds) for matrices of type 1 through 12 on an nCUBE-2



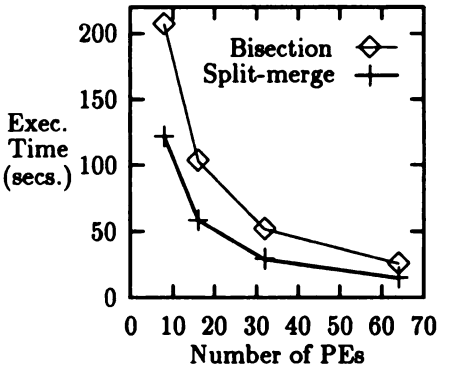
(a) Matrix with 2048 entries of type 1.



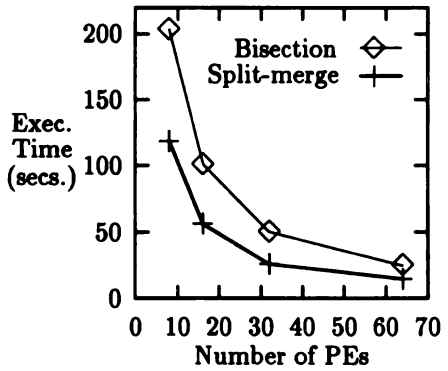
(b) Matrix with 2048 entries of type 2.



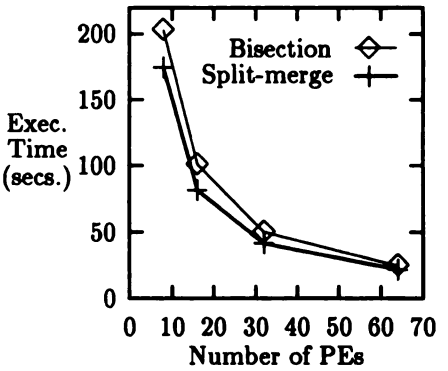
(c) Matrix with 2048 entries of type 3.



(d) Matrix with 2048 entries of type 4.

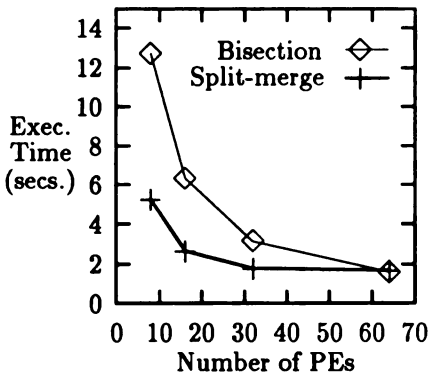


(e) Matrix with 2048 entries of type 5.

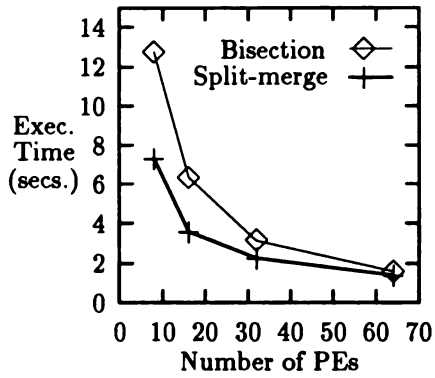


(f) Matrix with 2048 entries of type 6.

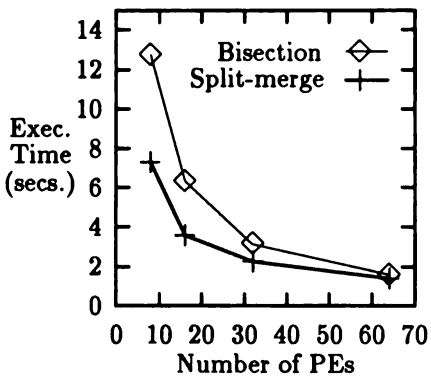
Figure 3.11. Execution times of parallel eigenvalue solvers on an nCUBE-2.



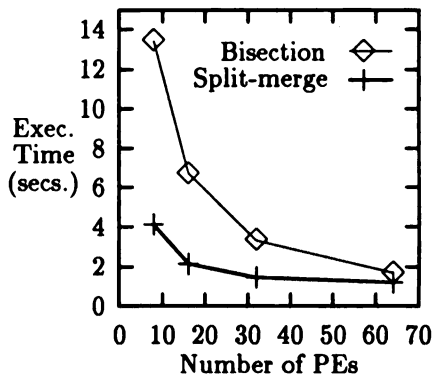
(a) Matrix with 512 entries of type 7.



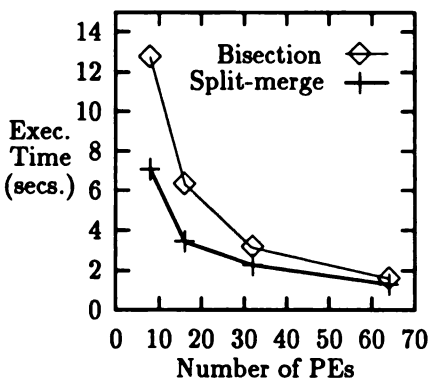
(b) Matrix with 512 entries of type 8.



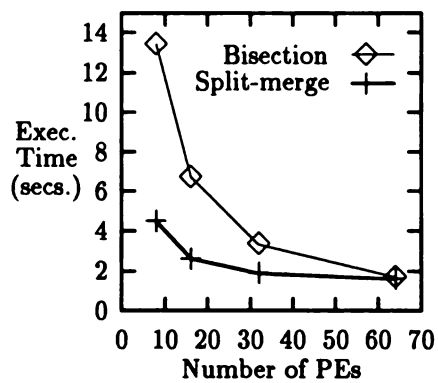
(c) Matrix with 512 entries of type 9.



(d) Matrix with 512 entries of type 10.



(e) Matrix with 512 entries of type 11.



(f) Matrix with 512 entries of type 12.

Figure 3.12. Execution times of parallel eigenvalue solvers on an nCUBE-2.

types of matrices, although in matrices of type 6, the advantage of split-and-merge is more modest.

On the nCUBE-2, the parallel version of split-and-merge is faster, in most cases, than the parallel version of bisection, even though the bisection algorithm has much lower communication requirements. Reducing communication costs, particularly broadcast, is critical to maintaining a performance advantage with larger numbers of processors. The result is that the parallel version of split-and-merge is faster than the parallel version of bisection for most of the matrices in this environment. There are some exceptions for small matrices where parallel bisection has lower execution time than split-and-merge: matrices with 128 or 256 entries executed on 32 or 64 processors. Observation reveals that the parallel version of split-and-merge is faster than the parallel version of bisection when there are at least 8 eigenvalues to be calculated by every PE. When the number of eigenvalues per PE is more than 8, the advantage of split and merge is more pronounced. This can be observed by comparing Figures 3.11 and 3.12 where, with 64 processors, the advantage of split-and-merge is clear for the larger matrices (order 1024 and higher).

3.6 Related Work

Several parallel algorithms have been developed to address this problem. The algorithm of choice in sequential computers has been QR, which finds simultaneously both eigenvalues and eigenvectors. A theoretical parallelization of that algorithm was reported in 1977 in [97]. More recently, other researchers have modified the QR algorithm and applied it on a variety of parallel machines and clusters of workstations [93]. That study focused on a modified version of QR for finding both eigenvalues and eigenvectors. Interestingly, as in the study described here, those researchers also

found the benefit of performing redundant computations in order to reduce communication costs. The finding of the eigenvalues was performed redundantly while the calculation of the eigenvectors was done in parallel.

The algorithm devised by Cuppen [98] was used as the basis for the implementation reported in [92] for shared memory machines. Gates [99] reports a variation on the same approach also for a shared memory machine. Cuppen's method has also been implemented on an hypercube [94].

A third group of implementations is based on bisection and multisection. Using Gerschgorin disks, an interval containing all eigenvalues is calculated. That interval is recursively divided into two (for bisection) or more (for multisection) sub-intervals until one of the following conditions is true: Either the interval contains no eigenvalue, or the interval contains exactly one eigenvalue or the interval is very small, this is, there are several very close eigenvalues. The number of eigenvalues in the interval can be determined by calculating the Sturm sequence at the extremes of the interval. The number of negative terms in the Sturm sequence evaluated at a particular points p is the number of eigenvalues that are smaller than p. Once it has been determined that the interval contains exactly one eigenvalue, there are different alternatives for finding the exact value of the eigenvalue. If the eigenvectors are desired, they can be found by using inverse iteration. If several eigenvalues are very close to each other, it might be necessary to perturb them before finding the eigenvectors. An implementation for shared memory machines is reported in [100] and for hypercubes in [94]. Lu and Qiao [101] developed an algorithm that parallelizes the evaluation of Sturm's sequence to improve the bisection process. They implemented their algorithm in a shared memory machine.

A fourth approach is based on homotopy methods [102, 103]. To find the eigenvalues of a given matrix A, another matrix D is chosen, such that

$$A(t) = (1 - t)D + tA$$

is an unreducible tridiagonal matrix for $t \neq 0$. The eigenpairs of matrix D are known in advance. There are disjoint paths between the eigenvalues of matrix D and matrix A. Those paths can be followed in parallel.

Clearly, this is an active area of research in parallel numerical algorithms. Research in this area is being pursued in the mathematics department at Michigan State University by Li and his group.

3.7 Summary

In this chapter, the implementation and performance evaluation of a parallel eigenvalue solver on an nCUBE-2 MPC have been described. The general structure of the split-merge algorithm for finding eigenvalues in symmetric tridiagonal matrices is well-suited to efficient parallelization. The algorithm uses Laguerre's iteration and exploits the separation property in order to create subtasks that can be solved independently.

The split-merge algorithm was implemented and studied on an nCUBE-2, a wormhole-routed hypercube. In the initial parallel version of the algorithm, the only *required* communication among processes occurs only between stages of the algorithm; the number of stages is at most $\log_2(n)$, where n is the order of the matrix. The inherent variance in the number of iterations in the split-merge algorithm justified more sophisticated approaches to load balancing. The communication used to implement dynamic load balancing is, in some sense, *programmable*.

Experiments on a variety of input matrices confirmed that split-merge is significantly faster than bisection for large matrices (order 512 or larger) on the nCUBE environment.

CHAPTER 4

The Split-merge Algorithm on a Conventional Cluster

Clusters of workstations interconnected via Ethernet are very common across academic, research and even industrial institutions. With tools like PVM and P4, these clusters constitute an economic alternative to MPCs. How does the performance of an ordinary cluster of workstations compare with that of an MPC? Is it competitive? Should the programming be different? What factors affect the performance of an application on this environment? How are these factors different from those on an MPC?

In order to answer these questions, the split-merge algorithm was implemented and tested on a cluster of Sun Sparc-10, model 30 and model 40 workstations, interconnected by a typical Ethernet network. The workstations used for the experiments were also available for general use by students and faculty, that is, other users could freely access the workstations at any time. This environment is consistent with our goal of testing the hypothesis that general-purpose, shared workstations can provide competitive performance for scientific computing tasks.

The eigenvalue algorithm was implemented using two different programming environments, PVM and P4. PVM [42], a public domain package from Oak Ridge

National Laboratory, provides a software infrastructure for network-based heterogeneous concurrent computing. PVM-based applications are structured as a set of *components*, with one or more *instances* of each component. PVM daemons are created on each node in the system and in turn spawn the application instances. Primitives are provided for process management, communication, synchronization, and so on. Communication among daemons uses ordinary sockets. P4 [43], developed at Argonne National Laboratory, comprises a library of macros and subroutines that support monitors for shared-memory programming, message-passing primitives, and support for heterogeneous cluster computing. Since there were no significant differences in performance between the PVM and P4 implementations, only the PVM implementation is described here.

4.1 PVM Implementation

Porting the eigenvalue code from the nCUBE-2 to PVM (version 2.4 and later version 3.2) was relatively straightforward. Both environments are based on message passing, and their functionality is similar: sends are non-blocking, receives can be either blocking or non-blocking, and messages can be identified with tags indicating their type. Both environments support programming in C and Fortran and allow the exchange of messages between program components written in either language.

The nCUBE is a single machine and the same format is used to represent the data uniformly across all nodes. The only time that the format of the data may need to be converted is when it is sent from the front-end workstation to the nodes in the hypercube. PVM, on the other hand, is designed to be used in an heterogeneous environment. Therefore, in the nCUBE, the system can be oblivious to the type of the data being passed in a message: The receiving and sending nodes have the responsibility on being consistent on the use of the data passed in the messages. In

contrast, in PVM it might be necessary to convert from the format of one machine to a different format for another machine and it is the responsibility of PVM to call XDR appropriately to perform the conversion.

In terms of allocating processes to processors, the two environments are quite different. In the nCUBE-2, full subcubes are allocated to each application; one instance of the program runs on each node of the allocated subcube. Every node in the subcube is devoted exclusively to the execution of the program, and I/O operations can be performed from any node. In PVM, the *hosts* file contains the names of the workstations that are available for use. Once started, the program "enrolls" one or more instances of each component that is required to solve the problem. Those instances are allocated to the available workstations. It is important to know that the number of instances can be larger than the number of workstations available, that is, multiple processes can be assigned to the same workstation, even if that workstation has only a single processor. The usefulness of this strategy will be described later.

Executing parallel programs on a cluster of general-use workstations introduces randomness in the performance of the program from two sources: the load of the workstations and the load of the network. Both resources are shared among applications. In the nCUBE-2, each node is devoted exclusively to the application assigned to that node. Furthermore, because only full subcubes are assigned to programs, the hypercube topology and the underlying routing algorithm prevent communication conflicts among the messages of different programs. In other words, the communication network of the subcube is also dedicated exclusively to the execution of the application assigned to it. Hence, there is no contention among different applications for access to the network.

The programming model in PVM assumes that a master process instantiates several server processes. The master process can perform I/O operations without intervention of PVM. The servers perform their I/O operations through the PVM

daemon on the starting workstation. If one of the servers writes a message to the standard output, the message will be written to a log file on the workstation that started PVM. If the program stops execution prematurely, those messages may not appear on the log file.

The syntax of the send and receive operations on the nCUBE-2 require pointers to the beginning of the area being sent or received and the number of bytes that one wishes to transmit. In one single call, one specifies the sender or receiver, the address of the area, its size and the type of the message.

For the reasons mentioned previously, PVM takes a different approach. When sending a message, one uses a call to specify that a message will be sent, and then uses calls that are type dependent to move data into the buffer that will be sent later. It is possible send a message with different types of data. Once the buffer is ready, another call actually sends the buffer, specifying the tag of the message. The process receiving the message, specifies that it will receive a message of a certain tag (or alternatively with any tag) and then it moves the data from the buffer to the appropriate variables by using type dependent calls.

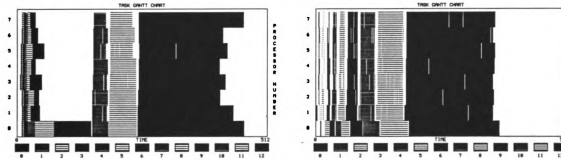
4.2 Load Balancing in Local Stages

As described in chapter 3, the nCUBE-2 version of the algorithm used dynamic load balancing to accommodate the variance in the Laguerre iteration routine. Traces of our cluster implementations demonstrated that the load balancing algorithm was also effective in the new environment, although larger workload sizes (the number of eigenvalues assigned by the coordinator following a request for additional work) resulted in better efficiency due to decreased communication overhead. The results in the cluster indicate that workloads of 16 eigenvalues produce best results for random matrices of order 2048; in the nCUBE-2, the optimal workload size was 6 eigenvalues.

In the hypercube environment, load balancing was useful only in the distributed stages of the algorithm. However, traces of execution on the cluster indicated that the problem of load imbalance was also present in the local stages, that is, in those stages involving no interprocessor communication. This behavior can be attributed to a PVM process relinquishing the processor to another user application on a particular workstation. When such an event occurs, the execution of the entire program is delayed until that PVM instance processes finishes its (local) share of the work. In order to alleviate this problem, load balancing was also applied to the last several local stages. The load balancing algorithm was the same as that used for the distributed stages, that is, in which a single node acts as the coordinator. The same workload size is used in all stages.

Figure 4.1 compares execution traces the PVM implementation on a matrix of order 4096, with and without load balancing in the local stages. Both traces use the same scale. Figure 4.1(a) shows the execution without load balancing in the early stages, where node 0 happens to require much more time than the other nodes to solve its share of the eigenvalues. Figure 4.1(b) corresponds to an execution of the algorithm on the same matrix with load balancing implemented in the last 5 local stages; the advantage of this strategy is clear. To summarize, in a cluster of workstations, load balancing may be critical to parts of the application where it was not effective in an MPC implementation.

An alternative approach to load balancing is to create more worker processes than there are workstations, which is possible because PVM can map several processes to the same PE. In the eigenvalue algorithm, assigning two processes to each processor reduced the negative effect of processes becoming blocked while waiting for communication to complete; communication delays are masked by overlapping communication of one process with computation in another. When one process is blocked, the presence of another process in the same PE increases the probability of more useful work



(a) no load balancing in local stages (b) load balancing in local stages

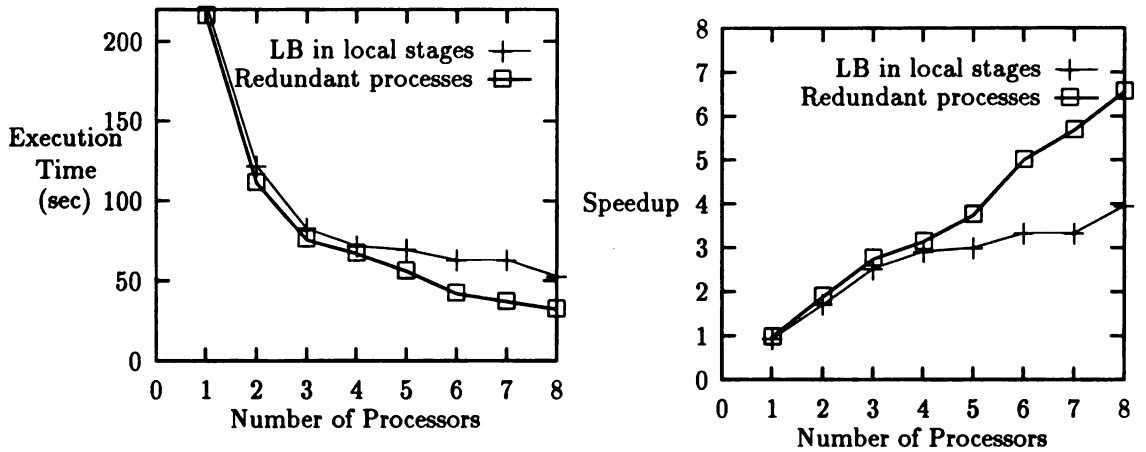
Figure 4.1. Execution traces of parallel eigenvalue solver on network of workstations

being done. In the PVM implementation, this approach proved even more successful than load balancing in local stages, as can be observed in Figure 4.2. Figures 4.2(a) and 4.2(b), respectively, plot the execution times and corresponding speedups when 16 processes were created and assigned to the workstations. As shown, dual-process approach achieved better speedup than load balancing in local stages. We did not try to use more than two processes per workstation.

4.3 Effects of the Cluster Environment

According to traces of program executions, the time that a worker processor has to wait between the moment it has sent its results to the coordinator and the time when it receives another piece of work is approximately 400 microseconds on the nCUBE-2. On the cluster, the same action requires approximately 5000 microseconds (in the best cases, when there is no contention for the network and the coordinator is ready to receive the request). In addition to software overhead, the increased communication latency results from the context switch at the coordinator node.

Broadcast communication is particularly important to the load balancing implementation of the eigenvalue solver. Broadcast is used to distribute the eigenvalues



(a) execution times

(b) speedups

Figure 4.2. Comparison of load balancing approaches for a matrix or order 4096.

found in the previous stage to all the worker processes. In the nCUBE-2, as described earlier, the tree-based system-supported broadcasting primitive and the DT algorithm [37] both provided performance advantage. Both broadcast implementations take advantage of the hypercube topology delivering a message to N nodes in $O(\log N)$ time. Broadcasting a message of 32 kbytes takes approximately 32000 microseconds in a subcube with 4 nodes and 47400 microseconds in a subcube with 8 nodes.

Using PVM, it is also possible to specify that a message should be delivered to multiple processes. Internally, however, this function is implemented as multiple point-to-point, or *unicast* messages, which requires time linear in the number of destinations. Cluster broadcast times are further affected by two factors. First, contention for the network among multiple applications can delay the sending of messages. Second, the nodes executing the algorithm were not always directly connected to the same Ethernet cable, that is, some nodes reside on different cables connected by bridges. For

32 kbyte messages, each of the constituent unicast messages required approximately 30000 microseconds. The advantage of the nCUBE-2 in terms of broadcasting is clear. Improving the performance of the broadcast operation by using different alternatives will be discussed in chapter 5.

The execution times of the split-merge algorithm in the cluster can vary from one execution to another, depending on the load on the workstations from other applications. Figure 4.3 shows the load of the workstations and the execution time of the eigenvalue program at different times of the day. To obtain the plot, eight Sparc-10 workstations were used to solve a matrix of order 4096. The program did not use load balancing in the early stages and only one process was assigned to each workstation. The load statistics were obtained using the *rup()* system command, and the average load was recorded. The load averages have been scaled up by a factor of ten to provide better contrast. The figure clearly illustrates the correlation between the average load on the workstations and the execution time of the program. The "spike" at 4:30 a.m. is due to automatic daily system maintenance. Load balancing on the program can only partially compensate for the additional load on the workstations. When poor speedups occur, they can be attributed to the use of the workstations by other users and to the communication costs.

As discussed previously, the environments of the nCUBE-2 and the cluster are very different. The nCUBE-2 has a more efficient network and its PEs are not shared. The cluster, on the other hand, has faster PEs: the sequential version of the split-and-merge program runs approximately 9 times faster in a lightly loaded Sparc-10 than in an individual node in the nCUBE-2. Although the communication costs are substantially higher on the cluster environment, the greater computing power of the PEs results in good performance. Since the split-merge algorithm is not communication-intensive, it benefits substantially from this characteristic of the cluster.

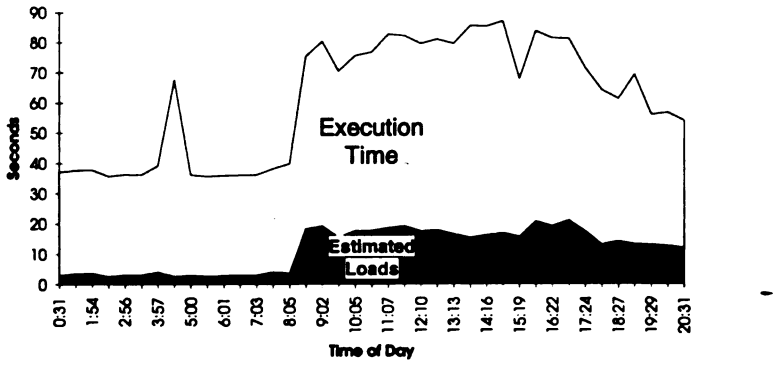


Figure 4.3. Effect of workstation load on execution times.

Although the speedup figures are not always particularly high, when comparing cluster performance with that of the nCUBE-2, it is necessary to examine the *absolute* execution times and to consider the cost and flexibility of the systems. Even in a general-use cluster environment, the times on a cluster of 8 workstations are competitive with the time of the nCUBE-2 with 64 processors and better than those of smaller hypercubes. Finding the eigenvalues for a matrix of order 4096 required 32 seconds on an 8-node cluster; the nCUBE-2 requires 118 seconds on a 16-node subcube, 58 seconds with 32 processors, and 32 seconds with 64 processors. Even when the load on the workstations is high, the worst recorded execution time for this matrix, 89.425 seconds, using 8 workstations, is still better than that of a 16-node subcube in the nCUBE-2. Again, a small cluster is very competitive.

In light of these results, and taking into consideration the cost element (given university discounts on equipment, 16 Sparc-10s cost approximately the same as 16-node nCUBE-2), it can be argued that networks of workstations do provide a cost/effective alternative to MPCs for certain scientific applications, once the higher communication costs are accounted for. In this case, the use of dual processes per node was effective in

hiding communication latency. Improving performance by reducing broadcast times is part of our ongoing research.

4.4 Performance Study

The same input matrices that were used to test the performance of the program on the nCUBE were used to test the performance on the cluster of workstation.

The experiments on the cluster of workstations were performed on 1, 4 and 8 workstations. The execution times for the programs on matrices of types 1 through 12 can be observed in Table 4.1. The execution times are reported in seconds. Figures 4.4 and 4.5, respectively, plot the execution times for matrices 1 through 6 with 2048 entries and types 7 through 12 with 512 entries.

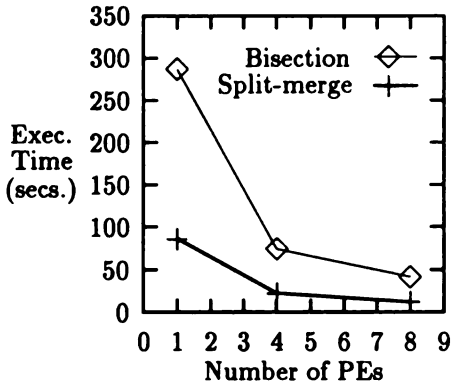
As previously discussed, a cluster of workstations interconnected through Ethernet has much higher communication latencies than the nCUBE-2. The experiments were generally conducted at times of little activity on the workstations, but as the workstations are in an open laboratory, it was not possible to guarantee exclusive access to the systems. The times reported in this section were collected with the PVM version on the cluster.

The results for the cluster of workstations are similar to those for the nCUBE-2. Again, the sequential version of the split-and-merge algorithm is significantly faster than the sequential version of bisection.

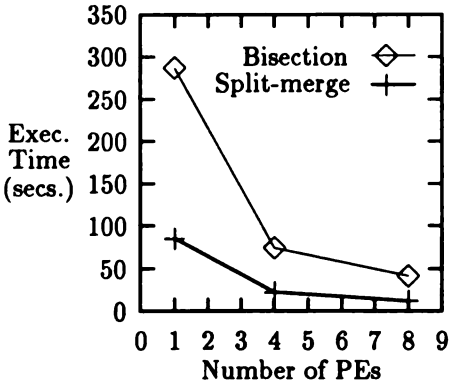
A phenomenon similar to the one observed on the nCUBE-2 can be observed in the clusters as well: for very small matrices (128 entries), bisection is faster than split-and-merge with more PEs (8 for the cluster and 32 for the nCUBE-2). The communication characteristics of the nCUBE-2, specifically, faster interprocessor communication and a faster broadcast primitive, make split-and-merge superior over a wide range of numbers of processors. The high communication overhead of the cluster reduces that

		Bisection			split-merge		
Type	Size	1 PE	4 PE	8 PE	1 PE	4 PE	8 PE
1	128	1.07	0.39	0.21	0.52	0.57	0.77
	256	3.99	1.16	0.65	2.00	0.92	0.95
	512	16.25	4.29	2.50	7.58	2.35	1.66
	1024	65.12	17.26	9.59	28.78	7.97	5.01
	2048	286.32	74.93	41.77	86.46	22.99	12.93
2	128	1.04	0.40	0.26	0.50	0.44	0.81
	256	3.97	1.10	0.66	2.00	0.78	0.96
	512	16.06	4.26	2.46	7.14	2.32	1.68
	1024	64.95	17.24	9.58	27.81	7.79	4.78
	2048	286.33	74.90	41.70	86.00	22.81	12.94
3	128	1.08	0.36	0.21	0.41	0.40	0.76
	256	4.07	1.18	0.65	1.74	0.76	0.95
	512	16.10	4.27	2.46	7.07	2.25	1.63
	1024	65.52	17.39	9.57	24.99	7.13	4.39
	2048	286.23	75.10	44.51	71.84	19.42	11.11
4	128	1.13	0.40	0.27	0.55	0.50	0.85
	256	4.15	1.16	0.71	2.02	0.81	0.92
	512	16.30	4.44	2.56	7.23	2.42	1.67
	1024	66.47	17.53	9.78	30.88	8.56	5.17
	2048	291.62	76.85	42.44	124.42	32.84	18.29
5	128	1.08	0.33	0.27	0.52	0.52	0.77
	256	4.06	1.11	0.68	1.69	0.78	0.95
	512	16.06	4.26	2.41	7.44	2.50	1.66
	1024	65.08	17.40	9.62	17.25	5.08	3.15
	2048	286.90	75.58	41.64	104.24	27.78	15.73
6	128	1.08	0.36	0.24	0.36	0.39	0.76
	256	4.11	1.16	0.68	1.48	0.66	0.91
	512	16.04	4.28	2.42	4.75	1.69	1.42
	1024	64.97	17.27	9.61	20.21	5.79	3.99
	2048	286.26	75.11	41.7	81.68	22.06	13.10
7	128	1.08	0.36	0.25	0.43	0.45	0.76
	256	4.07	1.22	0.68	1.02	0.53	0.93
	512	16.09	4.45	2.47	4.37	1.60	1.49
	1024	65.00	17.23	9.58	15.67	4.70	3.18
	2048	286.19	75.33	41.67	67.23	18.37	10.50
8	128	1.11	0.49	0.29	0.48	0.44	0.76
	256	4.09	1.17	0.81	1.80	0.77	0.91
	512	16.15	4.28	2.48	6.97	2.15	1.52
9	128	1.08	0.44	0.21	0.34	0.49	0.76
	256	4.04	1.23	0.67	1.22	0.58	0.85
	512	16.18	4.28	2.47	6.92	2.15	1.58
10	128	1.08	0.45	0.25	0.32	0.40	0.82
	256	4.05	1.21	0.66	0.98	0.68	0.85
	512	16.48	4.19	2.41	4.53	1.74	1.24
11	128	1.08	0.35	0.26	0.46	0.39	0.87
	256	4.06	1.20	0.67	1.68	0.71	0.89
	512	16.0	4.34	2.45	6.48	2.11	1.43
12	128	1.07	0.37	0.22	0.30	0.43	0.80
	256	4.10	1.21	0.66	1.01	0.62	0.83
	512	16.20	4.19	2.46	3.77	1.42	1.16

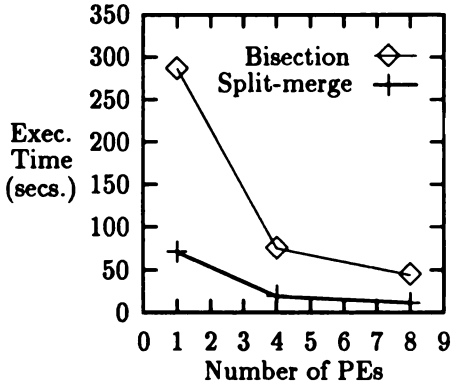
Table 4.1. Execution times (in seconds) for matrices of type 1 through 12 on a cluster of workstations



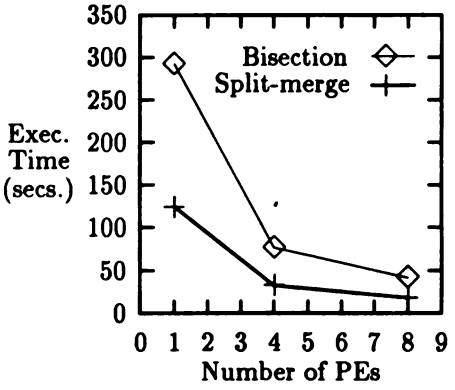
(a) Matrix with 2048 entries of type 1.



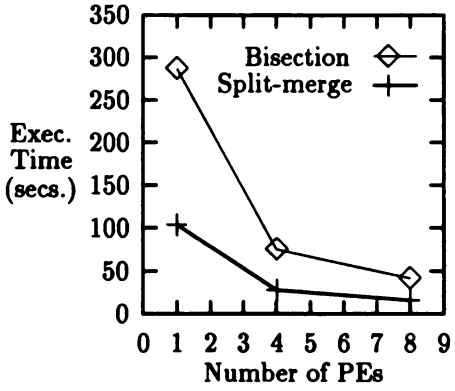
(b) Matrix with 2048 entries of type 2.



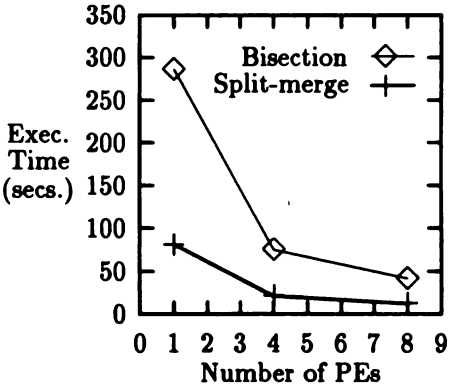
(c) Matrix with 2048 entries of type 3.



(d) Matrix with 2048 entries of type 4.

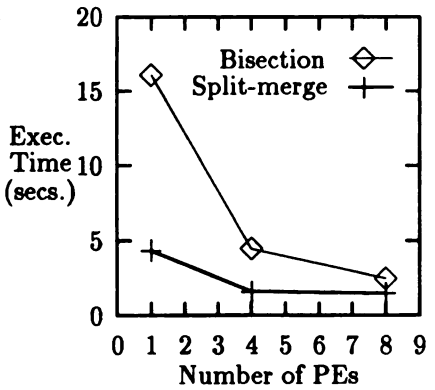


(e) Matrix with 2048 entries of type 5.

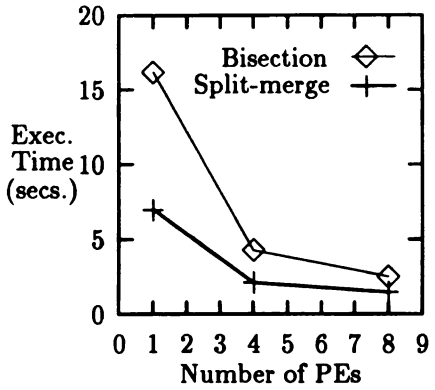


(f) Matrix with 2048 entries of type 6.

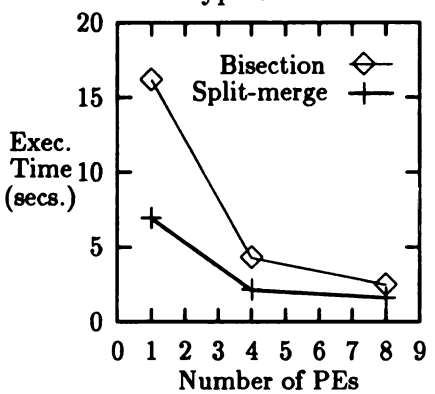
Figure 4.4. Execution times of parallel eigenvalue solvers on a cluster.



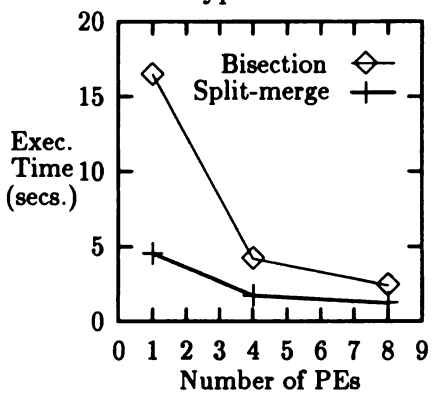
(a) Matrix with 512 entries of type 7.



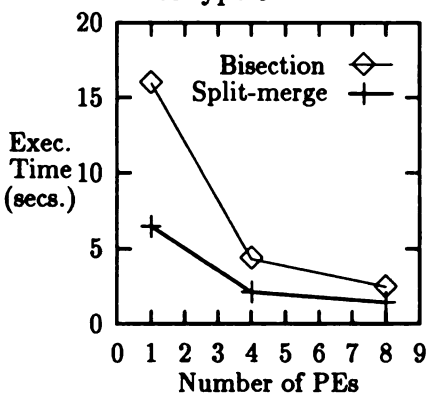
(b) Matrix with 512 entries of type 8.



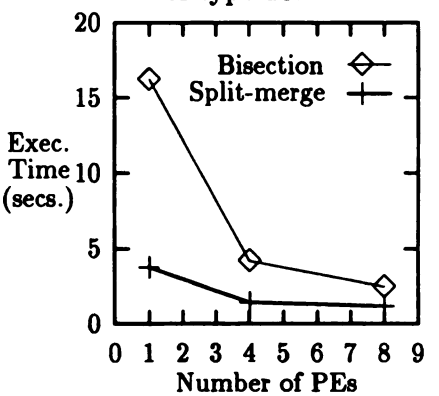
(c) Matrix with 512 entries of type 9.



(d) Matrix with 512 entries of type 10.



(e) Matrix with 512 entries of type 11.



(f) Matrix with 512 entries of type 12.

Figure 4.5. Execution times of parallel eigenvalue solvers on a cluster.

advantage, although the use of dual processes is effective in improving performance. On large matrices, the advantage of split-and-merge over bisection remains significant even on the cluster. For example, bisection requires 41.77 seconds to find the eigenvalues of a matrix with 2048 entries of type 1 on a cluster with 8 nodes, while split-and-merge requires only 12.93 seconds. Even if linear speedups could be obtained with bisection, 22 workstations would be required with bisection to obtain the same execution time as with split-and-merge on 8 workstations. On the nCUBE-2, for matrices of type 6, the advantage of split-and-merge over bisection was relatively small. That situation changes on the cluster, as the advantage of split-and-merge over bisection on matrices of type 6 on the cluster was very significant, as can be observed by comparing Figures 4.4(f) and 3.11(f).

Figure 4.5 shows that for matrices with 512 entries, the split-and-merge execution time begins to rise with 8 PEs compared to 4 PEs. The additional number of processes introduces communication in an additional stage in the execution of the algorithm, as this is similar to using more physical processing elements in an nCUBE-2. When the experiments were repeated on 8 workstations for matrices of 128 entries with 8 logical processes, instead of 16, slightly shorter times were observed. This suggests that the load balancing techniques discussed previously might be more useful for large matrices than for relatively small matrices. In other words, doubling the number of processes on every workstation is an useful strategy only for large matrices.

It should also be noted that faster networks and better interfaces between the workstation and the networks are becoming available and they will reduce the communication overhead of the clusters. As communications latencies decrease, the advantage of split-and-merge will likely increase. Our continuing study of such factors is discussed in chapter 5.

In examining absolute execution times, one can notice that, for matrices with 2048 entries, the execution times of the parallel version of split-and-merge on a cluster with 4 nodes are similar to the execution times of split-and-merge on a 32-node nCUBE-2, confirming the observations made previously for random matrices that the cluster implementations are competitive with those on an nCUBE-2. The execution times of the parallel version of bisection on 4 nodes on the cluster fall between the execution times with 16 and 32 nodes of bisection on the nCUBE-2. That is, for both the bisection and the split-and-merge algorithms, a very small cluster of Sparc 10 workstations represents a viable alternative to an nCUBE-2 with 32 nodes.

The reader may notice that the ratio of execution times between the sequential versions on an individual node on the nCUBE-2 and a Sparc 10 is not uniform for the algorithms. For example, split-and-merge requires 520.8 seconds to find the eigenvalues of a matrix of type 7 with 2048 entries on an individual node of the nCUBE-2 and 63.233 seconds on a Sparc 10, resulting on a ratio of 8.24. For bisection, the times are 1708.97 and 317.79 respectively, yielding a ratio of 5.38. A possible explanation for this difference is that split-and-merge may take better advantage of the cache of the Sparc 10 than bisection. Recall that only in the last stage of split-and-merge involves accesses to the entire matrix, whereas in bisection it is necessary to access the entire matrix at every iteration.

4.5 Summary

In this chapter, the implementation and performance evaluation of a parallel eigenvalue solver on a cluster of workstations connected via Ethernet have been described. Even though the communication costs are much higher on the cluster environment than on the MPC, the performance of a cluster for this particular algorithm is quite competitive. The presence of a full-fledged operating system on every workstation

permits the use of two instances of the program per node, which helps to mask the higher communication costs. On the other hand, the performance is very sensitive to the presence of other users on the workstations. It proved beneficial to include load balancing in earlier stages in order to account for the load imbalance caused by other users.

CHAPTER 5

Performance on Switch-Based Clusters

The need for higher interconnection rates among workstations has lead to the development of local area networks built from *high-speed switches.* These switches have been designed to serve other purposes besides cluster-based computing, but their higher speeds and capacities make them especially attractive for cluster-based computing. Two switch-based clusters of workstations are available in our department and we were interested in examining the performance of the cluster version of the split-merge algorithm on these environments.

The workstations in the High Speed Networking and Performance (HSNP) laboratory are interconnected using Ethernet and a set of three ATM switches. It is possible to use the TCP/IP and UDP/IP protocols over the ATM connections enabling the use of PVM. Another laboratory in the department, the Advanced Computing Systems (ACS) laboratory, is equipped with DEC Alpha 3000 workstations. The ALPHA workstations are interconnected through regular Ethernet and through a GIGAswitch, a crossbar switch. The connections between the workstations and the GIGAswitch use FDDI. Again, it is possible to use TCP/IP and UDP/IP, and hence PVM, across the GIGAswitch.

Two forms of hardware-supported multicast are available in the HSNP laboratory. First, the operating system of a subset of the workstations in the HSNP laboratory has been recompiled to incorporate the IP-Multicast protocol [26]. Second, the ATM environment provides hardware support for multicasting from a given node to a set of destinations [41]. In addition, the local version of PVM has been extended to provide a broadcast operation based on the recursive doubling [104], a software tree approach to multicast.

Experiments were conducted to evaluate the effects of the different environments, the different broadcast alternatives, and the enhancement to PVM. The experiments were conducted using matrices of type 7, as they seemed to be representative of the other types of matrices studied before. The results are discussed in the following sections.

5.1 Cluster Environments

The HSNP laboratory in our department includes an ATM testbed. Figure 5.1 depicts the laboratory configuration.

The testbed equipment includes 12 Sparc-10 workstations, 4 Sparc-2 workstations, and miscellaneous networking equipment. The workstations are networked together via both conventional Ethernet as well as via the ATM network. The latter comprises three FORE Systems ASX-100 ATM switches. Each ASX100 ATM switch provides up to 16 full-duplex ports, allowing for system growth. A Sparc RISC processor on each switch is dedicated to tasks, such as connection management and traffic monitoring. A 1.2 Gbps time-division-multiplexed bus constitutes the switch fabric, enabling each I/O port to operate at the full channel rate. The switch fabric is shared among the 16 ports and the switch processor. Among other features, the bus-based switch fabric provides an efficient method of supporting multicast communication among the hosts.

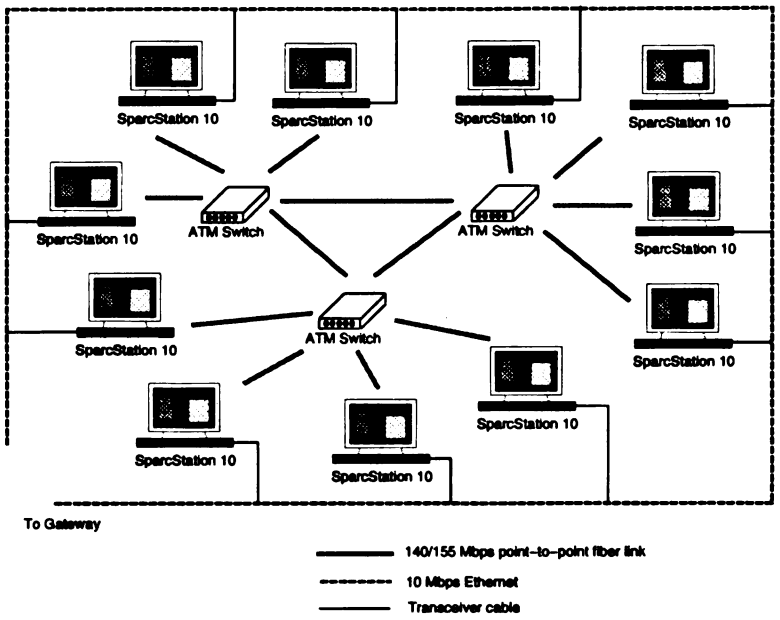


Figure 5.1. ATM cluster testbed at Michigan State University

Each workstation is equipped with an SBA-200 SBus adapter card, which is connected to a switch port by two unidirectional fiber optic links. The SBA-200 contains a dedicated Intel i960 RISC processor and features DMA with scatter-gather capabilities, as well as custom hardware support for segmentation and reassembly of ATM cells. The device driver for the SBA-200 not only provides an interface to the standard TCP/IP protocol suite, but also supports the Application Program Interface (API), a set of socket-like system utilities that allow user-level programs to directly access ATM-specific operations, such as hardware multicast. The SBA-200 card and its driver support ATM Adaptation Layers (AAL) 3/4 and 5. AAL5 [105] is an international standard designed primarily to provide efficient data communications over ATM networks.

Another cluster-based parallel computing platform has been recently added a to the ACS laboratory. The cluster includes six DEC Alpha-3000 workstations interconnected by a DEC GIGAswitch. The workstations are also interconnected using

regular Ethernet. Each port on the GIGAswitch supports 200 Mbps full-duplex FDDI. An FDDI connection exists between every workstation and a corresponding port on the GIGAswitch. The GIGAswitch, configured internally as a crossbar, has a maximum aggregate bandwidth of 3.6 Gigabits/second. It allows multiple simultaneous connections. Figure 5.2 represents the ACS cluster configuration.

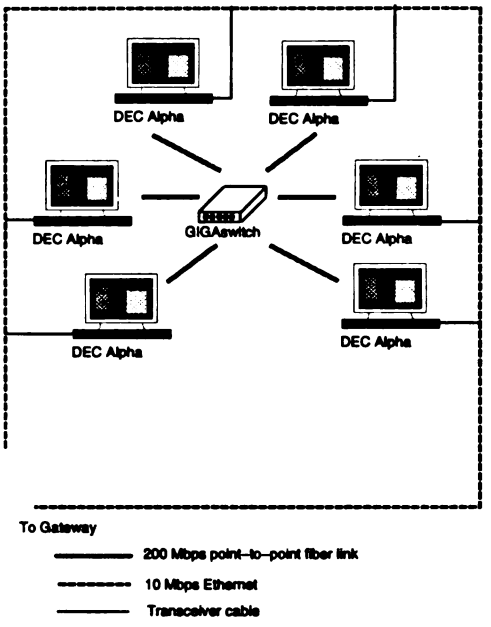


Figure 5.2. GIGAswitch cluster testbed at Michigan State University

5.2 Experiments Using Original PVM

Every workstation in the HSNP laboratory has two IP addresses and corresponding names. One corresponds to the Ethernet connection and the other one to the ATM interface. When executing PVM, one specifies in a *hosts* file the workstation names of the workstations that will form the "virtual machine". By including in the hosts

file the workstations names that correspond to the ATM connections, PVM directs all traffic through the ATM interfaces instead of using the Ethernet connections.

An experiment was conducted using 8 workstations to calculate the eigenvalues of matrices of sizes ranging from 128 to 2048 entries to compare ATM with Ethernet. Figure 5.3 shows that, for small matrices, the difference in performance is negligible, but as the size of the matrix grows, the faster communications that the ATM environment provides, result in a shorter execution time.

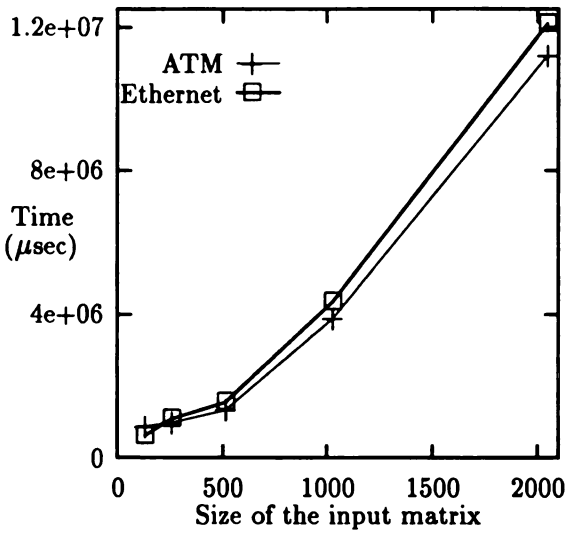


Figure 5.3. Comparison of the performance of Ethernet and ATM

A similar situation, regarding connections and addresses, exists in the cluster of DEC Alpha workstations. That is, every workstations has two addresses, one corresponding to the connection to the GIGAswitch and another one to the regular Ethernet connection. As with the ATM switch, it is possible to use the TCP/IP and UDP/IP protocols across the GIGAswitch.

The microprocessor used on the ALPHA workstations has better performance than the Sparcs on 64 bits floating point operations and hence the speed of execution is much faster. Figure 5.4 shows the execution times on a matrix of type 7 of size 2048 on clusters of different sizes using regular Ethernet. It can be observed that on sequential mode, an Alpha is significantly faster than a Sparc-10. A cluster of 4 Sparcs has a performance similar to that of a single ALPHA.

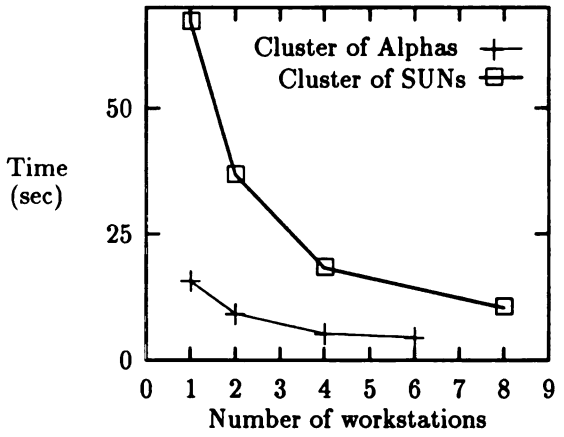


Figure 5.4. Comparison of the performance of a cluster of DEC and a cluster of SUN workstations

Experiments were performed on a cluster of 4 workstations with input matrices of different sizes using the Ethernet and the GIGAswitch networks. The procedure was similar to the one followed in the experiments on the ATM switch. Figure 5.5 presents the results of the comparison of the two networks. As in the case with the ATM and the Sparcs, the GIGAswitch provides a significant advantage for larger matrices. It should be noted that 4 is a very small number of workstations and that experiments with larger numbers of workstations would likely be more indicative of the scalability

of the switch. Nevertheless, the results indicate the performance gain to be obtained by using an alternative to Ethernet.

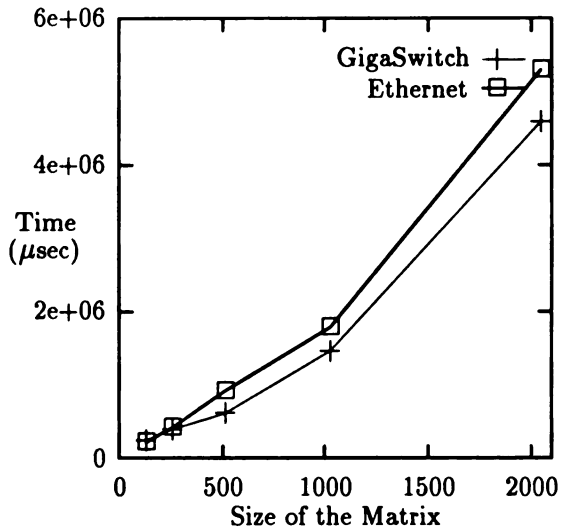


Figure 5.5. Comparison of the performance of the program on Ethernet and the GIGAswitch

5.3 Using IP-Multicast

In the late 80s, extensions to the UDP/IP protocols were proposed and implemented to multicast a message from one machine to a set of destinations [26]. The programming interface is very similar to regular socket programming. The only differences are that is necessary to set certain options to the socket and that one needs to specify the id of a group that the program joins. Any node that has joined the group can broadcast to the other members of the group, and it can also receive the broadcasts from any other member of the group. Several processes on the same workstation can join the broadcasting group.

This multicast primitive is not reliable. There are no guarantees of correct delivery as in TCP/IP. An experiment was conducted to test the reliability of the IP-Multicast in which an array of 256 doubles (2K bytes) was broadcast 1000 times using IP-Multicast. The receiver compared the received values with the correct values and no errors were detected over the 1000 tries. Given the high reliability of Ethernet, this result was to be expected. Nevertheless, production code would require that a reliability protocol be implemented above IP-Multicast.

It is possible to overrun the receiver when using IP-Multicast, that is, if two multicasts are executed over a short period of time, the second one might be lost because the receiver has not finished processing the first one. According to our experiments, the multicast messages could be of a size of up to 2K bytes (256 doubles). As a result, messages larger than 2K bytes needed to be split into pieces of 2K bytes and the broadcaster had to wait for ACKs from the receivers before proceeding with the next piece.

The split-merge program was modified to use IP-multicast instead of using the regular PVM *mcast* primitive, which sends the message from the sender to each destination sequentially. The original and modified versions of the program were executed on the six Sparc-10 workstations in the HSNP laboratory that support IP-multicast. The results can be observed in Figure 5.6. The differences are minimal.

Two factors explain the lack of improvement with IP-Multicast: The number of workstations involved is very small, so the broadcast operation of PVM is still relatively efficient. Second, the need for ACKs from the receivers for larger matrices serializes the communication process. As new protocols for the Internet and for cluster computing are explored, it would be beneficial for cluster computing users to have a reliable IP-Multicast primitive that allows large packets to be broadcast. While such a reliable multicast will require the use of either ACKs or NAKs, those will not be issued by the user, resulting in better performance.

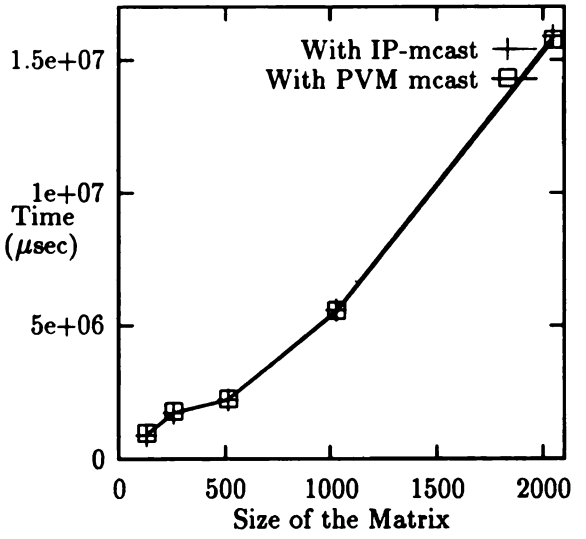


Figure 5.6. Comparison of the performance of the program using IP-Multicast and regular PVM bcast

5.4 An Improved Broadcast Implementation in PVM

PVM has been modified [104] by adding a more efficient group broadcast operation. The broadcast operation is based on the U-cast algorithm [36]. From the programmer's point of view, one of the nodes creates a group of processes by calling a new routine and passing as a parameter the task ids of the processes that belong to the group. The routine returns a *group id.* When a node needs to broadcast, it issues a regular send statement but indicating the group id as the destination. The receivers perform a regular receive operation. It is feasible to use two processes per workstation.

Two sets of experiments were performed. Both compare the performance of a regular version of PVM versus the new enhanced version of PVM. One set of experiments was performed over regular Ethernet. The results can be observed in Figure 5.7. The other one was run using the ATM environment. The results can be observed in

Figure 5.8. It can be observed that the new version of PVM results in slightly better performance on both environments for most of the sizes of the matrices.

In these experiments, the performance gain obtained with the new version of PVM is superior to the original PVM, which uses separate addressing to implement multicast. The number of messages and acknowledgments at any node is at most logarithimic in the number of destinations, implying that this method should scale better. The number of workstations used for these experiments is rather small. A larger number of workstations would probably show a more significant advantage for the new broadcasting alternatives. In the nCUBE-2, with small numbers of nodes, using a naive broadcast did not cause a significant difference in performance in comparison with a tree based broadcast. The fact that even on a small number of workstations there is a difference in performance seems to indicate that the results on larger number of workstations should be similar to those of the nCUBE.

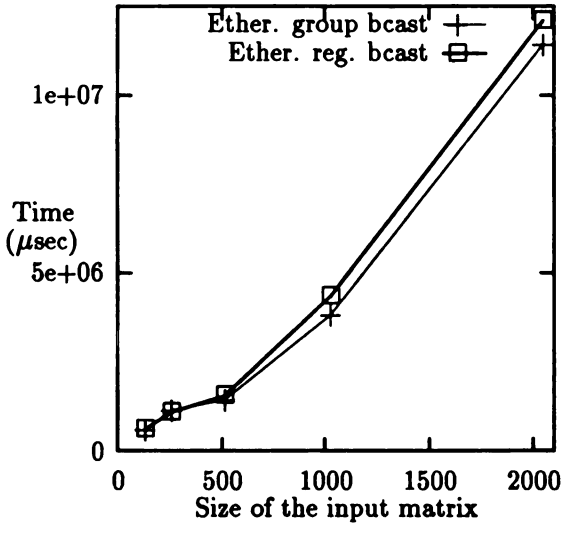


Figure 5.7. Comparison of the performance of regular PVM and the enhanced PVM over Ethernet

It should be noted, as well, that only 4 broadcast operations are taking place for the matrices of sizes between 512 and 2048, and only 3 broadcasts are taking place for the matrices smaller than 512.

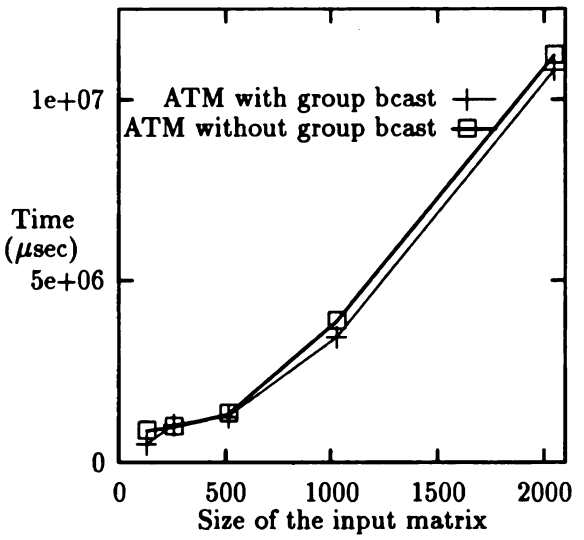


Figure 5.8. Comparison of the performance of regular PVM and the enhanced PVM over ATM

5.5 Experiments with ATM Multicast

ATM was originally designed as a standard for telecommunications. Its high bandwidth, though, makes it attractive for data communications as well. To make it feasible to use ATM in computing applications, a programming interface standard has been designed. It is called ATM Adaptation Layer 5 (AAL5) [105].

An unreliable version of the hardware supported multicast operation was incorporated into the split-merge program. In ATM multicast, every process that is to join a multicast group has to specify whether it is going to be the broadcaster (and only

one process can be the broadcaster) or a receiver. The maximum size of a packet that can be sent using ATM multicast is 4096 bytes (512 doubles). As in IP-multicast, it is possible to "swamp" the receiver if two broadcasts occur very close to each other. Again as in IP-multicast, for large packets, it is necessary to send ACKs from the receivers to the broadcaster before more broadcasts are sent. Unlike IP-multicast, though, only one process per workstation can join a broadcast group with the same address, which precludes doubling the number of processes per workstation.

Figure 5.9 presents the results of the test on a group of 8 workstations. One set of tests was run using regular PVM broadcast, while the other was run using ATM multicast. All other (non-multicast) communication was performed over regular Ethernet. The ATM multicast results in slightly better performance in matrices of size of up to 512, which corresponds to the maximum packet size that can be sent without acknowledgments from the receivers. Beyond the size of 512, the regular PVM broadcast results in better performance as the gain in broadcast performance obtained by using the ATM broadcast operation is dominated by the cost of the transmission of the acknowledgments. Again, if the flow control mechanisms are incorporated into lower-level protocols, better performance for ATM may result.

In this particular application, the fact that the broadcast operation is not symmetric, that is that only one member of the group can broadcast, was not relevant. But, comparing the IP-multicast with the ATM-multicast user interface, one notices that the IP interface is more flexible as any node in the group can broadcast and two or more processes in a given workstation can join the group without requiring different addresses.

Recently, a reliable implementation of multicast over ATM networks has become available [41]. The set of routines that implement multicast includes daemons on both the sending and receiving sides that take care of processing the acknowledgments. The

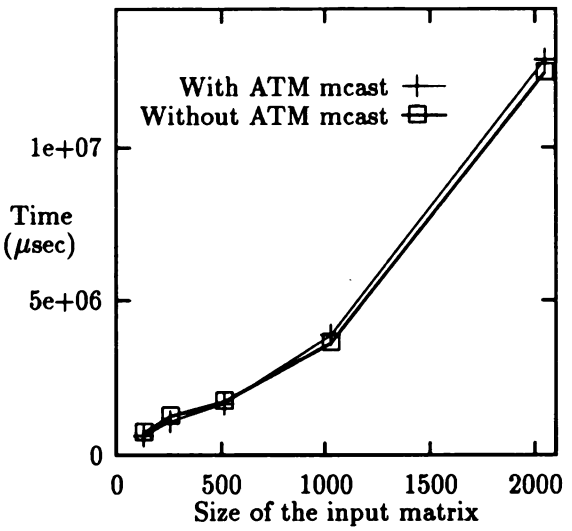


Figure 5.9. Comparison of the performance of the program using unreliable ATM-Multicast and regular PVM bcast

experiments were repeated with this multicast primitive. Removing the acknowledgments from the application process produces an improvement in performance that can be observed in Figure 5.10. These results are particularly encouraging, given that only three broadcasts occur in the 8-processor case.

5.6 Summary

The experiments described in this chapter compared the performance of the program under different communication environments and the results of using different implementations of the broadcast operation. The use of both switch-based environments, the ATM cluster and the GIGAswitch cluster, resulted in noticeable, albeit small, performance improvements over regular Ethernet. The standard deviations of the run times were smaller than the performance differences observed. For instance, the standard deviation for the executions of the program on the cluster of Sparcs

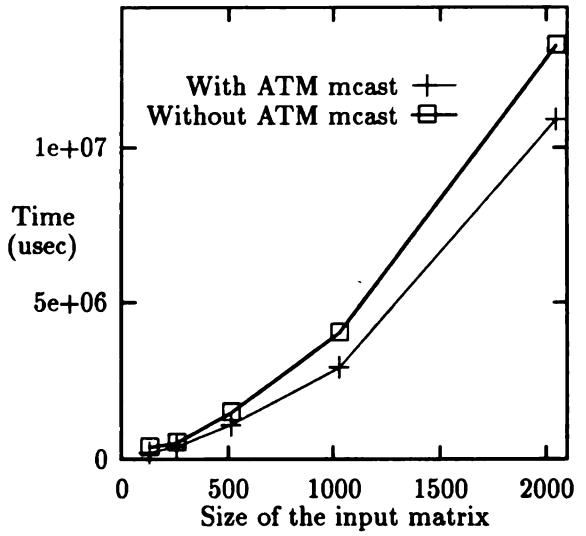


Figure 5.10. Comparison of the performance of the program using reliable ATM-Multicast and regular PVM bcast

using Ethernet was 184620 microseconds and the mean was 12.09 seconds. The same program, using the ATM cluster, had a standard deviation of 198902 microseconds and a mean of 11.21 seconds. The ALPHA workstations are significantly faster than the Sparc-10s for this numerically intensive algorithm.

The different broadcasting alternatives that were explored produced minor performance improvements when the acknowledgments were handled at the user level and an appreciable improvement when the acknowledgments were handled by the routines implementing multicast. In the case of IP-mcast based broadcast and unreliable AAL5 ATM broadcast, the need to send acknowledgments back from the destinations to the broadcaster seems to diminish the gain obtained by the more efficient broadcast operation. The reliable implementation of ATM multicast produced better performance that can be attributed to a more efficient handling of the acknowledgments. The enhanced implementation of the broadcast operation in PVM also produced performance gains. Two caveats should apply to these observations:

the number of workstations used in the tests was relatively small and the algorithm uses a small number of broadcast operations. When we compared the performance of the "naive" one-to-one broadcast against the performance of the tree-based broadcast on the nCUBE-2, the differences became noticeable only after more than 16 nodes were used. Other algorithms, like the tridiagonalization of a full matrix, use broadcast operations much more intensively and would likely benefit more from better broadcast implementations.

CHAPTER 6

A Model of Split-merge Performance

After performing the experiments, a model was developed to capture the most relevant aspects of the behavior of the program under different environments. A model allows us to better understand the observed behavior of the program on different environments, and to make some predictions about the behavior under different conditions.

This model, given a set of values of a particular matrix and the parameters of the runtime environment, produces a prediction of the execution time of the program. The time spent in I/O operations is ignored in this model.

6.1 Parameters

Table 6.1 contains a short description of the parameters that were considered for this model. The left column contains the symbol used to denote the parameter in the equations.

The distribution of the number of iterations of the input matrix is taken into account in the parameter $iter(i, j)$. As can be observed in Figure 3.7, the particular distribution of the number of iterations required by a particular matrix is very

Parameter	Short Description
n	Size of the input matrix
k	$\lceil log_2(n) \rceil$
$iter(i,j)$	Number of iterations required by eigenvalue i at stage j
sp	Speed of the microprocessor on a node (FLOPS)
p	Number of nodes
d	$log_2(p)$
c_i	Size of initial workload
c_s	Size of subsequent workload
$comm$	Communication Speed (node to node)
$W()$	Waiting time for additional work
$B()$	Time taken by the broadcast operation
$G()$	Time taken by the gather operation
$S()$	Time taken by the scatter operation

Table 6.1. Parameters considered in the model

unique and does not seem to fit easily into well-known statistical distributions. The distributions of other types of matrices were examined and were found to have similar shapes. The number of iterations is inherent to the particular matrix and the algorithm. It is independent from any particular implementation of the algorithm (except for considerations of precision).

The speed of the microprocessor is a difficult parameter to estimate as it depends not only on the speed of the processor itself but on other factors as the size of the cache and the memory access pattern of the program. For simplicity, it will be assumed that the number of processors is a power of 2.

$W()$,$B()$, $G()$ and $S()$ are functions of other parameters. These functions will be discussed in the following sections. $W()$ depends on the sizes of the initial and subsequent workloads, and on speeds of the microprocessor and the network. $B()$, $G()$ and $S()$ depend on the particular implementations of broadcast, gather, and scatter respectively, as well as on *comm*, the speed of the node-to-node communication and the n the number of nodes.

6.2 Iteration Characteristics

Every iteration requires a pass through the values of the diagonal and off-diagonal entries of the submatrix whose eigenvalues are being calculated in the current stage. Approximately 8 additions and 8 multiplications are required for each value, roughly 16 floating point operations.

Although there is computation involved in finding the eigenvalues of the submatrices of size 2×2 and in the merging operations, the dominating term in computation time is the time spent in Laguerre's iterations. Thus, a rough estimate of the total amount of floating point operations is given by:

$$\sum_{i=2}^{k} \sum_{\ell=1}^{n} iter(\ell, i) * 2^i * 16 \tag{6.1}$$

The first summation corresponds to the stages in the algorithm, while the second corresponds to the work needed for all eigenvalues. The complexity of the algorithm is $O(n^2 log_2(n))$ time. If all other parameters are kept constant, as n grows, the computation cost dominates all other costs since the complexity of the communications grows at a rate of at most $O(n)$.

In a parallel machine with $p = 2^d$ processors, the first $k - d$ stages do not require communication. The workload is distributed statically. All nodes have to wait for the last one (the one with the maximum workload) to finish before proceeding with the distributed stages. Thus, the time to finish the first $k - d$ stages is approximately:

$$MAX_{over\ all\ p}\left(\frac{\sum_{i=2}^{k-d} \sum_{\ell=1}^{n/p} iter(\ell, i) * 2^i * 16}{sp}\right) \tag{6.2}$$

All distributed stages involve a gathering at the coordinator from the sinks at that particular stage, followed by a broadcast from the coordinator to all workers.

Recall that one node is acting as a coordinator, so the number of workers is $p-1$. Out of the n eigenvalues to be calculated, $c_i * (p-1)$ are assigned initially to the workers. This means that $n-(c_i*(p-1))$ will be distributed later and that $\lceil \frac{n-(c_i*(p-1)}{c_s} \rceil$ requests for work will take place. Let $lreq_j$ denote the number of requests for work from node j. So an approximate value for execution time of the distributed stages will be:

$$\sum_{i=k-d+1}^{k} G() + MAX_{all\ p}[B() + \frac{(\sum_{\ell=1}^{c_i} iter(\ell,i)*2^i*16)}{sp} + lreq_j * W() * (\frac{\sum_{\ell=1}^{c_s} iter(\ell,i)*2^i*16}{sp})] + S().$$

The total estimated execution time is the sum of the estimates for the local and the distributed stages.

6.3 Waiting Time

The time $W()$ that a processor waits for more work to arrive is a stochastic variable. Unfortunately, it is not clear how to model the behavior of this variable. In a sense, this system could be described, approximately, with a queuing theory model, since there is one single server whose service times should be approximately uniform. The problem arises in modeling the distribution of the requests, as they are related to the distribution of the numbers of iterations, the stage in which the algorithm is in and the size of the workload. The closest model that seems applicable is the "machine repair model" which has been used to model systems with a fixed number of customers, and one single server [106]. This model assumes that the time between the requests for service from the customers has an exponential distribution with an average value of $E[t] = \frac{1}{\alpha}$ and the service time also has an exponential distribution with an average of $E[s] = \frac{1}{\mu}$. This queueing systems always reaches a steady state because the number of customers is finite. The actual system differs from this model in that the service time seems to be fixed (disregarding interferences from other processes on the server)

and the time between requests depends on the distribution of the required number of iterations.

The average time between requests depends on the amount of work that every node performs on a "chunk" and the speed of the microprocessor . The number of operations in turn depends on the size of the workload, and the size of the matrices being solved. The average service time depends on the speed of the network (disregarding contention for the network), the speed of the microprocessor and the overhead for protocol processing and context switching. The expression for the mean response time is:

$$W = \frac{pE[s]}{\rho} - \frac{1}{\alpha} \tag{6.3}$$

where ρ, the server utilization, is calculated as:

$$\rho = 1 - [\sum_{m=0}^{p-1} \frac{(p-1)!}{((p-1)-m)!}(\frac{\alpha}{\mu})^m]^{-1} \tag{6.4}$$

The main result of this part of the model, which is confirmed by the trace files, is that the amount of time spent solving a "chunk" of eigenvalues by a worker should be significantly greater than the service time of the coordinator, to avoid queuing requests for work at the server. The mean response time is also affected by the number of workers. The larger the number of workers, the larger the ratio should be between the amount of time required to finish a "chunk" and the service time to avoid the queuing of requests. An example of this phenomenon is illustrated in the next chapter in section 7.3. In terms of the model, if the server utilization is very low, then W is asymptotic to the line $E[s]$, the value for $p - 1 = 1$. As $p \rightarrow \infty$ then W is asymptotic to the line $pE[s] - \frac{1}{\alpha}$. The two asymptotic lines intersect at a value

called by Kleinrock the *system saturation point*:

$$SSP = \frac{E[s] + E[t]}{E[s]}. \tag{6.5}$$

As was discussed in chapter 3, there is a tradeoff between small workloads, which minimize the variance in the finishing times but increase the time waiting for more work and large workloads which minimize the time waiting for more work but increase the risk of load imbalance.

The effect of the period waiting for more work can be diminished by doubling the number of processes per processor. On the other hand, doubling the number of processes increments the number of distributed stages by one, and hence the communication overhead.

6.4 Broadcast Time

The broadcast function depends on the topology of the interconnection among the processing elements. On the nCUBE-2, the time complexity of the broadcasting function is $O(n*log_2(p))$ and the broadcast primitive acts as a barrier synchronization, all receivers resume their activity at approximately the same time. The performance of different broadcast implementations atop of ATM is presently under study [41]. For large messages (as the ones involved in this particular program), that work shows that the time complexity of PVM's mcast primitive is $O(n*p)$, recursive doubling has a complexity of $O(n * log_2(p))$, while hardware based (AAL5) multicast has basically a cost that is independent of the number of processors and depends only on the size of the message being broadcast, $O(n)$. The hardware based multicast operation that they examined is reliable and the acknowledgments from the receivers are combined in a tree to reduce the collection times. Because the acknowledgments are very

small in comparison with the message being broadcast, the broadcast time dominates and the time required for receiving the acknowledgments can be ignored. The same researchers also compared the performance of several broadcast primitives on top of Ethernet: Recursive doubling, iterative pvm_sends, and pvm_mcast. The complexity of all these operations is essentially linear, $O(p)$, although there are some performance differences depending on the size of the messages.

The broadcast cost is not a significant factor in performance with a small number of processors, but as the number of processors grows, as illustrated by our experiments on the nCUBE, the cost of the broadcast operation can become dominant. As the number of workstations increases, the number of distributed stages increases as well and the amount of computation time per node is bound to decrease. Therefore, if large numbers of workstations are to be used in a cluster, a better broadcast primitive is needed. The results of [41] indicate that an ATM switch can provide a significant advantage in the broadcast operation. Our experiments confirm that just using the ATM environment provides a performance advantage. On the other hand, it seems that if one uses Ethernet with a large number of workstations, communications (broadcast especially) are likely to become a bottleneck and affect the efficiency of the program.

Figure 6.1 plots the efficiency of using different numbers of workstations for a matrix of a given size with various broadcast implementations. The plot was produced with a simplified version of this model in which all factors, except the broadcast cost, were assumed to remain constant. As the number of processors increases, the amount of work per processor diminishes and the participation of the broadcast time on the total time increases, diminishing the efficiency.

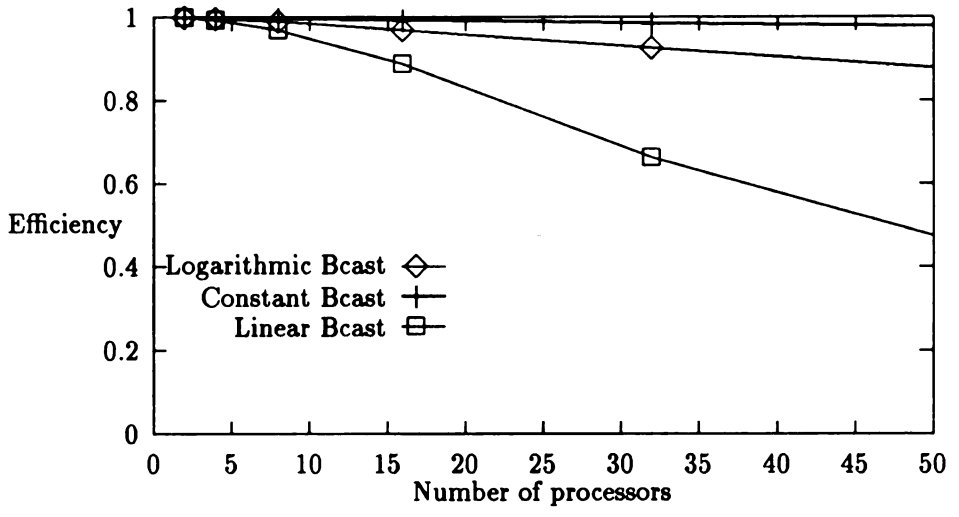


Figure 6.1. Efficiency as a function of the number of processors with different broadcasting functions.

6.5 Gather and Scatter Times

The gather and scatter operations, described in section 2.3, might also become bottlenecks with a large number of processors. Unlike the broadcast primitive, though, whose cost remains constant through all distributed stages, the cost of the gather and scatter operations changes from one distributed stage to the next one. In the first distributed stage, half the processors are sinks so the coordinator has to perform a gather and a scatter from and to $\frac{p}{2} - 1$ sinks. The number of sinks decreases by half in each successive distributed stage and in the last stage the coordinator gathers from only one sink.

A result concerning efficiency similar to the one obtained for broadcast could be obtained regarding the impact of the gather and scatter operations. That is, given a matrix of a certain size, as the number of processors increases, the work per node

decreases and the amount of time spent in the gather and scatter operations might become dominant as a fraction of the total execution time.

6.6 Scalability

How many processors can be used efficiently with this implementation of the algorithm? Different answers are obtained for different environments.

The rich topology of the hypercube makes it scalable in terms of both broadcast and gather. Our experiments confirmed that up to 64 processors can be used efficiently. The centralized load balancing approach might become a bottleneck when the *system saturation point* is reached. Once that point is reached, a hierarchical load balancing approach would become attractive. Even though more processors would be devoted to the role of coordinators, the reduction in the service time at the coordinator would maintain a high efficiency.

In the Ethernet environment, it might be feasible to design an efficient broadcast primitive that takes advantage of the "broadcast" nature of the medium. With such a broadcast implementation, it would be feasible to increase (scale) the number of processors. On the other hand, it is much more difficult to make gather and scatter operations scalable. The same is true for the load balancing approach. Eventually, the single medium becomes a bottleneck. The scalability of Ethernet-based clusters of workstations is limited.

Switch-based clusters, in contrast, allow efficient implementations of all communication primitives: Broadcast, gather, scatter and load balancing. By allowing simultaneous communication among different pairs of nodes, scalable tree-based broadcast, scatter and gather can be implemented. The solutions suggested for the nCUBE-2 regarding load balancing could also be applied in this environment. That is, Ethernet-based cluster computing is a cost-effective alternative for parallel computing but its

scalability is limited. Switch-based cluster computing offers better performance and better scalability.

6.7 Summary of Eigenvalue Study

In the previous 4 chapters, the implementation and performance evaluation of a parallel eigenvalue solver on both an MPC and a cluster of workstations have been described. The general structure of the split-merge algorithm for finding eigenvalues in symmetric tridiagonal matrices is well-suited to efficient parallelization. The algorithm uses Laguerre's iteration and exploits the separation property in order to create subtasks that can be solved independently.

The split-merge algorithm was first implemented and studied on an nCUBE-2, a wormhole-routed hypercube. In the initial parallel version of the algorithm, the only *required* communication among processes occurs only between stages of the algorithm; the number of stages is at most $\log_2(n)$, where n is the order of the matrix. The inherent variance in the number of iterations in the split-merge algorithm justified more sophisticated approaches to load balancing. The communication used to implement dynamic load balancing is, in some sense, *programmable*. This characteristic of the communications in the algorithm gives it the potential to also perform well on a cluster of workstations.

In order to test this hypothesis, the split-merge algorithm was implemented on a cluster of Sun Sparc-10, models 30 and 40 workstations, interconnected by a regular Ethernet network and by an ATM switch. Some experiments were also performed on a cluster of DEC Alpha workstations interconnected by a regular Ethernet and by a GIGAswitch.

Experiments on a variety of input matrices confirmed that split-merge is significantly faster than bisection for large matrices (order 512 or larger) in both parallel

environments and revealed the performance ratio between an nCUBE-2 and a cluster of workstations. Split-merge performed better than bisection for matrices of size 2048 on both environments and across the tested number of processors. It was observed that as the size of the matrices increased, the relative advantage of split-merge also increased. This trend seems to indicates that the advantage of split-merge should be even greater for matrices larger than the ones tested. On the other hand, beyond a certain number of processors, which depends on the particular environment, bisection performed better on small matrices (order 128). This indicates that it might be better to use bisection for small matrices when the available number of processors exceeds a certain threshold that depends on the particular environment.

The small number of workstations available did not allow the experiments to show clearly the advantage of the better broadcast options. But the model of the behavior of the system indicates that with larger numbers of processors, the broadcast, scatter and gather operations might become bottlenecks, hence the need for faster and more efficient broadcast operations as well as better scatter and gather operations.

CHAPTER 7

A Parallel Singular Value Algorithm

The singular value decomposition (SVD) of a matrix is an important tool in numerical linear algebra and it has applications in many scientific fields. The problem of finding the singular value decomposition (SVD) of an $m \times n$ real matrix A, with $m \geq n$, can be stated as follows: Find the values $\sigma_1, \ldots, \sigma_n$ such that $U^T A V = diag(\sigma_1, \ldots, \sigma_n)$, where $U \in \mathbf{R^{m \times m}}$ and $V \in \mathbf{R^{n \times n}}$ are orthogonal matrices. The SVD can be used to detect matrices that are singular or numerically very close to singular. In some cases, it will even provide an approximate answer for an ill-conditioned system of linear equations [78]. SVD is also used very often to solve linear least squares problems [78]. Linear least squares problems arise in many different fields, where, given a set of experimental measurements, one wants to find the parameters of the system. The SVD is also used in cartography to adjust field measurements to a representation in a map.

Solving for the SVD of a matrix is a computationally-intensive task, and reducing its execution time will improve the performance of the applications that require it. An approach that is frequently used is to transform an $m \times n$ real matrix A into a bidiagonal matrix B, whose singular values are the same as those of A. Therefore, a

fast algorithm to find the singular values of a bidiagonal matrix is an important part of the overall process of finding the singular values of a real matrix A.

Different methods have been used to solve the SVD problem of bidiagonal matrices. Given an $n \times n$ bidiagonal matrix B, its singular values can be found by computing the eigenvalues of the symmetric tridiagonal matrix B^TB, where B^T is the transpose of B, and taking the square roots of those eigenvalues. The disadvantage of this approach is that the values calculated for the smallest singular values are not very accurate because the values of the original entries of B are squared in the process of calculating B^TB, leading to roundoff errors. A more accurate method that can be used to find the small singular values is to create a symmetric tridiagonal matrix T of size $2n \times 2n$, with zeroes in the main diagonal and the entries of B in the offdiagonal. The positive eigenvalues of this matrix are identical to the singular values of B [107]. The drawback of this method is the larger size of the matrix T, which implies a greater computational effort.

Li *et al* [108] recently proposed a new SVD algorithm that combines both methods in order to find all the singular values accurately and efficiently. A *threshold* is calculated based on the matrix B^TB (details of the calculation will be given in section 7.1). The singular values below the threshold are computed by finding the eigenvalues of the matrix T, while those above the threshold are found by calculating the eigenvalues of the matrix B^TB.

In this chapter, we report the results of parallelizing Li's algorithm on an nCUBE-2 and across a cluster of workstations. Two versions of the new algorithm were tested. One uses the *split-merge* algorithm [77] for finding the eigenvalues of symmetric tridiagonal matrices. The other one uses the well known *bisection* algorithm for the same problem.

7.1 A New Algorithm

Given an $m \times n$ real matrix A, with $m \geq n$, the process of calculating its SVD can be described as: Compute the values $\sigma_1, \ldots, \sigma_n$ and the matrices U and V, such that $U^T AV = diag(\sigma_1, \ldots, \sigma_n)$, where $U \in \mathbf{R^{m \times m}}$ and $V \in \mathbf{R^{n \times n}}$ are orthogonal matrices. An approach that is frequently taken when finding the SVD of a matrix of size $m \times n$ is to reduce the matrix by orthogonal transformations to an $n \times n$ bidiagonal matrix B as shown in Figure 7.1. It can be assumed without loss of generality that all of the

$$B = \begin{bmatrix} s_1 & e_1 & & & \\ & s_2 & e_2 & & \\ & & s_3 & \ddots & \\ & & & \ddots & e_{n-1} \\ & & & & s_n \end{bmatrix}$$

Figure 7.1. Bidiagonal matrix

$s_i, i = 1, \ldots, n$ and $e_i, i = 1, \ldots, n-1$ are nonzero. This assumption implies that all $\sigma_i, i = 1, \ldots, n$, are positive and distinct. That is,

$$\sigma_1 > \cdots > \sigma_n > 0.$$

Two different approaches can be used to find the singular values. First, the singular values of B are the square roots of the eigenvalues of the symmetric tridiagonal matrix $B^T B$ [109](p. 320). Therefore, finding the eigenvalues of $B^T B$ and then taking the square roots of them produces the singular values of B. The drawback of

this approach is that the small singular values of B cannot be calculated accurately because the entries have been squared.

The second approach is based on finding the eigenvalues of T, another symmetric tridiagonal matrix, that is constructed from B as follows. Let

$$\tilde{T} = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}. \tag{7.1}$$

Then the positive eigenvalues of $\tilde{T}$ are the singular values of B [107]. Let P be a permutation matrix which reorders the rows and columns of $\tilde{T}$ in the order $1, n+1, 2, n+2, \ldots, n, 2n$, as shown in Figure 7.2. The matrix T is a symmetric tridiagonal matrix of order $2n \times 2n$. The matrices T and $\tilde{T}$ are similar, therefore they have the same eigenvalues, so the positive eigenvalues of T are also the singular values of B.

$$T = P^T \tilde{T} P = \begin{bmatrix} 0 & s_1 & & & & & \\ s_1 & 0 & e_1 & & & & \\ & e_1 & 0 & s_2 & & & \\ & & s_2 & 0 & e_2 & & \\ & & & e_2 & \ddots & \ddots & \\ & & & & \ddots & 0 & s_n \\ & & & & & s_n & 0 \end{bmatrix}$$

Figure 7.2. Symmetric Tridiagonal Matrix T

The following example illustrates the above transformations. Given the bidiagonal matrix B in Figure 7.3(a), one can easily construct the symmetric tridiagonal matrices B^TB, shown in Figure 7.3(b), and T, which can be seen in Figure 7.3(c). Notice that because the value of $B[2,2]$ is so small, when one calculates B^TB, $B^TB[2,2]$ becomes 1.0. It turns out that the singular values of the matrix B in this example are

1.4142 and 7.0710e-11, as calculated by the routine DBDSQR in LAPACK [55]. The eigenvalues of the matrix B^TB are, according to the routine DSTEQR of LAPACK, 0.0 and 2.0. The square root of 2.0, 1.4142, coincides with one of the singular values of B but the other one (square root of 0) does not. The positive eigenvalues of the matrix T, again calculated with DSTEQR, are identical to the singular values of B: 1.4142 and 7.0710e-11.

$$B = \begin{bmatrix} 1.0 & 1.0 \\ 0 & 1.0e-10 \end{bmatrix}$$

(a) sample bidiagonal matrix B

$$B^TB = \begin{bmatrix} 1.0 & 1.0 \\ 1.0 & 1.0 \end{bmatrix}$$

(b) sample product B^TB

$$T = \begin{bmatrix} 0 & 1.0 & & & \\ 1.0 & 0 & 1.0 & & \\ & 1.0 & 0 & 1.0e-10 & \\ & & 1.0e-10 & 0 & \end{bmatrix}$$

(c) sample matrix T

Figure 7.3. Numerical example of a bidiagonal matrix and the associated matrices

It is possible to compute all the singular values of B with high relative accuracy if the eigenvalue algorithm used on T introduces small relative perturbations entrywise on T. The drawback of this second approach is that the size of the matrix doubles, and therefore the SVD is more expensive to calculate. It is faster to find the eigenvalues of B^TB because the matrix is smaller, but this method introduces errors in the smallest singular values.

Li's algorithm uses a novel approach that combines both methods. Let σ_i be an actual singular value, and let σ_i' be an approximation of σ_i. Let ϵ be the machine precision. Then, based on the assumption that the eigenvalue algorithm can achieve

a relative accuracy of $3n\varepsilon$ when finding the eigenvalues of T, Li *et al* [108] set the following goal of relative accuracy to find the singular values:

$$\frac{|\sigma_i' - \sigma_i|}{\sigma_i} \leq 3n\varepsilon.$$

It has been shown [108] that this degree of relative accuracy can be obtained by calculating a threshold. The threshold is calculated using the ∞ norm and the size of the matrix B^TB. The threshold is:

$$\frac{7\|B^TB\|_\infty}{12n}.$$

The largest singular values of B are found using the first approach, this is, calculating the eigenvalues λ_i of B^TB in the interval $[\frac{7\|B^TB\|_\infty}{12n}, \|B^TB\|_\infty)$ and then setting $\sigma_i = \sqrt{\lambda_i}$. The smallest singular values of B are computed by finding the eigenvalues of T in the interval $(0, \sqrt{\frac{7\|B^TB\|_\infty}{12n}})$. An outline of Li's algorithm is given in Figure 7.4.

The threshold described previously was calculated using the best relative accuracy attainable with the split-merge algorithm, discussed in the previous chapters. The split-merge algorithm described in the previous chapter can be used to calculate the eigenvalues of symmetric tridiagonal matrices. The algorithm needed some modifications, described later in section 7.3.

We wanted to compare the split-merge version of the singular value algorithm against another version that uses a different eigenvalue solver. As discussed previously, different versions of the bisection algorithm have been used in both sequential (DSTEBZ in LAPACK [55]) and parallel systems [94] in order to find the eigenvalues of symmetric tridiagonal matrices. The algorithm is based on the evaluation of Sturm's sequence [80] at a particular value. Sturm's sequence determines the number of eigenvalues that are smaller than the point where the sequence is evaluated. Given an initial interval that is known to contain the eigenvalue of interest, Sturm's

Algorithm 1: Singular Values
Input: Bidiagonal matrix B of order n
Output: The singular values σ_i, $i = 1, \ldots, n$, of B
Procedure:
begin
Calculate matrix B^TB
Calculate the ∞ norm of B^TB, $\|B^TB\|_\infty$
Calculate the threshold $\frac{7\|B^TB\|_\infty}{12n}$
Construct matrix T
Calculate ℓ, the number of eigenvalues of T in the interval $(0, \sqrt{\frac{7\|B^TB\|_\infty}{12n}})$.
Calculate the ℓ eigenvalues $\lambda_i, i = 1, \ldots, \ell$ of T in the interval $(0, \sqrt{\frac{7\|B^TB\|_\infty}{12n}})$
and set $\sigma_i = \lambda_i$
Calculate the $n - \ell$ eigenvalues $\lambda_i, i = \ell + 1, \ldots, n$ of B^TB
in the interval $[\frac{7\|B^TB\|_\infty}{12n}, \|B^TB\|_\infty)$ and set $\sigma_i = \sqrt{\lambda_i}, i = \ell + 1, \ldots, n$
end

Figure 7.4. Algorithm to calculate singular values

sequence is evaluated in the middle of the interval to determine which part of the original interval contains the desired eigenvalue. The process is performed repeatedly in the manner of a binary search until a small enough interval has been obtained.

A particular implementation of bisection starts by finding the interval containing all the eigenvalues using Gerschgorin Theorem [96], which states that every eigenvalue λ_i of a complex matrix A is inside a disk in the complex plane centered at $a_{i,i}$ and with radius $\sum_{j \neq i} \|a_{i,j}\|$ [109]. The theorem applies to real matrices as well, but in this case the eigenvalues are inside an interval in the real line. Then each eigenvalue is obtained, independently and in parallel, by bisecting that interval with the same number of steps. This version of bisection is very appropriate for parallelization: once the initial data has been broadcast to the participating nodes, each node can work on its part of the problem without any communication with other nodes, except

for reporting the final results to the initiating node. Demmel, Heath and van der Vorst [110] note that it is possible to use different number of iterations for different eigenvalues, since tightly clustered eigenvalues require more iterations. However, this approach in turn might require load balancing to keep the work evenly distributed among the processors and load balancing requires communications.

Our experiments in finding the singular values of matrices with very small singular values revealed that obtaining high relative accuracy indeed required the use of a variable number of iterations for every singular value. Therefore we use a load balancing approach, similar to the one used in the parallel split-merge algorithm, in the implementation of the bisection based singular value solver. One process servers as a coordinator, distributing small numbers of singular values to the rest of the processes to try to ensure that all processes finish at approximately the same time. Let a and b be the lower and upper limits of the interval containing the singular value being calculated. In order to be consistent with the relative accuracy goal employed in the split-merge version of the algorithm, the criteria for stopping used in the bisection implementation reported here was set as follows:

$$b - a \leq 3n\varepsilon b$$

where n is the size of the matrix, and ε is the precision of the machine.

7.2 MPC SVD Study

Two versions of the new singular value algorithm were implemented on the nCUBE-2. They differ in the method used to find the eigenvalues of the symmetric tridiagonal matrices. One uses the split-merge algorithm and the other one uses the bisection-based routine described above. Both algorithm possess the ability to find only the

eigenvalues that lie within a given interval, without requiring the calculation of all eigenvalues of the matrix.

Recall that in the new algorithm, only the eigenvalues of B^TB and T that lie in certain ranges, defined by the threshold, are of interest. It is not necessary to calculate all the eigenvalues of both matrices. The parallel implementation of split-merge that was described earlier and reported in [111] calculated all the eigenvalues in the input matrix. Additional parameters were added to that implementation in order to define the interval of interest. For every submatrix in the "task-tree" depicted in Figure 3.3, only the eigenvalues in the appropriate range are calculated. The range of eigenvalues of interest is calculated by evaluating Sturm's sequence in the edges of the interval. Recall that Sturm's sequence will return the number of eigenvalues that are smaller than a certain value. The rest of the eigenvalues are set to values slightly above or below the limits of the interval, depending on their position with respect to the interval.

The load balancing task is more complicated in this version of split-merge, since the coordinator does not dispense additional eigenvalues to the workers from a continuous sequence of eigenvalues but out of a set of discontinuous intervals. Consider Figure 7.5, where every solid line represents all the eigenvalues of a given matrix. In the original implementation, every eigenvalue in every submatrix is calculated. That is, the entire solid line of eigenvalues would be calculated. But in this version, we are interested only in those eigenvalues in a particular interval, and those eigenvalues are only part of the entire set of eigenvalues. In Figure 7.5, those eigenvalues are represented with the dotted lines. As before, the eigenvalues calculated in the smaller matrices become initial approximations to compute the eigenvalues of the larger matrices.

The task of finding which eigenvalues are of interest in every submatrix was assigned to the "sink" nodes at every stage. After a "sink" has merged the results of

two smaller submatrices, it finds the range of eigenvalues that needs to be calculated and it sends this information to the coordinator. In the nCUBE implementation, this required sending an additional message to the coordinator from the "sinks" in the gather operation. A problem arose with the fact that two messages were being sent from every "sink" to the coordinator: The communication subsystem of the nCUBE was not able to handle more than 3 "sinks" (the case when 8 nodes were being used). Hence, it was necessary to "sequentialize" the messages from the sinks to the coordinator. Every sink waits for a message from the coordinator before sending its submatrix and the range of eigenvalues to calculate.

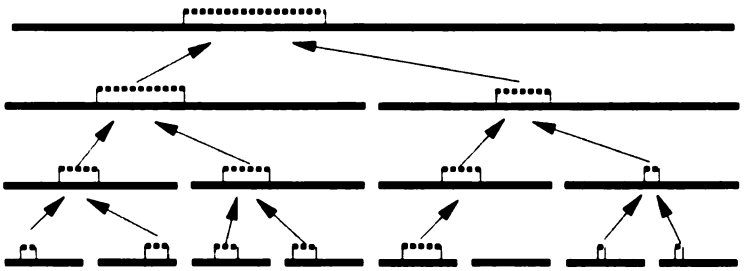


Figure 7.5. Representation of the intervals of eigenvalues of interest

Since this algorithm uses the split-merge algorithm as a routine, the size of the workloads dispensed by the coordinator has a significant impact on the performance of the program. Even more so in this SVD algorithm, as the version of split-merge works on finding only eigenvalues in a particular range.

For the purpose of comparison, the singular value algorithm was implemented using the bisection-based eigenvalue algorithm, which was described in Section 7.1. Both versions of the program were executed on input matrices of different sizes to examine the effects the size of the matrix. Figure 7.6 shows the execution times for the algorithms on random matrices of sizes 2048 and 4096. It can be observed that

the split-merge based version of the algorithm is faster than the bisection based one. The advantage holds even for a 64 processors, although the advantage is not so large with a large number of processors.

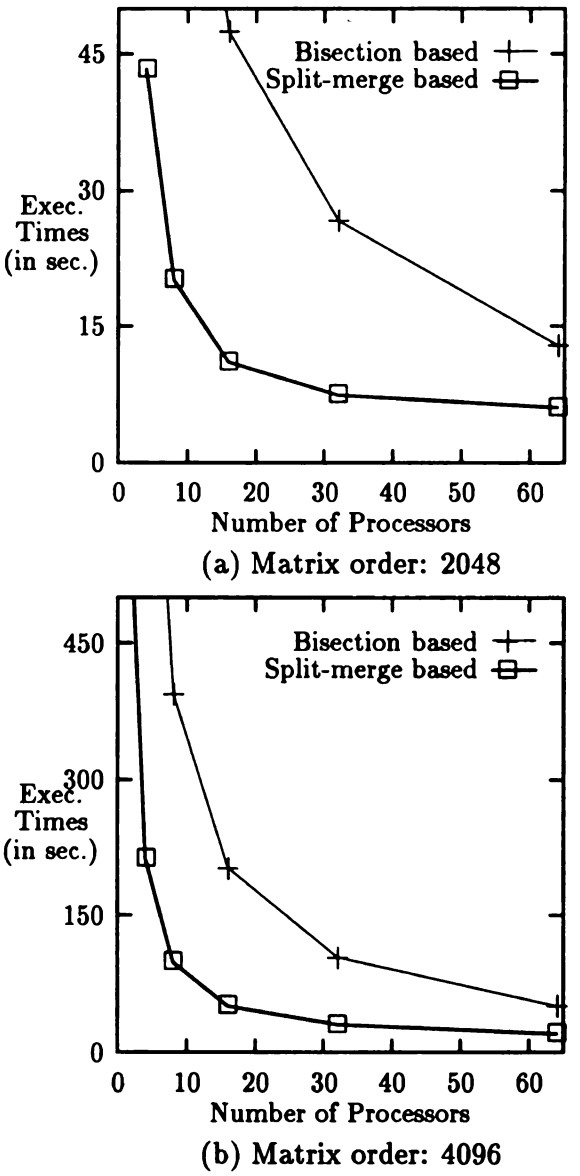


Figure 7.6. Comparison of the two versions of the SVD program on an nCUBE

7.3 Cluster implementation

The program was also implemented on the cluster of Sparc-10s described in the previous chapter. The implementation on the cluster was very sensitive to the size of the workload.

As discussed previously, if the workloads are too small, then the workers will have to request more work frequently, each time incurring the overhead of a waiting period. The risk exists again of using workloads that take very little processing time, thus increasing the response time from the coordinator. This compounds the problem of the frequent requests. Larger workloads imply less waiting, but they increase the risk of load imbalance. Specifically, a processor may receive a set of eigenvalues that requires a large number of iterations causing this processor to continue working after other processors become idle. This issue is illustrated in Figure 7.7, which shows the execution times of the singular value program on a certain random matrix with 2048 entries, as executed on a cluster with 8 processors. The x axis represents different sizes of workloads dispensed by the coordinator, while the y axis represents the execution times.

As described in the previous chapter, in order to better understand the effects of load imbalance, the executions of the program were instrumented so as to generate tracefiles compatible with the Paragraph visualization tool [81]. The traces in Figure 7.8 show the execution of the program with different sizes of workloads, on the same matrix used to produce Figure 7.7. All the traces use the same scale; every unit in the x axis represents 50 milliseconds and levels in the y axis represent the different processors. The shaded areas represent areas of processor activity, blank areas represent areas of inactivity. Different shades correspond to the stages of the algorithm. Recall that node 0 works as a coordinator in the last $d = \log_2(p)$ stages for every matrix, where p is the number of processes in the execution of the program.

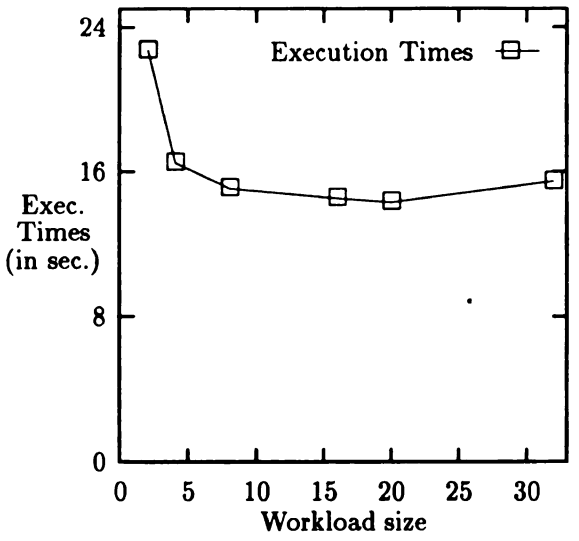


Figure 7.7. Execution times as a function of the workload size

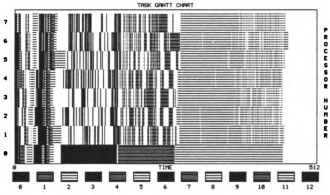
Notice that the algorithm was implemented with two consecutive calls to the split-merge eigenvalue solver. The first call finds some of the eigenvalues of the matrix T, which has twice the size of the original matrix. The second call calculates the rest of the singular values by finding another subset of eigenvalues in the matrix B^TB. Hence, in the traces, both calls are represented. The first twelve stages correspond to solving some of the eigenvalues of T and the next eleven stages to the calculations of the eigenvalues of B^TB. In this particular matrix, only 117 of the singular values needed to be calculated using matrix T and therefore the stages corresponding to B^TB take much longer. Figure 7.8(a) corresponds to a workload size of 2 eigenvalues. Numerous "gaps" of inactivity can be observed in the traces of the workers, although, the workers finish at approximately the same time in the distributed stages. A similar situation can be observed in Figure 7.8(b), which corresponds to a workload size of 4. The execution with a workload size of 20 is shown in Figure 7.8(c). The workers are busy most of the time, and the distributed stages finish at approximately the same

time. The gaps in the traces represent idle times, and as expected, the smaller the gaps, the better the execution time.

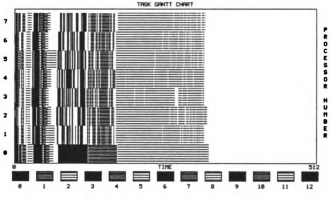
These traces also confirm the discussion about the response time of the server of chapter 6. A detailed observation of Figure 7.8(a) reveals, that the gaps of inactivity are more prominent in stage 9. The size of the submatrices being solved in stage 9 is 512. For a workload size of only 2, the average time to solve such a workload in matrices of this size is around 7000 microseconds, slightly above the 5000 microseconds of the service time. This is further compounded by having 7 workers. As a result, the response time from the server dominates and the workers are kept idle most of the time. In the next stage, stage 10, the average time for solving a chunk increases to around 15000 microseconds and as a result the response time diminishes. This results in a better utilization of the workers. It can be observed that stage 9 takes almost as long as stage 10, even though the amount of computational work of stage 10 is by far larger than that of stage 9. Finally, in stage 11, the last stage, the average time to solve a workload increases again to close to 30000 microseconds. The congestion phenomenon at the coordinator diminishes and the gaps of inactivity are smaller. Observe that when the size of the workload is 2, the time spent in stages 9 and 10 are significantly larger than with any other size. This can be attributed to the congestion in the server.

This phenomenon suggests using different workload sizes for the different matrices and maybe even at different stages. A small improvement was obtained by setting the workload size for the matrix T to 8 and the size for the matrix B^TB to 16. The execution time decreased to 13.7 seconds (from 14.3 for 20 and 20). It is possible to determine in advance how many singular values will be generated from the matrix T and how many will be generated from the matrix B^TB. In the current implementation the size of the workload is a user determined parameter, but it would be possible to

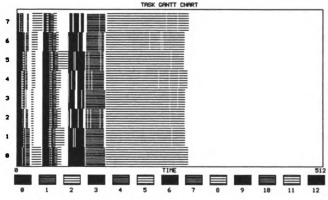
adjust the value at run time in order to divide the work more evenly among the processes.



(a) Workload size: 2



(b) Workload size: 4



(c) Workload size: 20

Figure 7.8. Traces of the program with different workload sizes

An additional factor has to be considered: The number of singular values that are calculated out of each of the matrices B^TB and T. The particular number depends on every matrix, it does not depend on the algorithm. If the number of singular values that are calculated out of any of the two matrices is very small, then it might be necessary to use smaller chunks to guarantee that all processors are active. For example, assume that the singular values of a matrix of size 2048 are being calculated on 8 processors. Furthermore, assume that 120 of the singular values are calculated by finding the eigenvalues of T and the remaining 1928 are found using B^TB. If one uses a workload size of 32, then in the distributed stages of calculating the singular values of T, only 4 of the 7 workers will be active, diminishing the efficiency of the algorithm.

Notice that the number of broadcast operations for a given matrix is double the one of the split-merge algorithm, precisely because the split-merge algorithm is called twice. This is, given a certain tridiagonal matrix of size n, if we used p processes to find its eigenvalues then $log_2(p)$ broadcasts will take place in the execution of the algorithm. Given a bidiagonal matrix of the same size and the same number of processes, then $2(log_2(p))$ broadcasts will take place. Thus, the performance of the parallel implementations of this algorithm are even more sensitive to the complexity of the broadcast operations than the split-merge implementations.

The experiments with the eigenvalue solver showed that using an ATM switch and using an improved version of the broadcast routine in PVM could improve the performance of the application. Given that this program uses twice as many broadcasts with approximately the same amount of computation, we tested the program on both environments to see the effects on this program.

Table 7.1 contains the execution times of the program on random input matrices of sizes 2048 and 4096. The results for both sets of tests were obtained on clusters of 7 workstations. The table shows that using an ATM network provides an advantage

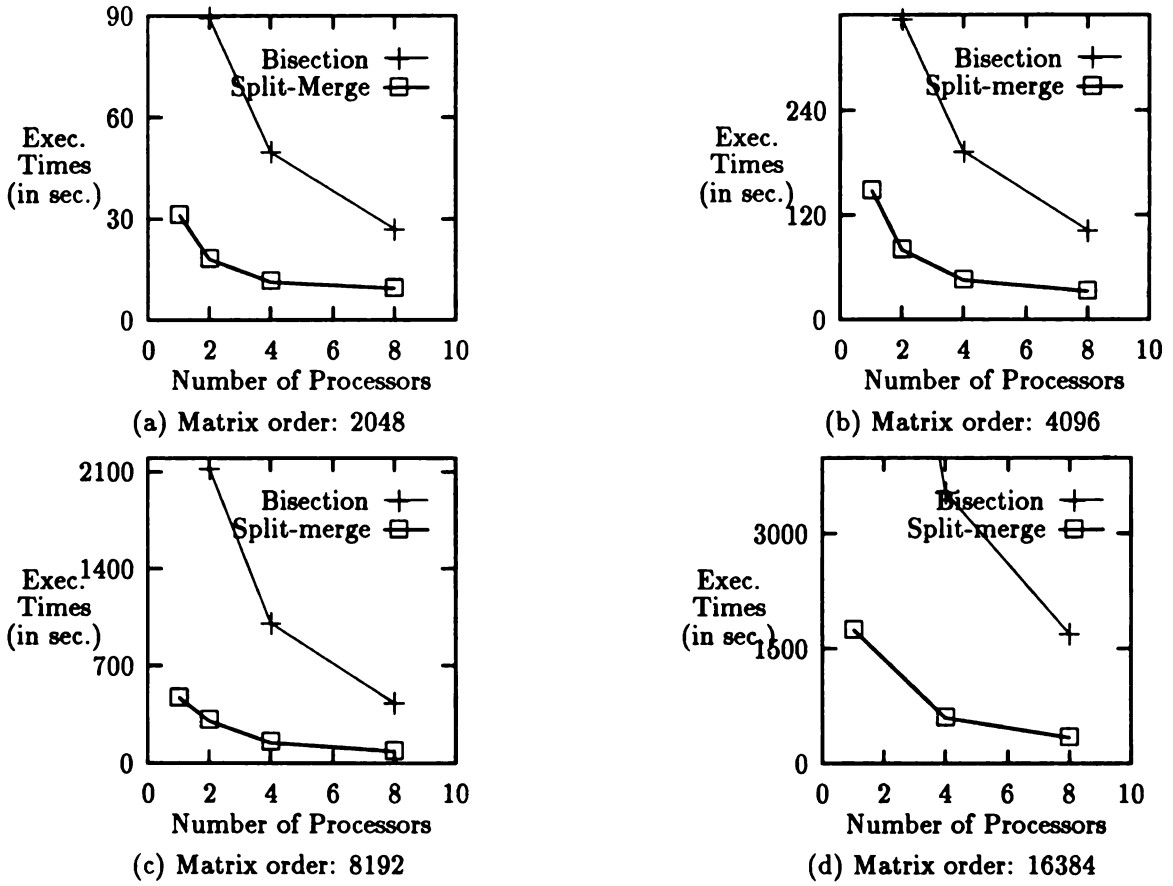


Figure 7.9. Execution times for random matrices of different sizes on a cluster of workstations

Matrix Size	*Ethernet*		*ATM*	
	Reg.Bcast	Improved Bcast	Reg.Bcast	Improved Bcast
2048	9.85	8.24	8.49	7.28
4096	33.60	31.30	32.97	30.79

Table 7.1. Execution times on a cluster with different broadcast algorithms and different interconnecting networks

over using regular Ethernet. Using the improved, tree based, version of the multicast operation [41] also improves the performance of the program.

7.4 A Model of the Performance of the SVD Algorithm

The split-merge based implementation of the SVD algorithm behaves very similarly to the split-merge algorithm reported in the previous chapter. The main difference is that in a given call to the split-merge procedure, only part of the eigenvalues are required. This in turn might lead to instances of the execution of the program where the workload size is such that some of the processes might be idle in the distributed stages. The size of the matrix whose eigenvalues are being found also varies as matrix T has size $2n$.

Let n_T be the number of singular values that are extracted from the matrix T and let n_{B^TB} be the number of singular values that are extracted from the matrix B^TB. Clearly, $n_T + n_{B^TB} = n$ and their values depend on the particular matrix.

Lets consider the call to split-merge to find the singular values using T. Because the size of T is $2n$, the total number of stages is incremented by one. The number of distributed stages remains constant (it depends on the number of processes), so there is one more local stage. The number of eigenvalues that are calculated in a given

submatrix along the process varies. For example, assume that matrix T is of size 128 and it contains 20 singular values. In the last stage of the split-merge call to find the eigenvalues of T we will be finding 20 singular values. In the previous stage, though, we will be finding eigenvalues of 2 submatrices of size 64 each. It is not possible to know in advance how many will be in each. 10 might come from one and 10 from the other one or 1 might come from one and 19 from the other one. Let $n_{p,i}$ denote the number of eigenvalues calculated by processor p at stage i, where $0 \leq n_{p,i} \leq n_T$. The expression for the local stages in the call for T is:

$$MAX_{over\ all\ p}\left(\frac{\sum_{i=2}^{k-d+1}\sum_{\ell=1}^{n_{p,i}} iter(\ell,i) * 2^i * 16}{sp}\right) \tag{7.2}$$

Previously we discussed the need to replace the value of the initial workload size c_i with a more appropriate value to avoid leaving some workers idle. If this situation arises, a reasonable option seems to replace the user supplied value of c_i with $\frac{n_T}{p-1}$. Let c_i' be the value that will be used during the execution of the program and which will assigned as: $c_i' = \begin{cases} c_i & \text{if } (p-1) * c_i \leq n_T \\ \frac{n_T}{p-1} & \text{otherwise} \end{cases}$

The expression for the distributed stages in the call for T is:

$\sum_{i=k-d+2}^{k+1} G() + MAX_{\text{all workers}}[B() + \frac{(\sum_{\ell=1}^{c_i'} iter(\ell,i)*2^i*16)}{sp} + lreq_j * W() * (\frac{\sum_{\ell=1}^{c_s} iter(\ell,i)*2^i*16}{sp})] + S()$.

For the call to split-merge to find the eigenvalues of B^TB, the expression for the local stages is:

$$MAX_{over\ all\ p}\left(\frac{\sum_{i=2}^{k-d}\sum_{\ell=1}^{n_{p,i}} iter(\ell,i) * 2^i * 16}{sp}\right) \tag{7.3}$$

and for the distributed stages

$\sum_{i=k-d+1}^{k} G() + MAX_{\text{all workers}}[B() + \frac{(\sum_{\ell=1}^{c_i'} iter(\ell,i)*2^i*16)}{sp} + lreq_j * W() * (\frac{\sum_{\ell=1}^{c_s} iter(\ell,i)*2^i*16}{sp})] + S()$.

7.5 Related work

In the realm of sequential SVD algorithms, the current method of choice is to reduce the original matrix to bidiagonal form and then find its singular values using an algorithm based on QR with zero shift [112]. The QR algorithm has been effectively parallelized when both eigenvalues and eigenvectors are needed [93]. But if only the eigenvalues are required, QR is difficult to parallelize efficiently, thus other algorithms have been proposed for finding singular values on parallel machines.

Jacobi methods operate directly on the original matrix, that is, there is no reduction of the original matrix to bidiagonal form. Bischof [113] presented an algorithm based on the two-sided Jacobi method. The matrix is partitioned in blocks among the processors. The process proceeds in sweeps. In every sweep, every processor works on a group of 4 blocks, say blocks (i,i),(i,j),(j,j), and (j,i). A transformation is applied to the entire group to make it block diagonal, this is, the norms of the blocks (i,j) and (j,i) will diminish. Once a group has been transformed, all processors exchange the updates done locally so that they can be performed globally. Eventually the matrix becomes block-diagonal. Once the matrix is block-diagonal, the SVD is computed by finding the SVD of each diagonal block. Bischof's implementation was done on a LCAP-1, an array of ten processors interconnected via global memory. Ewerbring and Luk [114] implemented Jacobi based singular value solvers on a CM-2, a SIMD machine. Zhou and Brent [115] are implementing a version of the Jacobi algorithm on a CM-5 that orders the sweeps taking advantage of the architecture of that machine. Lee *et al* [116] also report an implementation of Jacobi's method on a CM-5.

Cuppen's divide-and-conquer algorithm [98] to find the eigenvalues of symmetric tridiagonal matrices has been used to find SVDs as well. Arbenz [117] applied Cuppen's algorithm to the matrix T. The matrix T is not explicitly solved, though. Instead two matrices, B_1 and B_2, are formed (both smaller than B). The singular

values of each submatrix can be found in parallel and later recombined. This process is repeated recursively until the submatrices are small enough and it is more efficient to find their singular values with an EISPACK routine (svd). Arbenz reports an implementation of his algorithm on a sequential machine. Jessup and Sorensen [118] have also developed a parallel SVD algorithm based on divide-and-conquer. They apply the divide-and-conquer algorithm to BB^T and B^TB but they do not form these products explicitly to avoid numerical difficulties. Their program was implemented on a Sequent Symmetry and on an Alliant FX/8. Gu and Eisenstat [119] use an approach similar to the one of Jessup and Sorensen for computing the singular values, but they use a different approach for calculating the singular vectors. They do not report performance results from an implementation on a parallel machine. The algorithms of Arbenz, Jessup and Sorensen, and Gu and Eisenstat calculate both singular values and singular vectors and work on bidiagonal matrices.

7.6 Summary of Singular Value Study

The algorithm presented in this chapter provides the user with the speed advantages of finding the eigenvalues of B^TB and the accuracy of calculating the eigenvalues of T. The analysis performed by Li *et al* [108] to derive the expression for the threshold that dictates which singular values to find in each matrix could be adapted to the precision characteristics of other eigenvalue solvers for symmetric tridiagonal matrices. In the work reported in this chapter, two implementations were studied: one based on the split-merge algorithm and the other one uses a bisection based algorithm. Both implementations used the same load-balancing technique: a central coordinator which distributes work among the rest of the nodes.

The performance of the split-merge based implementation of the algorithm is sensitive to the size of the workload. The fact that the number of singular values

calculated out of one of the matrices can be small may require that the program overrides the workload size provided by the user to guarantee that all processors are active. The number of gathers, scatters and broadcasts operations is double that of the split-merge implementation and hence the performance of the program is even more sensitive to the implementation of the broadcast and gather primitives.

The implementation based on the split-merge algorithm achieved better performance than the one based on bisection, even though the communication and synchronization requirements of the split-merge version are considerably higher. This was true for both environments, the nCUBE-2 and the cluster of workstations.

CHAPTER 8

Conclusions and Future Work

Parallel computing is becoming more widely used as the software and hardware technologies involved are maturing, offering better performance and ease of use than in the past. Significant research is being conducted on how to solve efficiently a variety of problems in parallel, including numerical scientific applications. Clusters of workstations are becoming a cost/effective platform for parallel computing, allowing many more organizations to explore and take advantage of parallel computing. At the same time, more powerful, scalable distributed-memory parallel systems are appearing.

The research described in this dissertation has produced the following contributions: a study of the effects of collective communications, in particular, of different implementations of broadcast and multicast, with particular emphasis on their impact on scalability on both MPCs and clusters; a comparison of MPC and cluster environments, the effects of communication costs in each, and methods to accommodate them in algorithm design, such as load balancing and hiding communication costs; and case studies of the proposed techniques in new highly parallel numerical algorithms.

On both environments, clusters and MPCs, efficient implementation of collective communication operations can be used to improve performance, especially when large number of processors are being used. Hence, efficient collective communication is

required for good scalability, else the increased cost in communications eventually defeats the gains made by adding processing elements.

The research described in this dissertation has highlighted the effects of interprocess communication costs on the design and implementation of parallel numerical algorithms. A cluster of workstations has proven to be an effective alternative to a particular MPC, in spite of the much higher communication costs in the cluster. The characteristics of the cluster led to new approaches to application structure, for instance, doubling the number of processes in each node to mask the effect of communications.

In the particular case of Ethernet-based clusters, it is necessary to improve the implementation of the broadcast primitive in order to take advantage of the medium. For this particular algorithms, the gather, scatter, and load balancing operations pose an obstacle to the scalability of the algorithms. As manufacturers offer new alternatives for interconnecting clusters, such as switch-based LANs, there will be opportunities to implement collective communication more efficiently than on a shared medium. Gather, scatter and load balancing can be implemented more efficiently on a switch-based cluster, and hence the scalability of the algorithms, according to our model, should be significantly better. This research has concentrated on broadcasting and multicasting, as well as load balancing, and partially on gather and scatter, but a number of other primitives are equally important to other algorithms and their efficient implementation in clusters needs to be studied.

The alternative broadcast primitives that were explored in conjunction with PVM provided an initial approach to a more efficient implementation of broadcast in clusters. The implementation and study of the SVD algorithm provided better understanding of communication and dynamic load balancing when the amount of work varied in different stages of the algorithm. The experiments on the ATM and DEC

clusters contrasted the performance of new generation clusters with different characteristics. The model indicates that switch-based clusters should be scalable, thanks to the possibility of simultaneous communications among different pairs of nodes and, hence, better implementations of collective operations.

Load balancing proved to be very effective in improving the performance of the parallel versions of the algorithms. In these algorithms, where the memory requirements are modest, replicating certain data across all nodes involved to allow load balancing can be cost-effective. The response time of the server depends on the ratio between the expected service time and expected time between requests. The performance also depends on the size of the workloads. Very small workloads can cause congestion at the server while large workloads may cause load imbalance.

Further research can be done in several areas of the parallel implementation: exploring heuristics to assign automatically the sizes of the workloads, replacing the distributed sorting with performing all sorting at the coordinator to avoid all gather and scatter operations, except one gather, and using a hierarchy of coordinators, which might be necessary if larger numbers of nodes are to be used.

Both applications, the eigenvalue solver and the singular value solver, will benefit from research in better communication protocols for clusters of workstations. In particular, in the Ethernet environment, a reliable multicast protocol that allowed large messages would be very useful. On ATM-based clusters, the availability of efficient collective communication primitives at the user level will simplify the programmer's task and it will improve the performance of the program.

The insight gained in the parallelization in a distributed memory environment should be useful in parallelizing the programs in a shared memory multiprocessor or in a distributed memory machine that offers virtual shared memory. The fact that the number of iterations for Laguerre iterations is variable will help a programmer in that environment to use the appropriate directives for the parallel compiler. The

results obtained can also be used in the efficient implementations of other parallel algorithms.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] G. Bell, "Ultracomputers - a teraflop before its time," *Communications of the ACM*, vol. 35, pp. 27–47, August 1992.

[2] L. M. Ni and P. K. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer*, pp. 62–76, February 1993.

[3] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, "Graphical development tools for network-based concurrent supercomputing," in *Proceedings of Supercomputing 91*, IEEE CS Press, 1991.

[4] A. Agarwal, "Limits on interconnection network performance," *IEEE Transactions on Parallel and Distributed Processing*, vol. 2, pp. 398–412, October 1991.

[5] W. Dally, "Performance analysis of k-ary n-cube interconnection networks," *IEEE Transactions on Computers*, vol. 39, pp. 775–785, June 1990.

[6] C. E. Leiserson and et. al., "The network architecture of the Connection Machine CM-5," in *Proceedings of SPAA '92*, 1992.

[7] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Proceedings of the 19th. International Symposium on Computer Architecture*, pp. 278–287, 1992.

[8] D. H. Linder and J. C. Harden, "An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes," *IEEE Transactions on Computers*, vol. 40, pp. 2–12, January 1991.

[9] D. C. Grunwald and D. A. Reed, "Networks for parallel processors: Measurements and prognostications," in *Proceedings of the Third Conference on Hypercube Conncurrent Computers and Applications*, vol. 1, pp. 610–619, ACM, January 1988.

[10] W. J. Dally and C. L. Seitz, "The torus routing chip," *Journal of Distributed Computing*, vol. 1, no. 3, pp. 187–196, 1986.

[11] W. Dally, "Virtual-channel flow control," *IEEE Transactions on Parallel and Distributed Processing*, vol. 3, pp. 194–205, March 1992.

[12] R. Duzett and R. Buck, "An overview of the nCUBE-3 supercomputer," in *Proc. Frontiers'92: The 5th Symposium on the Frontiers of Massively Parallel Computation*, October 1992.

[13] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Flyer, "The message-driven processor: A multicomputer processing node with efficient mechanisms," *IEEE Micro*, pp. 23–39, April 1992.

[14] NCUBE Company, *NCUBE 6400 Processor Manual*, 1990.

[15] Intel, Supercomputing, Systems, and Division, *Intel Touchstone Delta Systems Description.* Advanced Information, Intel Corporation, 1991.

[16] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Sizovic, C. S. Steele, and W. K. Su, "The architecture and programming of the Ametek Series 2010 multicomputer," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, (Padadena, CA), pp. 33–36, ACM, January 1988.

[17] W. Myers, "Supercomputing 92 reaches down to the workstation," *IEEE Computer*, vol. 26, January 1993.

[18] H. T. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F. Bitz, F. Christianson, E. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill, "Network-based multicomputers: An emerging parallel architecture," in *Proceedings of Supercomputing '91*, pp. 664–673, 1991.

[19] W. Gropp and E. Lusk, "Users guide for the ANL IBM SP1 (draft)," Tech. Rep. ANL/MCS-Tm-00, Argonne National Laboratory, March 1994.

[20] A. Hac and H. B. Mutlu, "Synchronous optical network and broadband ISDN protocols," *IEEE Computer*, vol. 22, November 1989.

[21] E. Biagioni, E. Cooper, and R. Sansom, "Designing a practical ATM LAN," *IEEE Network*, pp. 32–39, March 1993.

[22] D. Banks and M. Prudence, "A high-performance network architecture for a PA-RISC workstation," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 191–202, February 1993.

[23] C. Chang, R. Flower, J. Forecast, H. Gray, W. R. Hawe, K. K. Ramarishnan, A. P. Nadkarni, U. N. Shikarpur, and K. M. Wilde, "High-performance TCP/IP and UDP/IP networking in DEC OSF/1 for Alpha AXP," *Digital Technical Journal*, vol. 5, no. 1, 1993.

[24] C. C. Huang and P. K. McKinley, "Efficient collective communication on ATM networks," *In preparation*, 1993.

[25] H. T. Kung, P. Steenkiste, M. Gubitoso, and M. Khaira, "Parallelizing a new class of large applications over high-speed networks," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallle Programming (PPoPP)*, 1991.

[26] S. E. Deering and D. R. Cheriton, "Multicast routing in datagram internetwork and extended LANs," *ACM Transactions on Computer Systems*, vol. 8, pp. 85–110, May 1990.

[27] C.-T. Ho and M. T. Raghunath, "Efficient communication primitives on hypercubes," *Journal of concurrency: Practice and Experience*, vol. 4, pp. 427–457, September 1992.

[28] H. Xu, P. K. McKinley, and L. M. Ni, "Efficient implementation of barrier synchronization in wormhole-routed hypercube multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 172–184, 1992.

[29] Y. Saad and M. H. Schultz, "Data communication in hypercubes," *Journal of Parallel and Distributed Computing*, vol. 6, pp. 115–135, 1989.

[30] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, December 1986.

[31] S. L. Johnsson and C. T. Ho, "Optimum broadcasting and personalized communication in hypercubes," *IEEE Transactions on Computers*, vol. 38, pp. 1249–1268, September 1989.

[32] C.-T. Ho, "Optimal broadcasting on SIMD hypercubes without indirect addressing capability," *Journal of Parallel and Ditributed Computing*, vol. 13, no. 2, pp. 246–255, 1991.

[33] M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn, "Efficient communication primitives on mesh architectures with hardware routing," in *Proceedings of the sixth SIAM conference on Parallel Processing*, pp. 943–948, 1993.

[34] M. Barnett, R. Littlefield, D. G. Payne, and R. van de Geijn, "Global combine on mesh architectures with wormhole routing," in *Proceedings of the 7th International Parallel Processing Symposium*, pp. 156–162, 1993.

[35] Y. Lan, A. H. Esfahanian, and L. M. Ni, "Multicast in hypercube multiprocessors," *Journal of Parallel and Distributed Computing*, pp. 30–41, January 1990.

[36] P. K. McKinley, H. Xu, A. Esfahanian, and L. M. Ni, "Unicast-based multicast communications in wormhole-routed networks." Accepted to appear in IEEE Transactions on Parallel and Distibuted Systems.

[37] P. K. McKinley and C. Trefftz, "Efficient broadcast in all-port wormhole-routed hypercubes," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. II, pp. 288–291, 1993.

[38] D. F. Robinson, D. L. Judd, P. K. McKinley, and B. H. C. Cheng, "Efficient collective data distribution in all-port wormhole-routed hypercubes," in *Proceedings of Supercomputing'93*, pp. 792–801, Nov. 1993.

[39] X. Lin, P. K. McKinley, and L. M. Ni, "Performance evaluation of multicast wormhole routing in 2D-mesh multicomputers." Accepted to appear in IEEE Transactions on Parallel and Distibuted Systems.

[40] A. S. Tanenbaum, F. Kaashoek, and H. Bal, "Parallel programming using shared objects and broadcasting," *IEEE Computer*, vol. 25, August 1992.

[41] C. Huang, E. P. Kasten, and P. K. McKinley, "Design and implementation of multicast operations for ATM-Based high performance computing," Tech. Rep. MSU-CPS-94-22, Michigan State University, March 1994.

[42] V. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, December 1990.

[43] J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, and R. Overbeek, *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, 1987.

[44] P. K. McKinley, H. Xu, E. Kalns, and L. M. Ni, "Efficient communication services for scalable architectures," in *Proceedings of Supercomputing '92*, pp. 478–487, 1992.

[45] J. G. Mills, L. J. Clarke, and A. S. Trew, "CHIMP concepts," Tech. Rep. EPCC-KTP-CHIMP-CON Version 1.2, Edinburgh Parallel Computing Centre,The University of Edinburgh, April 1991.

[46] A. Skjellum and A. P. Leung, "Zipcode, a portable multicomputer communication library atop the reactive kernel," in *Proceedings of the Fifth Distibuted Memory Computing Conference*, pp. 767–776, IEEE Press, April 1990.

[47] J. Flower, A. Kolawa, and S. Bharadwaj, "The Express way to distributed processing," *Supercomputing Review*, pp. 54–55, May 1991.

[48] A. Kolawa and J. Flower, *Software Engineering Productivity*, ch. The Impact of Parallel Processing. Elsevier Science Publishers B.V.(North Holland), 1992.

[49] G. Geist, M. Heath, B. Peyton, and P. Worley, "A machine-independent communication library," in *Proceedings of the fourth Conference on Hypercubes, Concurrent computers, and Applications*, 1990.

[50] N. Carriero and D. Galernter, "Linda in context," *Communications of the ACM*, vol. 32, April 1989.

[51] Message Passing Interface Forum, "The MPI draft, message-passing standard," *in preparation*, 1993.

[52] 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.* Lecture Notes in Computer Science, Berlin: Springer Verlag, 2nd ed., 1976.

[53] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK User's Guide.* Philadelphia, PA: SIAM, 1979.

[54] J. Dongarra, "Performance of various computers using standard linear software," tech. rep., Mathematical Sciences Section, Oak Ridge National Laboratory, 1993.

[55] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide.* Philadelphia PA: SIAM, 1992.

[56] J. J. Moré, D. C. Sorensen, B. S. Garvow, and K. E. Hillstrom, *Sources and Development of Mathematical software*, ch. The MINPACK project, pp. 88–111. Englewood Cliffs NJ: Prentice Hall, 1984.

[57] E. Piessens, E. deDoncker Kapenga, C. W. Überhuber, and D. K. Kahaner, *QUADPACK A Subroutine Package for Automatic Integration.* New York: Springer Verlag, 1983.

[58] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic linear algebra subprograms for FORTRAN usage," *ACM Trans. Math. Software*, vol. 5, pp. 308–323, 1979.

[59] J. Dongarra, J. Du-Croz, S. Hammarling, and R. Hanson, "An extended set of fortran basic linear algebra subprograms," *ACM Trans. Math. Software*, vol. 14, pp. 1–17, 1988.

[60] J. Dongarra, J. Du-Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Software*, vol. 16, pp. 1–17, 1990.

[61] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *Proceedings, Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 120–127, IEEE Computer Society Press, 1992.

[62] E. Anderson, A. Bnzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn, "LAPACK for distributed memory achitectures: progress report," in *Proceedings of the Fifth SIAM Conference on Parallel Processing*, 1991.

[63] J. Dongarra, R. van de Geijn, and R. Whaley, "Two dimensional basic linear algebra communication subprograms," in *Proceedings of the sixth SIAM conference on Parallel Processing*, pp. 347–352, 1993.

[64] R. Arlauskas, "iPSC/2 system: A second generation hypercube," in *Proceedings of the Third Conference on Hypercube Computers and Concurrent Applications*, pp. 38–42, ACM, January 1988.

[65] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, "A users' guide to PVM parallel virtual machine," Tech. Rep. TM-11826, Oak Ridge National Laboratory, July 1991.

[66] J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK++: An object oriented linear algebra library for scalable systems," in *Proceedings of the 1993 Scalable Parallel Libraries Confer ence*, pp. 216–223, Oct. 1993.

[67] I. D. Scherson and P. F. Corbett, "Communications overhead and the expected speedup of multidimensional mesh-connected parallel processors," *Journal of Parallel and Ditributed Computing*, vol. 11, pp. 86–96, 1991.

[68] D. Gannon and J. V. Rosendale, "On the impact of communication complexity in the design of parallel numerical algorithms," *IEEE Transactions on Computers*, vol. 33, pp. 1180–1194, December 1984.

[69] L. Brochard, "Efficiency of some parallel numerical algorithm on distributed systems," *Parallel Computing*, vol. 12, pp. 21–44, 1989.

[70] S. L. Johnsson, "Language and compiler issues in scalable high performance," Tech. Rep. 244, Thinking Machines, 1992.

[71] J. Dongarra and R. A. van de Geijn, "Lapack working note 30 - reduction to condensed form for the eigenvalue problem on distributed memory architectures," Tech. Rep. CS-91-130, University of Tennessee, 1991. To appear in Parallel Computing.

[72] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, pp. 483–485, Thompson Books, 1967.

[73] J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM Journal of Scientific and Statistical Computing*, vol. 9, pp. 609–638, July 1988.

[74] X. H. Sun and L. M. Ni, "Another view on parallel speedup," in *Proceedings of Supercomputing '90*, pp. 324–333, IEEE CS Press, 1990.

[75] P. H. Worley, "The effect of time constraints on scaled speedup," *SIAM Journal of Scientific and Statistical Computing*, vol. 11, pp. 838–858, September 1990.

[76] V. Kumar and V. N. Rao, "Parallel depth-first search, part ii: Analysis," *International Journal of Parallel Programming*, vol. 16, no. 6, pp. 501–519, 1987.

[77] T. Y. Li and Z. Zeng, "Laguerre's iteration in solving the symmetric tridiagonal eigenproblem - revisited," *SIAM Journal of Scientific Computing*, 1993. accepted to appear.

[78] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes, The Art of Scientific Computing.* Cambridge, UK: Cambridge University Press, 1986.

[79] G. H. Golub and C. F. Van Loan, *Matrix Computations.* Johns Hopkins University Press, 2nd. edition ed., 1990.

[80] P. G. Ciarlet, *Introduction to numerical linear algebra and optimisation*, pp. 200–203. Cambridge University Press, 1989.

[81] M. Heath and J. Etheridge, "Visualizing performance of parallel programs," *IEEE Software*, vol. 8, September 1991.

[82] N. G. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *IEEE Computer*, vol. 25, pp. 33–44, December 1992.

[83] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on Software Engineering*, vol. 12, pp. 662–675, May 1986.

[84] D. J. Lilja, "Expliting the parallelism available in loops," *IEEE Computer*, pp. 13–26, February 1994.

[85] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," in *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 236–240, 1984.

[86] C. Polychronopoulos and D. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, vol. 36, pp. 1425–1439, December 1987.

[87] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring, a practical and robust method for scheduling parallel loops," in *Proceedings of Supercomputing 91*, IEEE CS Press, 1991.

[88] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Processing*, vol. 4, pp. 87–98, January 1993.

[89] V. Kumar and G. Y. Ananth, "Scalable load balancing techniques for parallel computers," Tech. Rep. TR 91-55, University of Minnesota, November 1991.

[90] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE Transactions on Computers*, vol. 36, pp. 388–395, April 1987.

[91] H. Sullivan and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine," in *Proceedings of the 4th Annual Symp. Comput. Architecture*, pp. 105–124, March 1977.

[92] J. J. Dongarra and D. C. Sorensen, "A fully parallel algorithm for the symmetric eigenvalue problem," *SIAM Journal of Scientific and Statistical Computing*, vol. 2, pp. 139–154, March 1987.

[93] P. Arbenz, K. Gates, and C. Sprenger, "A parallel implementation of the symmetric tridiagonal QR algorithm," in *Proceedings of the Fourth Symposium on the Frontiers of Massively Paralllel Computation*, IEEE CS Press, 1992.

[94] I. C. F. Ipsen and E. R. Jessup, "Solving the symmetric tridiagonal eigenvalue problem on the hypercube," *SIAM Journal of Scientific and Statistical Computing*, vol. 11, pp. 203–229, March 1990.

[95] T. Gregory and D. L. Karney, *A Collection of Matrices for Testing Computational Algorithms.* Huntington, New York: Robert E. Krieger Publishing Company, 1978.

[96] J. H. Wilkinson, *The Algebraic Eigenvalue Problem.* Oxford: Clarendon Press, 1965.

[97] A. Sameh and D. Kuck, "A parallel QR algorithm for symmetric tridiagonal matrices," *IEEE Transactions on Computers*, no. C-26, pp. 81–91, 1977.

[98] J. J. M. Cuppen, "A divide and conquer method for the symmetric tridiagonal eigenproblem," *Numerische Mathematik*, vol. 2, no. 36, pp. 177–195, 1981.

[99] K. Gates, "A rank-two divide and conquer method for the symmetric tridiagonal eigenproblem," in *Proceedings of the Fourth Symposium on the Frontiers of Massively Paralllel Computation*, IEEE CS Press, 1992.

[100] S. S. Lo, B. Philippe, and A. Sameh, "A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem," *SIAM Journal of Scientific and Statistical Computing*, vol. 2, pp. 155–165, March 1987.

[101] M. Lu and X. Qiao, "Applying parallel computer systems to solve symmetric tridiagonal eigenvalue problems," *Parallel Computing*, vol. 18, pp. 1301–1315, 1992.

[102] T.-Y. Li, H. Zhang, and X.-H. Sun, "Parallel homotopy algorithm for the symmetric tridiagonal eigenvalue problem," *SIAM Journal of Scientific and Statistical Computing*, vol. 12, pp. 469–487, May 1991.

[103] T. Y. Li and N. H. Rhee, "Homotopy algorithm for symmetric eigenvalue problems," *Numerische Mathematik*, vol. 5, pp. 265–280, 1989.

[104] E. P. Kasten and P. K. McKinley, "An efficient multicast extension to pvm," Tech. Rep. MSU-CPS-94-21, Michigan State University, April 1994.

[105] CCITT, "Recommendation i.363," 1993.

[106] A. O. Allen, *Probability, Statisitcs and Queueing Theory with Computer Science Applications.* New York, NY: Academic Press, 1978.

[107] G. H. Golub and W. Kahan, "Calculating the singular values and pesudo-inverse of a matrix," *SIAM Journal On Numerical Analysis Ser. B*, vol. 2, no. 2, pp. 205–224, 1965.

[108] T. Y. Li, N. H. Rhee, and Z. Zeng, "An efficient and accurate parallel algorithm for the singular value problem of bidiagonal matrices," *Submitted for publication*, 1993.

[109] G. W. Stewart, *Introduction to Matrix Computations.* Academic Press, 1973.

[110] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, "Parallel numerical linear algebra," *Acta Numerica*, pp. 111–197, 1993.

[111] C. Trefftz, P. K. McKinley, T. Y. Li, and Z. Zeng, "A scalable eigenvalue solver for symmetric tridiagonal matrices," in *Proceedings of the sixth SIAM conference on Parallel Processing*, pp. 602–609, 1993.

[112] J. Demmel and W. Kahan, "Accurate singular values of bidiagonal matrices," *SIAM Journal of Scientific and Statistical Computing*, vol. 11, pp. 873–912, September 1990.

[113] C. H. Bischof, "Computing the singular value decomposition on a distributed system of vector processors," *Parallel Computing*, vol. 11, pp. 171–186, 1989.

[114] L. M. Ewerbring and F. T. Luk, "Computing the singular value decomposition on the Connection Machine," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 152–155, 1990.

[115] B. B. Zhou and R. P. Brent, "Parallel computation of the singular value decomposition on tree architectures," in *Proceedings of the 1993 International Conference on Parallel Processing*, vol. III, pp. 128–131, 1993.

[116] T. J. Lee, F. T. Luk, and D. L. Boley, "Singular value decomposition on a 2-d fat-tree architecture," in *Proceedings of the sixth SIAM conference on Parallel Processing*, pp. 358–362, 1993.

[117] P. Arbenz, "Divide-and-conquer algorithms for the computation of the SVD of bidiagonal matrices," Tech. Rep. 170, ETH Zürich, Institut für Wissenschaftliches Rechnen, Switzerland, 1991.

[118] E. R. Jessup and D. C. Sorensen, "A parallel algorithm for computing the singular value decomposition of a matrix," Tech. Rep. ANL-MCS-TM-102, Argonne National Laboratory, July 1991.

[119] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the bidiagonal SVD," Tech. Rep. YALEU/DCS/RR-933, Yale University, December 1992.