1995

Runtime Support for In-Core and Out-of-Core Data-Parallel Programs

Rajeev Thakur
Graduate School of Syracuse University, Computer Engineering

RUNTIME SUPPORT FOR IN-CORE AND OUT-OF-CORE DATA-PARALLEL PROGRAMS

by

RAJEEV THAKUR

B.E., University of Bombay, India, 1990
M.S., Syracuse University, 1992

DISSERTATION

Submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Computer Engineering
in the Graduate School of Syracuse University

May 1995

Approved _____
Professor Alok Choudhary

Date _____

Abstract

Distributed memory parallel computers or distributed computer systems are widely recognized as the only cost-effective means of achieving teraflops performance in the near future. However, the fact remains that they are difficult to program and advances in software for these machines have not kept pace with advances in hardware. This thesis addresses several issues in providing runtime support for in-core as well as out-of-core programs on distributed memory parallel computers. This runtime support can be directly used in application programs for greater efficiency, portability and ease of programming. It can also be used together with a compiler to translate programs written in a high-level data-parallel language like High Performance Fortran (HPF) to node programs for distributed memory machines.

In distributed memory programs, it is often necessary to change the distribution of arrays during program execution. This thesis presents efficient and portable algorithms for runtime array redistribution. The algorithms have been implemented on the Intel Touchstone Delta and are found to scale well with the number of processors and array size. This thesis also presents algorithms for all-to-all collective communication on fat-tree and two-dimensional mesh interconnection topologies. The performance of these algorithms on the CM-5 and Touchstone Delta is studied extensively. A model for estimating the time taken by these algorithms on the basis of system parameters is developed and validated by comparing with experimental results.

A number of applications deal with very large data sets which cannot fit in main memory, and hence have to be stored in files on disks, resulting in out-of-core programs. This thesis also describes the design and implementation of efficient runtime support for out-of-core computations. Several optimizations for accessing out-of-core data are presented. An Extended Two-Phase Method is proposed for accessing sections of out-of-core arrays efficiently. This method uses collective I/O and the I/O workload is divided among processors dynamically, depending on the access requests. Performance results obtained using this runtime support for out-of-core programs on the Touchstone Delta are presented.

Contents

List of Tables

List of Figures

Acknowledgments

I have been fortunate to have Alok Choudhary as my advisor throughout the course of my graduate studies. He has helped me in too many ways to mention. I greatly appreciate his guidance, support, encouragement and enthusiasm. Working with him has been a pleasure and a great learning experience.

I wish to thank Thong Dang, Geoffrey Fox, Salim Hariri, David Kotz and Sanjay Ranka for serving on my thesis committee and for their feedback on drafts of this thesis. I thank Geoffrey Fox and everyone else at NPAC for providing an excellent environment for my work. I thank Thong Dang for getting me interested in Computational Fluid Dynamics and inspiring me in many other ways. I am grateful to all my friends particularly Rajesh Bordawekar, Sachin Damle, Kanad Chakraborty, Ravi Ponnusamy, Sachin More, Kevin Roe and Mario Campuzano for their help at all times and making my stay at Syracuse pleasant and enjoyable.

I cannot express in words my gratitude towards my parents, grandparents and sister. All my achievements have been possible only because of their constant encouragement, love and support. I am fortunate to have such a wonderful family. I dedicate this thesis to them.

Chapter 1

Introduction

Massively Parallel Processors (MPPs) with peak performance greater than 100 Gflops have already made their advent into the supercomputing arena. As a result, parallel computers are increasingly being used in many applications which require a great deal of computational power. Examples of such applications include many large scale computations in physics, chemistry, biology, engineering, medicine and other sciences, which have been identified as Grand Challenge Applications [56, 131]. Many applications dealing with information technology, such as multimedia systems, video on demand, video compression and decompression, also require a large amount of compute power. It is estimated that a computer capable of 1 Tflops (10^{12} flops) or more would be required to solve these applications in a reasonable amount of time. It is widely recognized that rather than conventional vector supercomputers, parallel computers or distributed systems provide the only cost-effective means of achieving teraflop performance.

However, software support for parallel computers has lagged far behind advances in hardware. Programming a parallel machine can prove to be quite tedious. In order to get the best performance from a parallel computer, the programmer has to pay attention to many low-level implementation details. Also, the programs are very often not portable. One way to tackle this problem is by using advanced compilers and runtime support systems. The research presented in this thesis addresses several issues in providing runtime support for in-core as well as out-of-core programs on

distributed memory parallel computers. This runtime support can be directly used in application programs for greater efficiency, portability and ease of programming. It can also be used together with a compiler to translate programs written in a high-level data-parallel language like High Performance Fortran (HPF) to node programs for distributed memory parallel computers.

1.1 Distributed Memory Parallel Computers

In a distributed memory computer, the memory is physically distributed among processors. Each processor has its own local memory and all processors are connected together by an interconnection network such as a hypercube, mesh, torus, fat tree or some other topology. A processor can directly access only its own local memory. If data from some other processor is needed, it can be obtained by explicit message passing. Examples of such systems are the Intel iPSC/860, Touchstone Delta and Paragon; Thinking Machines CM-5; IBM SP-1, SP-2; Meiko CS-2; Cray T3D; nCUBE-2 etc.

The main advantage of distributed memory computers is that they are scalable. Advances in interconnection network technology have made it possible to connect hundreds or thousands of processors into one large high-performance parallel computer. Hence, such machines are also called Massively Parallel Processors (MPPs). Each processor of an MPP is typically a powerful microprocessor like a DEC Alpha, IBM PowerPC, Sun Sparc or Intel i860. Since these microprocessors are also used in workstations or personal computers, they are available at competitive prices. Hence, MPPs also prove to be very cost-effective.

1.2 Software for Distributed Memory Computers

Distributed memory machines are relatively hard to program. The most common programming model for distributed memory computers is the Single Program Multiple Data (SPMD) model in which each processor runs the same program, but on different data sets. The program running on each node is essentially a sequential program

(usually in C or Fortran) interspersed with calls to message passing routines. The node program is written in the local address space and the absence of a single global address space is what makes message passing programming difficult. The programmer has to decide how data should be partitioned among the processors and has to manage communication between processors explicitly. Interprocessor communication is at least an order of magnitude more expensive than accessing data in local memory. This forces the programmer to pay considerable attention to saving communication costs. Since each machine has its own architectural features and idiosyncrasies, the programmer tries to hardwire the program to exploit these peculiarities and get the best performance. Hence a great deal of effort is required to port such programs to other machines. These details also make it difficult to convert existing sequential programs to parallel programs. Another problem is that each parallel machine has its own message passing library which is quite different from that of other machines. Some of the common communication libraries are NX for Intel machines, CMMD for the CM-5, MPL for the IBM SP-1 and SP-2 etc. A program written using the NX library for the Intel Paragon cannot be directly run on the CM-5 and vice-versa because the message passing routines are totally incompatible.

There have been several attempts to provide some measure of portability in parallel programs. There are number of portable communication libraries like EXPRESS [90], PVM [114], PICL [47], P4 [19] etc. which provide a communication layer above the native message passing library of the system. These libraries provide a well-defined set of communication routines which remain the same for any system. For example, a program written using EXPRESS or PVM can be run on almost any parallel computer and even on a network of workstations. The EXPRESS or PVM routines make calls to the message passing library provided on the system. However, this portability is at the cost of slightly lower performance because of this additional layer of communication software.

There has also been an effort to develop a standard Message Passing Interface (MPI) [83]. The Message Passing Interface Forum, a group of researchers from industry, academia and research laboratories, has defined a set of library interface standards

for message passing called MPI. MPI has tried to make use of the most attractive features of a number of existing message passing systems. The goal is to establish a practical, portable, efficient and flexible standard for message passing. The definition of a message passing standard such as MPI provides vendors with a clearly defined set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby providing better performance. But the difficulty of explicitly doing message passing still remains.

1.3 Data-Parallel Languages

Since message passing programming is difficult, the user would prefer to write a program in global name space without any message passing, and have a compiler translate it into a message passing program. This requires advanced compiler technology, but it would result in parallel programming being easy and portable. Researchers have found it very hard to build compilers which can parallelize sequential programs written in standard C or Fortran 77. Hence, standard sequential languages have been augmented with directives and constructs to aid the compiler in generating message passing code. Fortran D [43, 58] is one such language. Fortran D consists of a set of extensions to Fortran 77 which specify how data is to be distributed among the processors of a distributed memory machine. The Fortran D compiler developed at Rice University [122, 59] can translate a Fortran D program into a Fortran 77 plus message passing node program. Another such language is Fortran 90D [128]. Fortran 90D consists of a set of extensions to Fortran 90, similar to those used in Fortran D. The Fortran 90D compiler developed at Syracuse University [13] translates Fortran 90D programs to Fortran 77 plus message passing node programs. Vienna Fortran [21, 130] and CM Fortran [121] also allow the user to write programs in global address space.

Recently, the High Performance Fortran Forum, comprising a group of researchers from industry, academia and research laboratories, developed a standard language called High Performance Fortran (HPF) [57, 67]. HPF was designed to provide a

portable extension to Fortran 90 for writing data-parallel applications. It includes features for mapping data to processors, specifying data-parallel operations, and methods for interfacing HPF programs to other programming paradigms. It is expected that HPF will be a standard programming language for computationally intensive applications on many types of machines, such as massively parallel MIMD and SIMD multiprocessors as well as traditional vector processors. In order for HPF to be successful, it needs a powerful compiler. Compiler technology for HPF is still in its infancy and current versions of HPF compilers are not robust and mature enough to compile large production codes efficiently. However once good compilers are available, the modern features and powerful capabilities of HPF are expected to make it the new popular version of Fortran for scientists and engineers solving complex large-scale problems [67].

1.4 Need for High Performance I/O

Another important issue in high performance computing is providing support for high performance parallel input-output (I/O). I/O for parallel systems has drawn increasing attention in the last few years as it has become apparent that I/O performance rather than CPU or communication performance may be the limiting factor in future computing systems. Large scale scientific computations, in addition to requiring a great deal of computational power, also deal with large quantities of data. At present, a typical Grand Challenge Application could require 1Gbyte to 4Tbytes of data per run [38]. These figures are expected to increase by orders of magnitude as teraflop machines make their appearance. Although supercomputers have very large main memories, the memory is not large enough to hold this much amount of data. Hence, data needs to be stored on disk and the performance of the program depends on how fast the processors can access data from disks. In order to have a balanced system [75], it is essential that the I/O bandwidth is comparable to the CPU and communication bandwidth. Unfortunately, the performance of the I/O subsystems of MPPs has not kept pace with their CPU and communication capabilities. It is still

orders of magnitude more expensive to do I/O than to do computation or communication. Improvements are needed in both hardware and software in order to improve I/O performance.

1.5 Contributions of this Thesis

This thesis addresses several issues in providing runtime support for in-core as well as out-of-core programs on distributed memory parallel computers. This runtime support can be used for performing many of the commonly required operations in application programs written using a distributed memory programming model. The use of runtime support makes it easier to write application programs and provides greater efficiency and portability. We mainly focus on runtime support for regular computations in which the communication and I/O patterns are known statically.

This runtime support can also be used together with a compiler to translate programs written in a high-level data-parallel language like HPF to node programs for distributed memory parallel computers. In fact it forms an essential part of the Fortran 90D/HPF compiler developed at Syracuse University [13]. Runtime support helps to separate the machine dependent aspects of compilation from the machine independent aspects. The compiler can do all the machine independent transformations and the runtime system can be optimized for each different machine. Thus, a portable and efficient compiler can be obtained by porting the runtime system to different machines. The runtime support discussed in this thesis is general and can be used in any other HPF compiler or a compiler for any other data-parallel language.

In distributed memory programs, arrays are distributed across processors in some fashion. For a number of reasons, it is often necessary to change the distribution, or redistribute the arrays during the course of program execution. This thesis presents efficient algorithms for runtime array redistribution. Since array distribution and redistribution is rigorously defined in HPF, the algorithms are developed for redistributing arrays as defined in HPF. The algorithms are independent of the communication mechanism used and hence are portable. A novel approach is proposed for

performing the general cyclic(x) to cyclic(y) redistribution using two methods, called the GCD Method and the LCM Method, which have low runtime overhead. We have implemented all the algorithms on the Intel Touchstone Delta and they are found to perform very well for different number of processors and array sizes.

A collective communication pattern which arises very often in many applications such as two-dimensional Fast Fourier Transform, parallel quicksort as well as in array redistribution, is the all-to-all communication pattern, also called complete exchange. This thesis presents several algorithms for all-to-all collective communication on fat-tree and two-dimensional mesh interconnection topologies. Previous work in this area tried to minimize link contention by increasing the number of communication steps. The algorithms proposed in this thesis take advantage of the fact that in many of the present generation machines like the Touchstone Delta and Paragon, the communication links have excess bandwidth and can tolerate a certain amount of link contention. Hence communication can be performed in fewer steps by allowing a small amount of link contention to exist. The performance of these algorithms on the CM-5 and Touchstone Delta is studied extensively. A model for estimating the time taken by these algorithms on the basis of system parameters has been developed and validated by comparing with experimental results.

A large number of applications deal with very large data sets which cannot fit in main memory, and hence have to be stored in files on disks. This thesis also describes the design and implementation of efficient runtime support for out-of-core computations. The runtime system supports three different models of data storage and access. A number of optimizations have been incorporated for improved performance. A new method, called the Extended Two-Phase Method, is proposed for accessing sections of out-of-core arrays efficiently. This method uses collective I/O in which processors cooperate to perform I/O in an efficient manner by combining several I/O requests into fewer larger requests, eliminating multiple file accesses for the same data and reducing contention for the I/O subsystem. A dynamic scheme is used for partitioning the I/O workload among processors, depending on the access requests. Performance results obtained using this runtime support for out-of-core

programs on the Touchstone Delta are presented.

A Parallel Compiler Runtime Consortium (PCRC) [45] has recently been formed with the goal of developing a common runtime support for high level parallel languages. We believe that the research presented in this thesis would be very useful to this consortium. It is also useful to vendors developing commercial HPF compilers, such as The Portland Group Inc. (PGI), Applied Parallel Research (APR), Digital Equipment Corporation (DEC) and others. Application programmers writing distributed memory programs would also benefit a great deal by using the ideas proposed in this thesis.

1.6 Related Work

This section briefly describes some of the related research in parallel languages, parallel compilers, runtime support systems and support for high-performance parallel I/O.

The concept of defining processor arrays and distributing data to them was first introduced in the programming language BLAZE [82] in the context of shared memory systems with non-uniform access times. This research was continued in the Kali [65] programming language for distributed memory machines. The Kali compiler uses the inspector/executor strategy to parallelize irregular computations. The compilation system SUPERB [129] parallelizes sequential programs semi-automatically for distributed memory machines. The SUPERB tool transforms sequential Fortran programs with data distribution annotations into parallel programs. Compilers for functional languages like Id Nouveau and Crystal have been developed for distributed memory machines. The Crystal compiler generates communication statements by studying the access patterns of the arrays in a statement. Split-C [32] is a parallel extension of C intended for high performance programming on distributed memory multiprocessors. It provides a global address space and allows a mixture of shared memory, message passing and data-parallel programming styles for both regular and

irregular problems. A compiler for Split-C is being developed at the University of California, Berkeley, with a runtime support system which uses Active Messages [124]. pC++, a data-parallel extension to C++, has been developed at Indiana University [5]. A Fortran D compiler is being developed at Rice University [122].

Some research has been done in developing compilers which can automatically determine a good distribution and alignment of arrays instead of having the user specify them. Gupta [50] has proposed a constraint based approach to automatically determine a good data distribution. This method has been incorporated in the PARADIGM compiler [49, 112]. The PARADIGM project at the University of Illinois aims at developing a fully automated compiler to translate sequential programs to parallel programs for distributed memory computers. The problem of automatic alignment of arrays has been studied by Chatterjee et al. [22] and Li et al. [79].

There has been some research in runtime support for applications with irregular data access patterns. The PARTI/CHAOS toolkit is a collection of runtime library routines to handle irregular computations [35, 102]. These primitives have been integrated with the Fortran D compiler at Rice University, the Fortran 90D/HPF compiler at Syracuse University and the Vienna Fortran compiler at the University of Vienna. Compilation methods for irregular problems have been investigated by Ponnusamy [93], Das [34] and Hanxleden [125]. Agrawal et al. [1] describe how runtime support can be integrated with a compiler to solve irregular block-structured problems.

Research has also been done in the area of array redistribution. Gupta et al. [53] and Koelbel [66] provide closed form expressions for determining the send and receive processor sets and data sets for redistributing arrays between block and cyclic distributions. An analytical model for evaluating the communication cost of data redistribution is presented in [63]. A multiphase approach to redistribution is discussed in [64]. Wakatani and Wolfe [126] describe a method of array redistribution called Strip Mining Redistribution in which parts of an array are redistributed in sequence, instead of redistributing the entire array at one time as a whole. The reason for doing

this is to try to overlap the communication involved in redistribution with some of the computation in the program. Kalns and Ni [62] present a technique for mapping data to processors so as to minimize the total amount of data that must be communicated during redistribution. Ramaswamy and Banerjee [99] discuss algorithms for redistribution based on a mathematical representation for regular distributions called PITFALLS. There has also been some research on the closely related problem of determining the local addresses and communication sets for array assignment statements like $A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2)$ where A and B have different cyclic(m) distributions [23, 110, 111, 52].

Algorithms for all-to-all communication (complete exchange) on a hypercube have been proposed by Bokhari [6], and also by Johnsson and Ho [61]. Complete exchange algorithms for a two-dimensional mesh are discussed in [103, 7, 113, 51, 54]. Optimal communication algorithms on the hypercube have been developed by Fox and Furmanski [42]. Algorithms for scheduling irregular communication patterns have been proposed by Wang and Ranka [100, 127] and also by Shankar and Ranka [106, 107, 108].

Compiling out-of-core data-parallel programs is a fairly new topic and there has been very little research in that area. A model and compilation strategy for out-of-core data-parallel programs is discussed in [10]. Bordawekar [11, 8] is developing a compiler for out-of-core HPF programs which uses the runtime library [116, 25, 115, 118] discussed in Chapters 5 and 6 of this thesis. Cormen and Colvin [31] are developing a compiler-like preprocessor for out-of-core C*, called ViC*, which translates out-of-core C* programs to standard C* programs with calls to a runtime library for I/O. Paleczny et al. [89] are also developing techniques for compiling out-of-core data-parallel programs. Brezany et al. [18] are working on compilation techniques for out-of-core programs in the context of Vienna Fortran. Language extensions for out-of-core data-parallel programs are proposed in [8, 17, 109, 18].

There has been a lot of effort to improve parallel I/O performance both by hardware and software means. Various RAID schemes (Redundant Arrays of Inexpensive

Disks) are proposed in [91] for better performance, reliability, power consumption and scalability. An excellent overview of RAID concepts is given in [24]. Disk striping where data is distributed across disks at a small granularity so that each block is distributed across all disks is proposed in [101]. File declustering, where different blocks of a file are stored on distinct disks is suggested in [81]. This is used in the Bridge File System [39], in Intel's Concurrent File System (CFS) [92] and in various RAID schemes [91]. Vesta is a parallel file system which supports logical partitioning of files [29, 27]. The RAMA [84] file system distributes file blocks across disks randomly using a hash function, instead of the usual striped layout. Runtime libraries for parallel I/O, such as the Panda Library [104, 105] and the Jovian Library [4], are being developed. Portable parallel file systems such as VIP-FS [55], PIOUS [85] and PPFS [60] have been developed recently. Techniques for improving I/O performance using collective I/O have also been proposed. Two-phase I/O is a technique for performing collective I/O using a runtime library [37, 12]. Disk-directed I/O is a technique for performing collective I/O at the file system level [69, 70, 71].

1.7 Organization of this Thesis

The rest of this thesis is organized as follows. Chapter 2 gives an overview of some of the issues in providing runtime support for in-core and out-of-core data-parallel programs. Runtime support for array redistribution is discussed in Chapter 3. Chapter 4 describes runtime support for all-to-all collective communication on fat-tree and two-dimensional mesh interconnection topologies. Runtime support for out-of-core programs is discussed in Chapters 5 and 6. Chapter 5 describes three models of data storage and access for out-of-core programs, and a number of local optimizations for accessing out-of-core data efficiently. Chapter 6 describes the Extended Two-Phase Method for accessing sections of out-of-core arrays using collective I/O. Finally, conclusions are presented in Chapter 7, along with some ideas for future work.

Chapter 2

Issues in Runtime Support

This chapter gives a brief overview of the various issues involved in providing runtime support for programs on distributed memory parallel computers. Runtime support can either be used directly in message passing application programs, or used together with a compiler to translate programs written in a high-level data-parallel language such as HPF.

2.1 Runtime Support for Regular Problems

There are many scientific applications which have very regular array access patterns. These access patterns may arise either from the underlying physical domain being studied, or the type of algorithm used. Examples of such applications include

- Matrix Multiplication, LU Decomposition and other operations in dense linear algebra
- Solving Partial Differential Equations (PDEs) on regular meshes
- Fast Fourier Transform

The main characteristic of such applications is that the communication pattern is known statically before program execution. Thus all the necessary optimizations can be performed beforehand at compile time. In such applications, data is usually

distributed using either a block, cyclic, or block-cyclic distribution [119]. The array subscripts used in the program are usually linear functions of the loop indices.

Many different communication patterns are possible depending on the array access pattern in the program. Li and Chen [78] characterize many of the commonly occurring communication patterns. A library of collective communication routines is needed for carrying out communication efficiently. A particular type of communication pattern which occurs frequently is the all-to-all communication pattern. Efficient algorithms for all-to-all communication are presented in Chapter 4 of this thesis. Runtime support is also needed for array redistribution. Although arrays are distributed among processors at the start of the program, it is very often necessary to change the distribution during program execution, which is called array redistribution. This involves calculation of source and destination processor and index sets as well as interprocessor communication. Runtime support for array redistribution is discussed in detail in Chapter 3 of this thesis.

2.2 Runtime Support for Irregular Problems

There is another set of scientific applications in which the array access pattern is irregular. This is usually due to the fact that the underlying physical domain is irregularly connected. Examples of such applications include

- Computational Fluid Dynamics (CFD) codes using unstructured meshes
- Molecular dynamics codes
- Sparse iterative linear systems solvers

In irregular problems, arrays are accessed using one or more level of indirection. An example of this is the following loop

```
do i=1, n
    A(x(i)) = B(y(i)) + C(z(i))
end do
```

A regular distribution of data, such as block, cyclic or block-cyclic, may not be the best

distribution for these problems because it may result in higher communication cost. A graph partitioning algorithm is normally used to determine the best distribution in the case of irregular applications. Such a distribution is called an irregular distribution. Due to the indirection in the array access, the communication pattern is not known statically at compile time. It depends on the values of the indirection arrays, which may not be known until runtime. Thus a runtime resolution scheme is needed to determine the communication required.

Runtime support for irregular problems has been studied by a number of researchers [93, 102, 34, 35, 68]. One way of detecting and performing the communication is by using an *inspector-executor* [68, 102] approach. A parallel loop is transformed into two constructs called an inspector and an executor. During program execution, the inspector examines the data references made by a processor, and calculates what off-processor data needs to be fetched and where that data will be stored once it is received. The executor loop then uses the information from the inspector to perform the actual communication and computation.

PARTI [35] is a library of runtime routines for solving irregular problems on distributed memory computers. PARTI primitives can be directly used to generate the inspector/executor pairs. Each inspector produces a communication schedule, which is essentially a pattern of communication for gathering or scattering data. In order to avoid duplicate accesses, a list of off-processor data references is stored locally for each processor in a hash table. For each new off-processor data reference required, a search through the hash table is performed in order to determine if this reference has already been accessed. If the reference has not previously been accessed, it is stored in the hash table, otherwise it is discarded. The primitives thus only fetch a single copy of each unique off-processor distributed array reference. The executor contains embedded PARTI primitives to communicate data. The primitives issue instructions to gather, scatter or accumulate (i.e. scatter followed by add) data according to the schedule created by the inspector. Latency or startup cost is reduced by packing small messages intended for the same destination into one large message.

2.3 Runtime Support for Compilers

Data-parallel languages like HPF [57] and pC++ [5] have recently been developed to provide support for high performance programming on parallel machines. These languages provide a framework for writing portable parallel programs independent of the underlying architecture and other idiosyncrasies of the machine. The same program can be run on different machines by simply using a different compiler for each machine. Compilers for such languages usually rely on runtime support to carry out many of the commonly required operations usually involving communication. Runtime support helps to separate the machine dependent aspects of compilation from the machine independent aspects. The compiler can do all the machine independent transformations and the runtime system can be optimized for each different machine. Thus, a portable and efficient compiler can be obtained by simply porting the runtime system to different machines.

We briefly describe some of the issues related to providing runtime support for an HPF compiler. We do not discuss runtime support for compiling irregular problems; that is explained in detail in [93]. We first outline the salient features of HPF in order to explain the runtime support needed for an HPF compiler.

2.3.1 Overview of HPF

HPF was designed to be a standard portable programming language for writing efficient computationally intensive parallel programs. HPF uses Fortran 90 as its base language and provides several extensions to Fortran 90. The new HPF language features fall into four categories with respect to Fortran 90: new directives, new language syntax, new library routines, and some language restrictions. The new directives are structured comments that suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but do not change the value computed by the program. Compiler directives form the heart of the HPF language. They start with the prefix `!HPF$`, `CHPF$` or `*HPF$` which would actually be treated as comments in Fortran 90. Some of the

new language features are `FORALL` statement and `FORALL` construct. HPF extends the Fortran 90 library of intrinsic functions to include additional reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.

The HPF approach is based on two key observations. First, the overall efficiency of the program can be increased if many operations are performed concurrently by different processors, and secondly, the efficiency of a single processor is likely to be the highest if the processor performs computations on data elements stored in its local memory. Therefore, HPF provides means for explicit expression of parallelism and data mapping. The data alignment and distribution directives in HPF allow the programmer to inform the compiler how to partition arrays. Arrays are first aligned to a *template* or index space using the `ALIGN` directive. The `DISTRIBUTE` directive specifies how the template is to be distributed among a set of abstract processors. The mapping of abstract processors to physical processors is not specified by HPF and is language-processor dependent. The combination of alignment (from arrays to templates) and distribution (from templates to processors) thus determines the mapping of an array to the processors. The data mapping is declared using the directives: `PROCESSORS`, `ALIGN`, `DISTRIBUTE` and, optionally, `TEMPLATE`. In addition, arrays may be remapped at runtime. For this, the array must be declared using the `DYNAMIC` directive and the actual remapping is specified using the executable directives `REALIGN` and `REDISTRIBUTE`.

In HPF, an array may be aligned with another in many ways including shifts, strides, or any other linear combination of a subscript (ie., $n \times i + m$), transposition of indices, and collapse or replication of array dimensions. Irregular alignments are not allowed. The template can be distributed as `BLOCK`, `CYCLIC` or `CYCLIC`(m). In a block distribution, contiguous blocks of the array are distributed among the processors. In a cyclic distribution, array elements are distributed among processors in a round-robin fashion. In a cyclic(m) distribution, blocks of size m are distributed cyclically. Irregular distributions are not allowed in version 1.0 of HPF.

2.3.2 Compiler and Runtime Support for HPF

An HPF compiler which translates in-core HPF programs to message passing node programs for distributed memory parallel computers has been developed at Syracuse University [13, 15]. It translates HPF programs to Fortran 77 programs with calls to a runtime library [2, 14, 119]. The compiler only exploits the parallelism expressed in data parallel constructs such as `FORALL`, `WHERE` and array assignment statements. It does not attempt to parallelize other constructs such as `DO` loops and `WHILE` loops, since they are used only as naturally sequential control constructs in HPF. The compiler mainly recognizes commonly occurring computation and communication patterns. These patterns are then replaced by calls to the optimized runtime support system routines. The runtime support system includes parallel intrinsic functions, data redistribution routines, communication primitives and several other miscellaneous routines.

The basic structure of the HPF compiler is organized around four major modules — parsing, data partitioning, communication detection and insertion, and code generation as shown in Figure 2.1. The compiler first creates a parse tree from the input HPF program. Also, each array assignment statement and `WHERE` statement is internally transformed into its equivalent `FORALL` statement, so that the subsequent steps only need to deal with `FORALL` statements. The partitioning module processes the data distribution directives, namely `TEMPLATE`, `DISTRIBUTE` and `ALIGN`, and partitions data and computation among processors. After partitioning, the parallel constructs in the node program are sequentialized since the code will be executed on a single processor. Array operations and `FORALL` statements in the original program are transformed into `DO` loops. The communication module detects communication requirements and inserts appropriate communication primitives.

Finally, the code generator produces *loosely synchronous* [44, 46] SPMD code. The generated code is structured as alternating phases of local computation and communication. Local computations consist of operations by each processor on the data in its own memory. The communication phase includes any transfer of data among processors, possibly with arithmetic or logical computation on the data as it

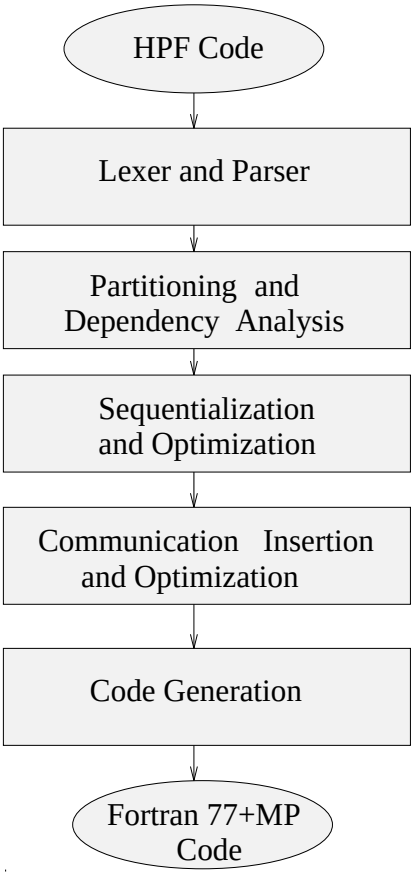

Figure 2.1: Phases of Compilation

is transferred (e.g. reduction functions). In such a model, processors do not need to synchronize during local computation. But, if two or more processors communicate with each other, they are implicitly synchronized during the communication.

Communication Library

The HPF compiler described above relies on a very powerful runtime support system which includes a library of collective communication routines [13], a library of intrinsic functions [2, 14] and other runtime routines such as for array redistribution [119]. The HPF compiler produces calls to collective communication routines instead of generating individual send and receive calls inside the compiled code. This is done

for the following reasons:

1. *Improved performance*: To achieve good performance, interprocessor communication must be minimized. By developing a separate library of interprocessor communication routines, each routine can be optimized.

2. *Increased portability of the compiler*: By separating the communication library from the basic compiler design, portability is enhanced because to port the compiler, only the machine specific low-level communication calls in the library need to be changed.

3. *Better estimation of communication costs*: The cost of collective communication routines can be determined more precisely, which enables one to make a better estimate of the time a program will take.

The compiler must recognize the presence of collective communication patterns in the program in order to generate the appropriate communication calls. This involves a number of tests on the relationship among subscripts of various arrays in a `FORALL` statement. These tests also include information about array alignments and distributions. The compiler uses pattern matching techniques to detect communication patterns [13, 15].

Intrinsic Library

Intrinsic functions form an important feature of Fortran 90 and HPF. They directly support many of the basic data-parallel operations on arrays and provide a means for expressing operations on arrays concisely. HPF includes all Fortran 90 intrinsic functions and also adds a number of new intrinsic procedures in three categories: system inquiry intrinsics, mapping inquiry intrinsics, and computational intrinsics. The computational intrinsic functions fall into the following main categories:-

- *Simple Reduction Functions*: `ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM`.

- *Combining Scatter Functions*: ALL_SCATTER, ANY_SCATTER, COUNT_SCATTER, MAXVAL_SCATTER, MINVAL_SCATTER, PRODUCT_SCATTER, SUM_SCATTER
- *Prefix and Suffix Functions*: ALL_PREFIX, ANY_PREFIX, COUNT_PREFIX, MAXVAL_PREFIX, MINVAL_PREFIX, PRODUCT_PREFIX, SUM_PREFIX, ALL_SUFFIX, ANY_SUFFIX, COUNT_SUFFIX, MAXVAL_SUFFIX, MINVAL_SUFFIX, PRODUCT_SUFFIX, SUM_SUFFIX
- *Array Sorting Functions*: GRADE_UP, GRADE_DOWN
- *Array Manipulation Functions*: CSHIFT, EOSHIFT, TRANSPOSE.
- *Array Location Functions*: MAXLOC, MINLOC
- *Array Construction Functions*: SPREAD, MERGE, PACK, UNPACK
- *Vector and Matrix Multiplication Functions*: DOT_PRODUCT, MATMUL.
- *Bit Manipulation functions*: IAND, IOR, POPCNT, POPPAR, LEADZ
- *Elemental Intrinsics functions*: SIN, COS, TAN

It is necessary to have a library of these intrinsic functions which can be called from the node programs of a distributed memory machine [14]. The HPF compiler detects calls to intrinsic functions in the HPF program and replaces them with calls to these routines. Since arrays in HPF can have up to seven dimensions and can be distributed in many different ways, it is not feasible to write intrinsic libraries for all possible distributions. A more practical approach is to write optimized routines for a few distributions. If the distribution of an array is different from what is expected by the intrinsic library, the array must first be redistributed to the desired distribution, and after returning from the intrinsic, it must be redistributed back to the original distribution. It is essential to use efficient algorithms for redistribution, so as to minimize the redistribution overhead. Efficient algorithms for redistributing arrays are discussed in Chapter 3 of this thesis.

2.4 Runtime Support for Out-of-Core Programs

An out-of-core program is one in which all the data required by the program cannot fit in main memory at the same time. Hence data needs to be stored in files on secondary storage such as disks. During program execution, only a portion of the data can be present in memory at any time. This makes it necessary to move data back and forth between main memory and disks. Since the cost of performing I/O is very high compared to computation and communication, this can prove to be very expensive. Hence it is necessary to do the I/O in out-of-core programs efficiently to get good performance.

The I/O performance is best when a processor makes large contiguous requests instead of many small requests. This is because of the high latency time associated with any I/O operation. In a parallel computer, when many processors need to do I/O simultaneously, there is the additional problem of contention for the I/O system. Hence it is necessary to schedule I/O requests of different processors as well as reorder and combine I/O requests within and across processors into large contiguous requests. We believe that this can best be done by having a library of optimized routines which can be directly called from an out-of-core program.

Another important factor is the portability of out-of-core programs. There is no standard parallel file system interface or Application Program Interface (API) at present. Each parallel machine has its own parallel file system with a completely different interface from that of any other parallel file system. Hence programs written directly using calls to the parallel file system are not portable to other systems. This problem can be overcome by using runtime support. The application programs can call a runtime library which provides a common interface for all machines. The routines in the runtime library can be implemented using the native parallel file system on each different machine.

Thus runtime support is needed in the case of out-of-core programs for efficiency and portability. As in the in-core case, this runtime support can also be used together with a compiler to translate out-of-core programs written in a high-level language like

HPF to node programs with calls to the runtime library for I/O and communication. Chapters 5 and 6 of this thesis describe in detail the design and implementation of runtime support for out-of-core programs, including a number of optimizations.

2.4.1 Out-of-Core Applications

There are a number of applications which deal with very large data sets, resulting in out-of-core programs. These applications exist in diverse areas such as large scale scientific computations, database and information processing, hypertext and multimedia systems, information retrieval and many others.

del Rosario and Choudhary [38] have done a survey of the I/O requirements of several Grand Challenge applications. They find that the data requirements of these applications range from 1 Gbyte to 4 Tbytes per run. Some of the applications they surveyed are:-

- *Imaging of Planetary Data*: The spacecraft Magellan has gathered more than 3 Tbytes of data from the surface of the planet Venus. To produce a three-dimensional rendering of the surface of Venus at 200 Mbytes of data per frame would require over 13 Gbytes/sec. of I/O throughput at 50 frames per second [48].

- *Climate Prediction*: A climate prediction code using the General Circulation Model (GCM) has the following requirements on the Intel Touchstone Delta. For a 100-year atmosphere run with 300 km^2 resolution and 0.2 simulated years/machine hour, the simulation takes 3 weeks run time and generates 1144 Gbytes of data at 38 Mbytes per simulation minute. For a 1000-year coupled atmosphere-ocean run with a 150 km^2 resolution, the atmospheric simulation takes about 30 weeks, while ocean simulation takes 27 weeks; the process produces 40 Mbytes of data per simulation minute, or a total of 20 Tbytes of data for the entire simulation [40].

- *Four-Dimensional Data Assimilation*: This application from NASA incorporates actual space-time observations into mathematical and computational models in order to create a unified, complete description of the atmosphere. The algorithm for this currently operates on a 3 Tbyte database with single runs producing 100Mbytes to several Gbytes of output. These figures are expected to increase by orders of magnitude in the near future.

Cormen [30] has also compiled a list of several applications which deal with huge data sets. These include applications in computational biology, computational fluid dynamics, combinatorial search, conjugate gradient solvers, genetic algorithms, geophysics, region growing, logic simulation, computational physics, meteorology, molecular dynamics, ocean modeling, partial differential equations solvers, synthetic aperture radar image processing, visualization and graphics. The Applications Working Group of the Scalable I/O Initiative has provided a description of the I/O requirements of several applications in biology, chemistry, physics, earth sciences, engineering and graphics [97]. Fox [41] presents a performance analysis of applications on a hypercube machine with a hierarchical memory at each node, consisting of a fast cache and slower main memory. The applications considered are the N-body problem, Poisson's equation solver, matrix multiplication, LU Decomposition, Fast Fourier Transform and neural networks. This performance analysis can be extended to out-of-core applications in which the memory hierarchy includes a much slower secondary memory.

There have also been studies of the file-access characteristics of application programs on multiprocessor systems. Kotz and Nieuwejaar [74] present the results of a three week tracing study in which all file-related activity on an Intel iPSC/860 running production, parallel scientific applications at NASA Ames Research Center was recorded. An interesting result of this study was that there were a large number of small strided requests to the file system. They found that 96.1% of all reads were for fewer than 4000 bytes, but those reads transferred only 2% of all data read. For writing, 89.4% of all requests were for fewer than 4000 bytes, but those writes transferred only 3% of all data written. Another study of the file-access characteristics

of production applications on the CM-5 at the National Center for Supercomputing Applications, University of Illinois, also found similar results of a large number of small requests [98]. This shows that although it is well-known that I/O should be done in large chunks to minimize the effect of high I/O latency, many parallel out-of-core applications actually access small strided data sets. Hence, it is necessary to provide runtime support for accessing small strided data efficiently. This issue is addressed in this thesis and optimizations, such as data sieving and collective I/O using an Extended Two-Phase Method, are proposed to access strided data in an efficient manner.

Chapter 3

Runtime Support for Array Redistribution

3.1 Introduction

In distributed memory parallel computers, arrays have to be distributed among processors in some fashion. The distribution can either be regular i.e. block, cyclic or block-cyclic as in Fortran D [43] and High Performance Fortran (HPF) [57, 67]; or irregular in which there is no function specifying the mapping of arrays to processors. The distribution of an array does not need to remain fixed throughout the program. In fact, it is very often necessary to change the distribution of the array at runtime, which is called *array redistribution*. This involves calculating source and destination processor and index sets as well as data communication. It is essential to use efficient algorithms for redistribution, otherwise the performance of the program may degrade considerably.

This chapter describes efficient algorithms for runtime array redistribution. Since array distribution and redistribution is rigorously defined in HPF, the algorithms are developed for redistributing arrays as defined in HPF. We consider block(m) to cyclic, cyclic to block(m) and the general cyclic(x) to cyclic(y) type redistributions. For the general cyclic(x) to cyclic(y) redistribution, there is no direct algebraic formula to calculate the source and destination processor and index sets [53]. We use a novel approach for doing the general cyclic(x) to cyclic(y) redistribution, where we first

develop optimized algorithms for two special cases — when x is a multiple of y, or y is a multiple of x. For the general case, we propose two algorithms called the *GCD Method* and the *LCM Method* which make use of the optimized algorithms developed for the above special cases. We initially describe algorithms for one-dimensional arrays and then extend the methodology to multidimensional arrays. The algorithms proposed have low runtime overhead. We study the performance of these algorithms on the Intel Touchstone Delta.

3.1.1 Need for Redistribution

HPF provides directives (`ALIGN` and `DISTRIBUTE`) which specify how arrays should be partitioned among the processors of a distributed memory computer. Arrays are first aligned to a *template* or index space. The `DISTRIBUTE` directive specifies how the template is to be distributed among the processors. In HPF, an array (or template) can be distributed as `BLOCK`(m) or `CYCLIC`(m) [57]. Though the distribution of an array is specified at compile time, there are a number of reasons for which it may be necessary to redistribute an array during the execution of the program.

- HPF has a directive called `REDISTRIBUTE` by which an array can be redistributed anywhere in the program provided the array was declared as `DYNAMIC`. `REDISTRIBUTE` can be considered as an executable directive.

- HPF provides intrinsic functions which operate on arrays. It is not practical to write intrinsic and runtime libraries for all possible distributions, especially since arrays can have up to seven dimensions. Libraries can be written for a few common distributions and for any other distribution it is necessary to redistribute before calling the subroutine. After returning from the subroutine it is necessary to redistribute back to the original distribution. We call this type of redistribution as a *circular redistribution*. This is illustrated in Figure 3.1 which shows an HPF program calling the `MATMUL` intrinsic with all arrays distributed as `(BLOCK,*)`. The `MATMUL` library routines have been written for `(BLOCK,BLOCK)` and `(CYCLIC,CYCLIC)` distributions. So it is necessary to redistribute to one

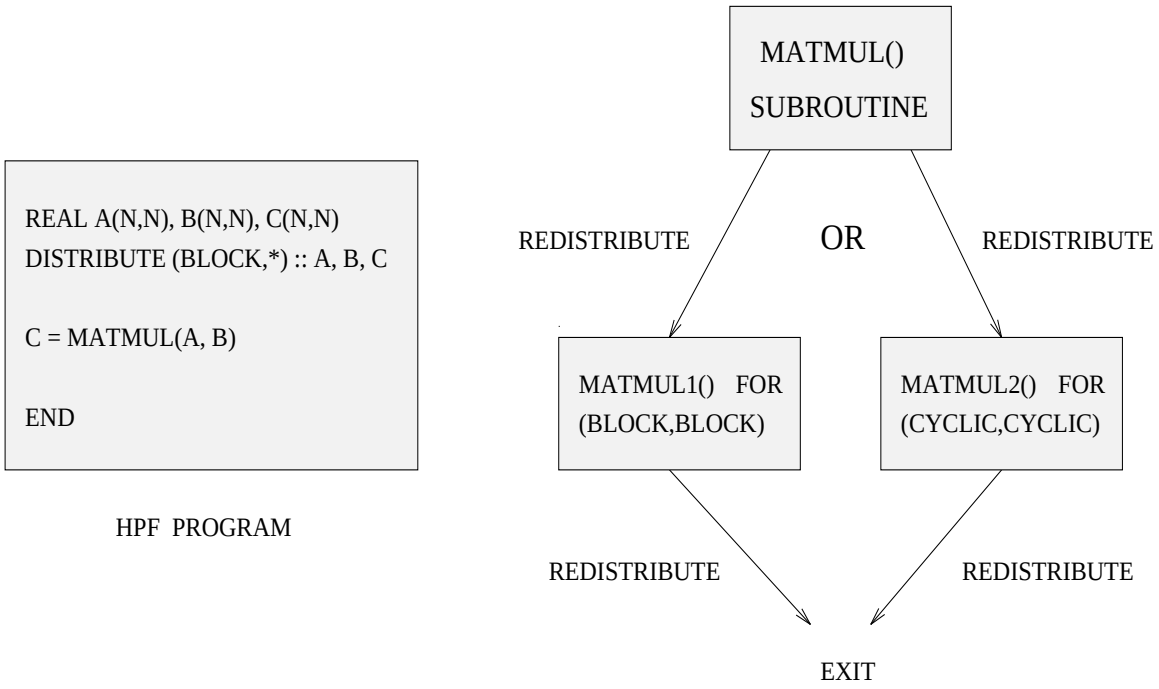

Figure 3.1: Need for Redistribution

of these distributions before calling the intrinsic and then redistribute back to `(BLOCK,*)` after the intrinsic.

- Arrays and array sections can be passed as arguments to subroutines. The array (or array section) can be specified to have any distribution in the subroutine. If this distribution is different from that in the calling program, the array (or array section) needs to be redistributed before the subroutine is called. At the end of the subroutine, the array (or array section) needs to be redistributed to the original distribution.

- In many applications such as 2D FFT and the ADI method for solving multidimensional PDEs, dynamic redistribution can result in significant performance improvement [20].

An example of an HPF program using redistribution is shown in Figure 3.2. This is a two-dimensional FFT program in which the array is first block distributed along

```
REAL A(1024,1024)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK,*) ONTO P
!HPF$ DYNAMIC A

.....

CALL ROW_FFT(A)
!HPF$ REDISTRIBUTE A(*,BLOCK) ONTO P
CALL COL_FFT(A)

.....

END
```

Figure 3.2: 2D FFT Program using Redistribution

rows. A one-dimensional FFT is first performed along each row of the array, which can be done without any communication. The array is then redistributed so that it is block distributed along columns. A one-dimensional FFT is performed along each column of the redistributed array, which again does not require any communication.

3.2 Notations and Definitions

The notations used in this chapter are given in Figure 3.3. We assume that all arrays are indexed starting from 1, while processors are numbered starting from 0 and that arrays are identically aligned to the template. The algorithms can be easily extended for the general case. We also assume that the number of processors on which the array is distributed remains the same before and after the redistribution.

The block(m) and cyclic(m) distributions in HPF are defined as follows. Consider an array of size N distributed over P processors. Let us define the ceiling division

N	global array size
P	number of processors
p_i	logical processor i
p	logical number of the processor executing the program
p_s	source processor
p_d	destination processor
L	local array size
m	block size

Figure 3.3: Notations

function $CD(j,k) = (j + k - 1)/k$ and the ceiling remainder function $CR(j,k) = j - k \times CD(j,k)$. Then block($m$) distribution means that index j of the array is mapped to logical processor number $CD(j,m) - 1$. Note that for a valid block(m) distribution, $m \times P \geq N$ must be true. Block by definition means the same as block($CD(N,P)$). In a cyclic(m) distribution, index j of the global array is mapped to logical processor number $MOD(CD(j,m) - 1, P)$[1]. Cyclic by definition means the same as cyclic(1).

Suppose we have 4 processors and an array of length 26. Distributing the array as `BLOCK` results in this mapping of array elements to processors :-

Proc. 0	1	2	3	4	5	6	7
Proc. 1	8	9	10	11	12	13	14
Proc. 2	15	16	17	18	19	20	21
Proc. 3	22	23	24	25	26		

[1] $MOD(a,b) = a \ \ modulo \ \ b$

Distributing the array as `CYCLIC` results in this mapping of array elements to processors :-

Proc. 0	1	5	9	13	17	21	25
Proc. 1	2	6	10	14	18	22	26
Proc. 2	3	7	11	15	19	23	
Proc. 3	4	8	12	16	20	24	

Distributing the array as `CYCLIC(3)` results in this mapping of array elements to processors :-

Proc. 0	1	2	3	13	14	15	25	26
Proc. 1	4	5	6	16	17	18		
Proc. 2	7	8	9	19	20	21		
Proc. 3	10	11	12	22	23	24		

In other words, in a block distribution, contiguous blocks of the array are distributed among the processors. In a cyclic distribution, array elements are distributed among processors in a round-robin fashion. In a cyclic(m) distribution, blocks of size m are distributed cyclically. The cyclic(m) distribution with $1 < m < \lceil N/P \rceil$ is commonly referred to as a block-cyclic distribution with block size m [43]. Block and cyclic distributions are special cases of the general cyclic(m) distribution. A cyclic(m) distribution with $m = \lceil N/P \rceil$ is a block distribution and a cyclic(m) distribution with $m = 1$ is a cyclic distribution. The formulae for conversion between local and global indices for the different distributions are given in Table 3.1.

The redistribution algorithms proposed in this thesis are intended to be portable. Hence, we do not specify how data communication should be performed because the best communication algorithms are often machine dependent. Redistribution requires all-to-many personalized communication in general and in many cases it requires all-to-all personalized communication. These communication algorithms are described in detail in Chapter 4 of this thesis and in [117, 95, 120]. The performance results presented in this chapter have been obtained using these algorithms. We do assume that all the data to be sent from any processor i to processor j has to be

Table 3.1: Data Distribution and Index Conversion

Note: This assumes that arrays are indexed starting from 1 and processors are numbered starting from 0

$$CD(j,k) = (j+k-1)/k \quad \text{and} \quad CR(j,k) = j - k \times CD(j,k)$$

	BLOCK(m)	CYCLIC	CYCLIC(m)
global index (g) to processor number (p)	$p = CD(g,m) - 1$	$p = MOD(g-1, P)$	$p = MOD(CD(g,m) - 1, P)$
global index (g) to local index (l)	$l = m + CR(g,m)$	$l = (g-1)/P + 1$	$l = MOD(g-1,m) + 1 + (g/(mP))m$
local index (l) to global index (g)	$g = l + mp$	$g = (l-1)P + p + 1$	$g = MOD(l-1,m) + 1 + (P((l-1)/m) + p)m$

collected together in a contiguous *packet* in processor i and sent in one communication operation, so as to minimize the communication startup cost. In the rest of this chapter, any division involving integers should be considered as integer division unless specified otherwise.

3.3 Block(m) to Cyclic Redistribution

We first consider the case of block(m) to cyclic redistribution, a few examples of which are shown in Figure 3.4.

Theorem 3.1 *Let l_{i1} and l_{i2} be the local array sizes in processor p_i corresponding to block(m) and cyclic distributions respectively. In a block(m) to cyclic redistribution, the number of processors to which p_i has to send data is*

$$\begin{array}{ll} P-1 & \text{if } l_{i1} \geq P \\ l_{i1} \text{ or } l_{i1}-1 & \text{if } l_{i1} < P \end{array}$$

The number of processors from which p_i has to receive data is at most

$$\begin{array}{ll} CD(N,m) & \text{if } l_{i2} \geq CD(N,m) \\ l_{i2} & \text{if } l_{i2} < CD(N,m) \end{array}$$

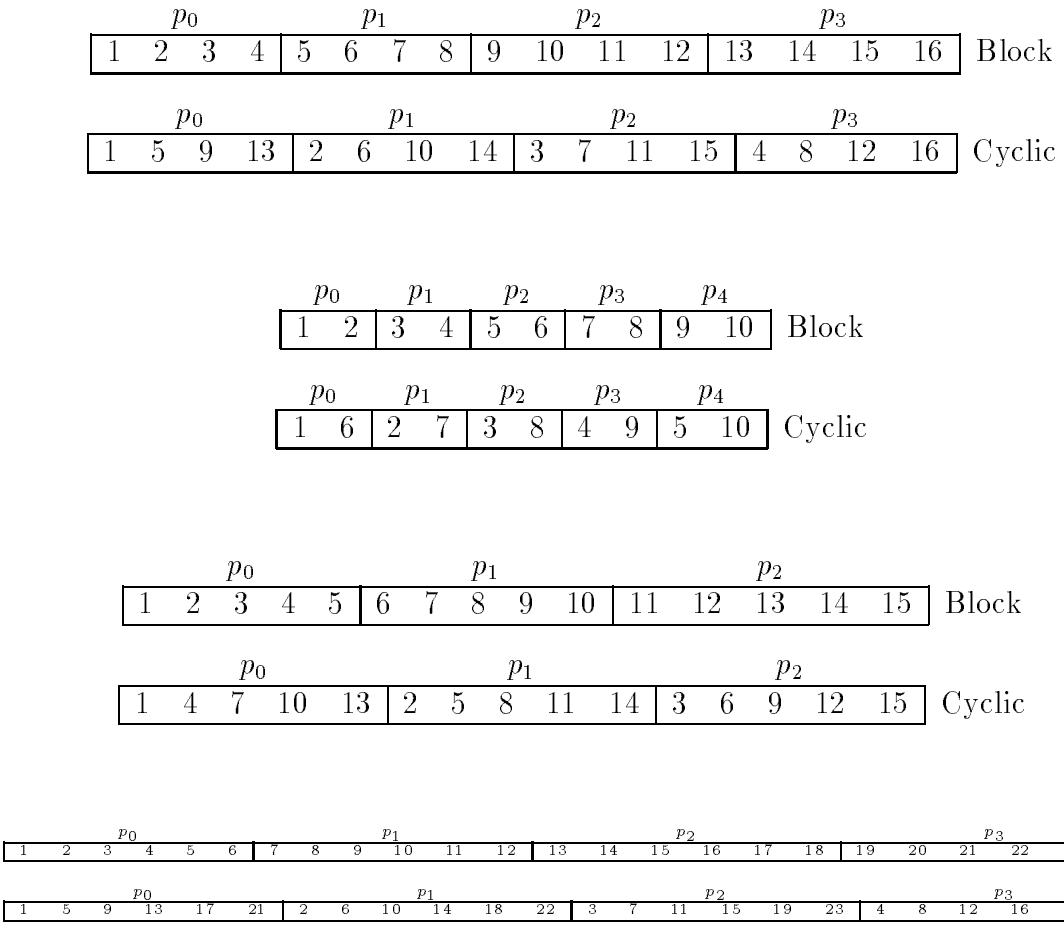

Figure 3.4: Block(m) to Cyclic Redistribution

Proof: Note that if N is not a multiple of P, l_{i1} may not be equal to l_{i2}. If $l_{i1} < P$, each of the l_{i1} elements of p_i corresponding to a block(m) distribution will lie in a different processor when the array is distributed cyclically. At most one of the l_{i1} elements will be remapped to processor p_i itself. Therefore, p_i will have to send data to either l_{i1} or $l_{i1} - 1$ processors. If $l_{i1} \geq P$, then clearly p_i has at least one element to send to every other processor.

In a block(m) distribution, the number of processors with data is given by $CD(N, m)$. Therefore, in the receive phase, if $l_{i2} < CD(N, m)$, each processor will receive data from at most l_{i2} processors. If $l_{i2} \geq CD(N, m)$, each processor will receive data from at most $CD(N, m)$ processors. □

The algorithm for block(m) to cyclic redistribution is given in Figure 3.5. In the send phase, each processor only needs to calculate the destination processor of the first element of the local array. The other elements have to be sent to the other processors in a round-robin fashion. Thus the array is scanned only once, which makes good use of the cache. In the receive phase, the data received from other processors has to be stored in contiguous memory locations in order of logical processor number. In every processor, the data received from processor 0 is stored first; from processor 1 second and so on. Hence addresses do not need to be calculated. If the amount of data to be received from all processors is known, the data can be directly received into appropriate locations in the array.

In a block(m) distribution, the last element N of the global array is mapped to processor $p_N = (N-1)/m$. If $p_N < P - 1$, no elements of the array are mapped to processors $p_N + 1, p_N + 2, ..., P - 1$. The index of the last element of the local array in processors $p < p_N$ is $last\_index = m$. The index of the last element in p_N is $last\_index = MOD(N-1, m) + 1$. The index of the first element of $p_s \leq p_N$ that is mapped to p in a cyclic distribution is given by

$$first\_index = MOD[p - MOD\{p_s\, MOD(m, P), P\} + P, P] + 1$$

Send Phase

1. The destination processor (p_d) of the first element of the local array is $p_d = MOD(p\,m, P)$.
2. Destination processor of element i, $i = 2 : N$, is $MOD(p_d + i, P)$.

Receive Phase

1. Last processor with data is $p_N = (N-1)/m$
2. For $p_s = 0$ to p_N do
3. If $(p_s = p_N)$ then
4. $last\_index = MOD(N-1, m) + 1$
5. Else
6. $last\_index = m$
7. The index of the first element of p_s mapped to p is $first\_index = MOD[p - MOD\{p_s\, MOD(m, P), P\} + P, P] + 1$
8. Number of elements to be received from p_s is 0, if $(last\_index < first\_index)$, and $(last\_index - first\_index)/P + 1$, otherwise
9. No data is to be received from processors $p_N + 1, p_N + 2, ..., P - 1$.
10 The received data is to be stored in order of processor number.

Figure 3.5: Algorithm for Block(m) to Cyclic Redistribution

If m is divisible by P, the first element of p_s that is mapped to p is $p + 1$. However, if m is not divisible by P, a shift is introduced in this simple mapping which is taken into account by the MOD expression in the above equation. Hence, the number of elements to be sent from any processor p_s to p is

$$\begin{cases} 0, & \text{if (last\_index} < \text{first\_index)} \\ & \text{or } (p_s > p_N) \\ (\text{last\_index} - \text{first\_index})/P + 1, & \text{otherwise} \end{cases}$$

where $first\_index$, $last\_index$ and p_N are calculated as above.

3.4 Cyclic to Block(m) Redistribution

A cyclic to block(m) redistribution is essentially the reverse of a block(m) to cyclic redistribution. The algorithm for cyclic to block(m) redistribution is given in Figure 3.6. In a cyclic distribution, the size of the local array in processor p is

$$L = \begin{cases} \lceil N/P \rceil & \text{if } MOD(N,P) = 0, \text{ or } p < MOD(N,P) \\ \lceil N/P \rceil - 1 & \text{otherwise} \end{cases}$$

In the send phase, each processor p calculates the destination processor p_d and the destination local address l_d of the first element of its local array as $p_d = CD(p+1, m) - 1$ and $l_d = m + CR(p+1, m)$. The first $(m - l_d)/P + 1$ elements from p_s have to be sent to p_d. The next set of elements starting from $i = (m - l_d)/P + 2$ have to be sent to processor $p_{d1} = CD((i-1)P + p + 1, m) - 1$. The destination local address of element i is given by $l_{d1} = m + CR((i-1)P + p + 1, m)$ and so $(m - l_{d1})/P + 1$ elements starting from i have to be sent to processor p_{d1}. This process is continued for the remaining elements of the array. Thus blocks of elements have to be sent to different processors.

In the receive phase, the data received cannot be directly stored in the array as the data has to be stored with a stride equal to the number of processors. Hence the data received has to be stored in a temporary buffer in memory. This gives us two choices in implementing the receive phase :–

1. *Synchronous Method*: Each processor waits till it receives data from all other processors, before placing any data in the local array. This increases the memory requirements of the algorithm and also increases processor idle time. These problems worsen as the number of processors is increased, so this method is not scalable.

2. *Asynchronous Method*: The processors do not wait for data from all processors to arrive. Instead, each processor takes any packet which has arrived and places the data from that packet into appropriate locations in the local array. This

Send Phase

1. $i = 1$
2. While $(i \leq L)$ do
3. The destination processor (p_d) and destination local address (l_d) of element i of the local array is $p_d = CD((i-1)P + p + 1, m) - 1$ and $l_d = m + CR((i-1)P + p + 1, m)$
4. The destination processor of elements i to $i + (m - l_d)/P$ is p_d.
5 $i = i + (m - l_d)/P + 1$

Receive Phase

Synchronous Method:
1. Receive packets from all processors.
2. The source processor of the first element of the local array $p_s = MOD(m\,p, P)$.
3. The source processor of element i, $i = 2 : N$, is $p_s = MOD(p_s + 1, P)$.

Asynchronous Method:
1. The source processor of the first element of the local array is $p_s = MOD(m\,p, P)$.
2. The source processor of the k^{th} element, $2 \leq k \leq P$, is $MOD(p_s + k - 1, P)$.
3. For $i = 0$ to $P - 1$ do
4. Receive a packet from any processor p_i
5. Starting from the location calculated above, place elements from the packet into the array with stride P.

Figure 3.6: Algorithm for Cyclic to Block(m) Redistribution

method *overlaps computation and communication.* Each processor posts non-blocking receive calls and waits for any packet to arrive. As soon as a packet is received, it places the data in appropriate locations in the local array. During this time, other packets may reach the processor. When the processor has placed all the data from the earlier packet into the local array, it takes up the next packet and so on. This reduces processor idle time. Since all packets do not have to be in memory at the same time, it also reduces memory requirements. This method is scalable as neither processor idle time nor memory requirements increase as the number of processors is increased.

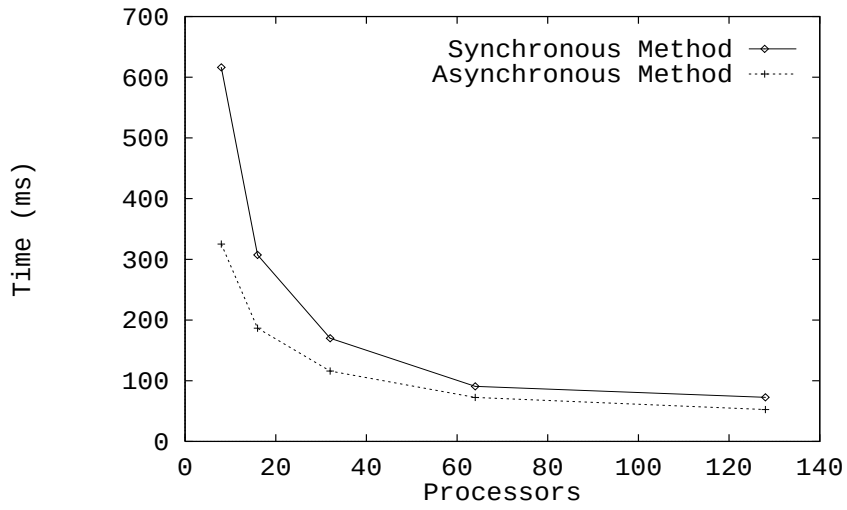

Figure 3.7: Performance of Cyclic to Block Redistribution, array size 1M integers

The array locations where incoming data has to be stored can be calculated as follows. The source processor (p_s) of the first element of the local array is given by $p_s = MOD(m\,p, P)$. The next $(P-1)$ elements will be received from the remaining processors in order of processor number. This cycle is repeated for all elements of the local array. If all packets are present in memory (Synchronous Method), the local array needs to be scanned only once to be filled. If the packets are processed one at a time (Asynchronous Method), the array elements are filled with stride P and the array has to be scanned P times. So, clearly the Synchronous Method makes better use of the cache than the Asynchronous Method. Figure 3.7 compares the performance of the Synchronous and Asynchronous Methods on the Intel Touchstone Delta for a global array with 1M (2^{20}) integers and the number of processors varied between 8 and 128. We observe that the Asynchronous Method performs much better than the Synchronous Method as it overlaps computation and communication and thus reduces processor idle time. This difference is larger for a small number of processors because the amount of data communicated per processor is larger.

3.5 Cyclic(x) to Cyclic(y) Redistribution

For a general cyclic(x) to cyclic(y) redistribution, there is no direct algebraic formula to calculate the set of elements to send to a destination processor and the local addresses of these elements at the destination [53]. Several complex schemes have been proposed for performing this redistribution [110, 111, 23, 99]. Since redistribution has to be done at runtime, a simple and efficient algorithm with minimum runtime overhead is necessary. We propose a new method for performing the general cyclic(x) to cyclic(y) redistribution, which has low runtime overhead. We first consider two special cases where x is a multiple of y, or y is a multiple of x, and develop optimized algorithms for these two special cases. For the general case where there is no particular relation between x and y, we have developed two algorithms called the GCD Method and the LCM Method, which make use of the optimized algorithms developed for the above two special cases.

3.5.1 Special case $x = k\,y$

We first consider the special case where x is a multiple of y. Let $x = k\,y$. An example of this is given in Figure 3.8.

Theorem 3.2 *In a cyclic(x) to cyclic(y) redistribution where $x = k\,y$, if $k < P$, each processor communicates with k or $k-1$ processors. If $k \geq P$, each processor communicates with all other processors.*

Proof: Assume $k < P$. Since $x = k\,y$, each block of size x is divided into k sub-blocks of size y and distributed cyclically. Consider any processor p_i. Assume that it has to send its first sub-block of size y to processor p_j. Then the remaining $k-1$ sub-blocks of the first block are sent to the next $k-1$ processors in order. The next $k(P-1)$ sub-blocks of the global array are located in the other $P-1$ processors. This results in a total of $k\,P$ sub-blocks. Hence the $(k+1)^{th}$ sub-block of size y in p_i is also sent to p_j. Thus all sub-blocks from p_i are sent to k processors starting from

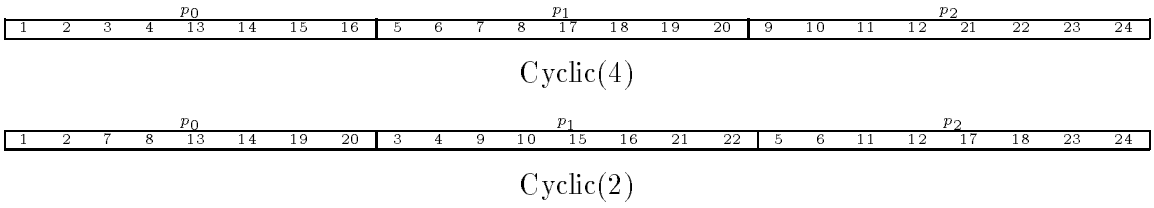

Figure 3.8: Cyclic(4) to Cyclic(2) Redistribution

p_j. One of these processors may be p_i itself, in which case p_i has to send data to $k-1$ processors. For the receive phase, consider the first $k\,P$ sub-blocks of size y in the global array corresponding to the first P blocks of size x. Let us number these $k\,P$ sub-blocks from 0 to $k\,P-1$. Out of these, the sub-blocks that are mapped to processor p_i in the new distribution are numbered p_i to $P(k-1)+p_i$ with stride P. These sub-blocks come from $\frac{\{P(k-1)+p_i\}-p_i}{P}+1 = k$ processors. One of these processors might be p_i itself, in which case p_i receives data from $k-1$ processors.

If $k \geq P$, each block of size x has to be divided into k sub-blocks and distributed cyclically, where the number of sub-blocks is greater than or equal to the number of processors. So clearly each processor has to send to and receive from all other processors (all-to-all communication). □

The algorithm for cyclic(x) to cyclic(y) redistribution, where $x = k\,y$ is given in Figure 3.9. We call this the KY_TO_Y algorithm. In the send phase, each processor p calculates the destination processor p_d of the first element of its local array as $p_d = MOD(k\,p, P)$. The first y elements have to be sent to p_d, the next y to $MOD(p_d + 1, P)$, the next to $MOD(p_d + 2, P)$ and so on till the end of the first block of size x. The next k sub-blocks of size y have to be sent to the same set of k processors starting from p_d. The sequence of destination processors can be stored and need not be calculated for each block of size x. In the receive phase there are two cases depending on the value of k :-

1. $(k \leq P)$ and $(MOD(P,k) = 0)$: In this case, each processor p calculates the source processor of the first block of size y of its local array as $p_s = p/k$. The next block of size y will come from processor $MOD(p_s + P/k, P)$, the next from $MOD(p_s + 2(P/k), P)$ and so on till the first k blocks. The above sequence of processors is repeated for the remaining sets of k blocks of size x and hence can be stored and used. If the Synchronous Method is used for receiving data, the local array needs to be scanned only once and the i^{th} block, $0 \leq i \leq \lceil L/y \rceil - 1$, of size y of the local array will be read from the packet received from processor $MOD(p_s + i(P/k), P)$. If the Asynchronous Method is used, the first block from the packet received from some processor p_i will be stored starting at the location calculated above. The remaining blocks will be stored with stride x.

2. If k does not satisfy the above condition, it is necessary to calculate the source processor of the first element $(j = i\,y + 1)$ of each block of size y, $0 \leq i \leq \lceil L/y \rceil - 1$, of the local array as $p_s = MOD[(i\,P + p)/k, P]$. The block is read from the packet received from p_s. The sequence of processors does not repeat itself and hence cannot be stored. In this case, the Synchronous Method is used.

We have tested the performance of the KY_TO_Y Algorithm using both Synchronous and Asynchronous Methods on the Intel Touchstone Delta. Figure 3.10 compares the performance of the Synchronous and Asynchronous Methods for a cyclic(4) to cyclic(2) redistribution of a global array of 1M integers for different number of processors. We observe that the Asynchronous Method performs better than the Synchronous Method, even though in this case each processor communicates with at most two other processors. This is because the Asynchronous Method overlaps computation and communication and thus reduces processor idle time. The better cache utilization of the Synchronous Method does not improve its overall performance. Figure 3.11 shows the performance of the KY_TO_Y Algorithm for a cyclic(4) to cyclic(2) redistribution on 64 processors for different array sizes. For small arrays, the difference in performance between the Synchronous and Asynchronous Methods is small, because of the small data sets. For large arrays, the difference is significant because

Send Phase

1. The destination processor (p_d) of the first element of the local array is $p_d = MOD(k\,p, P)$.
2. For each block of size x in the local array
3. For $i = 0$ to $k - 1$
4. The destination processor of elements $(i\,y + 1)$ to $(i + 1)y$ of this block of size x is $MOD(p_d + i, P)$.

Receive Phase

1. If $(k \leq P)$ and $(MOD(P, k) = 0)$ then
2. The source processor (p_s) of the first element of the local array is $p_s = p/k$.

Synchronous Method:

3. Receive data from all processors.
4. For $j = 1$ to $\lceil L/x \rceil$ do
5. For $i = 0$ to $k - 1$ do
6. Read the next block of size y from the data received from processor $MOD(p_s + i(P/k), P)$.

Asynchronous Method:

3. The i^{th} block of size y, $0 \leq i \leq k - 1$, is to be received from processor $MOD(p_s + i(P/k), P)$.
4. For $i = 0$ to $k - 1$ do
5. Receive data from any processor p_i.
6. Place the first block of size y in the local array starting from the location calculated above, and the other blocks with stride x.
7. Else
8. Receive data from all processors.
9. For $i = 0$ to $\lceil L/y \rceil - 1$ do
10. The source processor (p_s) of the first element $(j = i\,y + 1)$ of this block of size y is $p_s = MOD[(i\,P + p)/k, P]$
11. Read this block of size y from the data received from p_s.

Figure 3.9: KY_TO_Y Algorithm for Cyclic(x) to Cyclic(y) Redist., where $x = k\,y$

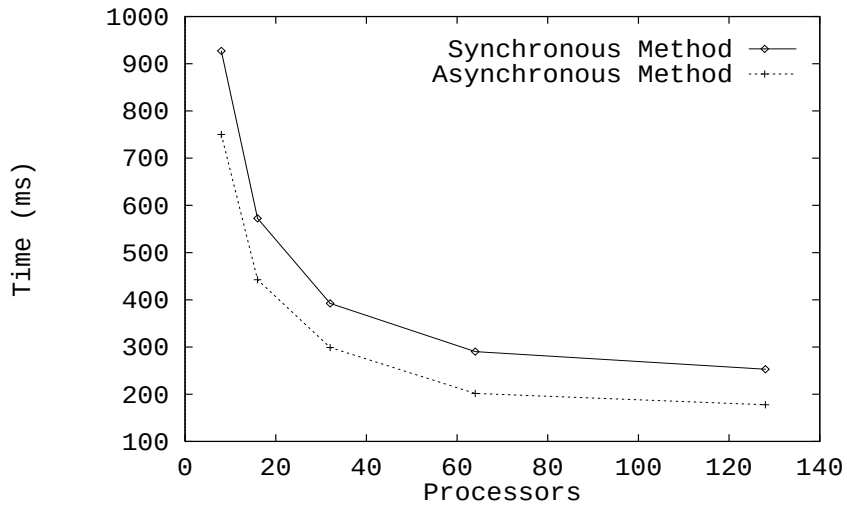

Figure 3.10: KY_TO_Y Algorithm, cyclic(4) to cyclic(2), array size 1M integers

of the higher processor idle time in the Synchronous Method.

3.5.2 Special case $y = k\,x$

We now consider the special case where y is a multiple of x. Let $y = k\,x$. This is essentially the reverse of the case where $x = k\,y$.

Theorem 3.3 *In a cyclic(x) to cyclic(y) redistribution where $y = k\,x$, if $k < P$, each processor sends data to k or $k-1$ processors and receives data from k or $k-1$ processors. If $k \geq P$, each processor has to communicate with all other processors (all-to-all communication).*

Proof: Assume $k < P$. Consider the first $k\,P$ sub-blocks of size x in the global array corresponding to the first P sub-blocks of size y. Let us number these $k\,P$ sub-blocks from 0 to $k\,P-1$. Out of these, the sub-blocks that are located in processor p_i are numbered from p_i to $P(k-1)-1+p_i$ with stride P. In the new distribution, these sub-blocks will be mapped to $\frac{\{P(k-1)-1+p_i\}-p_i}{P}+1 = k$ processors. One of these processors

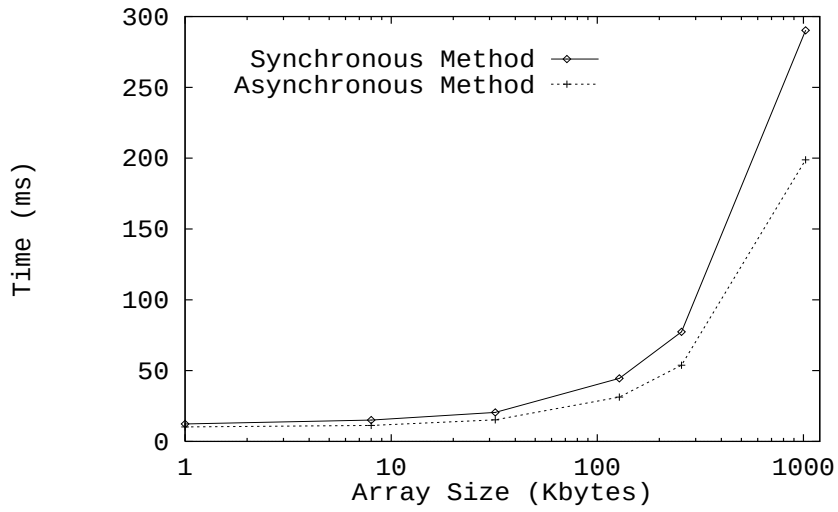

Figure 3.11: KY_TO_Y Algorithm, cyclic(4) to cyclic(2), 64 processors

might be p_i itself, in which case p_i sends data to $k-1$ processors. In the receive phase, since $y = k\,x$, each block of size y in the new distribution consists of k sub-blocks of size x which will come from k processors. Consider any processor p_i. Assume that it receives its first sub-block of size x from processor p_j. Then the remaining $k-1$ sub-blocks of the first block are received from the next $k-1$ processors in order. The other $P-1$ processors receive the next $k(P-1)$ sub-blocks of the global array. This results in a total of $k\,P$ sub-blocks. Hence the next sub-block in p_i, which is the first sub-block of the next block of size y, is also received from p_j. Thus all sub-blocks from p_i are received from k processors starting from p_j. One of these processors may be p_i itself in which case p_i receives data from $k-1$ processors.

If $k \geq P$, each block of size y will consist of k sub-blocks of size x, where the number of sub-blocks is greater than or equal to the number of processors. So clearly each processor has to send to and receive from all other processors (all-to-all communication). □

The algorithm for cyclic(x) to cyclic(y) redistribution, where $y = k\,x$, is given in Figure 3.12. We call this the X_TO_KX Algorithm. In the send phase, there are two cases depending on the value of k :-

1. $(k \leq P)$ and $(MOD(P,k) = 0)$: In this case, each processor p calculates the destination processor of the first block of size x of its local array as $p_d = p/k$. The next block of size x has to be sent to processor $MOD(p_d + P/k, P)$, the next to $MOD(p_d + 2(P/k), P)$ and so on till the first k blocks. The above sequence of processors is repeated for the remaining sets of k blocks of size x, and hence need not be calculated again.

2. If k does not satisfy the above condition, it is necessary to individually calculate the destination processor of each block i of size x, $0 \leq i \leq \lceil L/x \rceil - 1$, as $p_d = MOD[(i\,P + p)/k, P]$.

In the receive phase, each processor p calculates the source processor of the first element of its local array as $p_s = MOD[k\,p, P]$. As in the KY_TO_Y algorithm, the receive phase can be implemented using either the Synchronous Method or the Asynchronous Method. If the Synchronous Method is used to receive data, for each block of size y of the local array, the k sub-blocks of size x are read from the packets received from the k processors starting from p_s in order of processor number. If the Asynchronous Method is used, we know that the i^{th} block of size x of the local array, $0 \leq i \leq k-1$, will be received from processor $MOD(p_s + i, P)$. Thus the local index of the first block received from any source processor can be calculated. The remaining blocks have to be stored with stride y.

We have tested the performance of the X_TO_KX Algorithm on the Intel Touchstone Delta for different array sizes and number of processors. Figure 3.13 compares the performance of the Synchronous and Asynchronous Methods for a cyclic(2) to cyclic(4) redistribution of an array of 1M integers for different number of processors. Figure 3.14 compares the performance of the two methods for different array sizes on 64 processors. The results are similar to those obtained for the KY_TO_Y Algorithm.

Send Phase

1. If $(k \leq P)$ and $(MOD(P,k) = 0)$ then
2. The destination processor (p_d) of the first element of the local array is $p_d = p/k$.
3. For $j = 0$ to $\lceil L/y \rceil - 1$
4. For $i = 0$ to $k - 1$
5. The destination processor of the next block of size x of the local array is $MOD(p_d + i(P/k), P)$.
6. Else
7. For $i = 0$ to $\lceil L/x \rceil - 1$
8. The destination processor (p_d) of the first element $(j = i\,x + 1)$ of this block of size x is $p_d = MOD[(i\,P + p)/k, P]$.

Receive Phase

1. The source processor (p_s) of the first element of the local array is element of the local array is $p_s = MOD[k\,p, P]$.

Synchronous Method:
2. Receive data from all processors.
3. For each block of size y in the local array do
4. For $i = 0$ to $k - 1$ do
5. Read elements $(i\,x + 1)$ to $(i + 1)x$ of the current block of size y from the packet received from processor $MOD(p_s + i, P)$.

Asynchronous Method:
2. The i^{th} block of size x, $0 \leq i \leq k - 1$, is to be received from processor $MOD(p_s + i, P)$.
3. For $i = 0$ to $k - 1$ do
4. Receive data from any processor p_i.
5. Place the first block of size x in the local array starting from the location calculated above, and the other blocks with stride y.

Figure 3.12: X_TO_KX Algorithm for Cyclic(x) to Cyclic(y) Redist., where $y = k\,x$

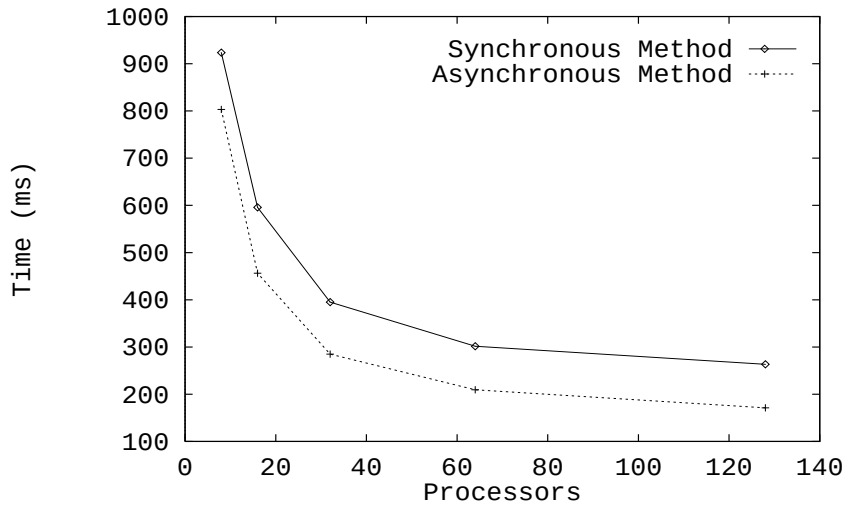

Figure 3.13: X_TO_KX Algorithm, cyclic(2) to cyclic(4), array size 1M integers

The Asynchronous Method is found to perform better in all cases.

3.5.3 General Case

Let us consider the general case of a cyclic(x) to cyclic(y) redistribution in which there is no particular relation between x and y (Figure 3.15). One algorithm for doing this is to explicitly calculate the destination and source processor of each element of the local array, using the formulae given in Table 3.1. We call this General Algorithm and is described below.

General Algorithm

In the send phase, the destination processor of each element of the local array can be determined by first calculating its global index based on the current distribution and then the destination processor based on the new distribution. These two calculations can be combined into a single expression to give the destination processor of element i of the local array as $p_d = MOD[\{MOD(i-1,x)+(P((i-1)/x)+p)x+y\}/y-1,P]$. Similarly in the receive phase, the source processor of each element of the local array

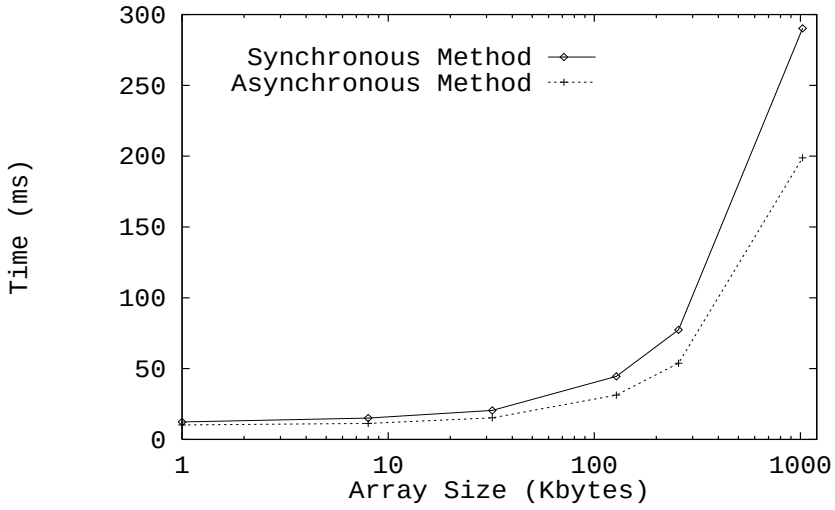

Figure 3.14: X_TO_KX Algorithm, cyclic(2) to cyclic(4), 64 processors

	p_0						p_1						p_2						
	1	2	3	10	11	12	4	5	6	13	14	15	7	8	9	16	17	18	Cyclic(3)
	p_0						p_1						p_2						
	1	2	7	8	13	14	3	4	9	10	15	16	5	6	11	12	17	18	Cyclic(2)

Figure 3.15: General Cyclic(x) to Cyclic(y) Redistribution

can be determined by first calculating its global index based on the new distribution and then the source processor based on the old distribution. These two calculations can be combined into a single expression to give the source processor of element i of the local array as $p_s = MOD[\{MOD(i-1,y) + (P((i-1)/y) + p)y + x\}/x - 1, P]$.

The drawback of this algorithm is that calculations are needed individually for all elements of the array. We propose two algorithms called the GCD Method and the LCM Method, which make use of the optimized KY_TO_Y and X_TO_KX algorithms and hence require a lot less calculations.

GCD Method	**LCM Method**
1. Let $m = GCD(x, y)$.	1. Let $m = LCM(x, y)$.
2. Redistribute from cyclic(x) to cyclic(m) using the KY_TO_Y Algorithm.	2. Redistribute from cyclic(x) to cyclic(m) using the X_TO_KX Algorithm.
3. Redistribute from cyclic(m) to cyclic(y) using the X_TO_KX Algorithm.	3. Redistribute from cyclic(m) to cyclic(y) using the KY_TO_Y Algorithm.

Figure 3.16: GCD and LCM Methods

GCD Method

This method takes advantage of the fact that we have developed special optimized algorithms for a cyclic(x) to cyclic(y) redistribution when $x = k\,y$ (the KY_TO_Y Algorithm) and $y = k\,x$ (the X_TO_KX Algorithm). In the GCD Method, the redistribution is done as a sequence of two phases — cyclic(x) to cyclic(m) followed by cyclic(m) to cyclic(y), where $m = GCD(x, y)$. Since both x and y are multiples of m, the KY_TO_Y Algorithm can be used for the cyclic(x) to cyclic(m) phase and the X_TO_KX Algorithm can be used for the cyclic(m) to cyclic(y) phase. This is described in Figure 3.16. The GCD Method involves the cost of having to do two separate redistributions. For small arrays, the increased communication may outweigh the savings in computation, but for large arrays in some cases, the performance is better than that of the General Method. This can be observed from Figure 3.17 which shows the performance of a cyclic(15) to cyclic(10) redistribution, for an array of size 1M integers on the Delta. We see that for small number of processors, the GCD Method performs considerably better than the General Method because of the saving in the amount of computation per processor. Since the size of the global array is kept constant, as the number of processors is increased, the size of the local array in each processor becomes smaller and each processor spends less time on address calculation. Hence the improvement of the GCD Method over the General Method is also small.

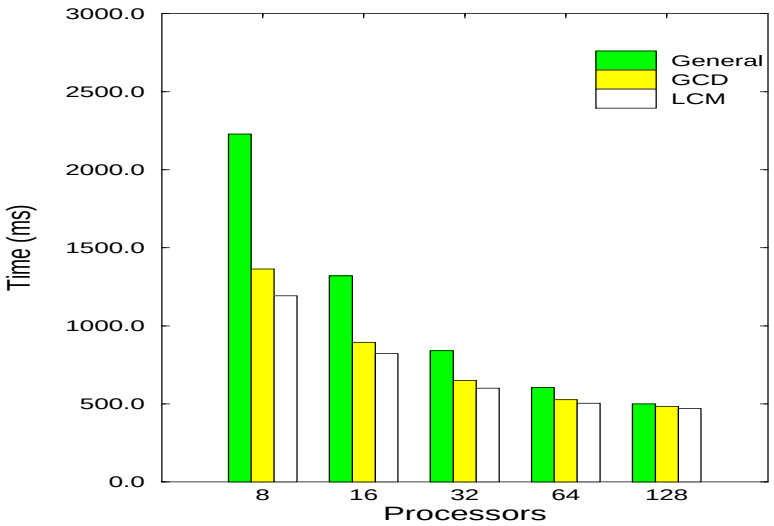

Figure 3.17: GCD, LCM and General Methods; cyclic(15) to cyclic(10); 1M array

One disadvantage of the GCD Method is that in the intermediate cyclic(m) distribution, the block size m is smaller than both x and y. In the KY_TO_Y and X_TO_KX algorithms, all the address and processor calculations are done for blocks of size x or y. Since m is the GCD of x and y, m can even be equal to 1 in some cases. When $m = 1$, calculations have to be done for each element, which is no better than in the General Method. In this case the General Method performs better than the GCD Method. Figure 3.18 shows an example of cyclic(11) to cyclic(3) redistribution on the Delta for an array of size 1M integers. Since the GCD of 11 and 3 is 1, we find that the General Method always performs better than the GCD Method.

LCM Method

The key to getting good performance in this two-phase approach for redistribution is to have a large value for m. One way of ensuring that m is always large is by choosing m as the LCM of x and y. Since m is a multiple of both x and y, the X_TO_KX Algorithm can be used for the cyclic(x) to cyclic(m) phase and the KY_TO_Y algorithm can be used for the cyclic(m) to cyclic(y) phase. This is described in Figure 3.16.

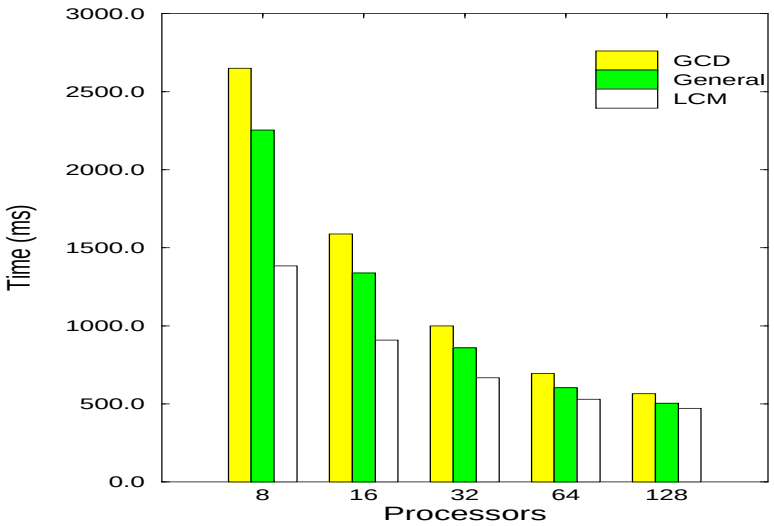

Figure 3.18: GCD, LCM and General Methods; cyclic(11) to cyclic(3); 1M array

Also, since m is larger than x and y, all calculations are done for this larger block size. This results in fewer calculations than in the GCD and General Methods. Figures 3.17 and 3.18 compare the performance of the LCM, GCD and General Methods for an array of 1M integers on different number of processors. We observe that the LCM Method performs better in all cases. Figure 3.19 compares the performance of the LCM and General Methods for a cyclic(11) to cyclic(3) redistribution keeping the number of processors fixed at 64 and varying the array size. We observe that for small arrays, the General Method performs better because it has less communication, but for large arrays the LCM method performs better because the saving in computation is higher than the increase in communication.

3.6 Redistribution of Multidimensional Arrays

The redistribution of two and higher dimensional arrays can be classified into two types :-

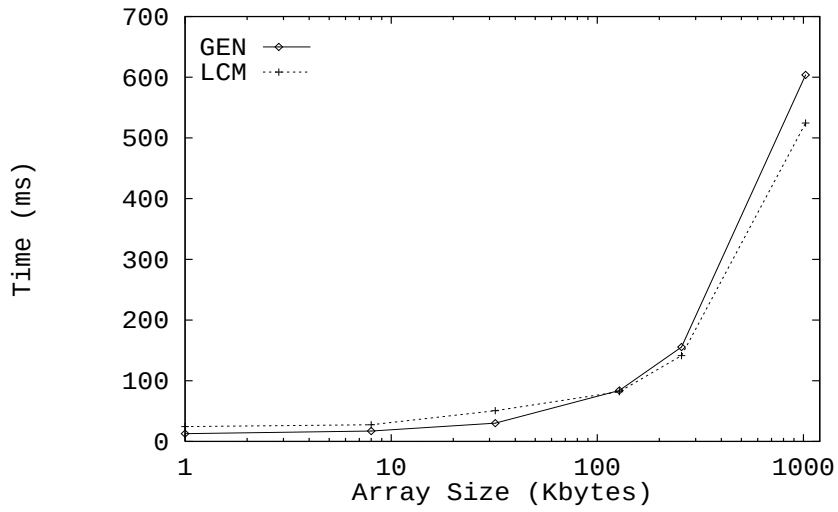

Figure 3.19: LCM v/s General Methods, cyclic(11) to cyclic(3), 64 processors

1. *Shape Retaining*: The shape of the local array remains unchanged after the redistribution, eg. (block,block) to (cyclic,cyclic).

2. *Shape Changing*: The shape of the local array changes after the redistribution, eg. (block,*) to (*,block) where '*' indicates that the corresponding dimension is collapsed. This type of redistribution is used for example in 2D FFT and the ADI method for solving multidimensional partial differential equations.

The *shape retaining* and *shape changing* redistributions are quite different from each other and require different algorithms.

3.6.1 Shape Retaining Redistribution

A shape retaining redistribution may involve redistribution in only one dimension {eg. (block,block) to (cyclic,block)} or more than one dimension {eg. (block,block) to (cyclic,cyclic)}. If the redistribution is only along one dimension, it is similar to the redistribution of one-dimensional arrays and the same algorithms described earlier can be used. If both dimensions have to be redistributed, it can either be done directly

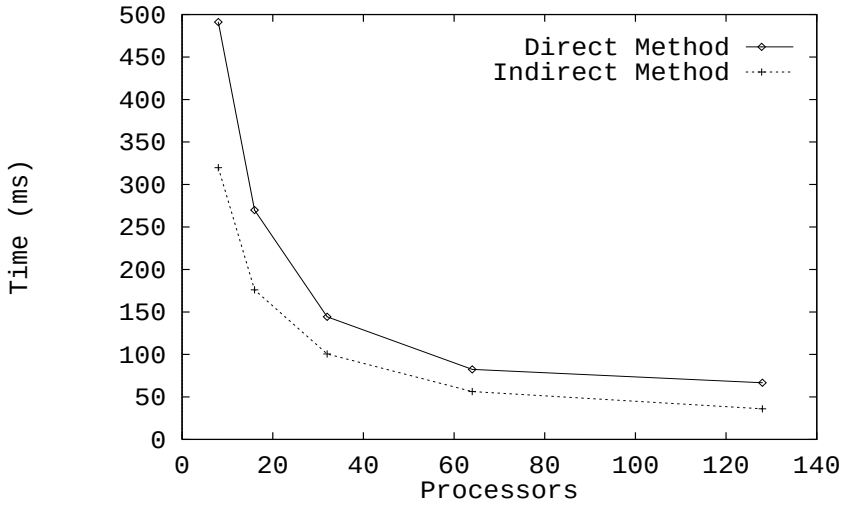

Figure 3.20: (block,block) to (cyclic,cyclic) Redistribution, 1K $\times$ 1K array

or indirectly as a series of one-dimensional redistributions. In the Indirect Method, the array is redistributed separately along each dimension. For example, if an array has to be redistributed from (block,block) to (cyclic,cyclic), it is first redistributed to (block,cyclic) and then to (cyclic,cyclic). This method has the advantage that all the optimizations developed for one-dimensional arrays in the previous sections can be easily extended to multidimensional arrays. The order in which the dimensions are redistributed does not affect the performance. This is because the order of dimensions chosen only results in a different set of data values being communicated, and does not affect the amount or type of communication. So, one could also do the above redistribution as (block,block) to (cyclic,block) to (cyclic,cyclic).

In the Direct Method, data is sent directly to the destination processor corresponding to the new distribution. Hence the optimized algorithms developed for the one-dimensional case cannot be used. This method requires different algorithms for different number of dimensions and different types of redistributions, and these algorithms cannot be optimized much. However, data needs to be communicated

only once in the Direct Method. Figure 3.20 compares the performance of the Direct and Indirect Methods for redistributing an array of size 1K $\times$ 1K integers from (block,block) to (cyclic,cyclic) on the Touchstone Delta. The Indirect Method is found to perform much better even though data is communicated twice. This is because the algorithms for one-dimensional redistribution are highly optimized and a lot of the communication during each one-dimensional redistribution actually takes place in parallel.

Another interesting observation comes from Figure 3.21 which compares the performance of the Direct and Indirect Methods for a (cyclic(11),cyclic(11)) to (cyclic(3),cyclic(3)) redistribution of a 1K $\times$ 1K array. The indirect redistribution is done in two ways — using the General Method and the LCM Method for each cyclic(11) to cyclic(3) redistribution. We find that the General Method performs better than the LCM Method. This is because in the General Method, for each one-dimensional redistribution, the destination and source processors need to be calculated for each row (or column) of the array and the entire row (or column) has to be communicated. In the LCM Method, this communication needs to be done twice. Since the amount of communication is much higher than the amount of computation, the General Method performs better. In the Direct Method, destination and source processor calculations have to be done for each element of the local array. If the local array size is $L \times L$, L^2 calculations have to be done. In the Indirect Method, calculations are done for each column during the column redistribution and for each row during the row redistribution. Hence the number of calculations is only $L + L = 2L$. Therefore, the Direct Method performs considerably worse than any of the Indirect Methods in this case.

3.6.2 Shape Changing Redistribution

This type of redistribution occurs when at least one dimension of the array is collapsed before or after the redistribution. Consider the redistribution from (X,*) to (*,Y) or vice-versa, where X and Y can be either block, cyclic or cyclic(m). This is basically a

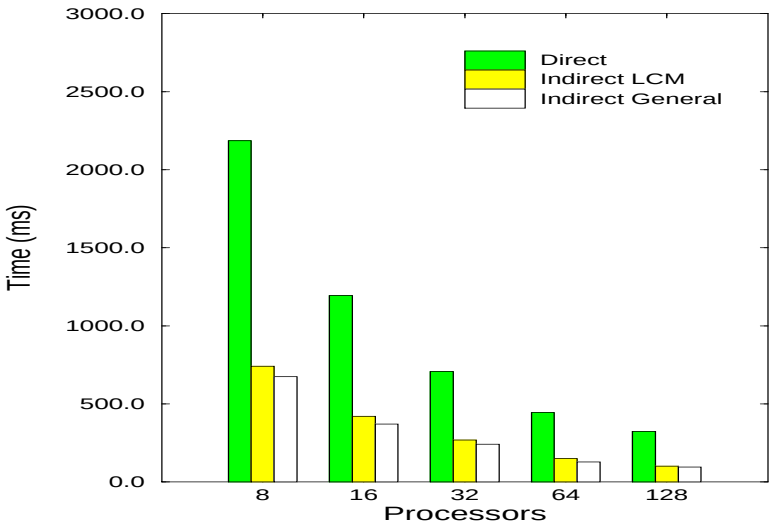

Figure 3.21: (cyclic(11),cyclic(11)) to (cyclic(3),cyclic(3)) Redist., 1K $\times$ 1K array

collapsed to distributed type of redistribution in one of the dimensions, which is done as follows. Each processor divides the local array into packets along the collapsed dimension, depending on the type of the new distribution Y. The processors then exchange packets with other processors. At the receiving end, packets are placed in the local array in order of source processor number. Data from the received packets may have to be placed in the local array either contiguously or with a stride, depending on the type of distributions X and Y.

The other type of redistribution involving change of shape of the local array is of the type (X,*) or (*,X) to (Y,Z), or vice versa. That is, in either the source or target distributions, one of the dimensions is collapsed and in the corresponding target or source distributions, both dimensions are distributed. Each case of this type requires a different algorithm and so has to be considered separately. Note that we do not use the Indirect Method for shape changing redistributions.

3.7 Circular Redistribution

Very often, it is necessary to redistribute from a distribution X to a distribution Y and then sometime later in the program it is required to redistribute back from Y to X. We call this $X \rightarrow Y \rightarrow X$ type redistribution as a *circular redistribution* and is denoted as $X \leftrightarrow Y$. The redistribution from X to Y is called a *forward* redistribution and from Y to X is called a *backward* redistribution. For example when the main program calls a subroutine with a different distribution, it is necessary to redistribute from say X to Y. After returning from the subroutine, it is necessary to redistribute back to X to restore the original distribution.

In the case of a circular redistribution, the backward redistribution is essentially the reverse of the forward redistribution. The calculation of the destination processors and addresses done during the forward redistribution can be saved and reused in the backward redistribution. Thus, no calculations need be done during the backward redistribution. This decrease in computation is at the cost of increased memory for saving the information calculated in the forward redistribution.

3.7.1 Circular Block(m) $\leftrightarrow$ Cyclic Redistribution

We first consider a circular block(m)$\leftrightarrow$cyclic redistribution. In the forward block(m)$\rightarrow$cyclic redistribution, each processor calculates the destination processor to which the first element of the local array is to be sent. This information can be saved (A). In the receive phase, each processor calculates how much data is to be received from other processors. This information can also be saved (B). In the backward cyclic$\rightarrow$block(m) redistribution, Information (B) is sufficient for each processor to know how many elements to send to other processors. Information (A) is sufficient for the receiving processor to know where to store incoming data. Thus, no calculations need to be done in the backward redistribution phase. However, this saving in computation is at the cost of increased memory requirements. Information (A) can be stored in a single variable. Information (B) requires an array of length equal to the number of processors.

3.7.2 Circular Cyclic $\leftrightarrow$ Block(m) Redistribution

This is essentially the reverse of a circular block(m) $\leftrightarrow$ cyclic redistribution. In the send phase of the forward cyclic$\rightarrow$block(m) redistribution, each processor stores the set of destination processors and the number of elements sent to them (A). In the receive phase, each processor stores the source processor of the first element of the local array (B). In the backward block(m)$\rightarrow$cyclic redistribution, Information (B) is sufficient to carry out the send phase and Information (A) can be used by the receiving processor to know how many elements are to be received from which processor. Information (B) can be stored in a single variable. Information (A) requires an array of length equal to the number of processors.

3.7.3 Circular Cyclic(x) $\leftrightarrow$ Cyclic(y) Redistribution

Let us first consider the special case of a circular cyclic(x) $\leftrightarrow$ cyclic(y) redistribution where $x = k\,y$. In the forward cyclic(x)$\rightarrow$cyclic(y) redistribution, each processor calculates the sequence of processors to which sub-blocks of size y have to be sent. This information is stored (A). In the receive phase, each processor calculates the source processors for blocks of size y of the local array. This information is also stored (B). In the backward cyclic(y)$\rightarrow$cyclic(x) redistribution where $x = k\,y$, information (B) can be used in the send phase to determine the sequence of processors to send blocks of size y. In the receive phase, information (A) can be used to determine the source processors for blocks of size y of the local array. Thus, no calculations need to be done in the backward redistribution phase. In the forward redistribution phase, since the sequence of processors to send sub-blocks of size y remains the same for every block of size x, information (A) can be stored in an array of size $x/y = k$. Similarly, information (B) also requires an array of size k.

The other special case $y = k\,x$ is similar.

For the general case where there is no particular relation between x and y, if the LCM or GCD Methods are used, information can be stored and reused in the same

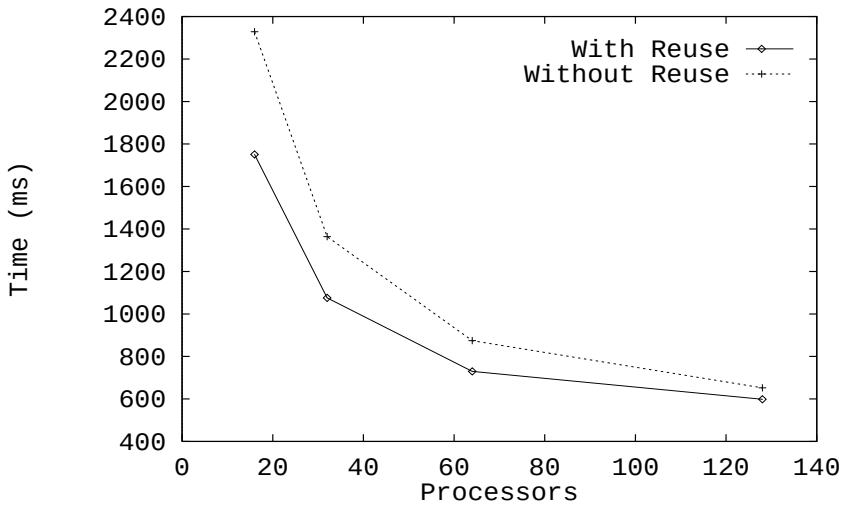

Figure 3.22: Circular cyclic(11) to cyclic(3) Redistribution, array size 1M integers

manner as described above for the $x = k\,y$ case.

If the General Method is used, then in the send phase of the forward redistribution, each processor calculates the destination processor of each element of the local array. This information can be stored (A). In the receive phase, each processor calculates the source processor of each element of the local array. This can also be stored (B). For the backward cyclic(y)→cyclic(x) redistribution, Information (B) gives the set of destination processors and Information (A) gives the set of source processors. This saves a lot of computation in the backward redistribution, since it would otherwise be required to compute the source and destination processors for each individual element. However the amount of storage required is twice the size of the local array. Figure 3.22 compares the performance of a circular cyclic(11) $\leftrightarrow$ cyclic(3) redistribution with and without saving any information in the forward redistribution, for an array of size 1M integers. We observe that there is considerable improvement in performance by reusing the information.

Chapter 4

Runtime Support for All-to-All Collective Communication

Programs on distributed memory parallel computers typically require interprocessor communication. In loosely synchronous parallel programs[46], all processors perform similar operations but on different data sets. Hence it is very likely that a group of processors or even all processors may need to perform communication at the same time. This makes it possible for processors to cooperate with each other to perform communication efficiently, which is known as *collective communication*. Depending on the type of communication required, different communication patterns are possible. These can be generally classified into the following types:-

- *All-to-all*: All processors need to send data to all other processors.
- *All-to-many*: All processors need to send data to some other processors.
- *Many-to-all*: Some processors need to send data to all other processors.
- *Many-to-many*: Some processors need to send data to some other processors.

Efficient algorithms are needed to implement these communication patterns on different network topologies. In this chapter, we consider the all-to-all communication pattern in detail. The other communication patterns, namely all-to-many, many-to-all and many-to-many, have been studied in [100, 127, 106, 107, 108].

The all-to-all communication pattern (also called complete exchange[1]) occurs frequently in many important parallel computing applications such as array redistribution, parallel quicksort, some implementations of two-dimensional Fast Fourier Transform, matrix transpose etc. It is the densest form of communication because all processors need to communicate with all other processors. This can result in severe link contention and degrade performance considerably. Hence, it is necessary to use efficient algorithms in order to get good performance over a wide range of message sizes and number of processors.

We describe several algorithms for all-to-all communication on fat-tree and two-dimensional mesh interconnection topologies. We use the Thinking Machines CM-5 as an example machine with a fat tree topology and the Intel Touchstone Delta as an example machine with a two-dimensional mesh topology. Since these machines have different architectures and communication capabilities, different algorithms are needed to get the best performance on each of them. We present four algorithms for the CM-5 and six algorithms for the Delta. Extensive performance results on the CM-5 and Delta are presented and analyzed. We have also developed analytical models to estimate the performance of the algorithms on the basis of system parameters. The analytical models are validated by comparing with experimental results.

4.1 Architecture and Communication Issues

4.1.1 CM-5

The CM-5 is a scalable distributed memory multiprocessor system which can be scaled up to 16K processors. It supports both SIMD and MIMD programming models. Each node on the CM-5 is a SPARC processor and has four optional vector processors. The CM-5 has two internal networks that support interprocessor communication — the *Control Network* and the *Data Network*. The control network is responsible for communication patterns in which many processors may be involved in the processing

[1] We use the words "all-to-all communication" and "complete exchange" interchangeably in this chapter.

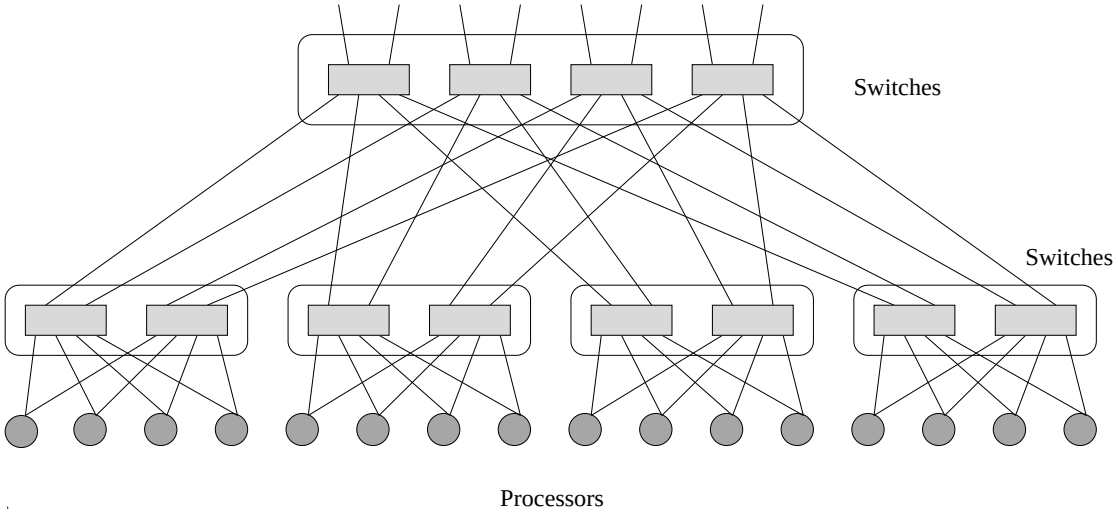

Figure 4.1: CM-5 fat tree

of each datum, such as global reduction operations, parallel prefix operations and processor synchronization. The data network supports point-to-point communication and has a fat tree topology [76, 77] as shown in Figures 4.1 and 4.2. It is actually a 4-ary fat tree or quad tree, where each node has four children. Each internal node of the fat tree is implemented as a set of switches. The number of switches per node depends on where it is in the tree. Nodes at level 1 have two switches. The number of switches per node doubles for each higher level till level 3, from where on it quadruples. Each level 1 or level 2 switch has two parents and four children; switches at higher levels have four parents and four children. The data network has a guaranteed system-wide bandwidth of 5 Mbytes/sec. Messages are divided into packets of size 20 bytes of which 4 bytes is the header. The routing algorithm for the Data Network compares the destination address with the source address to determine how far up the tree the message must travel. The message can then take any path up the tree. This allows the switches to perform load balancing on the fly. Once the message has reached the necessary height in the tree, it must follow a *particular* path down. A detailed discussion of the interprocessor communication overhead on the CM-5 is given in [96, 16, 94].

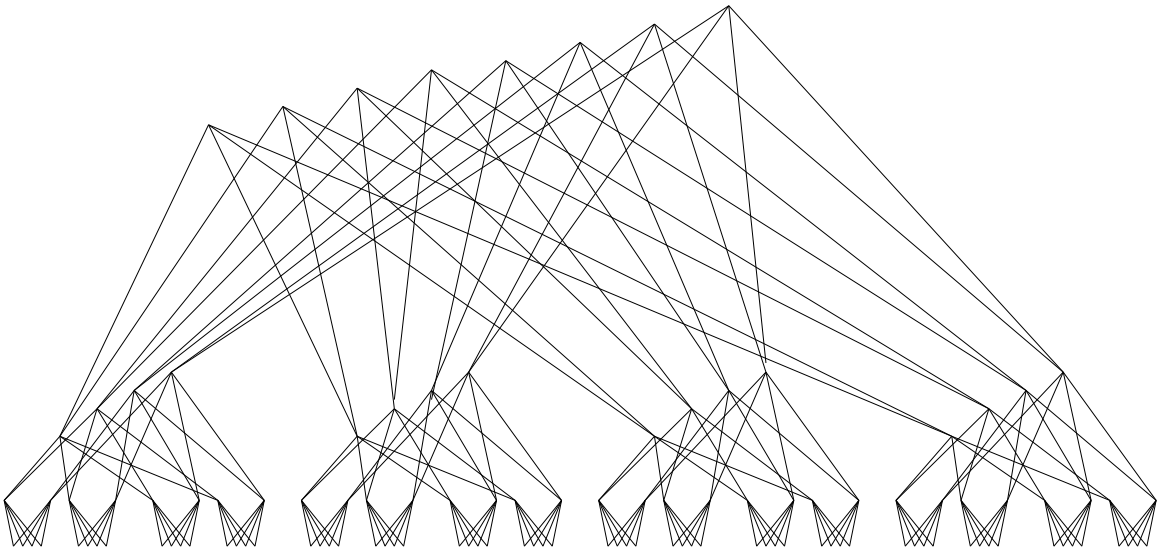

Figure 4.2: CM-5 Data Network with 64 Processing Nodes

4.1.2 Touchstone Delta

The Intel Touchstone Delta is a 16×32 mesh of computational nodes, each of which is an Intel i860/XR microprocessor. The two-dimensional mesh interconnection network has bidirectional links and uses wormhole routing. The links have a maximum bandwidth of 10 Mbytes/sec in each direction. Messages are divided into packets of size 512 bytes of which 32 bytes is the header. It uses deterministic XY routing in which packets are first sent along the X dimension and then along the Y dimension. In other words, at most one turn is allowed, and that turn must be from the X dimension to the Y dimension. For a 2D mesh, the XY routing algorithm is guaranteed to be deadlock free [33].

In wormhole routing, a packet is divided into a number of *flits* (flow control digits) for transmission. The size of a flit is typically the same as the channel width. The header flit of a packet determines the route. As the header advances along the route, the remaining flits follow in a pipeline fashion. If the header flit encounters a channel already in use, it is blocked until the channel becomes available. The flow control within the network blocks the trailing flits and they remain in flit buffers along the

established route. Once a channel has been acquired by a packet, it is reserved for the packet. The channel is released when the last flit has been transmitted on the channel. The pipelined nature of wormhole routing makes the communication latency almost insensitive to path length in the absence of network contention. The network latency for wormhole routing is $(L_f/B)D + L/B$, where L_f is the length of each flit, B is the channel bandwidth, D is the path length, and L is the length of the message [87]. Thus, if $L_f << L$, the path length D will not significantly affect the network latency unless it is very large. Further details of wormhole routing can be found in [87].

4.1.3 Performance Models

Barnett et. al. [3] have proposed algorithms and performance models for global combine operations on a wormhole routed mesh. We use similar models for our all-to-all communication algorithms, which take into account link conflicts and other characteristics of the underlying communication system. The following notations are used in our models :-

α	startup time per message
β_{ex}	transfer time per byte for an exchange with no link conflicts
β_{sr}	transfer time per byte to send to and receive from different processors with no link conflicts
β_{sat}	transfer time per byte on a saturated link
β_s	transfer time per byte for a single send-recv with no link conflicts
L	number of bytes to be exchanged per processor pair
$f(i)$	maximum number of messages contending for a saturated link at step i
P	total number of processors

The time taken for an exchange operation may be different from the time to send to and receive from different processors, because in the latter case the incoming and outgoing messages may traverse links with different amount of contention. Hence, we use β_{ex} or β_{sr} depending on the algorithm. We assume that the time taken is independent of distance, a property of both CM-5 and Delta [95, 3]. Thus, the time

required for an exchange step i is given by

$$T = \alpha + L\ max(\beta_{ex}, f(i)\beta_{sat})$$

We assume that conflicting messages share the bandwidth of a network link. The network may have excess bandwidth, enabling multiple messages to traverse a link in the same direction without conflict. In other words, $\beta_{sat} < \beta_{ex},\ \beta_{sat} < \beta_{sr},\ \beta_{sat} < \beta_s$.

Also, in any expression in this chapter, the division of two integer variables should be considered as *integer division*, ie. $5/2 = 2$.

4.2 All-to-All Communication on a Fat Tree

In this section, we describe four algorithms for all-to-all communication on the fat tree topology of the CM-5.

4.2.1 Linear Exchange (LEX)

The Linear Exchange algorithm is the simplest of the four algorithms. In step i, $0 \leq i < P$, processor i receives messages from every processor except itself. The algorithm clearly requires P steps. Since at every step one processor receives from all other processors, there is a lot of link contention. At step i, every processor sends data to processor i. Processor i has two links to its parent node and $P-1$ processors simultaneously need to use these links. Hence, the maximum number of messages contending for a link at any step is $\lceil \frac{P-1}{2} \rceil$. The time taken for any step i is

$$T(i) = \alpha + L\ max(\beta_s, \lceil \frac{P-1}{2} \rceil \beta_{sat})$$

The cost of LEX is obtained by summing over all steps of the algorithm :

$$T_{LEX} = \sum_{i=1}^{P-1} [\alpha + L\ max(\beta_s, \lceil \frac{P-1}{2} \rceil \beta_{sat})] = \alpha(P-1) + L(P-1) max(\beta_s, \lceil \frac{P-1}{2} \rceil \beta_{sat})$$

```
do i=1, P − 1
      destination = xor(mynumber, i)
      Exchange with destination
end do
```

Figure 4.3: Pairwise Exchange (PEX)

Table 4.1: Communication Schedule for PEX on 8 Processors

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
$0 \leftrightarrow 1$	$0 \leftrightarrow 2$	$0 \leftrightarrow 3$	$0 \leftrightarrow 4$	$0 \leftrightarrow 5$	$0 \leftrightarrow 6$	$0 \leftrightarrow 7$
$2 \leftrightarrow 3$	$1 \leftrightarrow 3$	$1 \leftrightarrow 2$	$1 \leftrightarrow 5$	$1 \leftrightarrow 4$	$1 \leftrightarrow 7$	$1 \leftrightarrow 6$
$4 \leftrightarrow 5$	$4 \leftrightarrow 6$	$4 \leftrightarrow 7$	$2 \leftrightarrow 6$	$2 \leftrightarrow 7$	$2 \leftrightarrow 4$	$2 \leftrightarrow 5$
$6 \leftrightarrow 7$	$5 \leftrightarrow 7$	$5 \leftrightarrow 6$	$3 \leftrightarrow 7$	$3 \leftrightarrow 6$	$3 \leftrightarrow 5$	$3 \leftrightarrow 4$

4.2.2 Pairwise Exchange (PEX)

We consider the Pairwise Exchange (PEX) algorithm which has been shown to be the best algorithm for a hypercube network, on which it guarantees no link contention at any step [6, 103]. The algorithm is described in Figure 4.3. It requires $P-1$ steps and the communication schedule is as follows. At step i, $1 \leq i \leq P-1$, each processor exchanges a message with the processor determined by taking the exclusive-or of its processor number with i. Therefore, this algorithm has the property that the entire communication pattern is decomposed into a sequence of pairwise exchanges. The communication schedule of the pairwise exchange algorithm for 8 processors is given in Table 4.1. The entry $i \leftrightarrow j$ in the table indicates that processors i and j exchange messages.

The time taken by this algorithm on the CM-5 can be estimated as follows. When a processor has to communicate with another processor in its cluster of 4^k processors, $k \geq 1$, the message has to travel a maximum of k levels up the tree. When a cluster

of 16 processors exchange among themselves, the messages have to travel either 1 or 2 levels up the tree depending on the source and destination. There are 32 links from level 0 to level 1 and 16 links from level 1 to 2 for this cluster, enough for the 16 processors to exchange among themselves without contention. However, when processors in a cluster of 16 need to exchange with processors in another cluster of 16, there are only 8 links from a level 2 node to a level 3 node, which results in 16 processors contending for 8 links. A similar bottleneck exists at higher levels. For example, a level 3 node has 32 links upwards and downwards, and 64 processors in its subtree.

The communication schedule of PEX is such that in the first 15 steps, processors exchange completely within a cluster of 16 processors and after that they exchange across clusters. Hence, in the first 15 steps there is no contention. From step 16 onwards, there are a maximum of 2 messages contending for a link. The time taken for step i is given by

$$T(i) = \begin{cases} \alpha + L\,\beta_{ex} & \text{for } 1 \leq i \leq 15 \\ \alpha + L\,max(\beta_{ex}, 2\beta_{sat}) & \text{for } i > 15 \end{cases}$$

The time for the entire PEX algorithm can be obtained by summing over all steps

$$T_{PEX} = \sum_{i=1}^{15}[\alpha + L\,\beta_{ex}] + \sum_{i=16}^{P-1}[\alpha + L\,max(\beta_{ex}, 2\beta_{sat})]$$

which can be simplified to

$$T_{PEX} = (P-1)\alpha + L\,[15\beta_{ex} + (P-16)\,max(\beta_{ex}, 2\beta_{sat})]$$

For a complete exchange on 16 processors, this algorithm has no contention.

4.2.3 Recursive Exchange (REX)

The Recursive Exchange algorithm is described in Figure 4.4. The number of processors is halved in each step and each processor exchanges data with the corresponding

Table 4.2: Communication Schedule for REX on 8 processors

Step 1	Step 2	Step 3
$0 \leftrightarrow 4$	$0 \leftrightarrow 2$	$0 \leftrightarrow 1$
$1 \leftrightarrow 5$	$1 \leftrightarrow 3$	$2 \leftrightarrow 3$
$2 \leftrightarrow 6$	$4 \leftrightarrow 6$	$4 \leftrightarrow 5$
$3 \leftrightarrow 7$	$5 \leftrightarrow 7$	$6 \leftrightarrow 7$

processor in the other half. A processor sends all the data intended for all processors in the other half to only one processor in that half, which forwards that data to the remaining processors in a later step. The number of steps required is $\lg P$ and each message is of size $L \times P/2$. The communication schedule for REX on 8 processors is given in Table 4.2. Although this algorithm takes less number of steps than LEX and PEX, the amount of data transmitted in each step is much higher. Since it is a store-and-forward type algorithm, each step incurs the additional overhead of reshuffling data

In step i, $1 \leq i \leq \lg P$ each processor j exchanges with processor $j \pm \frac{P}{2^i}$. Communication always takes place either entirely within a cluster of 16 processors or entirely across clusters. In steps 1 to $\lg P - 4$, communication takes place across clusters, so the maximum number of messages contending for a link is 2. In steps $\lg P - 3$ to $\lg P$, communication takes place within a cluster of 16 processors, so there is no contention. Hence, the time taken by REX is

$$T_{REX} = \sum_{i=1}^{\lg P - 4} [\alpha + L\,\frac{P}{2}\,max(\beta_{ex}, 2\beta_{sat})] + \sum_{i=\lg P - 3}^{\lg P} [\alpha + L\,\frac{P}{2}\,\beta_{ex}]$$

which can be simplified to

$$T_{REX} = \alpha \lg P + L\,\frac{P}{2}\,[4\beta_{ex} + (\lg P - 4)\,max(\beta_{ex}, 2\beta_{sat})]$$

```
size = P
pos = 0
bytes = L × P/2
do i=1, lg P
      size = size/2
      if (mynumber < (size + pos)) then
            dest = mynumber + size
      else
            dest = mynumber – size
            pos = pos + size
      end if
      exchange message of size "bytes" with dest
end do
```

Figure 4.4: Recursive Exchange (REX) on CM-5

4.2.4 Balanced Exchange (BEX)

In the PEX algorithm, the communication schedule is such that all processors in a cluster first exchange completely with each other and then exchange with processors in other clusters. In other words, all the communication is either entirely within the cluster or entirely across clusters. As explained above, this gives rise to contention from step 16 onwards. An improvement in performance can be expected if there is a balance of local and long distance communication at every step, which will reduce contention in step 16 onwards. The Balanced Exchange (BEX) algorithm provides such a schedule. BEX is a simple modification of PEX and is described in Figure 4.5. For the purpose of determining the communicating pairs of processors, we define a mapping between the physical number of a processor and its virtual number as

`virtual no = MOD(physical no + 1,` P`)`

Balanced exchange consists of using the pairwise exchange algorithm with this mapping and the virtual processor numbers. The communication schedule for BEX is given in Table 4.3.

```
virtual = MOD(mynumber + 1, P)
do j=1, P − 1
      dest = xor(virtual, j) − 1
      if (dest == −1)
            dest = P − 1
      end if
      exchange with dest
end do
```

Figure 4.5: Balanced Exchange (BEX)

The BEX algorithm has the property that in steps 0 to $P/2 - 1$, two processors in each cluster of size $P/2$ communicate across clusters while the rest communicate within the cluster. In steps $P/2$ to $P - 1$, two processors in each cluster of size $P/2$ communicate within the cluster, while the other processors communicate across clusters. In steps 0 to 15, there is no contention as in the PEX algorithm. In step $i > 15$, instead of $2^{\lfloor \lg i \rfloor}$ processors contending for $2^{\lfloor \lg i \rfloor - 1}$ links, there are $(2^{\lfloor \lg i \rfloor} - 2)$ processors contending for $2^{\lfloor \lg i \rfloor - 1}$ links. Hence the maximum contention for any link at step $i > 15$ is $\frac{2^{\lfloor \lg i \rfloor} - 2}{2^{(\lfloor \lg i \rfloor - 1)}}$ on an average. The total time taken by BEX is

$$T_{BEX} = \sum_{i=1}^{15} [\alpha + L\,\beta_{ex}] + \sum_{i=16}^{P-1} [\alpha + L\,max(\beta_{ex}, \frac{2^{\lfloor \lg i \rfloor} - 2}{2^{(\lfloor \lg i \rfloor - 1)}} \beta_{sat})]$$

which can be simplified to

$$T_{BEX} = (P-1)\alpha + L\,[15\beta_{ex} + (P-16)\,max(\beta_{ex}, \frac{2^{\lfloor \lg i \rfloor} - 2}{2^{(\lfloor \lg i \rfloor - 1)}} \beta_{sat})]$$

4.2.5 Performance of Algorithms on the CM-5

We have implemented the above algorithms on the CM-5 using the message passing library CMMD Version 3.0 Beta. Figure 4.6 compares the communication time of

Table 4.3: Communication Schedule for BEX on 8 processors

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
$0 \leftrightarrow 7$	$0 \leftrightarrow 2$	$0 \leftrightarrow 1$	$0 \leftrightarrow 4$	$0 \leftrightarrow 3$	$0 \leftrightarrow 6$	$0 \leftrightarrow 5$
$1 \leftrightarrow 2$	$1 \leftrightarrow 7$	$2 \leftrightarrow 7$	$1 \leftrightarrow 5$	$1 \leftrightarrow 6$	$1 \leftrightarrow 3$	$1 \leftrightarrow 4$
$3 \leftrightarrow 4$	$3 \leftrightarrow 5$	$3 \leftrightarrow 6$	$2 \leftrightarrow 6$	$2 \leftrightarrow 5$	$2 \leftrightarrow 4$	$2 \leftrightarrow 3$
$5 \leftrightarrow 6$	$4 \leftrightarrow 6$	$4 \leftrightarrow 5$	$3 \leftrightarrow 7$	$4 \leftrightarrow 7$	$5 \leftrightarrow 7$	$6 \leftrightarrow 7$

the four algorithms on a 32 node CM-5 with message size varied between 0 and 2048 bytes. As expected, LEX performs much worse than the other algorithms, so we do not consider it any further. For small message sizes, the performance of PEX, REX and BEX is virtually indistinguishable. However, for large message sizes, PEX performs much better than REX and BEX performs better than PEX. This is because of the following reasons. First, even though the number of steps in REX is only $\lg P$, as compared to $P - 1$ steps in PEX, the message size in REX remains constant at $L \times P/2$, whereas the size of each message in PEX is L. Also, REX uses a store-and-forward approach in which a message is sent from source to destination processor through one or more intermediate processors. Sending a message from source to destination through k intermediate processors costs k times more than sending it directly. In addition, each node needs to buffer and reshuffle data in REX so that appropriate data can be sent to the appropriate node. These two overheads outweigh the savings in the number of communication steps. BEX performs the best because it maintains a balance of local and remote communication at each step.

Figures 4.7 and 4.8 show the performance of the algorithms on up to 256 processors for message size 512 bytes and 2 Kbytes respectively. PEX and BEX perform better than REX for small number of processors because the overhead of message size and number of steps dominate for REX. As the number of processors increases, the overhead of the larger number of messages in PEX and BEX is higher than the overhead of larger message size and reshuffling in REX, and therefore, REX performs better.

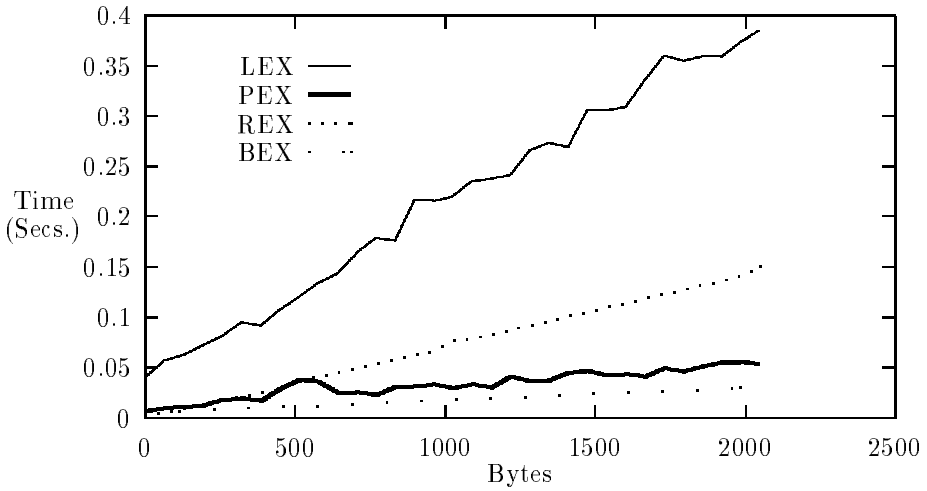

Figure 4.6: Performance on 32 node CM-5 for different message sizes

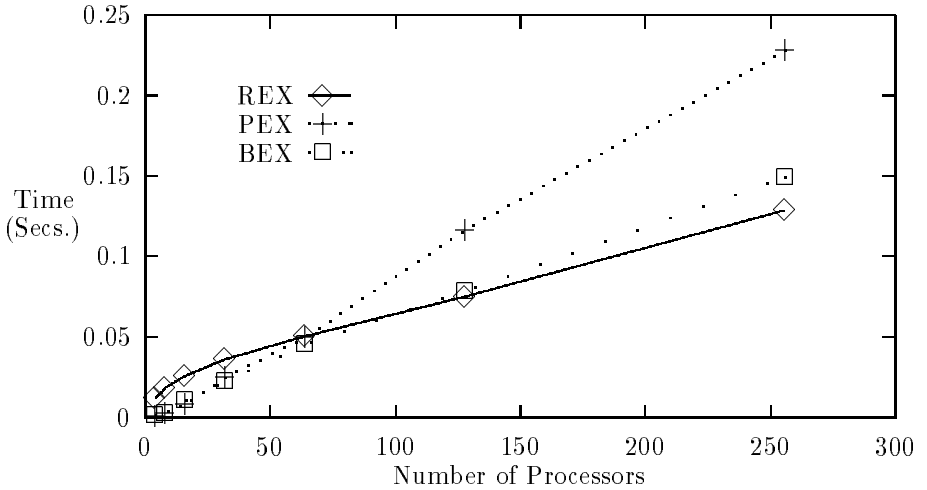

Figure 4.7: Performance on CM-5 for message size 512 bytes

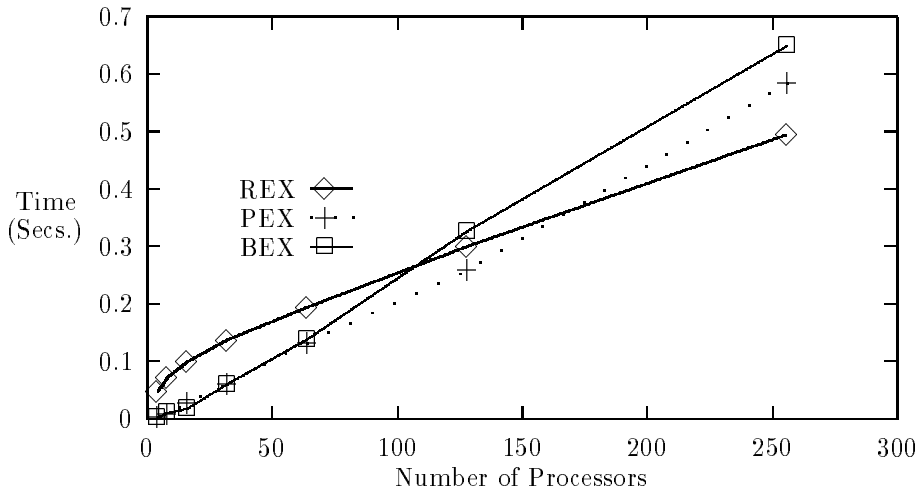

Figure 4.8: Performance on CM-5 for message size 2 Kbytes

4.3 All-to-All Communication on a 2D Mesh

This section describes algorithms for all-to-all communication on the two-dimensional mesh topology of the Intel Touchstone Delta. A mesh is a low-dimension high-bandwidth network which performs well when there is no link contention. However, a dense communication pattern like complete exchange results in a lot of link contention which can degrade performance considerably. Also, the algorithms discussed above for the CM-5 and the existing algorithms for a hypercube assume that the number of processors is a power-of-two. This is a valid assumption for hypercube and fat-tree architectures because the number of processors is always a power-of-two. However, on the Delta, the user can allocate a mesh which may not be a power-of-two and may even be an odd number (eg. 5×4 or 3×3). So, it is necessary to develop algorithms which work even on non power-of-two meshes.

Bokhari and Berryman describe two algorithms for a circuit-switched mesh, which assume that the number of processors is a power-of-two [7]. Scott has shown that $a^3/4$

is the lower bound on the number of phases required to perform complete exchange on an $a \times a$ mesh such that there is no link contention in any phase [103]. However, if we allow link contention to exist, the operation can be performed in fewer steps. We have adopted this approach of allowing a small amount of link contention to exist, thereby reducing the number of steps and keeping all processors active at every step. This approach also takes advantage of the fact that the communication links in the Delta have excess bandwidth [3], so that a small number of contending messages will not affect the communication time.

We consider six algorithms on the Delta, for power-of-two and non-power-of-two meshes. For the analysis of the algorithms, we assume that the mesh has r rows and c columns. Hence, $P = r \times c$.

4.3.1 Pairwise Exchange for Power-Of-Two Mesh (PEX)

The Pairwise Exchange algorithm described earlier for the CM-5 can also be used on the Delta without any modification, as long as the number of processors is a power of two. However, since the mesh architecture is different from a fat tree architecture, the link contention caused by this algorithm on the Delta is different from that on a fat tree. Figure 4.9 shows the communication pattern of PEX on a 2×4 mesh. The complete exchange requires seven steps. In steps 1, 4 and 5 there is no contention. In steps 2, 3, 6 and 7, the maximum number of messages contending for a link is 2. Messages traveling in opposite directions on a link do not contend, because the links on the Delta are bidirectional.

Since each step of the algorithm involves an exchange between pairs of processors, the maximum number of messages contending for a link at any step is limited by $max(r, c)/2$. An exact expression for the maximum number of messages contending for a link at step i is

$$f(i) = 2^{\lfloor \lg\{max(MOD(i,c), i/c)\}\rfloor}$$

This expression was obtained empirically. We studied the communication pattern of PEX for a large number of processors and mesh configurations. We found that there

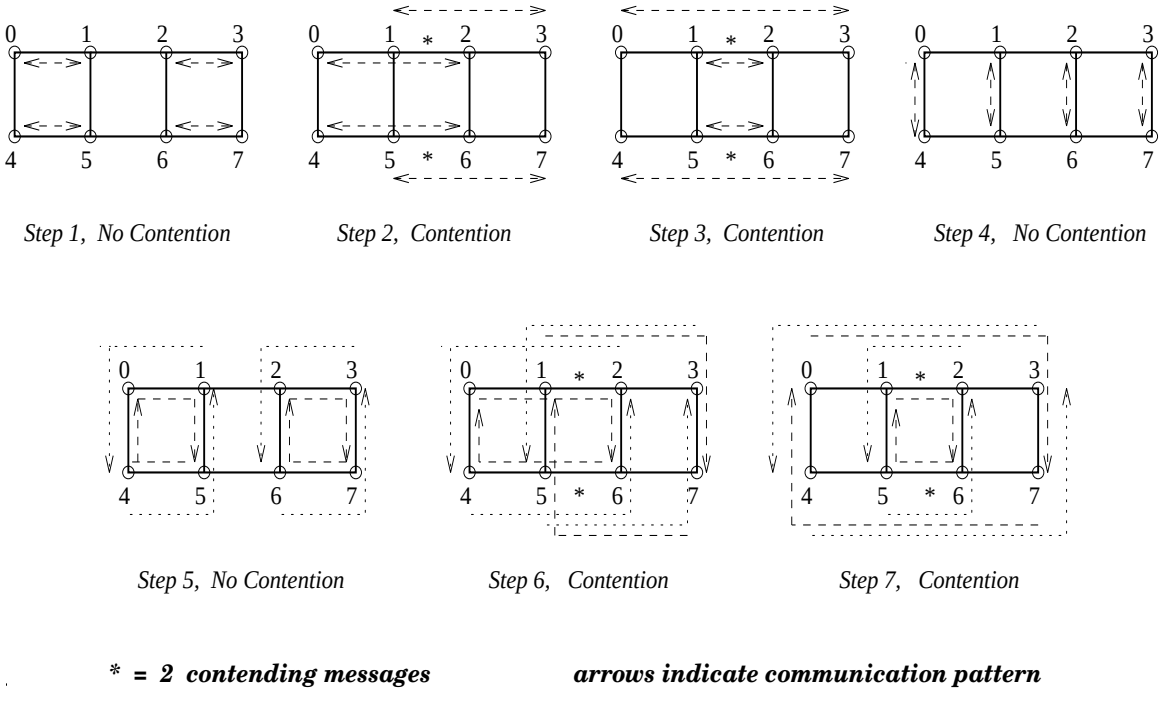

Figure 4.9: PEX on 2×4 mesh

is a relation between the step number i, the shape of the mesh and the maximum link contention in that step. That relation is given by the above expression for f(i).

The time taken for step i is

$$T(i) = \alpha + L\, max(\beta_{ex}, f(i)\beta_{sat})$$

The cost of PEX can be determined by summing over all steps of the algorithm :

$$T_{PEX} = \sum_{i=1}^{P-1} [\alpha + L\, max(\beta_{ex}, f(i)\beta_{sat})] = (P-1)\alpha + L \sum_{i=1}^{P-1} max(\beta_{ex}, f(i)\beta_{sat})$$

If the number of processors is not a power-of-two, the exclusive-or function does not create all the processor pairs in $P-1$ steps, so the algorithm is not directly applicable.

```
q = 2^⌈lg P⌉
do j=1, q − 1
      destination = xor(mynumber, j)
      if (destination < P) then
            Exchange with destination
      end if
end do
```

Figure 4.10: Pairwise Exchange for General Mesh (PEX-GEN)

4.3.2 Pairwise Exchange for General Mesh (PEX-GEN)

We have extended the basic pairwise exchange algorithm, so that it works even when the number of processors is not a power-of-two. We call this algorithm Pairwise Exchange for General Mesh (PEX-GEN) and is described in Figure 4.10. The algorithm first finds the smallest power-of-two (say q) greater than the number of processors and uses this number to schedule $q-1$ steps of the pairwise exchange. In each step, every processor checks to see if the calculated destination processor number is less than the actual number of processors. If so, it exchanges data with the processor, else it goes ahead to the next step. Thus, the algorithm requires $q-1$ steps where q is the nearest power-of-two larger than the number of processors. Clearly, the algorithm takes more steps than necessary and many processors remain idle in several of the steps. However, this reduces the link contention in each step. The maximum contention in each step is upper bounded by that in the PEX algorithm.

4.3.3 PEX-GEN with Shift (PEX-GEN-SHIFT)

The motivation for the Pairwise Exchange for General Mesh with Shift (PEX-GEN-SHIFT) algorithm can be explained with the help of Figure 4.11(a). Assume that the user has allocated a mesh of 20 processors which may be organized in any way (ie 4×5,

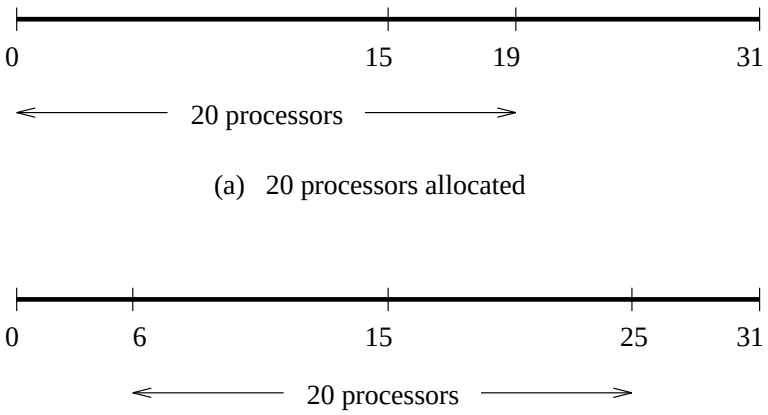

Figure 4.11: Processor Shift

or 2×10 etc.). The processor numbers will be 0 to 19. The nearest power of two larger than 20 is 32. The PEX-GEN algorithm will require 31 steps. The processor pairs created by the exclusive-or function are such that in steps 1 through 15, processors 0 to 15 exchange completely among themselves and do not communicate with any processor in the range 16 to 31. Similarly, processors 16 to 19 exchange completely among themselves and do not communicate with any processor in the range 0 to 15. In steps 16 through 31, the processors in one half (0 — 15) exchange with processors in the second half (16 — 31). Since there are only 4 processors in the second half, several of the processors in the first half do not do any communication. Thus the communication pattern is such that in the first 15 steps, all the processors in the first half are active and in the next 16 steps, several of them are inactive. Since each step involves communication with link contention, there is high link contention in steps 1 — 15 and very little or no link contention in steps 16 — 31. In general, if there are P processors where P is not a power of two, PEX-GEN will require $q-1$ steps where $q = 2^{\lceil \lg P \rceil}$. In the first $\lfloor (q-1)/2 \rfloor$ steps, the first $q/2$ processors are active and in the remaining steps, several of them are inactive.

A better algorithm is one which balances the contention such that all steps have more or less equal contention and equal number of inactive processors. This can be

```
q = 2^⌈lg P⌉
shift = (q − P)/2
myvirtual = MOD(mynumber + shift, P)
do j=1, q − 1
      virtual_destination = xor(myvirtual, j)
      destination = virtual_destination – shift
      if (destination < 0) then
            destination = destination + q
      end if
      if (destination < P) then
            Exchange with destination
      end if
end do
```

Figure 4.12: Pairwise Exchange for General Mesh with Shift (PEX-GEN-SHIFT)

achieved by defining virtual processor numbers such that the real processors 0 — 19 are numbered 6 — 25 as shown in Figure 4.11(b). The processor numbers are shifted by an amount equal to half the absolute difference between the number of processors and the nearest power of two. In other words, in the range 0 — 31, the actual processors are numbered 6 — 25, and there are no processors numbered 0 — 5 and 26 — 31. Thus the empty space which earlier existed only in the half 16 — 31 is now equally divided among the two halves. So, even in the first 15 steps of the algorithm, there are equal number of idle processors in both halves, which balances the contention among all the steps of the algorithm. We call this algorithm Pairwise Exchange for General Mesh with Shift (PEX-GEN-SHIFT) and is described in Figure 4.12. This algorithm also takes $q - 1$ steps where q is the smallest power-of-two larger than the number of processors. The maximum contention at each step is upper bounded by that for the PEX algorithm.

```
do j=1, P − 1
      destination = MOD(mynumber + j, P)
      source = MOD(mynumber − j + P, P)
      send to destination
      receive from source
end do
```

Figure 4.13: General Algorithm for any Mesh (GEN)

4.3.4 General Algorithm for any Mesh (GEN)

The above algorithms require one less than a power of two number of steps, because they use the exclusive-or function to obtain processor pairs which exchange with each other. For non power-of-two meshes, it would be advantageous to have an algorithm which requires only $P-1$ steps. Figure 4.13 describes such an algorithm, which we call the General Algorithm for any Mesh (GEN), because it works for any number of processors. In the GEN algorithm, processor pairs do not exchange with each other. Instead, at step i, a processor j sends data to processor $MOD(j+i, P)$ and receives data from processor $MOD(j-i+P, P)$. This algorithm requires only $P-1$ steps, for any value of P.

The maximum contention at step i is obtained empirically as

$$f(i) = min[MOD(i,c), c - MOD(i,c)] + min[i/c, (P-i)/c]$$

The total time for all steps is given by :

$$T_{GEN} = \sum_{i=1}^{P-1} [\alpha + L\, max(\beta_{sr}, f(i)\beta_{sat})] = (P-1)\alpha + L \sum_{i=1}^{P-1} max(\beta_{sr}, f(i)\beta_{sat})$$

4.3.5 Indirect Pairwise Exchange (IPEX)

The Indirect Pairwise Exchange (IPEX) algorithm aims at reducing link contention in the direct Pairwise Exchange (PEX) algorithm. In IPEX, each processor communicates only with the processors in its row and column. The algorithm is described in Figure 4.14. Each exchange along a row is followed by a complete exchange along a column. During the row exchange, each processor sends $L\,r$ bytes of data to the destination processor, out of which $L(r-1)$ bytes are intended for other processors in the same column as the destination processor. This is followed by a complete exchange along the columns (involving messages of L bytes), in which the data received during the row exchange is sent to the appropriate processors in the same column. This entire operation requires $r(c-1)$ communication steps. Finally, an additional complete exchange is required along the columns for processors to exchange their own data directly with processors in the same column. In this phase, data is sent directly from source to destination, requiring $r-1$ exchange steps. Hence, the total number of steps required is $r(c-1)+(r-1) = rc-1 = P-1$.

The maximum link contention at any step is the same as for pairwise exchange along a row or column which is $2^{\lfloor \lg i \rfloor}$, where i is the step number along the row or column. Hence, the total time required for IPEX is given by :

$$\begin{aligned} T_{IPEX} &= \sum_{i=1}^{c-1}[\alpha + L\,r\,max(\beta_{ex}, 2^{\lfloor \lg i \rfloor}\beta_{sat}) + \sum_{j=1}^{r-1}\{\alpha + L\,max(\beta_{ex}, 2^{\lfloor \lg j \rfloor}\beta_{sat})\}] \\ &\quad + \sum_{i=1}^{r-1}[\alpha + L\,max(\beta_{ex}, 2^{\lfloor \lg i \rfloor}\beta_{sat})] \end{aligned}$$

4.3.6 Recursive Exchange (REX)

The Recursive Exchange algorithm is described in Figure 4.15. It is similar to that for the CM-5, except it is recursively applied to both dimensions of the mesh. The mesh is first recursively halved in the x direction and messages are exchanged over

```
x = my x-coordinate
y = my y-coordinate
do i=1, c − 1
      destx = xor(x, i)
      dest = y × c + destx
      exchange L r bytes with dest
      do j=1, r − 1
            desty = xor(y, i)
            dest = desty×c + x
            exchange L bytes with dest
      end do
end do
do j=1, r − 1
      desty = xor(y, i)
      dest = desty ×c + x
      exchange L bytes with dest
end do
```

Figure 4.14: Indirect Pairwise Exchange (IPEX)

each cut. This takes $\lg c$ steps. The mesh is then recursively halved in the y direction and messages are exchanged over each cut, which takes $\lg r$ steps. Thus, the total number of steps is $\lg c + \lg r = \lg P$ and the message size in each step is $L \times P/2$. This algorithm also works only for power-of-two meshes, since the mesh is divided by two in each step. REX has an indirect form of communication, in the sense that data is sent from source processor to destination processor through one or more intermediate processors.

Since the mesh is recursively divided by two and each processor in one partition communicates with its mirror image in the other partition, the maximum number of messages contending for a link at step i is

$$f(i) = \begin{cases} c/2^i & \text{for } 1 \le i \le \lg c \\ \frac{r}{2^{i-\lg c}} & \text{for } \lg c < i \le \lg P \end{cases}$$

```
size = c
pos = 0
x = my x-coordinate
y = my y-coordinate
bytes = L × P/2
do i=1, lg c
     size = size/2
     if (x < (size + pos)) then
          destx = x + size
     else
          destx = x − size
          pos = pos + size
     end if
     dest = y × c + destx
     exchange message of size "bytes" with dest
end do
size = r
pos = 0
do i=1, lg r
     size = size/2
     if (y < (size + pos)) then
          desty = y + size
     else
          desty = y − size
          pos = pos + size
     end if
     dest = desty×c + x
     exchange message of size "bytes" with dest
end do
```

Figure 4.15: Recursive Exchange (REX) on Delta

Table 4.4: Performance of PEX (time in sec.)

Message Size (bytes)	Mesh Configuration					
	4×4	8×8	16×8	8×16	16×16	16×32
256	0.004	0.022	0.045	0.045	0.094	0.203
1K	0.008	0.064	0.120	0.115	0.290	0.860
4K	0.023	0.114	0.355	0.367	0.999	3.218
8K	0.034	0.228	0.692	0.773	2.068	6.794
16K	0.064	0.441	1.413	1.565	4.145	13.61

The cost of REX can be obtained by summing over all steps of the algorithm :

$$T_{REX} = \sum_{i=1}^{\lg P} [\alpha + \frac{L\ P}{2} max(\beta_{ex}, f(i)\beta_{sat})]$$

which can be expanded to

$$T_{REX} = \alpha \lg P + \frac{L\ P}{2} \sum_{i=1}^{\lg c} max(\beta_{ex}, c/2^i \beta_{sat}) + \frac{L\ P}{2} \sum_{i=\lg c+1}^{\lg P} max(\beta_{ex}, \frac{r}{2^{i-\lg c}} \beta_{sat})$$

4.3.7 Performance of Algorithms on the Delta

We implemented all the algorithms on the Intel Touchstone Delta and studied their performance for different mesh configurations and message sizes. As suggested in [80], we use *forced* messages (which provide higher bandwidth but also higher startup cost) if the message size is greater than or equal to 1.5 Kbytes and *unforced* messages if the message size is less than 1.5 Kbytes.

The performance of PEX is shown in Table 4.4. The number of processors is varied from 16 to 512 with message size varied from 256 bytes to 16 Kbytes. Message size refers to the amount of data communicated in each send and receive operation, so the total amount of data communicated increases as the number of processors is increased. Hence, the time taken increases almost linearly with the number of

Table 4.5: Performance of PEX-GEN (time in sec.)

Message Size (bytes)	Mesh Configuration					
	4×5	6×8	16×9	8×18	16×14	16×30
256	0.008	0.019	0.085	0.090	0.092	0.211
1K	0.017	0.038	0.191	0.230	0.270	0.899
4K	0.037	0.091	0.576	0.830	0.977	3.588
8K	0.073	0.174	1.188	1.743	2.007	7.616
16K	0.138	0.333	2.403	3.480	4.056	15.82

processors. In a mesh, the time taken depends not only on the number of processors, but also on the mesh configuration. The maximum contention in PEX is $max(r, c)/2$. Thus, for a fixed number of processors, the time taken will be minimum for a square mesh and maximum for a mesh which is a linear array.

The performance of PEX-GEN is given in Table 4.5. We have chosen some mesh sizes which are non power-of-two. We observe that for mesh sizes which are only slightly less than the nearest higher power-of-two, the performance is close to that of PEX for that power-of-two. But, if the mesh size is only slightly higher than the nearest smaller power-of-two, the time taken is almost twice the time taken by PEX for that power-of-two. For example, the time taken by PEX-GEN on a 16×9 mesh is much higher than the time taken by PEX on a 16×8 mesh, but the time taken by PEX-GEN on a 16×14 mesh is very close to the time taken by PEX on a 16×16 mesh. This is because of the difference in the number of steps required. Another interesting observation is that the time taken by PEX-GEN on a 16×30 mesh is in fact higher than the time taken by PEX on a 16×32 mesh. This is because since the processors are numbered in row major order, a change in the number of columns from a power-of-two to a non power-of-two, changes the communication pattern in the mesh completely for an algorithm which uses the exclusive-or function to determine processor pairs. Hence, there is more contention in the 16×30 case than in the 16×32 case.

Table 4.6 shows the performance of PEX-GEN-SHIFT. In all cases it performs

Table 4.6: Performance of PEX-GEN-SHIFT (time in sec.)

Message Size (bytes)	Mesh Configuration					
	4×5	6×8	16×9	8×18	16×14	16×30
256	0.008	0.019	0.085	0.089	0.092	0.211
1K	0.017	0.038	0.188	0.219	0.263	0.894
4K	0.036	0.091	0.543	0.782	0.933	3.526
8K	0.071	0.170	1.111	1.626	1.948	7.515
16K	0.129	0.333	2.242	3.282	3.844	15.74

no worse than PEX-GEN. In most cases, the improvement in performance is not significant.

Table 4.7 gives the performance of GEN on a power-of-two mesh. GEN performs better than PEX for small message sizes and small number of processors. However, for large number of processors (≥ 64) and large message sizes (> 1 Kbytes) PEX performs better. The GEN algorithm has a certain amount of asymmetry in the communication in the sense that each communication operation consists of a send to one processor and a receive from some other processor. Thus, the incoming and outgoing messages may traverse a different number of links with different amounts of contention, and the path which has the highest amount of contention adversely affects the communication time. On the other hand, in the PEX algorithm, processor pairs exchange with each other at every step, so the incoming and outgoing messages travel the same number of links with the same amount of contention.

The performance of GEN on non power-of-two meshes is given in Table 4.8. GEN reduces the number of steps from $q - 1$ in PEX-GEN and PEX-GEN-SHIFT, where $q = 2^{\lceil \lg P \rceil}$, to $P - 1$. For small number of processors, PEX-GEN performs the best and the improvement in performance is higher when $q - P$ is large. However, if $q - P$ is small and the number of processors is large, the performance of PEX-GEN-SHIFT tends to that of PEX and and the performance of GEN tends to that for a power-of-two mesh. So in this case, PEX-GEN-SHIFT performs better than GEN.

Table 4.7: Performance of GEN on power-of-two mesh (time in sec.)

Message Size	Mesh Configuration					
(bytes)	4×4	8×8	16×8	8×16	16×16	16×32
256	0.004	0.016	0.042	0.042	0.089	0.283
1K	0.008	0.042	0.123	0.132	0.346	1.217
4K	0.018	0.145	0.461	0.464	1.220	3.944
8K	0.037	0.290	0.933	0.927	2.511	8.007
16K	0.069	0.576	1.947	1.884	5.052	16.15

Table 4.8: Performance of GEN on non power-of-two mesh (time in sec.)

Message Size	Mesh Configuration					
(bytes)	4×5	6×8	16×9	8×18	16×14	16×30
256	0.004	0.015	0.046	0.048	0.074	0.246
1K	0.009	0.027	0.146	0.171	0.285	1.069
4K	0.025	0.083	0.527	0.566	0.998	3.706
8K	0.052	0.186	1.071	1.154	2.011	7.752
16K	0.098	0.369	2.182	2.360	4.005	15.94

We observe from Table 4.9 that REX does not perform well for any mesh configuration and message size even though it requires only $\lg P$ steps. There are several reasons for this. First, there is a lot of link contention in each step. Second, the message size per step is increased to $L\ P/2$ instead of L in the other algorithms. Third, the indirect form of communication requires a lot of data buffering and shuffling in order to send the appropriate data to the appropriate node. Also, this algorithm has high memory requirements because the large message size requires more memory per node. For example, with the other algorithms we could run tests with message sizes up to 2M bytes, but with REX we could only go up to 16 Kbytes on 512 processors.

Table 4.10 gives the performance of IPEX. For large meshes and large message sizes, IPEX performs better than any of the direct algorithms. This is because in

Table 4.9: Performance of REX on Delta (time in sec.)

Message Size	Mesh Configuration					
(bytes)	4×4	8×8	16×8	8×16	16×16	16×32
256	0.004	0.022	0.056	0.056	0.140	0.393
1K	0.011	0.082	0.222	0.221	0.552	1.581
4K	0.029	0.212	0.615	0.600	1.566	4.739
8K	0.059	0.415	1.216	1.203	3.151	9.390
16K	0.114	0.950	2.512	2.381	6.279	18.75

Table 4.10: Performance of IPEX (time in sec.)

Message Size	Mesh Configuration					
(bytes)	4×4	8×8	16×8	8×16	16×16	16×32
256	0.005	0.023	0.050	0.055	0.115	0.256
1K	0.010	0.059	0.144	0.145	0.334	0.829
4K	0.026	0.156	0.430	0.397	0.974	2.746
8K	0.049	0.294	0.823	0.755	3.151	5.499
16K	0.089	0.573	1.642	1.756	3.877	10.89

large meshes, direct algorithms result in a lot of link contention. The reduction in contention by IPEX is larger than the cost of sending messages indirectly, hence IPEX performs better. The message size and contention in each step in IPEX is much smaller than in REX. The additional memory required per node in IPEX is $L\,r$, compared to $L \times P/2$ in REX. In small meshes, the cost of communicating indirectly is higher than the saving in contention, so direct algorithms perform better.

A comparison of the power-of-two algorithms on a 16×32 mesh for different message sizes is shown in Figure 4.16. We can see that for this mesh size IPEX performs the best, for reasons explained above. This is followed by PEX and GEN. REX always performs the worst in spite of its $\lg P$ steps. The relative performance of non power-of-two algorithms on a 16×9 mesh is shown in Figure 4.17. For this

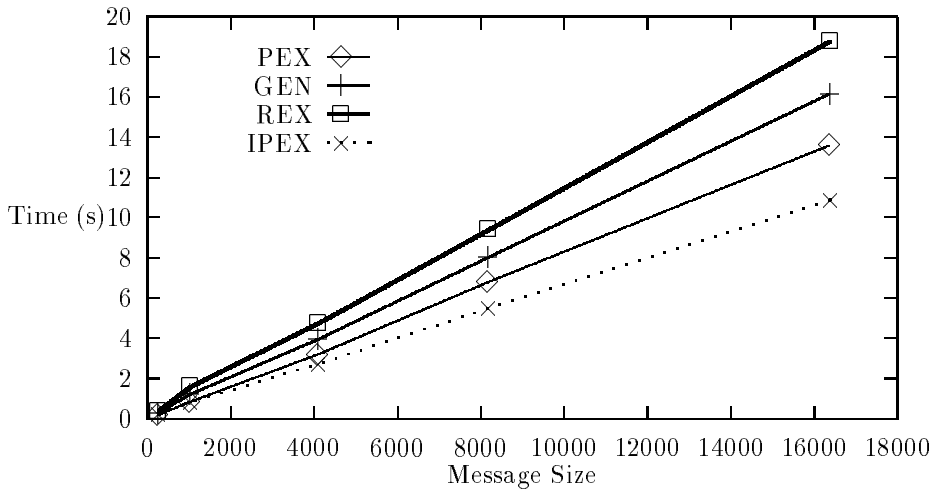

Figure 4.16: Performance of algorithms on a 16×32 mesh

mesh size, GEN clearly performs the best. For messages up to 2 Kbytes, the performance of PEX-GEN and PEX-GEN-SHIFT is almost indistinguishable. But for larger messages, PEX-GEN-SHIFT performs better than PEX-GEN.

The performance of the power-of-two algorithms for different mesh sizes keeping the message size constant at 16 Kbytes is shown in Figure 4.18. The corresponding graph for a message size of 1 Kbytes is shown in Figure 4.19. We observe that for a large message size of 16 Kbytes, PEX performs the best for meshes with up to 200 processors. For larger meshes, IPEX performs the best. For a message size of 1 Kbytes, we see that GEN performs the best on meshes with up to 125 processors. When the number of processors is between 125 and 420, PEX performs the best and for larger systems, IPEX performs the best.

4.3.8 Model Validation

We have validated the models developed to predict the performance of the algorithms by comparing the predicted times with the actual times observed on the Delta. For

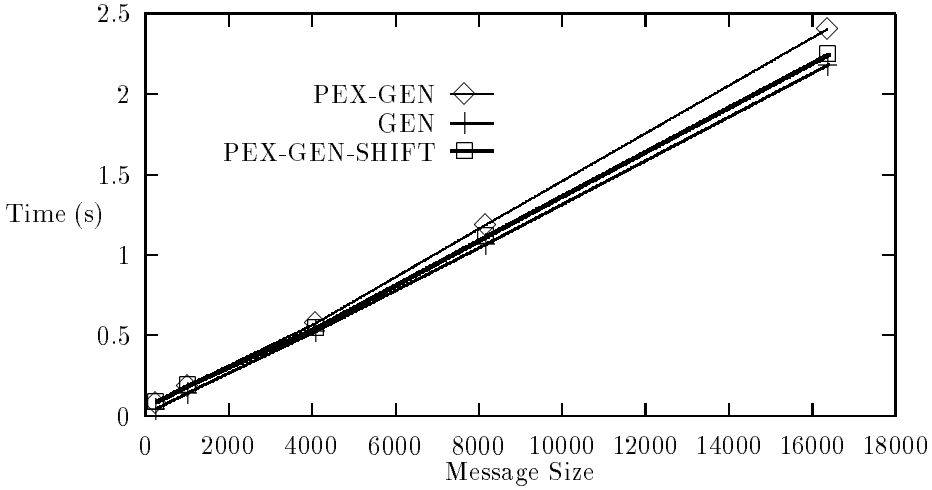

Figure 4.17: Performance of algorithms on a 16×9 mesh

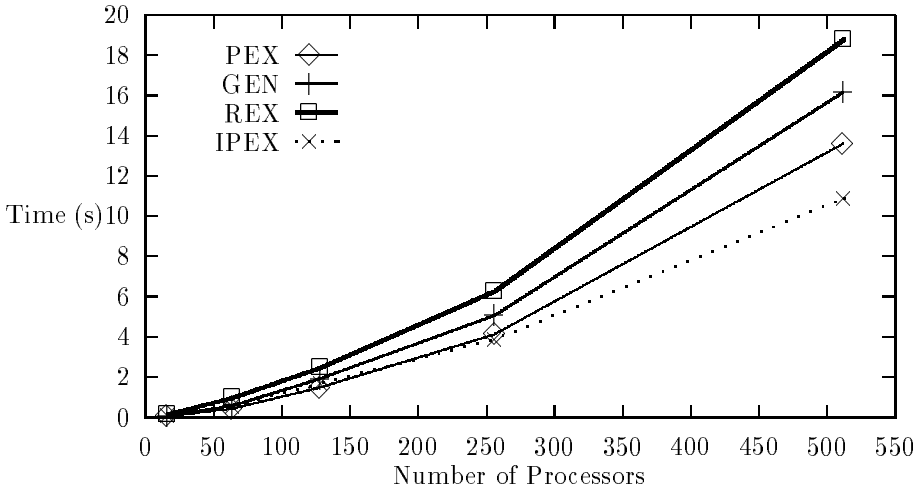

Figure 4.18: Performance of power-of-two algorithms for message size 16 Kbytes

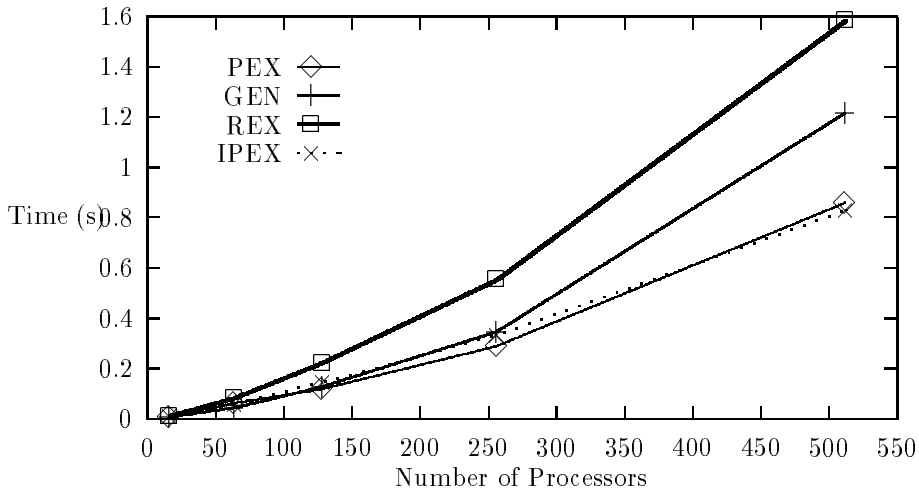

Figure 4.19: Performance of power-of-two algorithms for message size 1 Kbytes

this purpose, we use typical values for the communication costs on the Delta [80, 3], namely for unforced messages $\alpha = 75\mu s$, $\beta_{ex} = 0.35\mu s$ and for forced messages $\alpha = 150\mu s$, $\beta_{ex} = 0.2\mu s$. We assume that $\beta_{sr} \approx \beta_{ex}$ and $2\beta_{sat} \approx \beta_{ex}$, as done in [3], ie. two messages can travel on a link in the same direction without conflict. Figures 4.20 and 4.21 show that the observed and predicted times agree very closely.

We were not able to validate the models on the CM-5 because accurate values for β_{sat} relative to β_{ex} or β_{sr} for the CM-5 are not available in the literature.

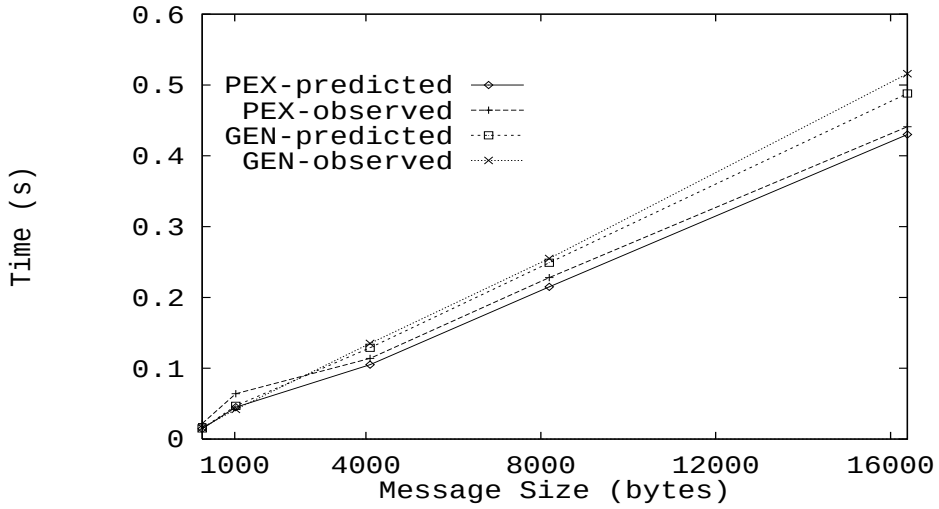

Figure 4.20: Observed and Predicted times (PEX, GEN)

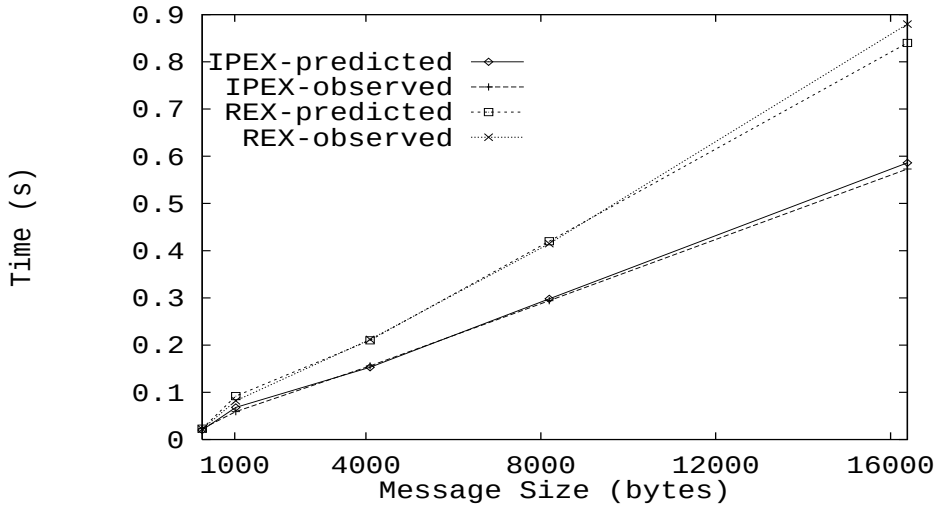

Figure 4.21: Observed and Predicted times (REX, IPEX)

Chapter 5

Runtime Support for Out-of-Core Programs: (I) Models and Local Optimizations

5.1 Introduction

There are a number of applications which deal with very large quantities of data. These applications exist in diverse areas such as large scale scientific computations, database applications, hypertext and multimedia systems, information retrieval, visualization etc. The number of such applications and their data requirements keep increasing day by day. Although supercomputers have very large main memories, the memory is not large enough to hold all the data required by these applications. For example, a typical Grand Challenge Application at present could require 1Gbyte to 4Tbytes of data per run [38]. These figures are expected to increase by orders of magnitude as teraflop machines make their appearance. Hence, data needs to be stored on disk and the performance of the program depends on how fast processors can access data from disks. A poor I/O capability can severely degrade the performance of the entire program.

Almost all present generation parallel computers provide some kind of hardware and system software support for parallel I/O [29, 92, 9, 36]. But, the I/O performance observed at the application level is usually much lower than what the hardware can

support. There are several reasons for this. First, the data access patterns of many parallel programs are such that they result in a large number of small requests to the file system [74]. Since the I/O latency is very high, this results is poor performance. Second, the interface to any parallel file system does not currently allow a programmer to specify strided accesses using a single read or write call; though there are some recent proposals to rectify this [28, 88]. Third, the interface does not provide support for processors to make a collective I/O request. So file systems cannot perform any optimizations based on the knowledge of the access requests of all processors. Finally, the programmer cannot specify access requests using a high level description, but instead has to explicitly manipulate file pointers. This makes it difficult for the programmer to perform optimizations at the application level, for example prefetching to overlap I/O with computation, because of the complexities involved in managing buffers and file pointers.

We have developed a runtime system called PASSION (Parallel and Scalable Software for Input-Output) [116, 25] which aims to alleviate many of these problems and provide better software support for out-of-core programs. We believe that high-level interfaces that facilitate the use of semantic knowledge about the accesses from parallel programs are necessary for simple and portable application programming. A high-level interface can at the same time provide enough information so that I/O can be done in an efficient manner. The PASSION Runtime Library accepts high-level requests from the application program, translates them to the low-level interface supported by the parallel file system, and performs optimizations for efficient I/O. This chapter discusses the basic design of PASSION and the various models, techniques and optimizations used in it.

5.2 PASSION Runtime Library

The PASSION Runtime Library provides routines to efficiently perform the I/O required in programs involving out-of-core multidimensional arrays. It provides support for loosely synchronous [46] out-of-core computations which use a Single Program

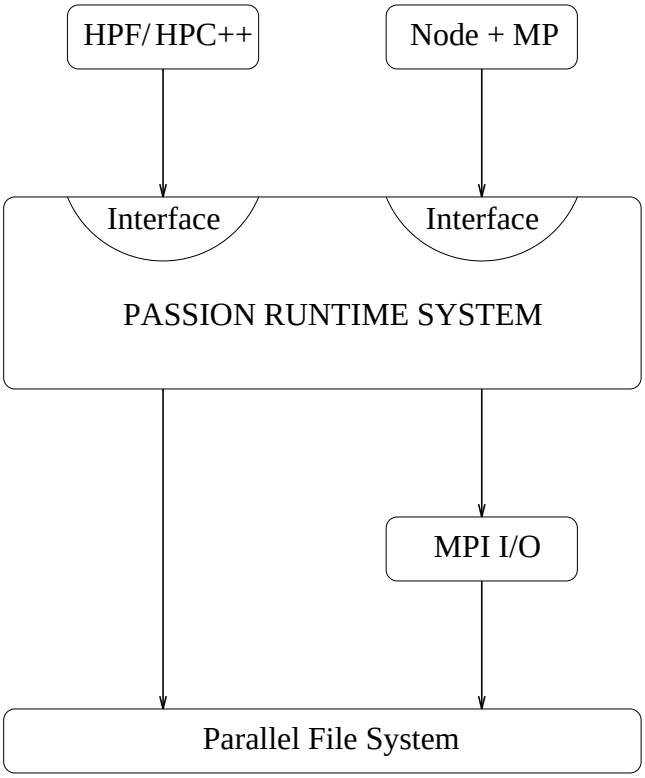

Figure 5.1: Software Architecture

Multiple Data (SPMD) Model. The library can either be directly used by application programmers, or a compiler can translate out-of-core programs written in a high-level data-parallel language like HPF to node programs with calls to the library for I/O. PASSION provides the user with a simple high-level interface, which is a level higher than any of the existing parallel file system interfaces, as shown in Figure 5.1. For example, the user only needs to specify what section of the array needs to be read/written in terms of its lower-bound, upper-bound and stride in each dimension, and the PASSION Runtime Library will fetch it in an efficient manner. A number of optimizations such as Data Sieving, Data Prefetching, Data Reuse, and the Extended Two-Phase Method have been incorporated in the library [116, 115, 118].

5.3 Models

This section discusses the architectural model and the data storage and access models used by the PASSION Runtime Library.

5.3.1 Architectural Model

An important goal in the design of PASSION has been to make it architecture independent as far as possible. The architectural model assumed by PASSION is that of any general distributed memory computer in which the processors are connected together in some fashion. The system is assumed to be provided with a set of disks and I/O nodes. The I/O nodes can either be dedicated processors or some of the compute nodes may also serve as I/O nodes [72]. Each processor may either have its own local disk or all processors may share the set of disks. We prefer if the file system allows us to control the way files are stored on disks, particularly the number of I/O nodes (or disks) across which the file is striped and the stripe size. The I/O subsystem may have a separate interconnection network or it can share the same network which connects the processors together.

PASSION has been implemented on the Intel Touchstone Delta using the native Concurrent File System (CFS) and the NX message passing library. Hence it can run without modification on other Intel machines such as the Paragon and iPSC/860. All the performance results in this chapter as well as in Chapter 6 have been taken on the Touchstone Delta. The computation and communication hardware on the Delta is described in Section 4.1.2. The I/O system on the Delta consists of 32 dedicated I/O nodes, each an Intel 80386 microprocessor. Each I/O node is connected to two disks, resulting in a total of 64 disks. A Concurrent File System (CFS) [92] is provided for parallel access to files. By default, a file is striped across all 64 disks in a round-robin fashion in blocks of size 4 Kbytes. It is possible for the user to specify the disks on which a file is to be stored, but the stripe size is fixed. The CFS provides the user with a Unix-like interface and the parallel reads and writes are handled transparently. The performance of the CFS on the Touchstone Delta has been studied in detail in [9]. The

CFS supports four *modes* of file access. In Mode 0, each processor has an independent file pointer. In Modes 1 – 3, processors have a common file pointer. All PASSION routines have been implemented using Mode 0.

5.3.2 Data Storage and Access Models

Since PASSION is used in programs having large arrays which do not fit in main memory, the arrays have to stored on disks in some fashion. PASSION supports three basic models of storing and accessing arrays, called the *Local Placement Model (LPM)*, the *Global Placement Model (GPM)* and the *Partitioned In-Core Model (PIM)*.

Local Placement Model (LPM)

In this model, the global array is divided into local arrays belonging to each processor. Since the local arrays are out-of-core, they have to be stored in files on disks. The local array of each processor is stored in a separate file called the **Local Array File (LAF)** of that processor as shown in Figure 5.2(I). The node program explicitly reads from and writes into the file when required. The simplest way to view this model is to think of each processor as having another level of memory which is much slower than main memory. If the I/O architecture of the system is such that each processor has its own disk, the LAF of each processor will be stored on the disk attached to that processor. If there is a common set of disks for all processors, the LAF will be distributed across one or more of these disks. In other words, we assume that each processor has its own *logical disk* with the LAF stored on that disk. The mapping of logical disks to physical disks depends on how much control the parallel file system provides the user. At any time, only a portion of the local array is fetched and stored in main memory. The size of this portion depends on the amount of memory available. The portion of the local array which is in main memory is called the **In-Core Local Array (ICLA)**. All computations are performed on the data in the ICLA. Thus, during the course of the program, parts of the LAF are fetched into the ICLA, the new values are computed and the ICLA is stored back into appropriate locations in

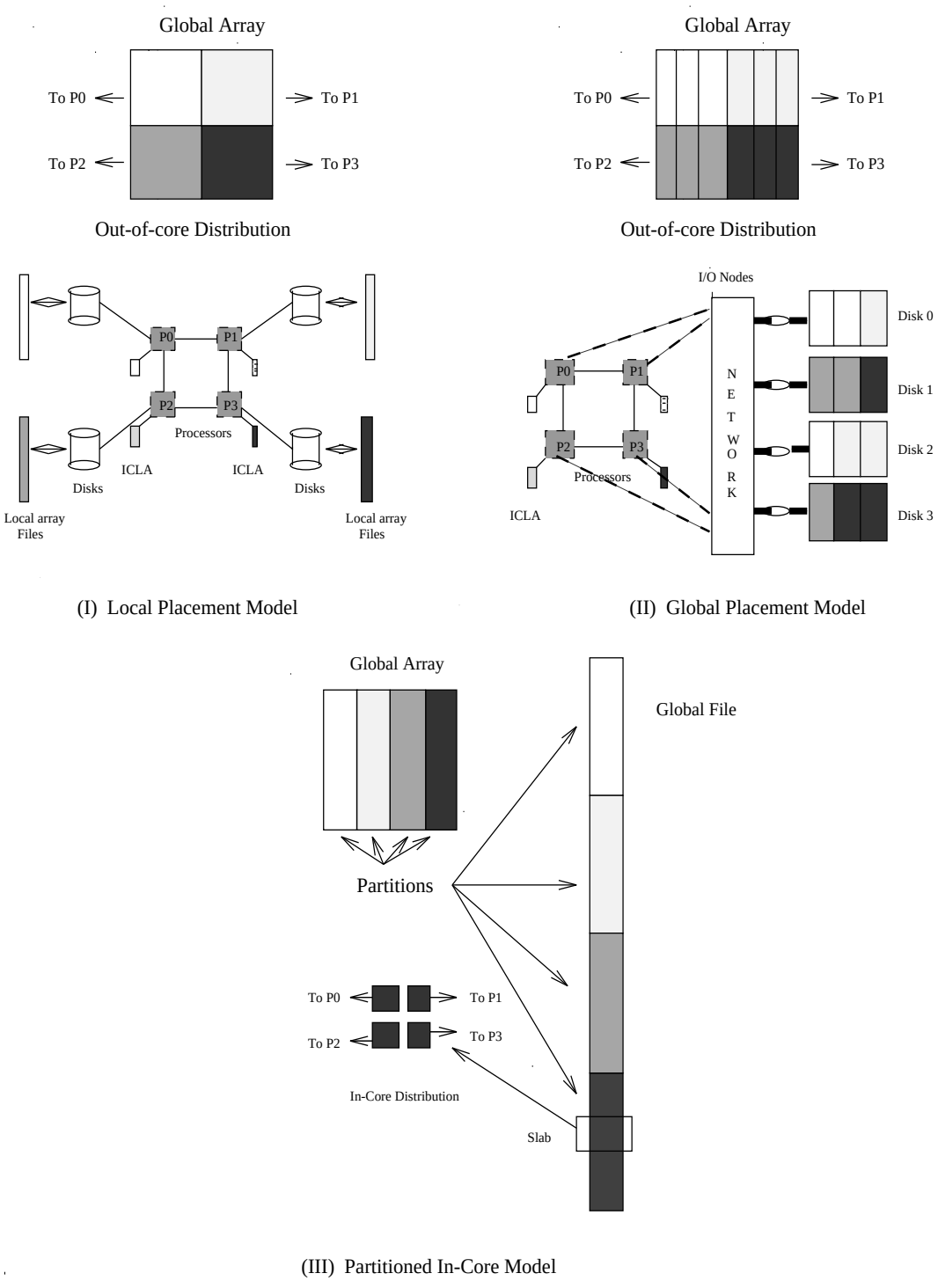

Figure 5.2: Data Storage and Access Models

the LAF.

Global Placement Model (GPM)

In this model, the global array is stored in a single file called the **Global Array File (GAF)** as shown in Figure 5.2(II) and no local array files are created. The global array is only logically divided into local arrays in keeping with the SPMD programming model. But, there is a single global array on disk. The PASSION runtime system fetches the appropriate portion of each processor's local array from the global array file, as requested by the user, in an efficient manner. The advantage of the Global Placement Model is that it does not require the initial local array file creation phase in the Local Placement Model. The disadvantage is that each processor's data may not be stored contiguously in the GAF, resulting in higher I/O latency time. Also, explicit synchronization is required when a processor needs to access data belonging to another processor. Hence an optimized method, such as the Extended Two-Phase Method proposed in Chapter 6, is needed to perform I/O efficiently.

Partitioned In-Core Model (PIM)

The Partitioned In-Core Model, illustrated in Figure 5.2(III), is a variation of the Global Placement Model. The array is stored in a single global array file as in the Global Placement Model, but there is a difference in the way data is accessed. In the Partitioned In-Core Model, the global array is logically divided into a number of *partitions*, each of which can fit in the main memory of all processors combined. Thus the computation on each partition is essentially an in-core problem and no I/O is required during the computation on the partition. Hence the name Partitioned In-Core Model. This model is useful when the data access pattern in the program has good locality. Otherwise, creating in-core partitions itself is difficult. The Extended Two-Phase Method is used for I/O in this model.

5.4 Runtime Support for the Local Placement Model

Let us first consider runtime support for the Local Placement Model. Consider the HPF program fragment shown in Figure 5.3, which solves Laplace's equation by Jacobi iteration method. Let us assume that arrays A and B are very large and hence out-of-core. We note that this may not be the best algorithm for solving Laplace's equation with an out-of-core data set as discussed in [41], but we only use it for the purpose of explanation.

The arrays A and B are distributed as (block,block) on a 4×4 grid of processors as shown in Figure 5.4. In the Local Placement Model, the out-of-core local array of each processor is stored in a separate local array file. Consider the out-of-core local array (OCLA) on processor P5, shown in Figure 5.4(B). This is stored in the local array file (LAF) shown in Figure 5.4(D). Depending on the amount of memory available, the OCLA is divided into slabs each of which can fit in the in-core local array (ICLA). Program execution proceeds by fetching a slab from the LAF to the ICLA, doing the computation on that slab, storing the results back to the file, and so on for the remaining slabs.

The computation in this example is a *stencil computation* in which the value of each element (i, j) is calculated using the values of its corresponding four neighbors, namely $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. Also, the computation in the current iteration uses values computed in the previous iteration. Thus to calculate the values at the four boundaries of the local array, P5 needs the last row of the local array of P1, the last column of the local array of P4, the first row of the local array of P9 and the first column of of the local array of P6. Before each iteration of the program, P5 needs to get these rows and columns from its neighboring processors. If the local array was in-core, these rows and columns would have been placed in the overlap areas shown in Figure 5.4(B). This is done so as to obtain better performance by retaining the DO loop even at the boundary. Since the local array is out-of-core, these overlap areas are provided in the local array file. The local array file basically consists of

```
parameter (n=1024)
real A(n,n), B(n,n)
..........
!HPF$ PROCESSORS P(4,4)
!HPF$ TEMPLATE T(n,n)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN with T ::  A, B
...........
    FORALL (i=2:n-1, j=2:n-1)
        A(i,j) = (B(i,j-1) + B(i,j+1) + B(i+1,j)
            + B(i-1,j))/4
...........
    B = A
```

Figure 5.3: HPF Program Fragment: Solving Laplace's Equation by Jacobi Iteration Method

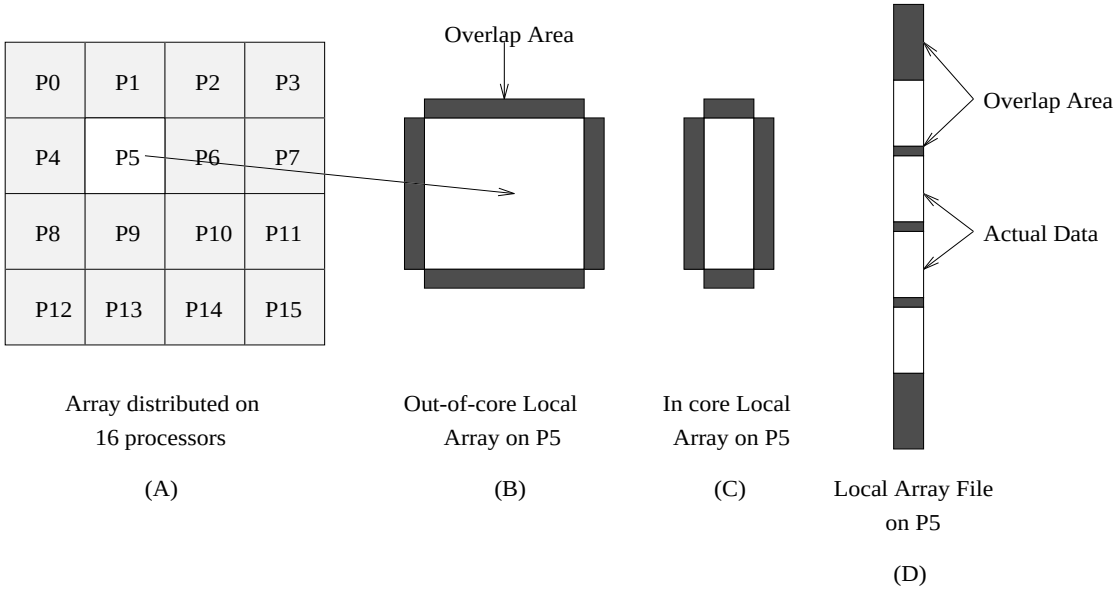

Figure 5.4: Example of OCLA, ICLA and LAF

Table 5.1: Some of the PASSION Routines

	PASSION Routine	Function
1	PASSION_read	Read entire LAF into ICLA
2	PASSION_write	Write entire ICLA to LAF
3	PASSION_read_section	Read a regular section (with stride) from LAF to ICLA
4	PASSION_write_section	Write a regular section (with stride) from ICLA to LAF
5	PASSION_read_prefetch	Prefetch a regular section
6	PASSION_prefetch_wait	Wait for a prefetch to complete
7	PASSION_read_reuse	read_section with data reuse
8	PASSION_global_read	Read a regular section (with stride) from global array file
9	PASSION_global_write	Write a regular section (with stride) to global array file
10	PASSION_oc_shift	Shift type collective communication on out-of-core data
11	PASSION_oc_multicast	Multicast communication on out-of-core data

the local array stored in either row-major or column-major order. In either case, the local array file will consist of the local array elements interspersed with overlap area as shown in Figure 5.4(D). The in-core local array also needs overlap area for the same reason as for the out-of-core local array.

At the end of each iteration, processors need to exchange boundary data with neighboring processors. In the in-core case, this would be done using a shift type collective communication routine to directly communicate data from the local memory of the processors. In the out-of-core case, this communication also requires I/O. In an out-of-core shift type collective communication, each processor reads the boundary data from its local array file and communicates it to the neighboring processor. The processor also receives the data sent by neighboring processors and stores it in appropriate locations in the local array file.

We have developed a library of routines to do the I/O as well as the out-of-core communication required in programs such as the example described above. The routines use a number of optimizations which are described in Section 5.5. Table 5.1 lists some of the PASSION routines and their function.

5.4.1 Out-of-Core Array Descriptor (OCAD)

The runtime routines require information about the array such as its size, distribution among the nodes of the distributed memory machine, storage pattern etc. All this information is stored in a data structure called the Out-of-Core Array Descriptor (OCAD) and passed as a parameter to the runtime routines. Before any of the runtime routines are called, the compiler or user must fill the necessary information in the OCAD. The structure of the OCAD is given in Figure 5.5. Rows 1 and 2 contain the lower and upper bounds of the in-core local array (excluding overlap area) in each dimension. The lower and upper bounds of the in-core local array in each dimension including overlap area are stored in rows 3 and 4. The size of the global array in each dimension is given in row 5. Row 6 contains the size of the out-of-core local array. Row 7 specifies the number of processors assigned to each dimension of the global array. The format in which the out-of-core local array is stored in the local array file is given in Row 8. The array is stored in the order in which array elements are accessed in the program, so as to reduce the I/O cost. The entry for the dimension which is stored first is set to 1, the entry for the dimension which is stored second is set to 2 and so on. For example, for a two-dimensional array, x,y = 1,2 means that the array is stored on disk in column major order and x,y = 2,1 means that the array is stored in row major order. This information enables the runtime system to determine the location of any array element (i,j) on the disk. Row 9 contains information about the distribution of the global array. Since the array can be distributed as BLOCK(m) or CYCLIC(m), where m is the block-size, the value of m is stored in Row 10 of the OCAD.

5.5 Optimizations

A number of optimizations, such as *data sieving*, *data prefetching* and *data reuse*, have been incorporated in the PASSION Runtime Library.

	Dimension						
	1	2	3	4	5	6	7
Incore lb							
Incore ub							
Incore lbo							
Incore ubo							
Global sz							
OCLA size							
Procs							
OOC storage	x	y					
Distribution							
Block sz							

Figure 5.5: Out-of-Core Array Descriptor (OCAD)

5.5.1 Data Sieving

Studies of file access characteristics by Kotz and Nieuwejaar [74] have shown that many scientific applications actually make strided accesses to the file. Hence, all the PASSION runtime routines for reading or writing data from/to files support the reading/writing of regular sections of arrays with strides. We define a *regular section* of an array as any portion of an array which can be specified in terms of its lower bound, upper bound and stride in each dimension. The need for reading array sections from disks may arise due to a number of reasons, for example `FORALL` or array assignment statements involving sections of out-of-core arrays.

Consider the array of size (11,11) shown in Figure 5.6, which is stored in a file. Suppose it is required to read the section (2:10:2,3:9:2) of this array. The elements to be read are circled in the figure. The interface to any parallel file system does not currently allow a programmer to read or write strided data using a single call. Hence the only direct way of reading the required data is to explicitly move the file pointer to each element and read it individually, which requires as many reads as the number of elements. We call this the *Direct Read Method*. A major disadvantage of this

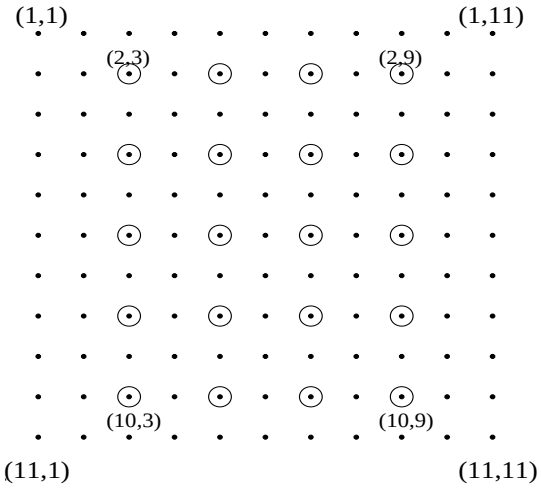

Figure 5.6: Accessing Sections of the OCLA

method is the large number of I/O calls and low granularity of data transfer. Since the I/O latency is very high, this method proves to be very expensive. For example, on the Intel Touchstone Delta using one processor and one disk, it takes 16.06 ms. to read 1024 integers from a file as one block, whereas it takes 1948 ms. to read all of them individually.

Suppose it is required to read a section of a two-dimensional array specified by $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$. The number of array elements in this section is $(\lfloor (u_1 - l_1)/s_1 \rfloor + 1) \times (\lfloor (u_2 - l_2)/s_2 \rfloor + 1)$. Therefore, in the direct read method,

No. of I/O requests = $(\lfloor (u_1 - l_1)/s_1 \rfloor + 1) \times (\lfloor (u_2 - l_2)/s_2 \rfloor + 1)$

No. of array elements read per access = 1

Thus in this method, the number of I/O requests is very high and the number of elements accessed per request is very low, which is undesirable.

We propose a much more efficient method called *Data Sieving* to read or write out-of-core array sections having strides in one or more dimensions. Data sieving can be explained with the help of Figure 5.7. As explained earlier, each processor has an out-of-core local array (OCLA) associated with it. The OCLA is (logically) divided into slabs, each of which can fit in main memory (ICLA). The OCLA shown in the figure has four slabs. Let us assume that it is necessary to read the array section

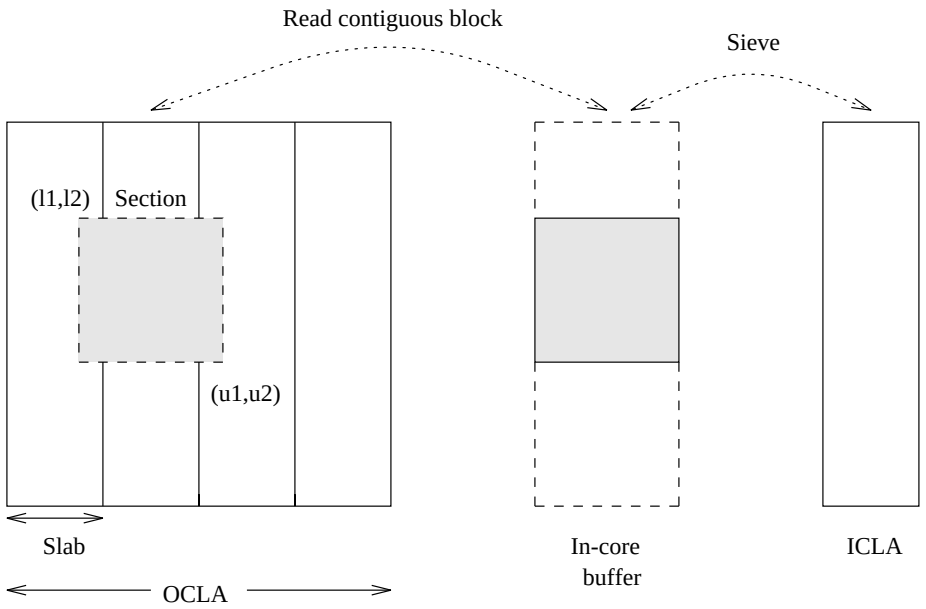

Figure 5.7: Data Sieving

shown in Figure 5.7, specified by $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$, into the ICLA. Although this section spans three slabs of the OCLA, because of the stride all the data elements can fit in the ICLA.

In data sieving, the entire block of data from column l_2 to u_2 if the storage is column major, or the entire block from row l_1 to u_1 if the storage is row major, is read into a temporary buffer in main memory using one read call. The required data is then extracted from this buffer and placed in the ICLA. Hence the name data sieving. A major advantage of this method is that it requires only one I/O call and the rest is data transfer within main memory. The main disadvantage is the high memory requirement. Another disadvantage is the extra amount of data that is read from disk. However, we have found that the savings in the number of I/O calls increases performance considerably. For this method, assuming column major storage,

No. of I/O requests = 1

No. of array elements read per access = $(u_2 - l_2 + 1) \times nrows$

Data sieving is a way of combining multiple I/O requests into one request so as to

reduce the effect of high I/O latency time. A similar method called *message coalescing* is used in interprocessor communication, where small messages are combined into a single large message in order to reduce the effect of communication latency. However, data sieving is different because instead of coalescing the required data elements together, it actually reads even unwanted data elements so that large contiguous blocks are read. The useful data is then filtered out by the runtime system in an intermediate step and passed on to the program. The unwanted data read into main memory is dynamically discarded.

Data sieving can also be considered from the following perspective. In the direct method, even though a separate read call is required for each individual element, it may not result in a disk access each time. There is usually some form of caching done at the I/O nodes and if the requested data lies in the cache, it can be read from the cache itself. In spite of this, we find that reading individual elements is a lot more expensive than reading one large chunk. This is probably because of the overhead of making several requests to the I/O nodes and looking up the software cache at the I/O node each time. Data sieving can be considered as a way of doing software caching in user space at the compute node itself. An entire chunk of data starting from the first element in the section is effectively cached at the compute node and all the required elements are supplied from this cache. The file system sees only a single request or at most a few requests, depending on the amount of memory available for this cache.

In the future when file systems support a strided interface, data sieving can also be implemented at the file system level. The I/O processors can read large chunks of data from the file, perform sieving, and send only the required data to the compute processors.

Reducing the Memory Requirement

If the stride in the array section is large or the number of rows in the section is small compared to the total number of rows in the out-of-core array, the amount of memory required to read the entire block from column l_2 to u_2 may be quite large.

There may not be enough main memory available to store this entire block. Hence instead of reading the entire section in a single call, a smaller subsection is fetched from the file, depending on the amount of memory available. Sieving is performed on this subsection, then the next subsection is read, sieving is performed on it, and so on. This reduces the memory requirements of the program considerably and increases the number of I/O requests only slightly. Let us assume that the array is stored in column major order and n columns of the OCLA can fit in main memory. Then for this case

$$\text{No. of I/O requests} = \lceil (u_2 - l_2 + 1)/n \rceil$$

$$\text{No. of array elements read per access} = n \times nrows$$

Writing Array Sections

Suppose it is required to *write* an array section $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ from the ICLA to the LAF. The issues involved here are similar to those described above for reading array sections. A *Direct Write Method* can be used to write each element individually, but it suffers from the same problems of large number of I/O requests and low granularity of data transfer. To reduce the number of I/O requests, a method similar to the data sieving method described above needs to be used. If we directly use data sieving in the reverse direction, i.e. elements from the ICLA are placed at appropriate locations in a temporary buffer with stride, and the buffer is written to disk, the data in the buffer between the strided elements will overwrite the corresponding data elements on disk. In order to maintain data consistency, it is necessary to first read the entire subsection from the LAF into the temporary buffer. Then, data elements from the ICLA can be stored at appropriate locations in the buffer and the entire buffer can be written back to disk. This is similar to what happens in cache memories when there is a write miss. In that case, a whole line or block of data is fetched from main memory into the cache and then the processor writes data into the cache. Thus, writing sections using sieving requires twice the amount of I/O compared to reading sections, because for each write to disk the corresponding block has to first be fetched into memory. Therefore, for writing array sections

$$\text{No. of I/O requests} = 2\lceil (u_2 - l_2 + 1)/n \rceil$$
$$\text{No. of array elements transferred per access} = n \times nrows$$

Performance

Table 5.2 gives the performance of data sieving versus the direct method for reading and writing array sections. An array of size 2K $\times$ 2K is distributed among 64 processors in one dimension along columns. So each processor's local array file consists of an array of size 2K $\times$ 32 stored in column major order. Each processor needs to read/write a section from/to its local array file. We measured the time taken by the `PASSION_read_section()` and `PASSION_write_section()` routines for reading and writing sections of the out-of-core local array on each processor. We observe that data sieving provides tremendous improvement over the direct method in all cases. The reason for this is large number of I/O requests in the direct method, even though the total amount of data accessed is higher in data sieving.

Table 5.3 gives the number of I/O requests and the total amount of data transferred for each of the array sections considered in Table 5.2. We observe that in the data sieving method, the number of data elements transferred is more or less the same for all cases. This is because the total amount of data transferred depends only on the lower and upper bounds of the section and is independent of the stride. Hence the time taken using data sieving does not vary much for all the sections we have considered. However, there is a wide variation in time for the direct method, because only those elements belonging to the section are read. The time is lower for small sections and higher for large sections.

We observe that even for writing array sections, data sieving performs better than direct write even though it requires reading the section before writing. As expected, `PASSION_write_section()` takes about twice the time as `PASSION_read_section()` when using data sieving. All PASSION routines involving array sections use data sieving for greater efficiency.

Table 5.2: Performance of Direct Read/Write versus Data Sieving (time in sec.)

Global array size 2K × 2K real nos. (single precision), 64 processors,
OCLA size 2K × 32, slab size = 16 columns

	PASSION_read_section		PASSION_write_section	
Array Section	Direct Read	Sieving	Direct Write	Sieving
(1:2048:2, 1:32:2)	52.95	1.970	49.96	5.114
(1:2048:4, 1:32:4)	14.03	1.925	13.71	5.033
(10:1024:3, 3:22:3)	8.070	1.352	7.551	4.825
(100:2048:6, 5:32:4)	7.881	1.606	7.293	4.756
(1024:2048:2, 1:32:3)	18.43	1.745	17.98	5.290

Table 5.3: I/O requirements of Direct Read and Data Sieving Methods

Global array size 2K × 2K real nos. (single precision), 64 processors,
OCLA size 2K × 32, slab size = 16 columns

	No. of I/O requests		No. of array elements read	
Array Section	Direct Read	Sieving	Direct Read	Sieving
(1:2048:2, 1:32:2)	16384	2	16384	65536
(1:2048:4, 1:32:4)	4096	2	4096	65536
(10:1024:3, 3:22:3)	2373	2	2373	40960
(100:2048:6, 5:32:4)	2275	2	2275	57344
(1024:2048:2, 1:32:3)	5643	2	5643	65536

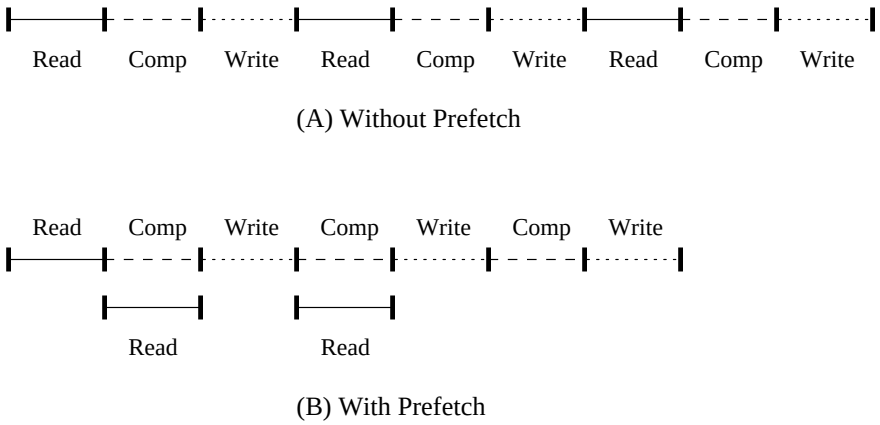

Figure 5.8: Data Prefetching

5.5.2 Data Prefetching

In the model of computation and I/O described earlier, the OCLA is divided into a number of *slabs*, each of which can fit in the ICLA. Program execution proceeds as follows:- a slab of data is fetched from the LAF to the ICLA; the computation is performed on this slab and the slab is written back to the LAF. This is repeated on other slabs till the end of the program. Thus I/O and computation form distinct phases in the program. A processor has to wait while each slab is being read or written as there is no overlap between computation and I/O. This is illustrated in Figure 5.8(A) which shows the time taken for computation and I/O on 3 slabs. For simplicity, reading, writing and computation are shown to take the same amount of time, which may not be true in certain cases.

The time taken by the program can be reduced if it is possible to overlap computation with I/O in some fashion. A simple way of achieving this is to issue a non-blocking I/O read request for the next slab immediately after the current slab has been read. This is called *Data Prefetching*. Kotz and Ellis [73] discuss in detail the effects of prefetching in parallel file systems. Since the read request is non-blocking, the reading of the next slab can be overlapped with the computation being performed on the current slab. If the computation time is comparable to the I/O time, this can

result in significant performance improvement. Figure 5.8(B) shows how prefetching can reduce the time taken for the example in Figure 5.8(A). Since the computation time is assumed to be the same as the read time, all reads other than the first one get overlapped with computation. The total reduction in program time is equal to the time for reading two slabs, as only two of the three reads can be overlapped in this example.

Note that the parallel file system may do some prefetching on its own. For example, for each read request on the Delta, the CFS automatically prefetches the next seven blocks of the file into the I/O node, resulting in a total of at least eight blocks being read. For many applications, the prefetching done at the file system level may not be sufficient and could even degrade performance, unless the user can control the prefetching policy. This is because different applications have different access patterns which may not match well with the prefetching policy of the file system. Most of the parallel file systems at present do not allow the user to control prefetching. On the other hand, the prefetching done by PASSION is controlled by the user. Since the user knows what data is needed next, it can be prefetched correctly. Prefetching can be done in PASSION using the routine `PASSION_read_prefetch()` and the routine `PASSION_prefetch_wait()` can be used to wait for the prefetch to complete.

Performance

We use an out-of-core median filtering program to illustrate the performance of data prefetching. Median filtering is frequently used in computer vision and image processing applications to smooth the input image. Each pixel is assigned the median of the values of its neighbors within a window of a particular size, say 3×3 or 5×5 or larger. We have implemented a parallel out-of-core median filtering program using PASSION runtime routines for I/O and communication. The image is distributed among processors in one dimension along columns and stored in local array files. Depending on the window size, each processor needs a few columns from its right and left neighbors. This requires a shift type communication which is implemented using the routine `PASSION_oc_shift()`.

Table 5.4: Performance of Median Filtering using 3×3 window (time in sec.)

	4 slabs		8 slabs		16 slabs	
Procs	Prefetch	No Prefetch	Prefetch	No Prefetch	Prefetch	No Prefetch
4	36.37	46.56	33.63	46.75	30.65	47.21
8	18.32	23.37	16.72	24.41	16.36	24.86
16	9.180	12.33	8.730	12.60	8.580	13.35
32	5.340	6.830	5.260	7.000	5.080	7.160
64	5.650	5.850	4.970	5.970	5.410	6.230

Table 5.5: Performance of Median Filtering using 5×5 window (time in sec.)

	4 slabs		8 slabs		16 slabs	
Procs	Prefetch	No Prefetch	Prefetch	No Prefetch	Prefetch	No Prefetch
4	81.47	94.09	79.25	95.63	78.58	96.88
8	41.81	47.76	41.35	49.32	41.01	50.59
16	21.57	25.41	21.40	27.28	21.74	27.81
32	11.36	12.83	11.40	13.64	11.43	14.81
64	7.110	9.010	6.810	9.110	8.090	9.197

Tables 5.4 and 5.5 show the performance of median filtering on the Intel Touchstone Delta for windows of size 3×3 and 5×5 respectively. The image is of size 2K $\times$ 2K pixels. We assume this to be out-of-core for the purpose of experimentation. The number of processors is varied from 4 to 64 and the size of the ICLA is varied in each case in such a way that the number of slabs varies from 4 to 16. Since the Touchstone Delta has 64 disks, each processor's LAF can be stored on a separate disk.

The following observations can be made from these tables:-

1. In all cases, prefetching improves performance considerably. In some cases, the improvement is close to 40%. Figures 5.9 and 5.10 show the relative performance with and without prefetching when the number of slabs is 8.

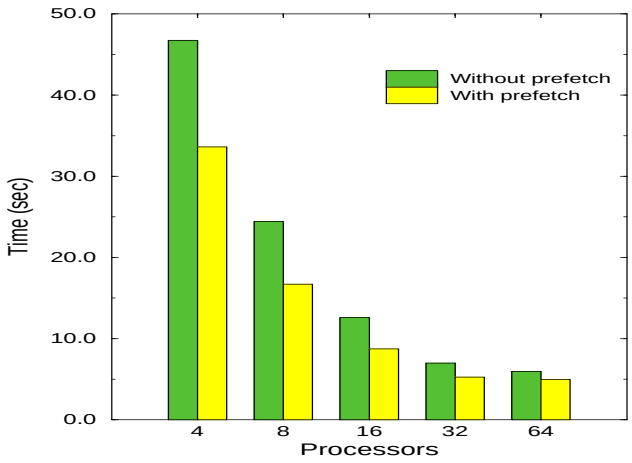

Figure 5.9: Median Filtering using 3×3 window

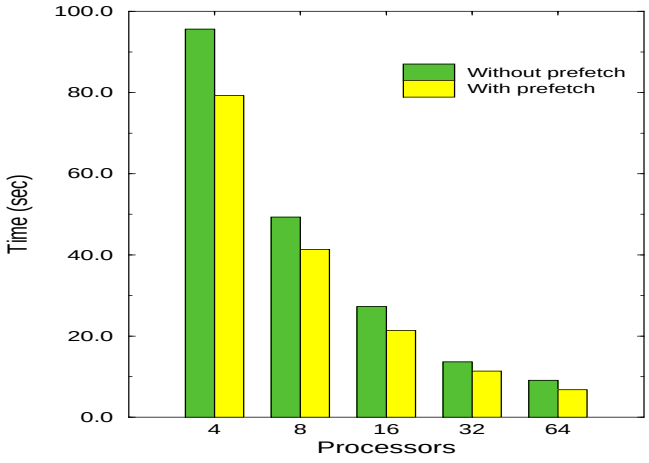

Figure 5.10: Median Filtering using 5×5 window

2. Without prefetching, as the number of slabs is increased, the time taken increases. This is because a larger number of slabs means a smaller slab size which results in a larger number of I/O requests.

3. With prefetching, as the number of slabs in increased, the time taken decreases in most cases. Since the first slab can never be prefetched, all processors have to wait for the first slab to be read. As the slab size is reduced, the wait time for the first slab is also reduced and there is more overlap of computation and I/O. However, the number of I/O requests increases. When the slab size is large, a reduction in the slab size by half improves performance because the saving in the wait time for the first slab is higher than the increase in time due to the larger number of I/O requests. But when the slab size is small (64 processor case with 8 or 16 slabs), the higher number of I/O requests costs more than the decrease in wait time for the first slab. Hence the performance actually degrades in this case.

5.5.3 Data Reuse

In out-of-core programs, data is fetched from files many times. If a portion of the data currently fetched into memory is also needed for the computation on the next data set, then that portion already in memory can be reused instead of reading it again with the next data set. For example, consider the Laplace equation solver discussed earlier (Figure 5.3). Suppose the array is distributed along columns. Then the computation of each column requires one column from the left and one column from the right. The computation of the last column requires one column from the overlap area and the computation of the column in the overlap area cannot be performed without reading the next column from the disk. Hence, instead of reading the column in the overlap area again with the next set of columns, it can be reused by moving it to the first column of the array and the last column can be moved to the overlap area before the first column (Figure 5.11). If this move is not done, it would be necessary to read the two columns again from the disk along with data for the next slab. The reuse thus

eliminates the reading of two columns in this example. In general, the amount of data reuse would depend on the intersection of the sets of data needed for computations involving two consecutive slabs. The PASSION routine `PASSION_read_reuse()` can be used to perform data reuse.

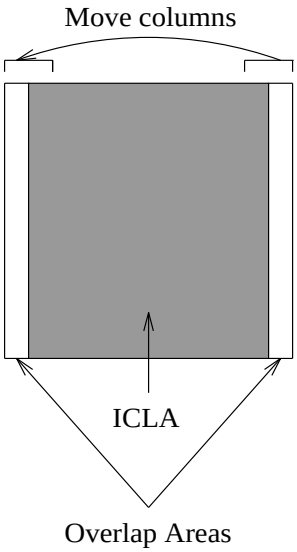

Figure 5.11: Data Reuse

Table 5.6: Performance of Data Reuse

Laplace Equation solver, 64 processors

Array size	Time in sec.	
	Without Reuse	With Reuse
2K × 2K	75.12	71.71
4K × 4K	274.7	269.1

Table 5.6 shows the performance improvement obtained by using data reuse for the Laplace Equation Solver program on the Intel Touchstone Delta using 64 processors. Two different global array sizes are used — 2K × 2K and 4K × 4K. Reuse provides good performance improvement even in this case where only two columns can be reused.

Chapter 6

Runtime Support for Out-of-Core Programs: (II) Collective I/O

6.1 Introduction

Chapter 5 discusses runtime support for the Local Placement Model where there is a separate local array file for each processor. Each processor can directly access only its own local array file. However, the situation is quite different in the Global Placement and Partitioned In-Core Models. In these two models, there is a single global array file containing the entire out-of-core array. Each processor accesses the data it needs from this common file. A processor may, in general, need to access any arbitrary section of the global array, with or without stride. The global array may be stored in the global array file in either row-major or column-major order. As a result, the data required by each processor may not be located contiguously in the file. Also, the requests of some processors may overlap. In the extreme case, all processors may want to access the same section of the array. If each processor directly tries to read the data it needs, it may result in a large number of low granularity requests and multiple requests for the same data.

In loosely synchronous SPMD programs, all processors perform similar operations but on different data sets [46]. Hence, if one processor needs to read data from disks, it is very likely that a group of processors or maybe all processors need to read data from disks at about the same time. This makes it possible for the requesting

processors to cooperate in reading or writing out-of-core data in an efficient manner, which is known as *collective I/O*. This chapter proposes a method, called the Extended Two-Phase Method, which uses collective I/O for accessing sections of out-of-core arrays efficiently. PASSION provides two routines, `PASSION_global_read()` and `PASSION_global_write()`, to read/write arbitrary array sections with strides from/to a global array file, using the Extended Two-Phase Method.

6.2 Need for Collective I/O

The need for collective I/O can be explained using the following example. Consider the large out-of-core array shown in Figure 6.1 and assume that it is stored in a file in column-major order. Four processors ($0 - 3$) need to access a block of rows each as shown. Since the access requests are orthogonal to the file storage order, the data requested by each processor is located non-contiguously in the file. Also, the requests of different processors lie interleaved in the file. A portion of 0's request lies at the start of the file, followed by some unwanted data (gap), then a portion of 1's request followed by a gap, then a portion of 2's request followed by a gap, then a portion of 3's request followed by a gap, then again a portion of 0's request, and so on. To read the data using the interface provided by most of the existing parallel file systems, each processor has to explicitly seek to the appropriate location in the file, read the small chunk of data, then seek to the next location, and so on. We call this the *Direct Method*. The Vesta file system [29] and the nCUBE file system [36] do provide support for the user to specify a logical view of the data to be read, and use a single call to read data. But each processor's request is serviced independently and there is no collective optimization based on the requests of all processors.

The drawback of the Direct Method is a large number of low granularity requests which may arrive from different processors in any order. Since I/O latency is very high, this usually results in poor performance. The basic premise behind collective I/O is that if a group of processors need to read/write data at the same time, they

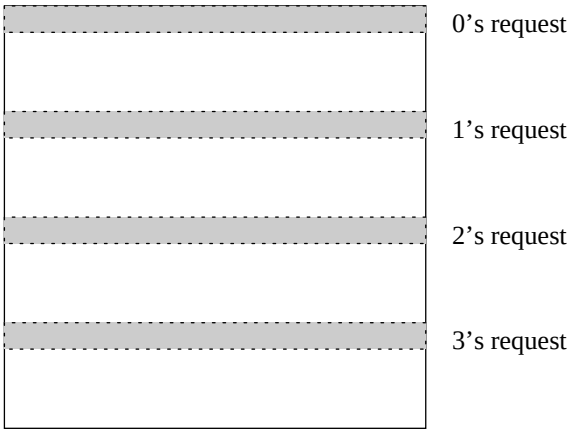

Figure 6.1: Accessing row blocks of a file stored in column-major order

can cooperate among themselves to perform I/O efficiently in large chunks and in the right order. This is usually possible in loosely synchronous SPMD programs in which all processors perform similar operations on different data sets. Hence, a group of processors may need to perform I/O at the same time. An appropriate interface needs to be provided for the user to specify a collective I/O request. The PASSION Runtime Library provides such an interface [26].

Given an appropriate collective I/O interface, the collective I/O operation can be implemented either as a library on top of the file system, or at the file system level itself. A technique called Two-Phase I/O [37, 12] has been proposed for doing collective I/O at the library level. In this method, I/O is done in two phases. In the first phase, processors cooperate to read data in large chunks, and in the second phase they do an in-core redistribution of the data. Disk-directed I/O [69] is a technique which proposes to do collective I/O at the file system level. In disk-directed I/O, a collective I/O request is sent to all I/O nodes which determine the order and timing of the flow of data.

6.3 Extended Two-Phase Method for Collective I/O

The Two-Phase Method [37, 12] is a collective I/O technique for reading an entire in-core array from a file into a distributed array in main memory, and conversely to write a distributed in-core array to a file. I/O is done in two phases. In the first phase, processors always read data assuming a *conforming distribution*. A conforming distribution is defined as a distribution of an array among processors such that each processor's local array lies contiguously in the file. This results in each processor reading a single large chunk of data. For an array stored in a file in column-major order, a column-block distribution is the conforming distribution. In the second phase, data is redistributed among processors to whatever is the desired distribution. Since I/O cost is orders of magnitude more than communication cost, the cost incurred by the second phase is negligible. This Two-Phase approach is found to give consistently good performance for all distributions [37, 12].

We have extended the basic Two-Phase Method to access arbitrary sections of out-of-core arrays. This Extended Two-Phase Method performs I/O for out-of-core arrays efficiently by

- partitioning the I/O workload among processors dynamically, depending on the access requests,
- combining several I/O requests into fewer larger granularity requests,
- reordering requests so that data is accessed in proper sequence,
- eliminating multiple disk accesses for the same data, and
- reducing contention for disks.

6.3.1 Reading Sections of Out-of-Core Arrays

We first describe the Extended Two-Phase Method for reading regular sections of out-of-core arrays. We define a regular section of an array as a section which can

be specified by a lower-bound, upper-bound and stride in each dimension. For the purpose of explanation, we consider the case where each processor needs to read some regular section of a two-dimensional array stored in the file in *column-major order*. The Extended Two-Phase Method can actually be used for arrays with any number of dimensions, stored in any order in the file and accessed by any number of processors. Section 6.6 discusses how the general case can be implemented.

In the Extended Two-Phase Method, the I/O workload is divided among processors. For this, we assign *ownership* to portions of the file such that a processor can directly access only the portion of the file it *owns*. The file is effectively logically divided into *domains*. The portion of the file which a processor can directly access is called its *File Domain (FD)*. For a file stored in column-major order, the file domain of each processor is some set of columns of the array. The issue of how to select file domains is important because it determines how the I/O workload gets divided among processors. This is discussed in detail in Section 6.4.

Let us assume that each processor needs to read some regular section $(l_1 : u_1 : s_1, l_2 : u_2 : s_2)$ of the array in global coordinates. In the first step of the Extended Two-Phase Method, processors exchange their own access information (the indices $l_1, u_1, s_1, l_2, u_2, s_2$) with other processors, so that each processor knows the access requests of other processors. This information is stored in a data structure called the *File Access Descriptor (FAD)*. The FAD contains exactly the same information on all processors. This exchange phase is not required if the collective I/O interface itself provides information about other processors' access requests.

Since each processor knows its own file domain and the access requests of other processors, it can determine what portion of the data in its file domain is needed by other processors. This is done by computing the intersection of the requests of other processors from the FAD and its own file domain. This information is stored in a data structure called the *File Domain Access Table (FDAT)*. Thus the FDAT of a processor contains information about which portions of its file domain have been requested by other processors.

Each processor now has to read data from its file domain, as determined by the

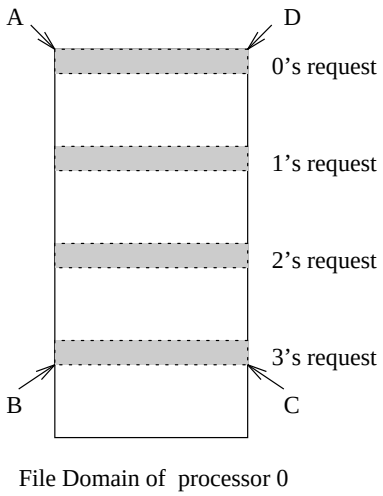

Figure 6.2: The requests in processor 0's file domain

FDAT. For example, Figure 6.2 shows the file domain of processor 0 and, for some access pattern, the portions of this file domain which have been requested by other processors. A simple way of reading would be to read all the data needed by processor 0, followed by that needed by processor 1 and so on in order of processor number. But, as in the case of Figure 6.2, this may result in too many small accesses which are not in sequence. For the read to be done efficiently, it is important that the FDAT be analyzed so that the file is accessed in sequence and contiguously, as far as possible.

We have devised a very general method for analyzing the information in the FDAT, which ensures that the file is read contiguously and in sequence. Each processor calculates the minimum of the lower-bounds and the maximum of the upper-bounds of all sections in its FDAT. This effectively determines the smallest section which contains all the data that needs to be read from the file domain (for example, section ABCD in Figure 6.2). This section may also contain some data which is not required by any processor. If the processor tries to read only the useful data, it may result in a number of small strided accesses. To avoid this, it uses the optimization *data sieving* described in Chapter 5. The processor reads a column of the section at a time in a single operation into a temporary buffer. This may include some unwanted data. The useful data is extracted from the temporary buffer and placed in communication

buffers depending on which processors need the data. The entire section is read from the file domain in this fashion. It is possible to read more than one column at a time if there is enough temporary memory available to do sieving on the set of columns. This forms the first phase of the Extended Two-Phase Method.

The second phase of the Extended Two-Phase Method consists of communicating the data read in the first phase to the respective processors. The information in the FDAT is sufficient for each processor to determine what data has to be sent to which processor. Since each processor knows the file domains of other processors and its own access request, it can calculate how much data it needs to receive from other processors, as well as the locations in main memory where the received data needs to be placed.

The two phases of the Extended Two-Phase Method can either be done distinctly by performing all the I/O first and then the communication, or they can be overlapped (pipelined) by reading smaller portions of data and communicating it.

6.3.2 Writing Sections of Out-of-Core Arrays

The algorithm for writing sections is essentially the reverse of the algorithm for reading sections. From the information in the File Access Descriptor (FAD), each processor can determine what portion of the section it needs to write lies in the file domains of other processors. Each processor also computes its own File Domain Access Table (FDAT), which indicates how much data it needs to receive from other processors, to be written to its file domain. The first phase of the Extended Two-Phase Method for writing is to perform communication to get this data from other processors.

The second phase is to write the data to the file in sequence and contiguously. The FDAT is analyzed in the same way as in the read algorithm. Each processor determines the minimum and maximum of all indices in its FDAT. This effectively determines the smallest section which includes all the data that needs to be written to the file domain. It may also include some data which is not being written by any processor. The processor writes the useful data in this section one column at a time using data sieving. However, for writing using data sieving, we cannot directly use

the reverse of the method used for reading. If the useful data is placed at appropriate locations (possibly with stride) in a temporary buffer and the buffer is written to the file, the contents of the buffer between the useful data elements will overwrite the data in the file. To maintain data consistency, it is necessary to first read the entire column from the file into the temporary buffer. Then, the data elements to be written in that column can be stored at appropriate locations in the buffer and the entire column can be written back to disk. Thus, writing sections requires twice the amount of I/O compared to reading sections, because to write each column, the corresponding column has to first be read into memory. It is possible to avoid this extra reading in cases where the entire column contains useful data to be written. This requires each processor to do a more extensive analysis of the FDAT, to make sure that there are no "holes" between the data sets being written by different processors in the same column, and also no strides in the section being written by each processor. As in the case of reading sections, it is possible to do sieving with more than one column at a time if there is enough temporary memory available.

If the sections requested to be written by different processors have some elements in common, there is a data consistency problem. The result depends on how the Extended Two-Phase Method has been implemented. In our implementation, if there are write requests from multiple processors to the same location, the data from the highest numbered processor is written to the file.

6.4 Partitioning I/O Among Processors

In the Extended Two-Phase Method, processors cooperate to perform I/O. The exact partitioning of the I/O workload among processors depends on how file domains are defined. In general, the partitioning of I/O can be done either statically or dynamically.

6.4.1 Static Partitioning

One way of partitioning I/O is to assign a block of columns of the entire out-of-core array to each processor, as if the array were distributed among processors in a column block fashion. Thus the file domain of each processor is a block of columns of the array, which is stored contiguously in the file. The file domains are predefined and fixed. The size of each domain can be determined from the size of the array and the number of processors, and is independent of the access requests. This is called a static partitioning of I/O among processors. Note that this is just a logical partitioning of the file among processors, so that each processor directly accesses only a particular portion of the file (its file domain). Figure 6.3(A) shows the file domains with static partitioning when there are four processors.

6.4.2 Dynamic Partitioning

The main drawback of static partitioning is that the partitioning is independent of the type of access requests. For many types of access patterns, this may result in an imbalance of I/O among processors. Some processors may perform more I/O than others and some may not perform any I/O at all. For example, consider the access pattern in Figure 6.3. If the partitioning is done statically, the access requests span the file domains of only two processors. Thus only two processors (1 and 2) perform all the I/O; the remaining two processors (0 and 3) only wait to receive data sent by 1 and 2. Another drawback of static partitioning is that as the size of the out-of-core array is increased keeping the number of processors fixed, the size of each file domain also increases. Hence, for the same access pattern, as the size of the out-of-core array is increased, the access requests span the file domains of fewer processors resulting in greater imbalance of I/O and lower performance.

The I/O throughput can potentially be improved by partitioning the I/O workload among processors dynamically, depending on the array sections requested. This is illustrated in Figure 6.3(B). For a file stored in column-major order, each processor calculates the lowest and highest among the columns of the sections requested by

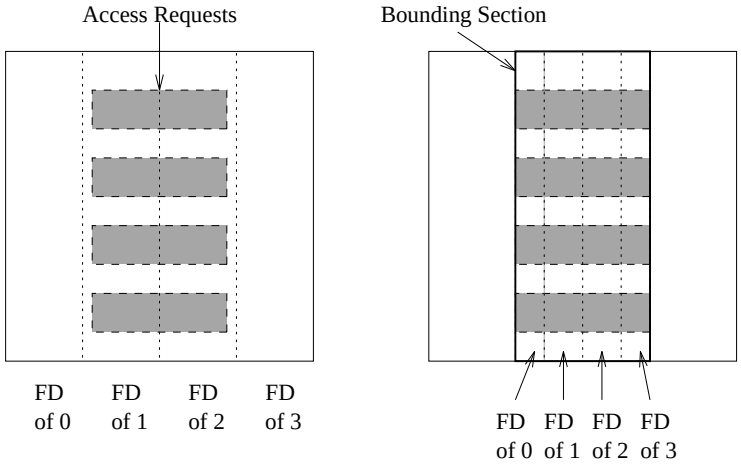

Figure 6.3: Static versus Dynamic Partitioning

all processors. The section formed by these columns and all the rows of the out-of-core array is called the *bounding section*. The bounding section includes the sections requested by all processors and it lies contiguously in the file. Figure 6.3(B) shows the bounding section for the given access requests. File domains are determined by dividing the bounding section among the processors in a column-block fashion. Thus the file domain of each processor is a contiguous chunk of the bounding section.

If the requested sections span all the columns of the out-of-core array, the dynamically selected file domains are exactly the same as those determined statically. But if the sections span only a few columns, as in Figure 6.3, dynamic partitioning provides a much better balance of I/O among processors. The memory requirements of the Extended Two-Phase Method are also reduced, because the file domain of each processor is smaller. In the static case, if all requested sections lie in a single processor's file domain, all the requested data may not fit in the memory of that processor. Hence I/O and communication may need to be done in stages several times. This is less likely to occur in the case of dynamic partitioning, because the requested data is more evenly divided among processors.

1. Exchange access information with other processors and fill in the File Access Descriptor (FAD).
2. Determine the smallest section which includes the sections requested by all processors, called the bounding section.
3. The file domain of each processor is determined by dividing this bounding section among the processors in a column-block manner for arrays stored in column-major order.
4. Compute the intersection of the FAD and this processor's file domain, and fill in the File Domain Access Table (FDAT).
5. Calculate the minimum of the lower-bounds and the maximum of the upper-bounds of all sections in the FDAT to determine the smallest section containing all the data needed from the file domain.
6. Read this section using data sieving and communicate the data to the requesting processors.

Figure 6.4: Extended Two-Phase Method for Reading Sections of Out-of-Core Arrays

The algorithm for reading sections of out-of-core arrays using the Extended Two-Phase Method with dynamic partitioning is given in Figure 6.4. If the array is stored in the file in row-major order instead of column-major order, the only difference would be that the file domains would be defined in terms of row-blocks and data sieving would be done along rows.

6.5 Performance

We extensively tested the performance of the Extended Two-Phase Method versus the Direct Method on the Intel Touchstone Delta, for many different access patterns,

using both static and dynamic partitioning. The access patterns can be classified into three basic types:-

1. *Common Sections*: All processors need to access exactly the same section of the array.

2. *Overlapping Sections*: Parts of the section requested by a processor may overlap with parts of the sections requested by other processors.

3. *Distinct Sections*: The section requested by each processor does not have any data in common with the section requested by any other processor.

6.5.1 Reading Common Sections

Table 6.1 compares the performance of the Extended Two-Phase Method and the Direct Method for reading common sections. The array size is 4K $\times$ 4K and the number of processors is 16. Figure 6.5 shows approximately where each of these sections is located in the array. The performance of the Extended Two-Phase Method was measured using both static and dynamic partitioning. We observe that in all cases, the Extended Two-Phase Method performs considerably better than the Direct Method. This is primarily because, in the Extended Two-Phase Method, the common section is read only once and then broadcast to other processors, whereas in the Direct Method, all processors simultaneously try to read the same section from the file, resulting in extra I/O overhead.

In all cases, the Extended Two-Phase Method takes a lot less time with dynamic partitioning than with static partitioning. In the case of static partitioning, for a 4K $\times$ 4K array with 16 processors, each processor's file domain is of size 4K $\times$ 256. Thus all sections except V lie in the file domains of only a few processors. But with dynamic partitioning, each section is evenly divided among all available processors, resulting in higher I/O throughput. Since Section V spans all 4096 columns, the statically and dynamically selected file domains are identical, resulting in identical performance.

Table 6.1: Time (sec.) for Reading Common Sections

Array size 4K × 4K real nos. (single precision), 16 processors

No.	Array Section	Direct Read	Extended Two-Phase	
			Static	Dynamic
I	(1:100:1, 1:100:1)	1.632	1.027	0.431
II	(200:300:1, 200:300:1)	1.867	0.883	0.363
III	(400:800:1, 400:800:1)	6.265	3.692	1.056
IV	(32:64:1, 128:1024:1)	9.995	2.780	1.318
V	(1:16:1, 1:4096:1)	52.06	3.241	3.241
VI	(1:4096:1, 1:16:1)	1.216	2.024	0.420

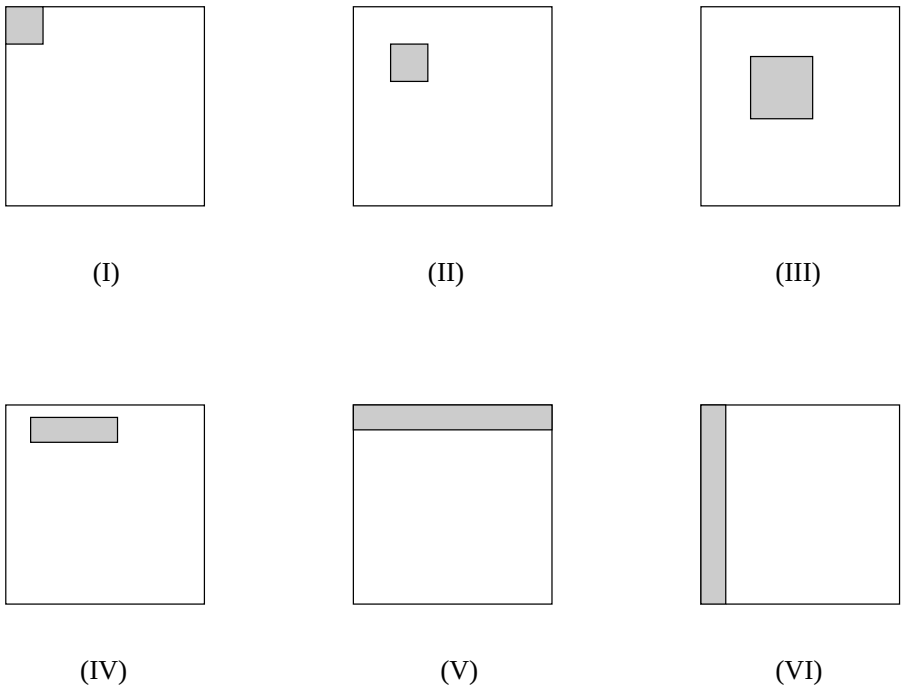

Figure 6.5: The common sections listed in Table 6.1 (not to scale)

In cases where the sections span a large number of columns (e.g. Section V), the Extended Two-Phase Method provides a significant improvement over the Direct Method. This is because for such cases, the Direct Method results in a large number of small requests spread across the entire file. In the Extended Two-Phase Method, the I/O gets evenly divided among all processors, the requests are reordered and data is read in large chunks.

6.5.2 Reading Overlapping Sections

Table 6.2 compares the performance of the Extended Two-Phase Method and the Direct Method for reading overlapping sections. Figure 6.6 shows approximately where these sections are located in the array. To represent these overlapping sections for all processors concisely, we use the following notation. Each processor's request is denoted by $(l_1 + ov1 \times p : u_1 + ov1 \times p : s_1, l_2 + ov2 \times p : u_2 + ov2 \times p : s_2)$, where p is the processor number and $ov1$, $ov2$ are some constants. The amount of overlap can be changed by varying $ov1$ and $ov2$. For example, the notation (1:100:1, 1+10p:100+10p:1) in row I of Table 6.2 represents a group of overlapping sections with processor 0 requesting section (1:100:1, 1:100:1), processor 1 requesting section (1:100:1, 11:110:1), processor 2 requesting section (1:100:1, 21:120:1) and so on. The sections in cases I — IV overlap along columns, the sections in cases V — VII overlap along rows and the sections in case VIII overlap in both dimensions.

We observe that the Extended Two-Phase Method with dynamic partitioning performs the best in all cases. The sections in cases I and II are of the same size, but they differ in the amount of overlap. The sections in case I have more overlap than those in case II. Since the total number of columns of the out-of-core array spanned by the sections in case I is less than that by the sections in case II, it takes less time to read the sections in case I. Sections in cases IV, V and VI span only a few columns. For these cases, the Direct Method performs better than the Extended Two-Phase Method with static partitioning. This is because static partitioning results in only a few processors performing I/O. But the Extended Two-Phase Method with

Table 6.2: Time (sec.) for Reading Overlapping Sections

Array size 4K × 4K real nos. (single precision), 16 processors

No.	Array Section (p = processor number)	Direct Read	Extended Two-Phase	
			Static	Dynamic
I	(1:100:1, 1+10p:100+10p:1)	2.000	1.830	0.693
II	(1:100:1, 1+50p:100+50p:1)	4.627	1.859	0.875
III	(400:800:1, 400+100p:800+100p:1)	8.097	3.348	2.477
IV	(1:4096:1, 1+8p:16+8p:1)	1.152	3.374	0.826
V	(1+50p:100+50p:1, 1:100:1)	1.579	1.994	0.524
VI	(400+100p:800+100p:1, 400:800:1)	7.442	11.84	1.361
VII	(1+8p:16+8p:1, 1:4096:1)	50.32	2.992	2.992
VIII	(200+100p:400+100p:1, 200+100p:400+100p:1)	3.104	2.986	1.739

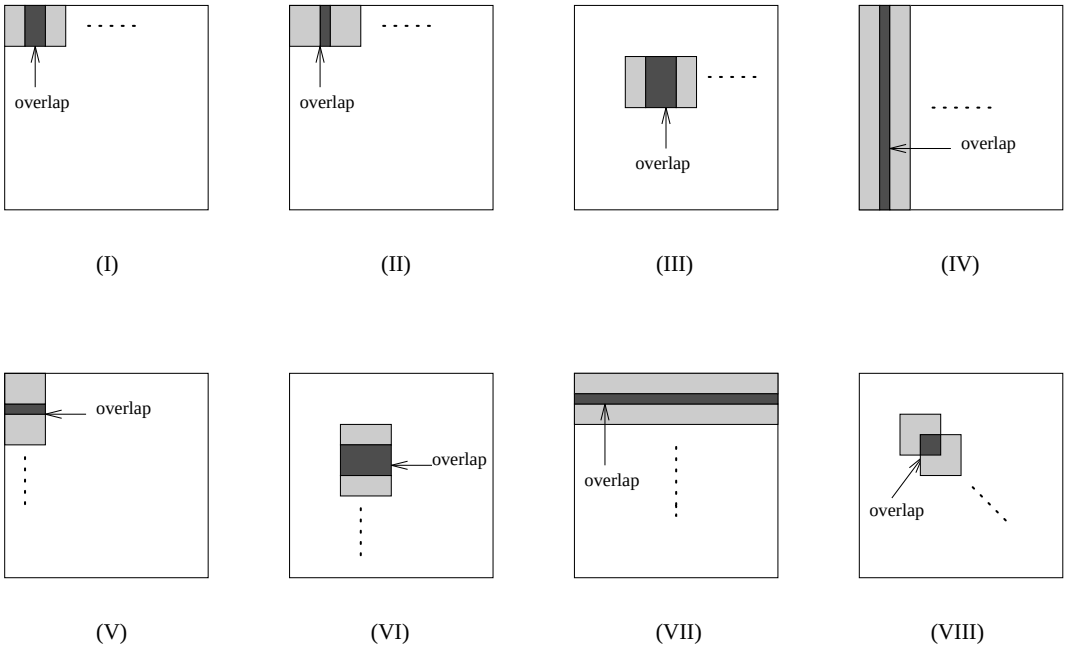

Figure 6.6: The overlapping sections listed in Table 6.2 (not to scale)

dynamic partitioning performs better than the Direct Method, since there is a better distribution of I/O among processors. The sections in case VII span all columns of the array, which is the worst case for the Direct Method. Finally, in case VIII, the sections have overlap in both dimensions and again the Extended Two-Phase Method with dynamic partitioning takes the least time.

6.5.3 Reading Distinct Sections

Table 6.3 compares the performance of the Extended Two-Phase Method and the Direct Method for reading distinct sections. Figure 6.7 shows approximately where these sections are located in the array. We use the same notation as above, $(l_1 + ov1 \times p : u_1 + ov1 \times p : s_1, l_2 + ov2 \times p : u_2 + ov2 \times p : s_2)$, for representing distinct sections. The overlap factors $ov1$ and $ov2$ need to be large enough to ensure that the sections are distinct.

In all cases, the Extended Two-Phase Method with dynamic partitioning performs the best. The relative performance of the three methods is similar to that for overlapping sections in Table 6.2. The sections in case I are located along rows, so the requests of different processors lie in separate locations in the file. Hence the Extended Two-Phase Method performs only slightly better. The sections in cases II — IV are located along columns and so the requests of different processors lie interleaved in the file. Hence the Extended Two-Phase Method performs considerably better. Static partitioning does not give good performance for the sections in case II because they span only a few columns. The best case for the Extended Two-Phase Method is case IV since the sections span all the columns. The sections in cases V and VI lie partly interleaved in the file. Even for these cases, the Extended Two-Phase Method performs the best.

Table 6.3: Time (sec.) for Reading Distinct Sections

Array size 4K × 4K real nos. (single precision), 16 processors

No.	Array Section (p = processor number)	Direct Read	Extended Two-Phase	
			Static	Dynamic
I	(1:100:1, 1+100p:100+100p:1)	1.976	1.954	1.304
II	(1+100p:100+100p:1, 1:100:1)	1.633	2.182	0.548
III	(200+200p:400+200p:1, 512:1024:1)	8.016	5.680	1.725
IV	(1+32p:16+32p:1, 1:4096:1)	51.63	4.823	4.823
V	(200+200p:400+200p:1, 1+200p:512+200p:1)	5.466	4.524	3.912
VI	(1+32p:32+32p:1, 1+100p:1024+100p:1)	12.02	2.991	2.371

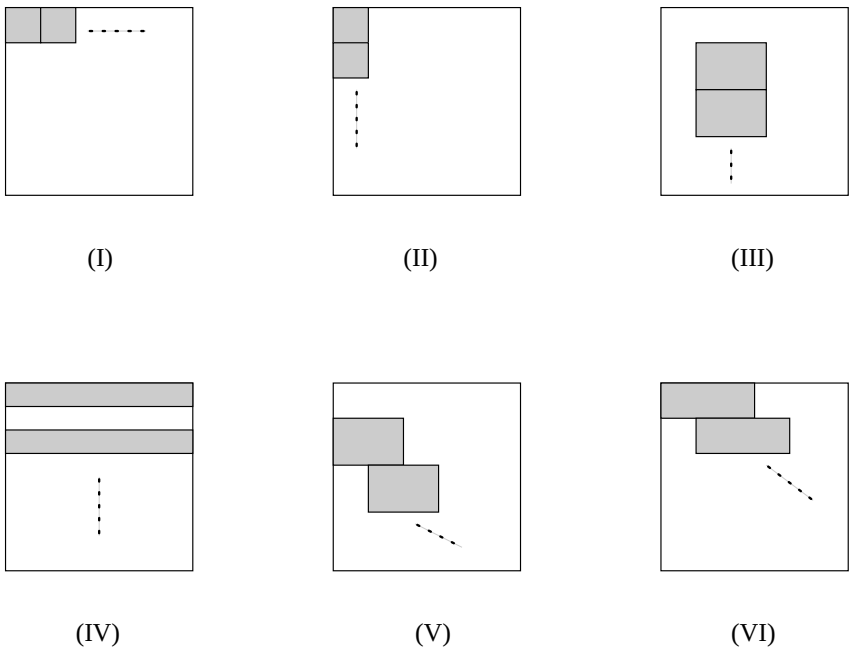

Figure 6.7: The distinct sections listed in Table 6.3 (not to scale)

Table 6.4: Time (sec.) for Writing Distinct Sections

Array size 4K × 4K real nos. (single precision), 16 processors

No.	Array Section (p = processor number)	Direct Write	Extended Two-Phase	
			Static	Dynamic
I	(1:100:1, 1+100p:100+100p:1)	1.944	3.250	2.801
II	(1+100p:100+100p:1, 1:100:1)	1.182	2.901	0.854
III	(200+200p:400+200p:1, 512:1024:1)	4.202	8.715	3.569
IV	(1+32p:16+32p:1, 1:4096:1)	24.85	10.25	10.25
V	(200+200p:400+200p:1, 1+200p:512+200p:1)	5.155	5.461	4.401
VI	(1+32p:32+32p:1, 1+100p:1024+100p:1)	8.233	4.994	4.274

6.5.4 Writing Distinct Sections

We only consider the case where each processor writes a distinct section to the file, because it is unlikely that processors will want to write overlapping or common sections. Table 6.4 compares the performance of the Extended Two-Phase Method and the Direct Method for writing distinct sections. The sections chosen are the same as those for reading in Table 6.3 (Figure 6.7).

We use the most general algorithm for writing in the Extended Two-Phase Method, which requires an extra read for each write. Hence for the sections in case I, the Direct Method performs better because it does not require the extra read and also these sections are small with few columns. The sections in cases II – IV lie interleaved in the file, so the Extended Two-Phase Method with dynamic partitioning performs much better than the Direct Method. The sections in cases V and VI lie partly interleaved in the file and even for these cases, the Extended Two-Phase Method performs considerably better.

6.5.5 Accessing Sections with Non-Unit Strides

We have also tested the performance for accessing sections with non-unit strides. When an array section has a non-unit stride, each element of the array lies strided

Table 6.5: Time (sec.) for Reading Sections with Non-unit Strides

Array size 4K × 4K real nos. (single precision), 16 processors

No.	Array Section (p = processor number)	Direct Read	Extended Two-Phase	
			Static	Dynamic
I	(p+1:4096:*nprocs*, p+1:4096:*nprocs*)	210.8	9.330	9.330
II	(1+250p:250+250p:2, 1+250p:250+250p:2)	53.13	3.610	2.842
III	(1+200p:500+200p:3, 1+200p:500+200p:3)	87.19	4.394	4.387
IV	(1+64p:64+64p:2, 500:2500:3)	96.20	4.759	3.848
V	(500:2500:3, 1+64p:64+64p:2)	130.7	4.574	2.340

in the file. The only way of reading such array sections using a direct method is to explicitly seek to each individual element and read only that element. This results in very low granularity of data transfer, which is very expensive. The Extended Two-Phase Method overcomes this drawback of the Direct Method by reordering requests and using data sieving for larger granularity reads.

Table 6.5 shows the performance of the Extended Two-Phase Method for reading sections with non-unit strides. The sections in case I span almost the entire array, with stride equal to the number of processors. Hence static and dynamic partitioning take exactly the same time. The sections in cases II and III are located diagonally across the out-of-core array. The sections in case IV are located along columns and the sections in case V are located along rows. In all cases, the Extended Two-Phase Method is more than 20 times faster than the Direct Method. Table 6.6 shows the performance of the Extended Two-Phase Method for writing sections with non-unit strides. The sections chosen are the same as in Table 6.5. Even for writing sections, the Extended Two-Phase Method provides considerable performance improvement.

6.5.6 Scalability

We have also studied the scalability of the Extended Two-Phase Method for large number of processors, large array sections and large out-of-core arrays. Since dynamic

Table 6.6: Time (sec.) for Writing Sections with Non-unit Strides

Array size 4K × 4K real nos. (single precision), 16 processors

No.	Array Section (p = processor number)	Direct Write	Extended Two-Phase	
			Static	Dynamic
I	(p+1:4096:*nprocs*, p+1:4096:*nprocs*)	53.28	22.77	22.77
II	(1+250p:250+250p:2, 1+250p:250+250p:2)	25.22	6.438	3.775
III	(1+200p:500+200p:3, 1+200p:500+200p:3)	44.64	8.696	7.516
IV	(1+64p:64+64p:2, 500:2500:3)	71.35	8.858	7.279
V	(500:2500:3, 1+64p:64+64p:2)	79.24	7.724	4.405

partitioning always performs better or at least as good as static partitioning, we only consider dynamic partitioning for the scalability experiments. Table 6.7 shows the timings obtained by varying the number of processors requesting array sections from 4 to 128, for both reading and writing. We have selected a few sections in each category, i.e. common, overlapping, distinct, and also sections with non-unit strides. Note that each processor is requesting a section, so as the number of processors is increased, the amount of I/O required increases.

The main observation is that the Extended Two-Phase Method scales well with the number of processors. In many cases, the time taken increases only slightly as the number of processors is increased, which indicates that we are able to get higher I/O throughput by increasing the number of processors. For example, for the sections in case I, the time taken increases from 1.282 sec. to only 2.130 sec. when the number of processors is increased from 4 to 128. In some cases, such as case II, the time taken even decreases. The Direct Method performs quite poorly as the number of processors is increased, especially for sections in cases II, IV and VIII. The Extended Two-Phase Method also scales well for writing sections. For small number of processors, the Direct Method takes less time for writing than the Extended Two-Phase Method. This is because of the extra read required for every write in the Extended Two-Phase Method. However, for large number of processors ($\geq$ 16), the Extended Two-Phase Method performs better. For sections with non-unit strides, the Extended Two-Phase

Table 6.7: Performance for different number of processors

I = (400:800:1, 400:800:1), Figure 6.5(III)

II = (1:16:1, 1:4096:1), Figure 6.5(V)

III = (400:800:1, 400+25p:800+25p:1), Figure 6.6(III)

IV = (1+8p:16+8p:1, 1:4096:1), Figure 6.6(VII)

V = (1+25p:16+25p:1, 1:4096:1), Figure 6.7(IV)

VI = (1+32p:32+32p:1, 1+24p:1024+24p:1), Figure 6.7(VI)

VII = (p+1:4096:*nprocs*, p+1:4096:*nprocs*)

VIII = (500:2500:3, 1+32p:32+32p:2)

Time in sec., 4K × 4K array, DR = Direct Read,

ETP = Extended Two-Phase (dynamic partitioning), DW = Direct Write

READING COMMON SECTIONS												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
I	2.620	1.282	3.184	1.040	4.421	1.056	8.734	1.169	16.28	1.436	32.64	2.130
II	12.16	4.315	13.95	3.099	19.65	3.241	32.96	2.647	60.11	3.432	116.7	3.219
READING OVERLAPPING SECTIONS												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
III	3.079	1.748	5.208	1.699	6.850	1.991	13.61	2.798	24.98	3.801	47.95	4.602
IV	13.75	4.450	13.77	3.391	19.63	2.992	32.70	3.696	60.58	4.791	115.9	7.401
READING DISTINCT SECTIONS												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
V	12.37	4.791	13.57	3.929	19.76	4.149	32.38	6.109	46.12	7.276	54.82	8.161
VI	3.704	1.893	2.396	1.585	4.125	1.638	7.806	2.418	19.77	2.970	26.23	4.110
WRITING DISTINCT SECTIONS												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP
V	3.129	7.900	6.971	6.861	12.45	8.554	27.52	12.74	37.70	18.52	52.41	24.74
VI	0.982	1.937	1.803	2.218	3.954	3.058	6.436	5.028	7.139	6.234	21.20	9.403
READING SECTIONS WITH NON-UNIT STRIDES												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
VII	799.2	22.82	216.6	15.83	210.8	9.331	103.1	10.89	54.94	8.307	50.60	9.657
VIII	56.44	1.342	77.78	1.440	83.87	1.870	163.1	3.123	331.5	5.062	867.4	7.711
WRITING SECTIONS WITH NON-UNIT STRIDES												
Sec-	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
tion	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP
VII	668.7	42.75	147.3	39.11	84.54	31.40	64.53	26.42	35.35	28.40	51.38	31.16
VIII	9.041	1.612	18.83	1.603	35.17	2.972	75.95	4.812	163.6	7.915	341.8	21.75

Method performs considerably better than the Direct Method. For the sections in case VIII, the stride depends on the number of processors, and so the total amount of I/O decreases as the number of processors is increased.

Table 6.8 shows the performance for accessing large sections of a large out-of-core array of size 16K $\times$ 16K single precision real numbers (file size 1Gbyte). Figure 6.8 shows approximately where these sections are located in the array. We consider common, overlapping and distinct sections for reading, and distinct sections for writing. The trend in the results is the same as in Table 6.7 for a 4K $\times$ 4K array, which confirms the scalability of the Extended Two-Phase Method and its superiority over the Direct Method. We observe that the Direct Method performs a lot worse for accessing large sections than for small sections, whereas the Extended Two-Phase Method performs consistently well for sections of any size. The relative performance of the two methods for reading and writing the sections in case VI of Table 6.8 is illustrated in Figures 6.9 and 6.10 respectively.

6.6 Advantages of Extended Two-Phase Method

The Extended Two-Phase Method with dynamic partitioning provides a very general way of accessing arbitrary sections of out-of-core arrays in an efficient manner. The first phase performs I/O optimizations at the cost of interprocessor communication in the second phase. Since communication cost is orders of magnitude lower than I/O cost, the overhead of communication is negligible. In all our experiments, we found the time spent on communication to be less than 5% of the total time. The Extended Two-Phase Method combines many small requests of different processors into single larger requests, thus providing larger granularity of data transfer and lower latency time. Another advantage is that multiple accesses by different processors to the same data in the file are eliminated. For example, if all processors need to read exactly the same section of the array, it will be read only once from the file and then broadcast to other processors over the interconnection network. Similarly, if the requests of two or more processors are overlapping, the overlapping portion will only be read once

Table 6.8: Performance for large sections of large arrays

I = (5000:6000:1,5000:6000:1)

II = (1+100p:300+100p:1,4000:8000:1)

III = (1+100p:400+100p:1,2000+20p:2800+20p:1)

IV = (4000:8000:1,1+4p:8+4p:1)

V = (1+100p:100+100p:1, 1+100p:1024+100p:1)

VI = (1+20p:16+20p:1,4000:12000:1)

Time in sec., 16K × 16K array, DR = Direct Read,

ETP = Extended Two-Phase (dynamic partitioning), DW = Direct Write

READING SECTIONS												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP	DR	ETP
I	23.65	7.880	43.43	7.795	78.99	7.935	151.3	9.085	302.7	9.368	605.1	11.86
II	53.30	26.51	103.3	28.10	132.3	28.50	157.6	32.49	162.3	40.03	182.4	52.08
III	13.31	5.061	24.11	6.489	31.49	7.400	39.81	9.253	41.28	10.12	44.29	13.23
IV	0.683	0.699	0.841	0.939	1.343	1.173	2.189	1.663	4.149	2.850	8.486	4.994
V	10.97	5.380	19.31	8.475	26.52	10.58	35.06	12.69	52.81	14.10	124.2	22.06
VI	57.29	21.94	74.05	23.05	127.3	32.88	240.8	51.26	500.2	112.2	799.7	98.68
WRITING SECTIONS												
Section	Procs=4		Procs=8		Procs=16		Procs=32		Procs=64		Procs=128	
	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP	DW	ETP
V	7.108	12.01	15.21	18.98	32.20	23.37	35.99	30.17	53.10	35.76	98.90	32.54
VI	48.35	44.18	71.85	52.07	151.4	73.34	272.8	122.3	548.1	174.1	746.6	164.2

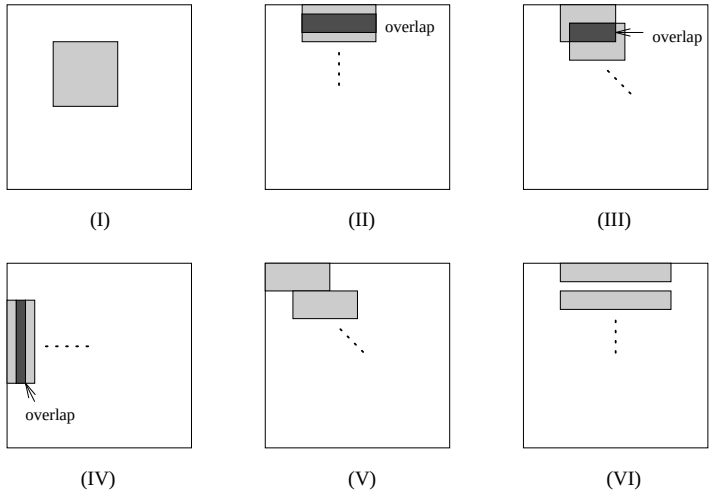

Figure 6.8: The sections listed in Table 6.8 (not to scale)

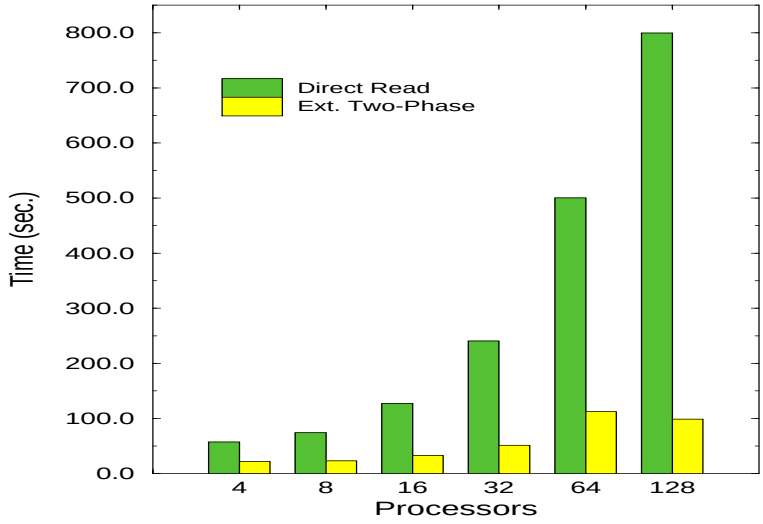

Figure 6.9: Scalability Results, reading sections in case VI of Table 6.8

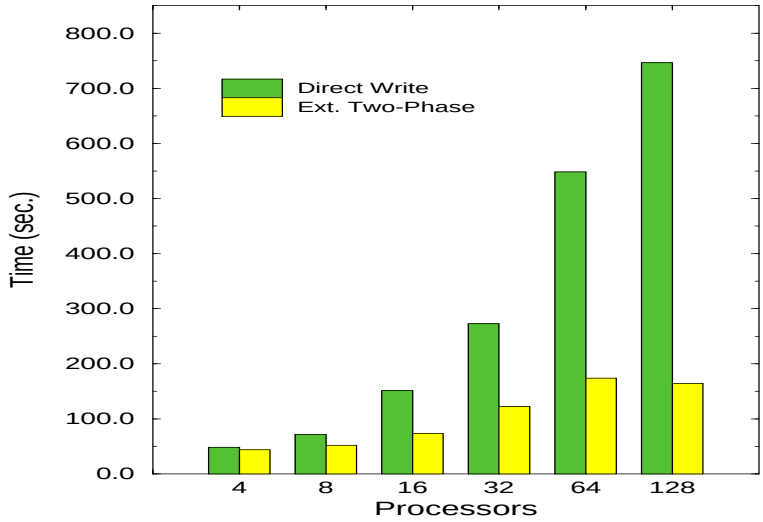

Figure 6.10: Scalability Results, writing sections in case VI of Table 6.8

from the file.

The Extended Two-Phase Method also provides a great deal of flexibility in dividing I/O among processors. This can be done by defining the file domains appropriately. We have described one way of doing this dynamically as a function of the access requests. This performs better than a static assignment, but it may be possible to do even better. For example, instead of dividing the bounding section in a column-block manner among the processors, it could be divided in a block-cyclic fashion, so that if the bounding section includes some unwanted columns, they get evenly distributed. Another approach is to divide I/O among processors so that the I/O requests of different processors go to different disks or I/O nodes. For this, we need to know on which disks the requested sections are located. This information is not provided by any of the parallel file systems at present, but we expect it to be available in future versions of the file systems. Furthermore, if the ratio of processors to disks on the machine is very high, it is possible to have only a few processors perform I/O, so as to reduce contention for disks.

The Extended Two-Phase Method can be used for accessing arrays with any number of dimensions and any storage order. For the dynamic partitioning scheme we have proposed, the file domains for an n-dimensional array can be obtained by first calculating the n-dimensional bounding section of all requests. This section is divided among processors such that the file domain of each processor is stored contiguously in the file.

Array sections other than those which can be represented by a lower-bound, upper bound and stride in each dimension, for example sections with non-uniform strides, can also be accessed using the Extended Two-Phase Method. This requires a more general notation for representing such sections. The data structures such as the File Access Descriptor and the File Domain Access Table need to be modified to handle this, but the basic idea remains the same.

It is not necessary that all processors must call the Extended Two-Phase read/write routine. It is possible to define a group of processors involving only those processors that need to access data from the file. This is similar to process groups in MPI [83].

Only the processors in the group need to call the Extended Two-Phase routine and participate in the two-phase process. The I/O workload can be divided among the processors in this group.

The Extended Two-Phase Method is not specific to any particular machine, file system or architecture. It can be easily implemented on top of any of the existing file system interfaces or portable interfaces such as the proposed MPI-IO interface [28], resulting in portable implementations. It can also be easily modified and tuned for any particular system. This only requires defining the file domains appropriately, and possibly using a different algorithm for interprocessor communication.

Chapter 7

Conclusions

This thesis addresses several issues in providing runtime support for in-core as well as out-of-core data-parallel programs. This runtime support can be used for performing many of the commonly required operations in application programs written using a distributed memory programming model. The use of runtime support makes it easier to write application programs and provides greater efficiency and portability. It can also be used together with a compiler to translate in-core as well as out-of-core programs written in a high-level data-parallel language like HPF to node programs for distributed memory parallel computers. Runtime support helps to separate the machine dependent and machine independent aspects of the translation. The compiler can do all the machine independent transformations and the runtime libraries can be optimized for each different machine. Thus, a portable and efficient compiler can be obtained by porting the runtime library to different machines.

This thesis proposes efficient algorithms for runtime array redistribution which is frequently required in parallel programs. We have considered block(m) to cyclic, cyclic to block(m) and the general cyclic(x) to cyclic(y) redistributions for one-dimensional and multidimensional arrays. In all cases, the Asynchronous Method was found to perform better than the Synchronous Method because it overlaps computation and communication, and hence reduces processor idle time. For the general cyclic(x) to cyclic(y) redistribution, the LCM Method performs the best. The thesis also shows how circular redistributions can be performed efficiently by saving address information

in the forward redistribution and reusing it in the backward redistribution.

This thesis presents several algorithms for all-to-all collective communication for the fat tree and two-dimensional mesh network topologies. The performance of these algorithms was studied on the CM-5 and Touchstone Delta. An analytical model for estimating the time taken by these algorithms was developed and validated by comparing with experimental results. The conclusion is that the choice of algorithm depends on the message size and the number of processors. No one algorithm performs the best for all message sizes and number of processors.

Due to the large memory requirements of many applications, it is becoming increasingly important to provide support for out-of-core programs. This thesis also describes the design and implementation of a runtime library for out-of-core computations. This library can either be directly used in out-of-core applications, or used together with a compiler to translate out-of-core programs written in a language such as HPF. The runtime library supports three different models for data storage and access. Runtime routines have been developed for all these models. Several optimizations, such as data sieving, data prefetching and data reuse, have been incorporated in the runtime library. The optimizations have provided significant performance improvement. The thesis also proposes the Extended Two Phase Method for reading and writing sections of out-of-core arrays efficiently. This method uses collective I/O in which processors cooperate to perform I/O in an efficient manner by combining several I/O requests into fewer larger requests, eliminating multiple file accesses for the same data and reducing contention for the I/O subsystem. A dynamic scheme is used for dividing I/O among processors, depending on the access requests. The Extended Two Phase Method showed impressive performance benefits over a Direct Method for many different access patterns.

There are a number of areas in which the research presented in this thesis can be extended. The area of runtime support for parallel I/O is particularly promising. A useful feature of the Extended Two-Phase Method is the flexibility it provides in defining file domains. We have studied one way of selecting file domains. A good research problem is to determine the best way to define file domains depending on

the pattern of access requests as well as the distribution of the file on disks. Scientific data is very often stored in standard data formats such as HDF [86] or NetCDF [123]. It would be useful to design and implement efficient runtime support for data stored in files using these formats. The PASSION Runtime Library has currently been implemented on the Intel Touchstone Delta, Paragon and iPSC/860 systems. It would be an interesting project to port it to the IBM SP-2 using the PIOFS file system. A unique feature of PIOFS (and its predecessor Vesta) is that it supports logical partitioning of files [29]. It is not entirely clear how widely this logical partitioning can be used. Implementing a general runtime system like PASSION using PIOFS can provide some insight into the usefulness of logical partitioning.

Bibliography

[1] G. Agrawal, A. Sussman, and J. Saltz. Compiler and Runtime Support for Structured and Block Structured Applications. In *Proceedings of Supercomputing '93*, pages 578–587, November 1993.

[2] I. Ahmad, R. Bordawekar, Z. Bozkus, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Implementation and Scalability of Fortran 90D Intrinsic Functions on Distributed Memory Machines. Technical Report SCCS–256, NPAC, Syracuse University, 1992.

[3] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn. Global Combine on Mesh Architectures with Wormhole Routing. In *Proceedings of the* 7^{th} *International Parallel Processing Symposium*, April 1993.

[4] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A Framework for Optimizing Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20, October 1994.

[5] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. Yang. Distributed pC++: Basic Ideas for an Object Parallel Language. In *Proceedings of the First Annual Object-Oriented Numerics Conference*, pages 1–24, April 1993.

[6] S. Bokhari. Complete Exchange on the iPSC/860. Technical Report 91–4, ICASE, NASA Langley Research Center, 1991.

[7] S. Bokhari and H. Berryman. Complete Exchange on a Circuit Switched Mesh. In *Proceedings of the Scalable High Performance Computing Conference*, pages 300–306, 1992.

[8] R. Bordawekar and A. Choudhary. Language and Compiler Support for Parallel I/O. In *Proceedings of IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, April 1994.

[9] R. Bordawekar, A. Choudhary, and J. del Rosario. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. In *Proceedings of the 7^{th} ACM International Conference on Supercomputing*, July 1993.

[10] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A Model and Compilation Strategy for Out-of-Core Data Parallel Programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, July 1995. To appear.

[11] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines. Technical Report SCCS–622, NPAC, Syracuse University, September 1994.

[12] R. Bordawekar, J. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, November 1993.

[13] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[14] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, R. Thakur, and J. Wang. Scalable Libraries for High Performance Fortran. In *Proceedings of the Scalable*

Parallel Libraries Conference, pages 67–75. Mississippi State University, October 1993.

[15] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu. Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers. *Journal of Parallel and Distributed Computing*, pages 15–26, April 1994.

[16] Z. Bozkus, S. Ranka, and G. Fox. Modeling the CM-5 Multicomputer. In *Proceedings of Frontiers of Massively Parallel Computation 92*, pages 100–107, October 1992.

[17] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent File Operations in a High Performance Fortran. In *Proceedings of Supercomputing '92*, pages 230–238, November 1992.

[18] P. Brezany, T. Mueck, and E. Schikuta. Language, Compiler and Parallel Database Support for I/O Intensive Applications. In *Proceedings of High Performance Computing and Networking 1995*, May 1995.

[19] R. Butler and E. Lusk. User's Guide to the P4 Programming System. Technical Report ANL–92/17, Argonne National Laboratory, October 1992.

[20] A. Carle, K. Kennedy, U. Kremer, and J. Mellor-Crummey. Automatic Data Layout for Distributed Memory Machines in the D Programming Environment. Technical Report CRPC-TR93298, Center for Research on Parallel Computation, Rice University, February 1993.

[21] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. Technical Report 92–9, ICASE, NASA Langley Research Center, March 1992.

[22] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Proceedings of Principles of Programming Languages (POPL) '93*, pages 16–28, January 1993.

[23] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating Local Addresses and Communication Sets for Data Parallel Programs. In *Proceedings of Principles and Practices of Parallel Programming (PPoPP) '93*, pages 149–158, May 1993.

[24] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[25] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, September 1994. Also available as CRPC Technical Report CRPC–TR94483–S.

[26] A. Choudhary, R. Bordawekar, S. More, K. Sivaram, and R. Thakur. A User's Guide for the PASSION Runtime Library Version 1.0. Technical Report SCCS–702, NPAC, Syracuse University, February 1995.

[27] P. Corbett and D. Feitelson. Overview of the Vesta Parallel File System. In *Proceedings of the Scalable High Performance Computing Conference*, pages 63–70, May 1994.

[28] P. Corbett, D. Feitelson, Y. Hsu, J. Prost, M. Snir, S. Fineberg, B. Nitzberg, B. Traversat, and P. Wong. MPI-IO: A Parallel I/O Interface for MPI, Version 0.3. Technical Report NAS-95-002, NASA Ames Research Center, January 1995.

[29] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[30] T. Cormen. *Virtual Memory for Data-Parallel Computing.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1993.

[31] T. Cormen and A. Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[32] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[33] W. Dally and C. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, pages 547–553, May 1987.

[34] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime Partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Runtime Environments for Distributed Memory Machines*, pages 185–220. Elsevier Science Publishers, 1992.

[35] R. Das, J. Saltz, and H. Berryman. A Manual for PARTI Runtime Primitives. Interim Report 17, ICASE, NASA Langley Research Center, May 1991.

[36] E. DeBenedictis and J. del Rosario. nCUBE Parallel I/O Software. In *Proceedings of* 11^{th} *International Phoenix Conference on Computers and Communications*, pages 117–124, April 1992.

[37] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Runtime Access Strategy. In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS '93*, pages 56–70, April 1993.

[38] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, pages 59–68, March 1994.

[39] P. Dibble, M. Scott, and C. Ellis. Bridge: A High-Performance File System for Parallel Processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[40] I. Foster, M. Henderson, and R. Stevens. Workshop Introduction. In *Proceedings of the Workshop on Data Systems for Parallel Climate Models at Argonne National Laboratory*, July 1991.

[41] G. Fox. *Domain Decomposition in Distributed and Shared Memory Environments – I: A Uniform Decomposition and Performance Analysis for the nCUBE and JPL Mark IIIfp Hypercubes*, volume 297 of *Lecture Notes in Computer Science*, pages 1042–1073. Springer-Verlag, 1987. Supercomputing, ed. E. Houstis, T. Papatheodorou, and C. Polychronopoulos.

[42] G. Fox and W. Furmanski. Optimal Communication Algorithms for Regular Decompositions on the Hypercube. In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume 1*, pages 648–713, January 1988.

[43] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. Fortran D Language Specifications. Technical Report COMP TR90-141, CRPC, Rice University, 1990.

[44] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, Vol. I.* Prentice-Hall Inc., 1988.

[45] G. Fox, S. Ranka, M. Scott, A. Malony, J. Browne, M. Chen, A. Choudhary, T. Cheatham, J. Cuny, R. Eigenmann, A. Fahmy, I. Foster, D. Gannon, T. Haupt, M. Karr, C. Kesselman, C. Koelbel, W. Li, M. Lam, T. LeBlanc, J. Openshaw, D. Padua, C. Polychronopoulos, J. Saltz, A. Sussman, G. Weigand, and K. Yelick. Common Runtime Support for High Performance Parallel Languages — Parallel Compiler Runtime Consortium. In *Proceedings of Supercomputing '93*, pages 752–757, November 1993.

[46] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works*. Morgan Kaufmann Publishers Inc., 1994.

[47] G. Geist, M. Heath, B. Peyton, and P. Worley. A User's Guide to PICL, A Portable Instrumented Communication Library. Technical Report ORNL/TM–11616, Oak Ridge National Laboratory, October 1991.

[48] M. Grossman. Modeling Reality. *IEEE Spectrum*, pages 56–60, September 1992.

[49] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, September 1992.

[50] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, pages 179–193, mar 1992.

[51] S. Gupta, S. Hawkinson, and B. Baxter. A Binary Interleaved Algorithm for Complete Exchange on a Mesh Architecture. Technical report, Intel Supercomputer Systems Division, April 1993.

[52] S. Gupta, S. Kaushik, C. Huang, and P. Sadayappan. On Compiling Array Expressions for Efficient Execution on Distributed Memory Machines. Technical Report OSU-CISRC-4/94-TR19, Computer and Information Science Research Center, The Ohio State University, April 1994.

[53] S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C. Huang, and P. Sadayappan. On the Generation of Efficient Data Communication for Distributed Memory Machines. In *Proceedings of International Computing Symposium, Taiwan*, pages 504–513, 1992.

[54] S. Hambrusch, F. Hameed, and A. Khokhar. Communication Operations on Coarse-Grained Mesh Architectures. *Parallel Computing*, 1995. to appear.

[55] M. Harry, J. del Rosario, and A. Choudhary. VIP-FS: A Virtual, Parallel File System for High Performance Parallel and Distributed Computing. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995.

[56] High Performance Computing and Communications: Grand Challenges 1993 Report. A Report by the Committee on Physical, Mathematical and Engineering Sciences, Federal Coordinating Council for Science, Engineering and Technology, 1993.

[57] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Version 1.0, May 1993.

[58] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An Overview of the Fortran D Programming System. Technical Report COMP TR91-154, CRPC, Rice University, March 1991.

[59] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler Support for Machine-Independent Parallel Programming in Fortran D. In *Languages, Compilers and Run-Time Environments for Distributed Memory Machines.* North-Holland, 1992.

[60] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal. PPFS: A High Performance Portable Parallel File System. Technical Report UIUCDCS-R-95-1903, University of Illinois at Urbana Champaign, January 1995.

[61] S. L. Johnsson and C. T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes . *IEEE Transactions on Computers*, pages 1249–1268, Sep 1989.

[62] E. Kalns and L. Ni. Processor Mapping Techniques Toward Efficient Data Redistribution. In *Proceedings of the* 8^{th} *International Parallel Processing Symposium*, pages 469–476, April 1994.

[63] S. Kaushik, C. Huang, R. Johnson, and P. Sadayappan. An Approach to Communication-Efficient Data Redistribution. In *Proceedings of the* 8^{th} *ACM International Conference on Supercomputing*, July 1994.

[64] S. Kaushik, C. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase Array Redistribution: Modeling and Evaluation. Technical Report OSU-CISRC-9/94-52, Computer and Information Science Research Center, The Ohio State University, September 1994.

[65] C. Koelbel. *Compiling Programs for Nonshared Memory Machines*. PhD thesis, Purdue University, August 1990.

[66] C. Koelbel. Compile-Time Generation of Regular Communication Patterns. In *Proceedings of Supercomputing '91*, pages 101–110, November 1991.

[67] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook*. The MIT Press, 1994.

[68] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, pages 440–451, oct 1991.

[69] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994. Revised as Technical Report PCS–TR94–226, Dept. of Computer Science, Dartmouth College.

[70] D. Kotz. Disk-directed I/O for an Out-of-Core Computation. Technical Report PCS-TR95-251, Dept. of Computer Science, Dartmouth College, January 1995.

[71] D. Kotz. Expanding the Potential for Disk-directed I/O. Technical Report PCS-TR95-254, Dept. of Computer Science, Dartmouth College, March 1995.

[72] D. Kotz and T. Cai. Exploring the Use of I/O Nodes for Computation in a MIMD Multiprocessor. Technical Report PCS–TR94–232, Dept. of Computer Science, Dartmouth College, Revised February 1995.

[73] D. Kotz and C. Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 218–230, April 1990.

[74] D. Kotz and N. Nieuwejaar. Dynamic File Access Characteristics of a Production Parallel Scientific Workload. In *Proceedings of Supercomputing '94*, November 1994.

[75] H. Kung. Memory Requirements for Balanced Computer Architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 49–54, 1986.

[76] C. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. In *Proceedings of International Conference on Parallel Processing*, pages 952–958, 1984.

[77] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, 1992.

[78] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, pages 361–376, July 1991.

[79] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-References Between Distributed Arrays. In *Proceedings of Frontiers of Massively Parallel Computation 92*, October 1992.

[80] R. Littlefield. Tuning Communication. *Proceedings of the Delta Advanced User Training Class Notes, CCSF Technical Report CCSF-25-92*, pages 99–119, jul 1992.

[81] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 69–77, May 1987.

[82] P. Mehrotra and J. Van Rosendale. The BLAZE Language: A Parallel Language for Scientific Programming. *Parallel Computing*, 5:339–361, 1987.

[83] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Version 1.0, May 1994.

[84] E. Miller and R. Katz. RAMA: Easy Access to a High-Bandwidth Massively Parallel File System. In *Proceedings of the 1995 Winter USENIX Conference*, pages 59–70, January 1995.

[85] S. Moyer and V. Sunderam. PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[86] National Center for Supercomputing Applications, University of Illinois. *NCSA HDF Reference Manual.* Version 3.3, February 1994.

[87] L. Ni and P. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, pages 62–76, February 1993.

[88] N. Nieuwejaar and D. Kotz. Low-level Interfaces for High-level Parallel I/O. Technical Report PCS–TR95–253, Dept. of Computer Science, Dartmouth College, February 1995.

[89] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Data Parallel Machines. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[90] Parasoft Corporation. EXPRESS Fortran User's Guide, 1990.

[91] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[92] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of* 4^{th} *Conference on Hypercubes, Concurrent Computers and Applications*, pages 155–160, March 1989.

[93] R. Ponnusamy. *Runtime Support and Compilation Methods for Irregular Computations on Distributed Memory Parallel Machines.* PhD thesis, School of Computer and Information Science, Syracuse University, April 1994.

[94] R. Ponnusamy, A. Choudhary, and G. Fox. Communication Overhead on CM-5 : An Experimental Performance Evaluation. In *Proceedings of Frontiers of Massively Parallel Computation 92*, pages 108–115, October 1992.

[95] R. Ponnusamy, R. Thakur, A. Choudhary, and G. Fox. Scheduling Regular and Irregular Communication Patterns on the CM-5. In *Proceedings of Supercomputing '92*, pages 394–402, November 1992.

[96] R. Ponnusamy, R. Thakur, A. Choudhary, K. Velamakanni, and G. Fox. Experimental Performance Evaluation of the CM-5. *Journal of Parallel and Distributed Computing*, pages 192–202, November 1993.

[97] J. Poole. Preliminary Survey of I/O Intensive Applications. Scalable I/O Initiative Working Paper Number 1, 1994.

[98] A. Purakayastha, C. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing Parallel File-Access Patterns on a Large-Scale Multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, April 1995.

[99] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. Technical Report CRHC-94-09, CRHC, University of Illinois, 1994.

[100] S. Ranka, J. Wang, and G. Fox. Static and Runtime Algorithms for All-to-Many Personalized Communication on Permutation Networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 1266–1274, December 1994.

[101] K. Salem and H. Garcia-Molina. Disk Striping. In *Proceedings of the International Conference on Data Engineering*, pages 336–342, 1986.

[102] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and Runtime Compilation. *Concurrency: Practice and Experience*, pages 573–592, December 1991.

[103] D. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *Proceedings of the* 6^{th} *Distributed Memory Computing Conference*, pages 398–403, 1991.

[104] K. Seamons and M. Winslett. An Efficient Abstract Interface for Multidimensional Array I/O. In *Proceedings of Supercomputing '94*, pages 650–659, November 1994.

[105] K. Seamons and M. Winslett. A Data Management Approach for Handling Large Compressed Arrays in High Performance Computing. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[106] R. Shankar, K. Alsabti, and S. Ranka. The Transportation Primitive. Technical report, Dept. of Computer and Information Science, Syracuse University, August 1994.

[107] R. Shankar and S. Ranka. Random Data Accesses on a Coarse-grained Parallel Machine – I. One-to-One Mappings. Technical report, Dept. of Computer and Information Science, Syracuse University, October 1994.

[108] R. Shankar and S. Ranka. Random Data Accesses on a Coarse-grained Parallel Machine – II. One-to-Many and Many-to-One Mappings. Technical report, Dept. of Computer and Information Science, Syracuse University, October 1994.

[109] M. Snir. Proposal for IO. Posted to HPFF I/O Forum by Marc Snir, July 1992.

[110] J. Stichnoth. Efficient Compilation of Array Statements for Private Memory Multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[111] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, pages 150–159, April 1994.

[112] E. Su, D. Palermo, and P. Banerjee. Automatic Parallelization of Regular Computations For Distributed Memory Multicomputers in the PARADIGM Compiler. In *Proceedings of International Conference on Parallel Processing*, pages II–30—II–38, August 1993.

[113] N. Sundar, D. Jayasimha, D. Panda, and P. Sadayappan. Complete Exchange in 2D Meshes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 406–413, May 1994.

[114] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2:315–339, dec 1991.

[115] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the* 8^{th} *ACM International Conference on Supercomputing*, pages 382–391, July 1994.

[116] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[117] R. Thakur and A. Choudhary. All-to-All Communication on Meshes with Wormhole Routing. In *Proceedings of the* 8^{th} *International Parallel Processing Symposium*, pages 561–565, April 1994.

[118] R. Thakur and A. Choudhary. Collective I/O Using an Extended Two-Phase Method with Dynamic Partitioning. Technical Report SCCS–704, NPAC, Syracuse University, March 1995.

[119] R. Thakur, A. Choudhary, and G. Fox. Runtime Array Redistribution in HPF Programs. In *Proceedings of the Scalable High Performance Computing Conference*, pages 309–316, May 1994.

[120] R. Thakur, R. Ponnusamy, A. Choudhary, and G. Fox. Complete Exchange on the CM-5 and Touchstone Delta. *The Journal of Supercomputing*, pages 305–328, March 1995.

[121] Thinking Machines Corporation. *CM Fortran Language Reference Manual*. Version 2.1, January 1994.

[122] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[123] Unidata Program Center, University Corporation for Atmospheric Research. *netCDF User's Guide*. Version 2.0, October 1991.

[124] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of International Symposium on Computer Architecture*, 1992.

[125] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. In *Proceedings of the* 5^{th} *Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[126] A. Wakatani and M. Wolfe. A New Approach to Array Redistribution: Strip Mining Redistribution. In *Proceedings of Parallel Architectures and Languages Europe (PARLE 94)*, July 1994.

[127] J. Wang. *Load Balancing and Communication Support for Regular and Irregular Problems*. PhD thesis, School of Computer and Information Science, Syracuse University, December 1993.

[128] M. Wu and G. Fox. Compiling Fortran 90 Programs for Distributed Memory MIMD Parallel Computers. Technical Report SCCS–88, NPAC, Syracuse University, April 1991.

[129] H. Zima, H. Bast, and M. Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, pages 1–18, 1988.

[130] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a Language Specification, Version 1.1. Interim Report 21, ICASE, NASA Langley Research Center, March 1992.

[131] G. Zorpette. Teraflops Galore. *IEEE Spectrum*, pages 26–76, September 1992.

Vita

NAME:	Rajeev Thakur
PLACE OF BIRTH:	Bombay, India
DATE OF BIRTH:	July 15, 1969
PREVIOUS DEGREES:	B.E., Computer Engineering, University of Bombay, 1990 M.S., Computer Engineering, Syracuse University, 1992
EXPERIENCE:	Summer Intern, Intel Supercomputer Systems Division, Beaverton, OR, May 1993 – August 1993
	Graduate Fellow Syracuse University, August 1994 – May 1995
	Graduate Research Assistant, Northeast Parallel Architectures Center, Syracuse University, August 1991 – August 1994
	Graduate Research Assistant, Department of Electrical and Computer Engineering, Syracuse University, January 1991 – August 1991