# ISSUES IN SOFTWARE SUPPORT FOR PARALLEL I/O

By

Rajesh R. Bordawekar

B.E., University of Bombay, India, 1991

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of Master of Science in computer engineering in the Graduate School of Syracuse University

May 1993

Approved ____________________

Date ____________________

© Copyright 1993

Rajesh R. Bordawekar

## Abstract

This thesis looks at various issues in providing application-level software support for parallel I/O. We show that the performance of the parallel I/O system varies greatly as a function of data distributions. We present runtime I/O primitives for parallel languages which allow the user to obtain a consistent performance over a wide range of data distributions.

In order to design these primitives, we study various parameters used in the design of a parallel file system. We evaluate the performance of Touchstone Delta Concurrent File System and study the effect of parameters like number of processors, number of disks, file size on the system performance. We compute the I/O costs for common data distributions. We propose an alternative strategy -two phase data access strategy- to optimize the I/O costs connected with data distributions. We implement runtime primitives using the two-phase access strategy and show that using these primitives not only I/O access rates are improved but also user can obtain complex data distributions like block-block and block-cyclic.

# Acknowledgments

I like to thank my advisor, Dr. Alok Choudhary for his invaluable guidance and encouragement during the course of this research; my thesis committee members, Professors Gary Craig, Geoffrey Fox and Salim Hariri for their suggestions. I want to thank Juan Migel del Rosario for useful technical discussions especially regarding two-phase data access strategy. I also want to thank Rajiv Raje for reading the entire thesis and giving many useful suggestions. Finally I like to thank Rajiv Thakur for suggestions and assistance.

Access to Touchstone Delta was provided by Center for Research in Parallel Computing and Concurrent Supercomputing Consortium. This work was sponsored by DARPA under contract # DABT63-91-C-0028 and NSF MIP-9110810. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

This thesis is dedicated to my family.

# Contents

**Acknowledgments** **i**

**1 Introduction** **1**

1.1 Thesis Objective . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Design Issues in Parallel I/O Systems** **6**

2.1 Design of the Disk Systems . . . . . . . . . . . . . . . . . . . . . . . 7

2.2 Data Distribution Protocols . . . . . . . . . . . . . . . . . . . . . . . 10

2.3 Processor-Disk Interconnection . . . . . . . . . . . . . . . . . . . . . 12

2.4 Parallel File Access Policies . . . . . . . . . . . . . . . . . . . . . . . 13

2.5 Disk Caching and Prefetch Policies . . . . . . . . . . . . . . . . . . . 16

2.6 OS Interface between the Processing Nodes and the File System . . . 18

2.6.1 Existing Parallel File System Interfaces . . . . . . . . . . . . 22

**3 Case Study of a Parallel I/O System** **24**

3.1 The Touchstone Delta System . . . . . . . . . . . . . . . . . . . . . . 25

3.2 Concurrent File System . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.2.1 CFS File Structure . . . . . . . . . . . . . . . . . . . . . . . 27

3.2.2 The File System Interface . . . . . . . . . . . . . . . . . . . . 27
3.2.3 The I/O Modes . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2.4 The I/O Network . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 CFS Evaluation Methodology . . . . . . . . . . . . . . . . . . . . . 31
3.4 CFS Performance Evaluation . . . . . . . . . . . . . . . . . . . . . 33
3.4.1 Single Compute Node . . . . . . . . . . . . . . . . . . . . . 33
3.4.2 Multiple Compute Nodes . . . . . . . . . . . . . . . . . . . . 37
3.5 Parallel File System: An Extension of Concurrent File System . . . . 47
3.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

**4 Cost Analysis of Data Mappings 50**

4.1 An I/O Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.1.2 An I/O Model for a Multiprocessor . . . . . . . . . . . . . . . 52
4.2 Analysis of the I/O costs . . . . . . . . . . . . . . . . . . . . . . 57
4.2.1 Analysis of Array Decompositions . . . . . . . . . . . . . . . . 58
4.2.2 Analysis of Common Data Distributions . . . . . . . . . . . . 61
4.2.3 Experimental Results . . . . . . . . . . . . . . . . . . . . . . 65
4.3 Two Stage Data Mapping . . . . . . . . . . . . . . . . . . . . . . 68

**5 Runtime I/O Support For Parallel Languages 73**

5.1 Languages Supporting Data Distribution . . . . . . . . . . . . . . . 74
5.1.1 Vienna Fortran . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.1.2 High Performance Fortran . . . . . . . . . . . . . . . . . . . 75
5.2 Runtime Primitives for Parallel I/O . . . . . . . . . . . . . . . . . 77
5.2.1 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.2.2 General Description . . . . . . . . . . . . . . . . . . . . . . 79

5.2.3 Syntax of the Runtime Primitives . . . . . . . . . . . . . . . . . 84

5.2.4 A Sample Program . . . . . . . . . . . . . . . . . . . . . . . . 89

5.2.5 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . 90

**6 Conclusions and Future Work** **96**

6.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

6.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

**Biographical Data** **114**

# List of Tables


1 Definitions of Various Terms Used in the Thesis . . . . . . . . . . . . 32

2 File Read Time is (ms)-Single Compute Node . . . . . . . . . . . . . 35

3 Multinode (4*4) Burst Mode Throughput as a Function of Disk Volumes (Mode 0) . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4 Throughput of Accessing 1 MB File in Mode 0 (Burst Mode,64 Disks) 40

5 Throughput Rates of Accessing 2 MB File in Mode 0(Burst Mode) . . 40

6 Read Throughput in Mbytes/sec (Burst Mode) . . . . . . . . . . . . 46

7 Write Throughput in Mbytes/sec (Burst Mode) . . . . . . . . . . . . 46

8 Number of I/O Requests as a Function of Data Distributions . . . . . 58

9 Array Distribution (Column Block) Throughput in MBytes/sec ($N_D$ = 64) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

10 Array Distribution (Column Cyclic) Throughput in Kbytes/sec ($N_D$ = 64) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

11 Array Distribution (Row Block) For Modes 2 and 3 ($N_D$ = 64) . . . 67

12 The File Descriptor Array (FDA): Fortran Version . . . . . . . . . . . 80

13 The Array Description Table (ADT) . . . . . . . . . . . . . . . . . . 81

14 The P_INFO Array for a Two-Dimensional Logical Grid . . . . . . . 82

15 Performance of **pread** for 16 Processors (5K*5K Array) . . . . . . . . 92

16 Performance of **pread** for 16 Processors (10K*10K Array) . . . . . . 92

17 Performance of **pread** for 64 Processors (5K*5K Array) . . . . . . . . . 93
18 Performance of **pread** for 64 Processors (10K*10K Array) . . . . . . . 93
19 Block-Block Distribution over 16 Processors using **pread** (time in msec) 94
20 Block-Block Distribution over 64 Processors Using **pread** (time in msec) 95

# List of Figures


1 Intel Touchstone Delta System . . . . . . . . . . . . . . . . . . . . . . 26

2 File block distribution across the disks . . . . . . . . . . . . . . . . . . 27

3 Single Compute Node - 32 I/O Nodes (64 Disks): Read Rates in KB/sec 34

4 Read Rates for Single Compute Node-Burst Mode . . . . . . . . . . . 36

5 Write Rates for Single Compute Node-Burst Mode . . . . . . . . . . . 37

6 Read Rates For Multiple Compute Nodes - Mode 0 . . . . . . . . . . 38

7 Read for Grid Size 4*4 . . . . . . . . . . . . . . . . . . . . . . . . . 41

8 Reading a 16M File using 4K Buffer for 64 Disk Volumes . . . . . . . 43

9 Multicompute Nodes Read (Mode 3 and 16 Mbytes File) . . . . . . . 43

10 Multicompute Nodes Read (Mode 2 and 16 Mbytes File) . . . . . . . 44

11 Multicompute Nodes File Read (Mode 1 and 16 Mbytes File) . . . . . 45

12 A Processing Element . . . . . . . . . . . . . . . . . . . . . . . . . . 51

13 A Mesh Connected Multiprocessor . . . . . . . . . . . . . . . . . . . 52

14 Data Mapping between the Processors and the Disks . . . . . . . . . 59

15 Block-Block Decomposition over 16 Processors . . . . . . . . . . . . . 70

16 Pairwise Exchange Algorithm . . . . . . . . . . . . . . . . . . . . . . 72

17 Array Mapping Using Vienna Fortran . . . . . . . . . . . . . . . . . . 75

18 Data Distributions in Fortran 90D/ HPF . . . . . . . . . . . . . . . . 76

19 **pread** Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

20 **pwrite** Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
21 A Sample Program For Performing Parallel I/O . . . . . . . . . . . . 91

# Chapter 1

# Introduction

In the last few years, the processor speeds have increased tremendously, and with multiprocessor organizations, the processing power of computer systems has been increasing at a rapid pace. It is believed that massively parallel machines will provide teraflops of computing power in near future.

Some of the commercially available parallel computers include Intel Paragon [Int92], nCUBE-2 [nCU92], KSR-1 [KSR92] and CM-5 [Thi91]. They are the computational instruments of choice in the scientific community and can be found in one form or another in almost every major academic and research institution. Several prototypes are also being developed at various industrial and academic institutions. Examples of such machines are DASH [DJK+92], Alewife [ACD+91], J Machine [DCF+86] and Tera [ACC+90]. These machines are aimed at achieving the teraflop goal set by the High Performance Computing and Communication Initiative of the Federal Government [Cho93].

One of the areas that has often been neglected in parallel systems, both at the software and hardware levels, is the I/O system. Currently, there exists a huge difference between the processing power and the I/O system's performance. As a

result of this disparity, the problem solving speed is determined by how fast the I/O can be done.

If the I/O is done in a serial fashion, even if the problem is solved considerably faster, the speed of the solution depends on the rate of the serial I/O. Kung showed that for certain types of applications increasing memory size alone is not sufficient, and I/O bandwidth must be increased proportionately to balance the computing system. This is called the *I/O problem* [Kun86]. Amdahl's rule of thumb for a balanced computer shows that a system should have 1 MBytes of main memory and 1 Mbit/sec I/O bandwidth per 1 MIPS of CPU performance [Amd67]. Akella and Siewiorek state that the I/O requirements are nearer to 1 MBytes/sec for 1 MIP CPU and 1 MBytes memory [Ake91]. A simple extrapolation will show that for teraflops machines the required bandwidth will range from tens of GBytes to at least several hundred GBytes per second.

There are a variety of applications that require a significant amount of I/O. These include image processing, weather forecasting and databases [Sha93]. Another important application that requires sufficient I/O support is multimedia. The multimedia applications require accesses to large image and video data requiring a tremendous bandwidth and storage capacity. Similarly, data visualization is another field which requires a lot of I/O operations [Dem92]. A real-time data visualization package requires a large amount of data to be transferred between the disks and the CPU.

Parallel I/O systems, are therefore, required for providing the large I/O bandwidth. A lot of work needs to be done both in software and hardware aspects of parallel I/O. The compilers and operating systems are less developed as compared to the development in the hardware. MIMD languages such as Vienna Fortran [CMZ92], Fortran D [FHK+90], High Performance Fortran (HPF) [For93] provide data decomposition as compiler directives. The distributed data needs to accessed from the

file system. This is a I/O sensitive job since each processor needs to access the disks, thus resulting in a large number of disk requests. Hence to optimize the I/O performance at the application level, an appropriate support from the compilers and runtime system is needed.

## 1.1 Thesis Objective

Our objective is to evaluate and design application-level software support for parallel I/O systems. We concentrate on I/O problems associated with data distributions in the multiprocessor environments. We develop a runtime software environment which allows the user to control the data mapping on both the processors and disks and at the same time relieves the user from worrying about data distribution costs by providing runtime distribution primitives. To meet these objectives, we first survey various design issues in parallel I/O systems. We then investigate the performance of Touchstone Delta Concurrent File System. We study various system parameters and evaluate their effects on the overall performance of the file system. We analyze the performance of the applications which require parallel data access from the disks. We choose array decompositions as our computational domain and compute the I/O costs for various patterns of data decompositions. We show that the performance of the parallel file system can vary greatly as a function of data distribution. Further, certain data distributions are not supported by parallel file systems. Motivated by these problems, we propose an alternative strategy - two-phase access strategy - which guarantees consistent performance over a wide spectrum of data decompositions. To facilitate the programmer to perform efficient data distributions from the application level, we develop runtime support to perform efficient collective I/O. We design library routines that use system parameters such as number of disks, data distribution on

the processors and disks, number of processors, number of disks, etc., to determine efficient strategy for performing I/O, and then redistribute the data as required by the target data distribution on the processors.

## 1.2 Thesis Outline

Chapter 2 presents a survey of various issues in the design of a parallel I/O system. It reviews both the hardware as well as software approaches in the design strategy. We study design of the disk systems and present prominent data distribution protocols. We evaluate the processor-disk interconnection. Various parallel file access strategies are presented and corresponding disk caching and prefetching policies required for such data access patterns are introduced. Finally we review the operating system interface between the processors and the disks.

In chapter 3, we choose the Touchstone Delta Concurrent File System (CFS) as a case study. We study the interaction between the processors and the file system by studying various parameters. We analyze the effects of data request size, number of participating processors and disks, file sizes and file access patterns. Based on our results, we present values of various parameters required for obtaining the peak performance.

Chapter 4 presents a simple I/O model for the multiprocessors associated with parallel file systems. We analyze the effects of I/O performance on the overall program bandwidth. Using this model, we will analyze and compute data distribution costs in these environments. We again use the Touchstone Delta CFS for our analysis. We show that the data distribution costs vary according to the distribution pattern and depend on the data mapping over processors and disks. Moreover, there exists a mapping which is optimal for both processor and the disk distributions. Using these

facts, we present an alternative strategy - two-phase data access strategy - for optimal data distribution.

Chapter 5 studies data distribution from the application level. We review various languages which support parallel I/O operations for data distribution. In order to allow a user to obtain optimal data distribution performance and exploit the underlying hardware support, we develop runtime primitives to perform efficient collective I/O. These library routines optimize the two-phase data access strategy to perform the I/O and then redistribute the data. These routines are linked at compile time and can be used with either Fortran or C MIMD/data parallel dialect. These runtime routines could be ported to various parallel file systems, thus providing common I/O interface for parallel programs.

Finally, chapter 6 summarizes significant conclusions of experiments performed in previous chapters. We also present important contributions of this thesis and discuss some future work.

# Chapter 2

# Design Issues in Parallel I/O Systems

In this chapter, we study various issues involved in designing a parallel I/O system. The aim of this study is to explore various design factors and to understand the effect of these factors on the performance of the parallel I/O system.

The disk access overhead is one of the most prominent of the factors affecting the performance of I/O system. We will briefly review in section 1 the recent studies done in improving the disk access time. The various ways of disk interleaving will be studied in detail. Various data distribution protocols will be analyzed in section 2.

Section 3 describes various approaches used in interconnecting the disks with the processing nodes. The next section deals with the parallel file access strategies. This section describes various data access patterns found in parallel I/O systems. This leads us to the topic of disk prefetching and caching. Section 5 overviews various issues in this field. Finally, we describe the file system interface between the processing nodes and the file system in section 6. In the same section we discuss the requirements of the interface and survey some of the available file system interfaces.

## 2.1 Design of the Disk Systems

The low value of the I/O bandwidth is mainly attributed to the costs involved in the accessing of data from the disks. The data access rates are mainly limited by the speeds of physical motion of the heads and it is unlikely that these speeds will improve dramatically in the near future [PGK88]. The cost associated with each disk request has three main components, namely, seek time, latency time and read time. *Seek time* is the time required to position the access mechanism to the cylinder containing the data. The time required for rotating the correct data or track area under the reading head is called the *latency time*. *Data Transfer time* is the time required to transfer the data between the disk and a memory buffer. Hence the total data access time can be reduced by using techniques to reduce the above mentioned costs.

As the computational power of the machines increases, the corresponding memory requirement increases [Kun86]. One can have a single large capacity disk or one can design an array of inexpensive disks [KOPS88]. Both alternatives have some deficiencies. The single denser disk is error-prone [Mat77] and the array of disks cause reliability problems [Kim85].

The popularity of personal computers have decreased the cost of the hard disks. This has allowed the increase in the bandwidth of the disk system by replacing a small number of large disks with a very large number of small inexpensive disks. There exist several alternative ways to configure a number of disks to achieve high performance. We will consider disk striping, data interleaving and combinations of both [RB89a].

The RAID project developed at UC Berkeley [KOPS88, CGK+88, KGP89, KOP+89, CGKP90, CK91] incorporates an array of small inexpensive disks. The problem of reliability is solved by using extra disks to store redundant information. This approach

is called *low level parallelism* by Ghosh *et al.* [Joy93]. The RAID system breaks the disk array into different reliability groups, with each group having extra "check" disks to store the redundant information. The extra disks helps in reducing the mean time to repair (MTTR). The main advantages of the RAID system are cost-effectiveness, reliability, less power consumption and scalability. The RAID system is beginning to replace the conventional disk system in most of the advanced parallel machines including Paragon XP/S [Int92].

The most important advantage of disk arrays is the increase in the bandwidth that can be achieved in operating all the disks in parallel. The data can be declustered over a number of disks, thus permitting parallel access. Studies by Linvy *et al.* [LKB87] and Salem *et al.* [GMS88] show that a system with data declustering is better than a system without data declustering. Livny *et al.* show that when the block of I/O transfer is a track, a declustered system outperforms a normal system. The data declustering can be done in three possible ways: *independent drives* [KOPS88], *synchronous arrays* [Kim86b, Kim86a] and *asynchronous arrays* [Mic91]. The simplest alternative is the *independent drives* approach. In this approach, the drives are plugged into one or more servers using off-the-shelf controllers. This declustering system stores each file on only one disk. Reddy *et al.* describe such an approach as an example of the *traditional system* [RB90b, RB89a, RB89b, RB89d, Red92b, Red92a]. The job of operating them in such a system is left to the operating system or the user level software. This approach is suitable for application domains requiring a large number of independent requests. However, this method of disk interleaving may not work equally well for a single application. For example, if the files are not uniformly distributed, a large number of requests will arrive at a single disk, causing load imbalance.

The remaining two approaches deal with different ways of controlling the disk

heads in the interleaved disk array system. Both approaches distribute the data over the disk array such that succeeding portion of the data are in different disks. Synchronized disk interleaving allows a group of interleaved disks to operate synchronously. Byte interleaving stores adjacent bytes of a data block at the same place on each disk. Thus it becomes possible to synchronize the rotation of all disk units. This scheme of disk interleaving allows simplified control and multiple disk units can be treated as a single unit. Thus the disk array system presents a *Single device image*, greatly simplifying control problems that are inherent in multidisk systems. The addition of new disk units to the array does not affect the efficiency of the overall system. Hence this disk array architecture is well suited for scalable systems. Synchronous interleaving provides performance improvement, parallelism through interleaving, uniform distribution of I/O requests over the disks and improved reliability with minimum redundancy. The synchronous interleaving can be used for many applications that reference large address spaces and require large blocks of data. A few applications are database, AI and scientific applications requiring large data transfers such as nuclear simulations.

Kim *et. al* propose an alternative system of disk interleaving. Synchronous disk interleaving stores adjacent blocks of data on separate disks at a predetermined position. However, these subblocks could be placed on adjacent disks independently of each other. This is called *asynchronous disk interleaving.* This approach treats disks to be independent of each other. Since, the data is stored at different positions on each disk, the seek and rotational delay involved in the data transfer will be different for each disk. Thus the access delay of a request for a data block in an $n$-disk system is the maximum of $n$ access delays. The asynchronous system provides a queue for each individual disk. The reduced data transfer time in an asynchronous system will reduce the number of requests in the queue. The asynchronous system provides

much more scalability than the synchronous system. Asynchronous interleaving can be used to group several independent synchronously independent disks. Each group can be considered as a single disk and may form another level of disk interleaving.

Data declustering is incorporated in most of the new generation parallel machines including Intel iPSC series [RBA88, RB89c, RB90a], Ncube-2 [Jua92a], Intel Touchstone Delta [Int91b], Paragon XP/S [Int91a] and CM-5 [TMC91].

## 2.2 Data Distribution Protocols

Disk declustering allows multiple accesses to the disk system leading to an improvement in the I/O bandwidth. The main advantage of the disk interleaving is that the data can be distributed over the specified number of disks in the disk array. The time required to access the distributed data depends on the data distribution protocols used in the disk system.

For a given file size and the number of disks, the data could be distributed over the disk system in many ways [Fal91]. All these strategies distribute the data into blocks and distribute the blocks(buckets) over the disks using a specific distribution protocol. The data distribution protocols can be classified in three main groups,

1. The Random Data Allocation Methods.

   These methods distribute the blocks with the help of a random number generator. Generalized method of the random data allocation is the partition data allocation or PDA. These methods result in highly probabilistic data distribution patterns. Hence these methods are not generally favored because of the extra work required for finding the blocks of the distributed file.

2. Modulo Allocation Methods.(MAM)

   This method distributes the data across the processors using the generic modulo function. For a disk system having $j$ disks, the $i^{th}$ block of the file lies in the (i mod j)th disk. Such kind of modular mapping leads to an efficient round-robin distribution.

3. Minimum Spanning Tree and Shortest Spanning Path Declustering.

   Fang *et al.* proposed Minimal Spanning Tree (MST) and Shortest Spanning Path (SSP) declustering methods [Fan86]. The main idea behind this to distribute a set of data into groups such that the these groups share some specific attribute. These groups are then distributed over the disks. These methods are generally used only for distributions involving relations, hence these are useful for database operations. We will not examine these methods further since these are not of any practical interest.

Out of these three methods, the most important is the Modulo Allocation Method. MAM distributes the data over the disks in the most optimal way. The most important parameter in the MAM declustering is the *Granularity of Blocks*, which is the size of the primitive blocks into which the given set of data is distributed. The choice of the *granularity of blocks* is made by the specifications of the operating systems. The *granularity* determines the *degree of declustering*, i.e. the number of partitions of a given set of data. Larger is the granularity, smaller is the degree of declustering. Traditional computers use various degrees of granularities. For example, CM-5 uses 512 bytes as a basic block size, [Thi91] whereas the intel file systems use 4096 bytes as their basic block size [Int91b, Int90].

The data distribution protocol, along with the corresponding disk interleaving strategy define the data mapping on the disks. Hence both these parameters are

important in the design of parallel I/O system.

## 2.3 Processor-Disk Interconnection

This section reviews various issues in disk control and processor-disk interconnection.

One way to control an interleaved disk array is to provide a single controller doing all the file system and networking services. An excellent example of this is the microvax controller provided for the CM-2 data vault [TMC87]. Another approach is to provide individual controllers for the sets of disks. This approach is followed in the parallel file systems provided by the Intel i860 [Int90], the Intel Touchstone Delta [Int91b] and the nCUBE-2 [nCU92] system. This is an example of the *high level parallelism* [Joy93]. The Intel Touchstone Delta provides a separate I/O node per two disks. These nodes provide all the operating system services for the parallel I/O execution. The processing nodes access the disks through these I/O nodes.

The choice of the interconnection network decides the effect of the position of the I/O nodes on the I/O throughput. If the processing and the I/O nodes are connected by a high bandwidth network, then the position of the I/O nodes does not affect the I/O performance. If there is little overlap between the I/O access and the interprocess communication, then there is negligible degradation in using the common network. For example, the Intel Touchstone Delta uses the same network both for the interprocess communication and the I/O transactions. The CM-5 uses the fat-tree data network for its I/O messages.

As an alternative, Ghosh *et al.* propose using an independent network for the I/O services [Joy93, Joy91]. The network serves multiple I/O nodes operating under a distributed operating system. The data traffic in this I/O network is controlled by a pipelined circuit-switched routing technique called *worm-hole routing*. The extra

network makes the I/O bandwidth relatively independent of the data locality and the file allocation policy. This approach provides a balanced and scalable high bandwidth I/O network. However, provision of an extra network will make the disk control and the processor-disk interconnection more complex.

However, advances in the hardware and in the routing algorithms has made communication network equally suitable for the I/O purposes. Hence most of the advanced parallel machines use the same interconnection network for both communication and I/O.

## 2.4 Parallel File Access Policies

The data distribution over the disks allows the computing processors to access the data from the disks simultaneously. This permits the processors to read and write files in a parallel fashion. The various types of parallel access strategies decide the type of workload on the parallel I/O subsystem. In this section, we will review various data access strategies and their impact on the design of the parallel I/O subsystem.

*A file* is a collection of logically related data items. The file is composed of records which may or may not be of equal sizes. The records in the file are generally addressed by a file pointer. This pointer is used for various file related operations such as file rewind, file seeking etc.

From a parallel programming point of view, there exist two different views of a file. One is the *global View*, that is logical view of the file as an unit, where as the *local view* explains the file structure as viewed from the individual processor's point of view. Global view classifies files as either SEQUENTIAL or DIRECT, whereas the local view classifies the files according to the parallel file access.

When the file is opened in parallel, it is more appropriate to call it as a parallel

file. Parallel files can be classified into standard parallel files and special parallel files [Cro88, Cro89], [Kot92]. Standard parallel files are present independent of the execution of the parallel program. Examples of such files are system files such as .login files which are required by each processor independent of the program. The special files are created by the program and accessed by the processors only during the execution of the program. Such files include the data input and output files.

The workload on the parallel file system depends on the type of file accesses. There are many ways a parallel file can be accessed in a concurrent fashion. From a global view, these access patterns can be classified as SEQUENTIAL accesses or DIRECT accesses. Each group is again classified according to the local point of view. Let us first consider possible SEQUENTIAL access patterns.

1. Global Sequential All (GSA):

   In this pattern all the processors read the array from the start to the end sequentially. This process is mainly used for reading the standard system files.

2. Global Sequential Subset (GSS):

   In many cases, only part of the participating processors perform the I/O. This pattern is produced when the some of the participating processors read the entire file in a sequential manner.

3. Global Sequential Partitioned (GSP):

   This sequential pattern distributes the file across the processors, so the global view is that the entire file is read. The file is distributed into contiguous blocks, one for each processor. The processors read the contiguous partitioned data.

4. Global Sequential Interleaved (GSI):

This access pattern distributes the file into non-contiguous blocks separated by a constant stride. From the processor point of view, this is a round-robin access pattern. All the processors participate in this type of access.

5. Global Sequential Scheduled Access (GSSA):

   The GSSA allows an asynchronous data access. Each processor can independently access the records of the file. The sequence in which the records are read is decided by the order in which the processor requests arrive. The amount of data read by each request must be same, thus the amount of data read by each processor remains the same.

6. Global Sequential Overlapped (GSO):

   This is also an asynchronous data access pattern. But this access pattern allows the size of data request to be different, hence some requests might access overlapped data sets.

7. Global Sequential Partitioned Subset (GSPS):

   This is a modification of the GSP. In this form, some processors can remain idle. Thus the amount of data read by each processor is much greater than in the case of GSP.

DIRECT access files can also be classified from a global point of view.

1. Global Direct Random (GDR):

   The direct access pattern allows the processors to access the data from any position. The GDR allows the processors to access the data in any order and from any position. This is the most general case of data access.

2. Global Direct Partitioned (GDP):

In this access pattern, the processors read(write) data in a pre-determined order. But they use the DIRECT access pattern, reading(writing) the data in any arbitrary order.

There can be many more types of access patterns possible, but they are basically modifications of the basic file access patterns described here. The file access patterns are very important in designing the parallel I/O system, since it decides the memory access strategy used in the parallel machine. The file accessing pattern has a profound effect on deciding the caching policies in multiprocessor systems.

## 2.5 Disk Caching and Prefetch Policies

The provision of a multiple disk system helps improve the I/O bandwidth of a multi-processor system. The effective I/O bandwidth can be further improved by providing effective caching and prefetching policies.

The choice of a caching policy depends on the workload model. The lack of real parallel I/O workload has led designers to use artificial workloads for the designing caching policies. We have to consider file access patterns other than the disk access patterns. The disk access patterns refer to analyzing the accesses to the physical blocks of the data residing on disks. This approach is complicated because the data is striped over the disk array and is accessed by multiple nodes. Instead of studying individual disk accessing, we study the file access patterns.

The applications access distinct data from a file (records). This record access is translated into the corresponding block access by the file system. The blocks are either read(written) on the target disks according to the application specifications.

The file access patterns have been studied extensively. Survey of IBM systems and Cray systems conclude that most files are accessed sequentially [Pow77, Ber78].

Even in parallel computers, the most used access is the sequential access.

Disk caching is heavily used in uniprocessors for improving the disk I/O performance. The hit ratio of the cache can be improved by reading the blocks into the cache before they are requested. This is called prefetching of data blocks. The most common technique used for caching is the one-block lookahead or OBL. This strategy prefetches the (i+1)th block when the i'th block is prefetched [Smi78].

However in the parallel file systems, the prefetching strategy depends upon many factors [Dav90, EK89, KE92, KE91, Kot91]. These include what to fetch, when to fetch, which disk to fetch from, which processor should make the decision of prefetching and where the data should be stored. The common prefetching strategy used for the uniprocessors may not work here because of different file access patterns. When there are a large number of processors accessing the data from several blocks, it may be beneficial to prefetch the blocks that can be accessed in near future. Thus the many-block lookahead (MBA) might be a good replacement for (OBA).

Kotz *et al.* have performed experiments on the parallel file system testbed, the RAPID Transit System [Dav90]. According to their results, the prefetching contributes to the decrease in the execution time. The improvement ranged from 15 percent to 70 percent. Similar work was carried out by French *et al.* [FPD91a, FPD91b]. They studied the effect of prefetching and caching on the performance of an Intel iPSC-860 hypercube. They have found 217% gain for prefetching and 57% gain for caching.

The read access times can be improved by providing prefetching mechanisms. Similar improvement in write throughput can be obtained by providing suitable write-back strategies [KE93]. Also in the write(read) operation the choice of replacement policy is important. The replacement policy depends on the locality of access. In addition to *special* and *temporal* localities, Kotz *et al.* define another form of locality

called *interprocess locality*. In the interprocess locality, the data block will be used in near future by another process. Analogous to one-block lookahead in prefetching, the sequential locality results in *toss-immediately* replacement policy [Sto81]. In the parallel access environment a slight modification of the toss-immediately policy will be more useful. The parallel toss-immediately policy retains more than one MRU blocks resulting in a efficient cache performance.

The *write-behind* policy can improve the write performance by allowing the process to continue while the data is written to the disk. This policy is used in the Touchstone Delta Concurrent File System [Dav92]. A slight variation of this policy writes the data to the disks after a slight delay resulting in *delayed write-back*. Another strategy is to write to the disk when the write-buffer is full. This *write full* strategy is more useful for sequential access patterns. This policy writes the old, not used blocks after a certain delay. Thus write-full policy becomes write-back for non-sequential patterns [KE93]. Kotz *et al.* have shown that write-full strategy is the best write strategy for scientific workloads.

A lot of work is still needed for analyzing both read and write cache performance under various file system parameters. Some important issues to be considered include cache coherence protocols, prefetching policies and parallel replacement strategies.

## 2.6 OS Interface between the Processing Nodes and the File System

Parallel file systems allow parallel access to the data distributed over multiple disks. Conventional (UNIX-like) interfaces cannot fully satisfy various conditions required for allowing access from a parallel program. This section tries to emphasize pertinent points in the design of the parallel I/O interface.

The traditional UNIX system interface treats the file as a single stream of bytes. The file is opened either by *create* or *open*. The files are accessed sequentially using a file pointer. System calls *lseek*, *read*, *write* and *close* use this file pointer. File information is stored in the global *inode* structures. All file accesses are sequential and at one instant only one process can access the given file.

A parallel interface must also take into account various other factors such as the workload models (explained in section 2.4), parallel system calls, distributed data management and parallel request scheduling. The main aim of a parallel file system interface is to provide the user a transparent service and allow him to utilize the parallel I/O system to the fullest extent. We describe below some important issues that affect the design of a parallel file system interface.

1. Uniform directory structure.

   The files should have simple and uniform hierarchical structure. It should not be dependent on the actual storage pattern of the file data and on the pattern of file access (sequential or parallel). A user should visualize a unified file system consisting of both the sequential and parallel files. This would mean that the same call can access both the sequential as well as the parallel files. This makes the file system implementation completely transparent to the user.

2. Ability to execute multiple file accesses.

   The main task before the parallel file system interface is to serve multiple file access requests concurrently. These include file open, file close and file pointer operations such as lseek. For a file to be accessed concurrently, there should be a provision for multiple file pointers. One can have a file pointer per process or all the processors can share the file pointer. This

choice depends on the file access pattern. When all the processors need to read the same data in different order (GSA), independent file pointers are suitable. For a distributed data access (read/write)(GSP/GSI/GSSA), processors require a common file pointer. The resulting file structure depends on how the processor synchronization issue is handled. Other system calls such as lseek involve file pointer manipulations. When the files are distributed over the disks and opened in the interleaved fashion (GSP/GSI/GSSA), the lseek operation becomes quite complicated. Hence the file system should take care of irregular file pointer operations.

The multiple file operations should be executed in an optimal way, so as to reduce the number of requests to the file system controller. Also the file system should support operations on extremely large files (larger than 1 GBytes).

3. Availability of efficient read/write system calls.

   The most important factor in the design of the operating system interface is to provide for read/write system calls for multiple file access requests. The data to be read/written will be distributed over a number of disks. The design of the read/write system calls decide the time required to access the required amount of data from the disks. The larger is the amount of data read per request, the smaller is the total time required for reading the total data.

   To facilitate a large access capacity, parallel file systems normally use buffered read-write. Buffer-reads read requested block of bytes in a serial fashion. Position of data block in file depends on file pointer position. The file pointer will be controlled by the operating system and will vary depending on the file distribution pattern. Hence, it is necessary to know

file distribution pattern before data read and write.

4. Provision of a transparent data management scheme.

   Traditional file systems use various data structures to effectively track the positions of the stored data. The data book-keeping becomes very important in parallel file systems because multiple processes access distributed data concurrently.

   Distributed data management stores the file information such as the file size, the mode of file access and the number of disks on which the file is stored. The file information is stored in the usual inode form which is accessed by the I/O controller. This bookkeeping hides the actual data storage from the user. The user just decides the total number of disks on which the file is stored. All the internal file details are maintained by the data management scheme.

   I/O controller is an important part of the distributed data management scheme. The I/O controller receives the requests from the users (I/O requests) and translates them into the corresponding disk requests. The I/O controller acts as a gateway to the disks. The I/O controller implementation is file system dependent. There may be a single I/O controller for the entire disk system or each subset of disks will have a separate I/O controller. The I/O controller maintains a transparent file structure and allows the user to write simple read/write routines.

5. Portable with standard file systems.

   A parallel file system treats file data in a different way than the traditional file systems. The file system provides specific data structures and data management policies for supporting multiple accesses of the distributed

data. There are separate file system calls for serving the multiple file requests.

However, the parallel file system should be able to run the traditional file system calls. The user should be able to run the normal operations on the file (for example, tar,compress etc). Also the file system should accept common file system verification calls such as df or du. Finally to make the parallel file system fully portable with the standard file systems, the internal file structure of the parallel file system should be similar to that standard system (hierarchical file structure, simple consistent interface to peripheral devices etc).

### 2.6.1 Existing Parallel File System Interfaces

Several new parallel processing systems now provide parallel file systems. These file systems consider most of the previously discussed issues. In this section we will review the recent trends in this field.

The CUBIX file system for the CrOS system was designed on the MARK range of hypercubes [GMG+88], [Sal86]. The CUBIX system proposed a single programming paradigm for the hypercubes. A universal host program acts as a file server serving the requests from the nodes. There are two modes of file accesses: crystalline also called loosely synchronous and amorphous. In the crystalline case, the files have a single file pointer and have two accesses- single or multi. In single access mode, each node has to send the request at the same time resulting in a single input/output. In the multi mode, each node can send individual requests having different parameters. In the amorphous mode each file has different file pointer.

The nCUBE hypercube multiprocessor range also supports parallel file system

[DeB91, Jua92a, nCU92, DdR92, DM91, Jua92b]. The nCUBE system is designed around the UNIX system. This system uses byte mapping for providing the routing information for the data in the file. In nCUBE system, the user can control the data storage on the disks by providing the stripe size of the declustering. The nCUBE system distributes the files over the disks in a round robin fashion.

Intel iPSC/2 and iPSC/860 support the Concurrent File System or CFS [Int88, Pie89], [Int89, Nit92, BR89], [FPD93, FPD91a, Int90]. The CFS provides three interfaces [AS89],*standard,random-sequential access* and *coordinated interleaving.* This is carried out using distinct file access modes. User has the ability to control the disk storage of the file data. Similar file system is used for the Intel Touchstone Delta [Int91b, Raj93]. The Intel Paragon XP/S will use a distributed operating system based on the OSF/1 [Int92]. Both the file systems support multiple file accesses using various file access modes.

Other commercial computers providing parallel file interfaces include the Kendall Square Research KSR1 [KSR92] and the CM-5 [TMC91]. The KSR1 uses shared memory address space for file mapping and provides a RAID architecture for the disk system. The CM-5 uses the data vault with a microvax as a I/O controller. The CM-5 system software provides file access modes for random-sequential and coordinated access.

Several researchers have proposed prototype parallel file interfaces. Kotz explains a parallel file interface that provides multiple file operations [Kot92]. Similar parallel file operations have been defined in the Bridge file system [DSE88, DS92]. Witkowski *et. al* describe a parallel I/O interface for hypercubes [And90]. PARAM - a transputer based parallel machine developed in India also provides a parallel file system [USP91].

# Chapter 3

# Case Study of a Parallel I/O System

Chapter 2 reviewed various issues in designing a parallel file system. To understand the effects of these issues in a greater depth, we perform an experimental analysis of Touchstone Delta Concurrent File System (CFS). The chapter focuses on file read/write rates for various configurations of I/O and compute nodes. The study attempts to show the effect of file access modes, buffer sizes and volume restrictions on the system performance.

This chapter is organized as follows: section 1 describes Touchstone Delta system. The Concurrent File System or CFS is introduced in section 2. Section 3 explains the evaluation methodology used in the experiments. Detailed explanation of various experiments along with the results is presented in section 4. Section 5 introduces Parallel File System for Paragon. Finally we conclude in section 6.

## 3.1 The Touchstone Delta System

The Touchstone Delta system was developed by the Intel Corporation as a part of the Touchstone program [Int91b]. The Intel Touchstone Delta system is a message passing multicomputer consisting of processing nodes that communicate across the two dimensional mesh interconnection network.

The system supports various types of processing nodes (numeric, mass storage, gateway and service). Numeric nodes form the computational core of the system. The Delta system uses Intel i860 processors as the core of computational nodes. In addition the Delta has 32 Intel 80386 processors as the core of I/O nodes. Each I/O node has 8Megabytes memory which serves as I/O cache. There are other processor nodes such as service nodes and ethernet nodes. The Delta is arranged as a mesh of 16*32 compute nodes and has 16 I/O node on each side(as seen in figure 1).

## 3.2 Concurrent File System

The Intel Touchstone Delta Concurrent File System consists of I/O nodes connected to disks. Each I/O node is connected to 2 disks, each with 1.4 Gigabytes of space. I/O nodes do not run any application processes but provide disk services for all users. A file is uniformly distributed over all 64 disks by default in a round-robin manner. The stripe unit is 4Kilobytes(one block) per disk. When a file is opened for reading or writing, data is accessed by default from 64 disks. A user, however, can restrict the number of disks on which a file is distributed. All read-write transactions are carried out in an integral number of blocks, where each block size is 4 Kilobytes.

Figure 1: Intel Touchstone Delta System

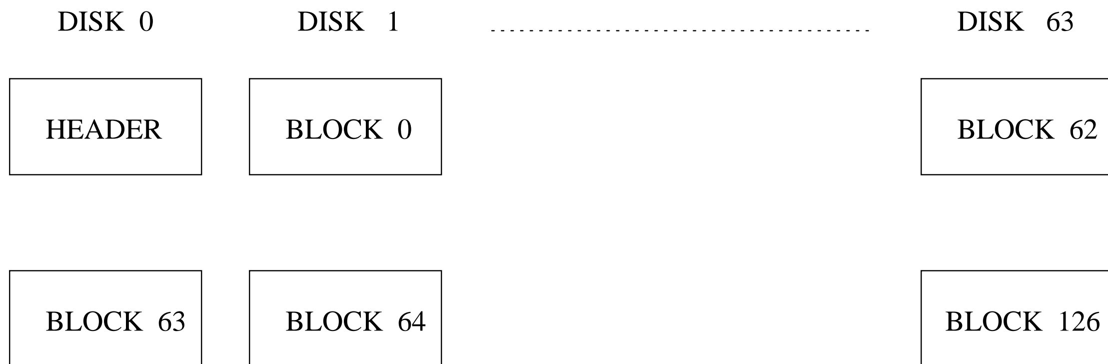


Figure 2: File block distribution across the disks

### 3.2.1 CFS File Structure

The CFS provides a UNIX view of a file to the application programs. Each CFS file has a header and a body. CFS file header stores file information such as file size, permission and link count. The file header is always allocated the first file block. In case of small files the header contains the data whereas for large files the header contains the pointers to the indirect blocks that store the data. When the file is striped across the disks, the file header is stored on the first disk and all the subsequent blocks are distributed in a round-robin fashion over the disks (figure 2).

### 3.2.2 The File System Interface

Touchstone Delta CFS has four major components: CFS libraries, file Checkers, DiskProc and NameProc [Dav92].

CFS libraries are linked with source programs. These libraries create and send messages to DiskProc and NameProc. In addition, the job of file offset maintenance and execution of I/O modes is assigned to these libraries. CFS libraries are also responsible for asynchronous I/O.

File checkers check the file information and create the necessary file structures.

These are installed on the i860 (compute) nodes. These check the directory and the CFS file headers for errors. If any errors are found, file checkers modifies it. Otherwise, theya creates the block map of the file and sends it to DiskProc.

NameProc is resident on the *nameserver*. It provides the CFS naming services and maintains the CFS directory file. It translates the CFS pathnames into corresponding file header blocks. DiskProc is resident on all the I/O nodes. DiskProcs provide interface between CFS libraries and I/O devices. In addition DiskProc executes the CFS file open calls.

When a compute node opens a file, the operating system sends a message to *nameserver*, which responds with a copy of the meta data for the file so that the compute node knows where each block of the file resides. When there is a read or a write request, the compute node translates the logical file offset obtained from the nameserver into the physical block reads (writes). Subsequently, the compute node sends the read or write requests directly to the I/O processor that owns the disk. When a read request is presented to an I/O node, it prefetches 8 blocks of 4Kbytes each into it's cache (if these blocks do not exist in the cache). Therefore, for each read request each I/O node prefetches 32 KBytes of data. During writing, the I/O node stores the data to be written in the I/O memory up to 12 blocks before actually writing to the disk.

### 3.2.3 The I/O Modes

Four I/O modes are supported in the CFS. These are described below.

- Mode 0: In this mode, each node process has its own file pointer. This mode is specifically useful for large files to be shared among the nodes. Here, sharing implies that the same data is accessed by nodes (replicated). This should be

distinguished from sharing a file but distributing the data, i.e, when different nodes access different (and distinct) parts of a file. This mode is useful for accessing the file in GSA access pattern.

- Mode 1: In this mode, compute nodes share a common file pointer. I/O requests are serviced on a first-come-first-serve basis. Nodes can read and write at any point, but they use the same file pointer. Thus GSP file access pattern is obtained.

- Mode 2 : Mode 2 treats reads and writes as global operations. The set of compute nodes that open a concurrent file must read the file in a specified order (in the increasing order of the node-numbers). This mode performs global synchronization in the sense that the second request by any node is blocked until the first request by all nodes in the set is complete. This mode supports synchronized common file pointer. Using this mode, nodes can perform variable length read-write operations. Hence, the requests are serviced in a predefined order.

- Mode 3: Mode 3 is a synchronized ordered mode. The difference between mode 2 and mode 3 is that in mode 3, all read/write operations must be of the same size. This mode also supports global synchronization. Hence, the requests can be serviced in any order, but still the second request by a node is blocked until the first request of all nodes is completed. Hence this mode can be used for obtaining GSI form of file access.

The data accessed by each processor depends upon the mode. During the write operation, the resultant size of the file created depends on the file mode used. Many scientific applications involve automatic distribution of the data across processors.

Using different I/O modes, it is very easy to decompose the data across the disks. Note the distinction between mode 0 and the other 3 modes. In mode 0, reads/writes are to the same data in a shared file, whereas in other modes reads/writes are to distinct data (for each node) in the shared file.

### 3.2.4 The I/O Network

Touchstone Delta does not provide an independent I/O network. The compute and I/O nodes share a common interconnection network. The same network is used for both interprocess communication and I/O communication.

In Touchstone Delta, both interprocess communication and I/O, messages travel in the form of packets. Touchstone Delta uses *packet switched wormhole* routing as a communication protocol [Int91b, Lio93, Rik92]. Each node of the machine is connected to the mesh using a mesh routine chip (MRC). Messages travel from MRC to MRC until they reach the destination node.

Each message is split into packets of a fixed size (512 Bytes). On the physical level, the packet travels through the network in form of *flits* or *flow control digits* [Lio93, Bar93]. The packet follows a XY routing protocol. The XY direction is specified in the message header. In Touchstone Delta, the message packets always travel first along X direction. During the journey, each MRC decrements the X offset. When the X offset becomes zero, the packet travels in the Y direction. The message reaches the destination when both the X and the Y offset of the message become zero.

Using the same network for interprocessor communication and I/O may cause serious network contention. Also since the I/O nodes are physically at the edge of the mesh, the position of the compute nodes might affect the I/O performance. We will investigate these points in the experimental analysis of Touchstone Delta.

## 3.3 CFS Evaluation Methodology

The overall performance of accessing data in the CFS depends on several factors such as the number of compute nodes participating in an I/O operation, size of access (buffer size), number of disks, block size, I/O mode and the overall available bandwidth from the I/O system as well as that of the interconnection network (Figure 1). In principle, it is difficult to decouple the influence of some parameters on the performance. Our study includes the following experiments:

- Single Compute Node
  - Single compute node and paged I/O

    In paged I/O experiments, we study the effects of buffer size, node position, number of disks and the file size on the throughput. These experiments are carried out for smaller buffer sizes and relatively small file sizes.
  - Single compute node and burst I/O

    For burst I/O experiments, the buffer size is very large. The factors that affect the burst mode throughput include the buffer size and the number of disks.
- Multiple Compute Nodes
  - Multiple compute nodes and paged I/O

    Multiple processors access the file system using various access modes. During multinode accesses, the CFS performance not only depends on the buffer size, number of processors and disks, but also on the file access modes. Hence, the paged I/O experiments are carried for all the four file access modes.

Table 1: Definitions of Various Terms Used in the Thesis

| Term | Definition |
|---|---|
| F | The size of the file distributed over the disks. |
| $N_p$ | The number of processors accessing the file. |
| $F_p$ | The amount of file data per processor.<br>$F_p = \frac{F}{N_p}$.<br>(True only for modes 1,2 and 3.) |
| $N_{io}$ | The number of I/O requests per processor. |
| $b_{io}$ | The size of the data buffer.<br>Access size requested by the processor |
| $N_D$ | Total number of disks (for Delta $N_D$ is 64). |
| $B_D$ | Block Size ( Amount of data per block on each disk). |

– Multiple compute nodes and burst I/O

In burst I/O mode, the buffer size is quite large compared to that in the paged mode. For such file accesses, the parameters that affect the throughput include the number of processors and the number of disks. Furthermore, we study the performance of the CFS when data is distributed using common data distributions found in typical scientific programs.

- Effect of Common Interconnection network on I/O Performance

We will investigate the effects of using the common network for both interprocess communication and I/O. We will especially study the effect of compute node position on the I/O performance. The results will help in analyzing the mesh network performance.

Table 1 presents some definitions that will be used throughout this thesis.

## 3.4 CFS Performance Evaluation

The main objective of the CFS performance evaluation is to determine the maximum read-write rates observed for different configurations. Therefore, the experiments try to saturate the I/O system with the I/O requests so as to obtain a peak performance. Similar performance measurements have been used in the study of Intel iPSC I/O system [FPD91a, FPD91b, FPD91c, FPD91b, Nit92].

### 3.4.1 Single Compute Node

The first part of the study aims at determining the maximum I/O rates obtained for a single compute node. These studies are performed both for paged as well as burst I/O modes. Paged I/O performance is important for implementing and supporting node virtual memory to fetch or store pages on disks. Burst-mode I/O is important for file accesses when a node requires reading/writing large files containing data for an application. For both types of workload we study the maximum throughput obtained from the I/O systems.

#### Paged I/O

Since there is no virtual memory support currently available on the Delta system, virtual memory was simulated by opening a file and reading (writing) it using fixed size buffers. The buffer size $b_{io}$ indicates the amount of data fetched in each I/O request. In each experiment, the compute node opens a file and reads (writes) it using fixed size buffers. Other parameters varied for the following experiments include the file size, buffer size, node position and the number of disks $N_D$.

Figure 3 illustrates the performance of implementing paged I/O using various buffer sizes. As the buffer size increases from 1K, $N_{io}$ reduces and consequently

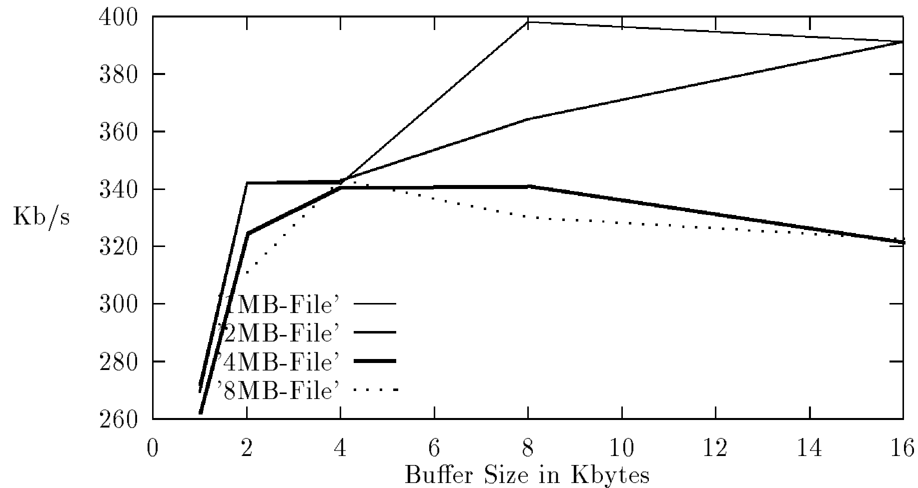


Figure 3: Single Compute Node - 32 I/O Nodes (64 Disks): Read Rates in KB/sec

the throughput increases. For a buffer size of about 4k, the read rate is about 340Kbytes/sec for all file sizes. This convergence of performance occurs because the buffer size and the block size are both 4Kbytes, and therefore, the requested size is same as the size of data read in one operation from the disk. Thus, each I/O request for 4K buffer results in reading the prefetched blocks from the node cache. Beyond 4K buffer size, the throughput increases for small files as a function of buffer size whereas it degrades slightly for larger files. In summary, 4K buffer size seems optimal for most of the files but for very small files, larger buffer sizes perform well. It should be noted that the throughput is not limited by the I/O system, but is limited by how fast a node request is generated. For the paged I/O experiments, new I/O request is generated only when the previous one is completed. Similar results were obtained for a write operation.

In order to determine if the position of a compute node in the network has any

Table 2: File Read Time is (ms)-Single Compute Node

| Node Position | 1 MB Read | 2 MB Read | 4 MB Read | 8 MB Read |
|---|---|---|---|---|
| 56 | 2954 | 5836 | 11642 | 23230 |
| 11 | 2926 | 5896 | 11754 | 23195 |
| 24 | 3018 | 5836 | 11749 | 23314 |
| 319 | 2968 | 5843 | 11694 | 23205 |
| 268 | 2951 | 5936 | 11729 | 23215 |
| 567 | 2994 | 5903 | 11941 | 23264 |
| 535 | 3078 | 6096 | 11790 | 23460 |
| 517 | 2972 | 5961 | 11897 | 23383 |

effect on read and write operations, different nodes at various locations in the mesh were chosen as shown in Table 2. Three were on one side of the mesh (Node Nos. 56,11,24), two in middle of the mesh (Node Nos. 319,268), and the remaining three on the other side of the mesh. Keeping the buffer size fixed at 4K, file read times were observed for file sizes varying from 1M to 8M. The files were distributed over 64 disks, so each side had an equal amount of data distributed on the nearby disks (32). The read times do not change significantly as the position of the node varies. This experiment shows that the distance that a request travels in the network does not have any significant impact on the performance when there is very little contention in the network. This also shows that the inter-node "hop times" between the nodes are negligible.

### Burst Mode I/O

When the compute nodes require to read (write) a large amount of data (large fraction or an entire file) then the operation may be performed in a burst mode. In burst mode, the buffer size is very large, maximum being the file size being accessed. Figures 4

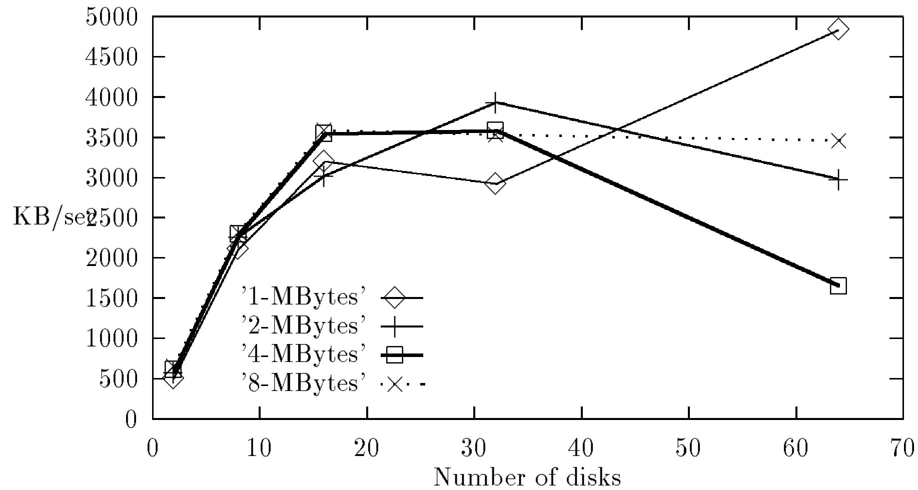


Figure 4: Read Rates for Single Compute Node-Burst Mode

and 5 show that burst mode operation is much faster than the paged mode. Using 64 disks, 1MB file was read in 203 ms giving a peak rate of 4.83 MBytes/sec. The peak write rate was 1.39 MB/sec. The peak read rate obtained for paged mode was about 400 KB/sec. This increase in the throughput, compared to that in paged mode is observed due to the large number of I/O requests in paged I/O mode where each request must be sent explicitly.

Generally for the single processor configuration, both for the paged and for the burst mode I/O, the read rates are much higher than the write rates. Also as the number of disk volumes increases the throughput increases upto a threshold. For large files, the trend will be the same as the "8-Mbytes" case shown in Figure 5.

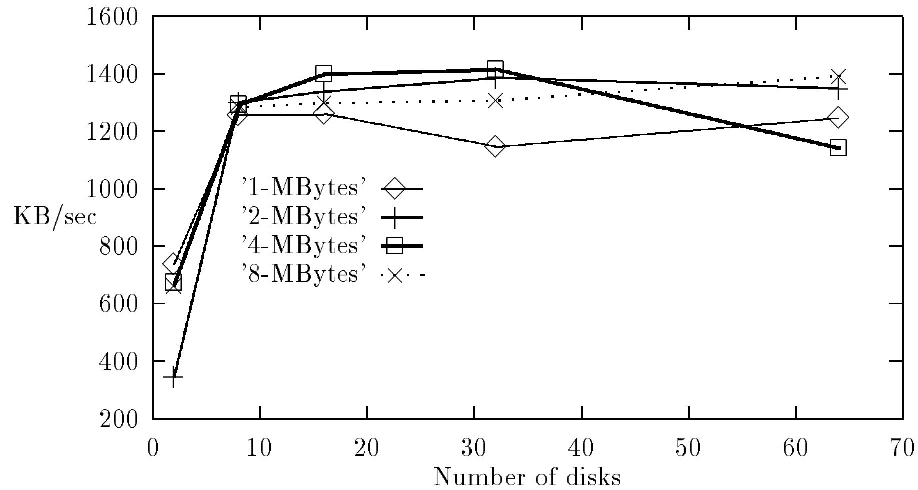


Figure 5: Write Rates for Single Compute Node-Burst Mode

### 3.4.2 Multiple Compute Nodes

The most important use of a parallel I/O system and CFS is concurrent accesses by multiple processors. This section presents performance of the file system by varying different parameters such as the number of disks, the access modes and the number of processors.

#### Mode 0: Paged I/O

Mode 0 is useful for accessing shared files by multiple compute nodes. Each processor has its own file pointer. Note that write operations are not protected in the sense that the processors can overwrite each others data.

Figure 6 shows the read throughput for paged I/O as the function of number of processors for various number of disks. There exists a threshold in terms of the

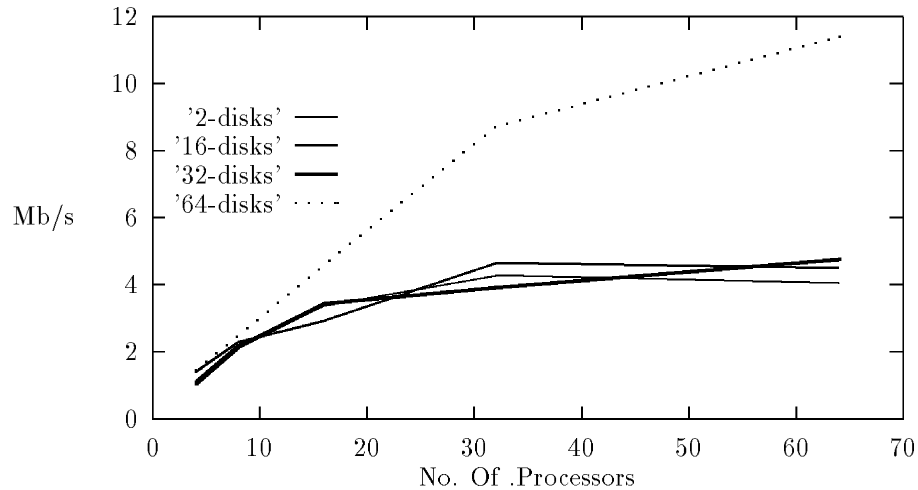


Figure 6: Read Rates For Multiple Compute Nodes - Mode 0

number of disks beyond which a substantial performance gain can be expected. As the number of processors is increased, the performance does not change significantly when $N_d$ is increased from 2 to 32 disks. However for 64 disks, there is a significant jump in the performance. The throughput obtained for 64 processors is 11.2 MBytes/sec. Whereas for 2 disks, the throughput is about 4.5 Mbytes/sec. This shows that the "declustering" of the file data is very effective.

The throughput increases as the number of processors increases. Since, each processor reads the same data, as the number of processors increases, the total amount of shared data accessed increases. However, the total time required to read the data increases slowly because one node's read acts as a prefetch command for others.

Another interesting point to observe is that for a small number of processors, the effect of the number of disks on the performance is negligible. That is, the performance is limited by the bandwidth available at the computational node side rather than at

Table 3: Multinode (4*4) Burst Mode Throughput as a Function of Disk Volumes (Mode 0)

| File size | Buffer size | Disks | Read (MB/s) | Write (MB/s) |
|---|---|---|---|---|
| 1M | 1M | 2 | 0.9329 | 0.748 |
| 1M | 1M | 16 | 5.88 | 3.069 |
| 1M | 1M | 32 | 11.11 | 3.147 |
| 1M | 1M | 64 | 11.21 | 3.506 |
| 4M | 4M | 2 | 0.74 | 0.603 |
| 4M | 4M | 16 | 3.630 | 2.0878 |
| 4M | 4M | 32 | 7.705 | 4.947 |
| 4M | 4M | 64 | 12.11 | 6.055 |

the I/O subsystem side.

**Mode 0: Burst I/O**

Table 3 shows the performance of burst-mode read/write throughput as a function of number of disks (processor size = 16). The specified buffer size at the application level is equal to the size of the file to be read/written. As we can observe, the performance improves as the number of disks is increased. For a given processor grid size, the performance depends on two parameters, namely, the file size and the number of disks. For smaller files (1 Mbytes/node), the performance saturates for smaller number of disks (32 disks) and beyond which the improvements diminish. However, as the file size increases, the threshold number of disks for which performance improves also increases as seen for the case of 4 MByte read/write per node.

Tables 4 and 5 presents the performance of burst mode I/O when the number of computational nodes is varied from 16 to 512 ($N_D = 64$). In general for burst I/O, good performance is obtained and the saturation occurs due to bandwidth limitation of the I/O subsystem. Reads perform better than writes. This is because reads can

Table 4: Throughput of Accessing 1 MB File in Mode 0 (Burst Mode,64 Disks)

| Mesh Size | Write Rate(MB/sec) | Read Rate(MB/sec) |
|---|---|---|
| 4*4 | 3.506 | 11.21 |
| 4*8 | 5.421 | 11.75 |
| 8*8 | 7.027 | 11.672 |
| 8*16 | 3.314 | 11.766 |
| 16*16 | 3.244 | 11.813 |
| 16*32 | 2.839 | 11.992 |

Table 5: Throughput Rates of Accessing 2 MB File in Mode 0(Burst Mode)

| Mesh Size | Write Rate(MB/sec) | Read Rate(MB/sec) |
|---|---|---|
| 4*4 | 4.37 | 10.51 |
| 4*8 | 3.89 | 11.73 |
| 16*16 | 2.860 | 12.103 |
| 16*32 | 3.678 | 23.83 |

be performed from the I/O cache if a desired block exists in the cache due to an access by some other processor. However, all writes must be written onto the disks. The difference between paged and burst I/O performance is not significant because the system bandwidth is not limited.

### Modes 1, 2 and 3 : Paged I/O

These modes are useful for accessing data when the data set is distributed over multiple nodes.

The first experiment was used to observe the effect of different modes and the number of disks. In this experiment a data file of size 16 Mbytes was read using modes 1, 2 and 3. As Figure 7 shows, for a 4*4 processor grid the maximum throughput is

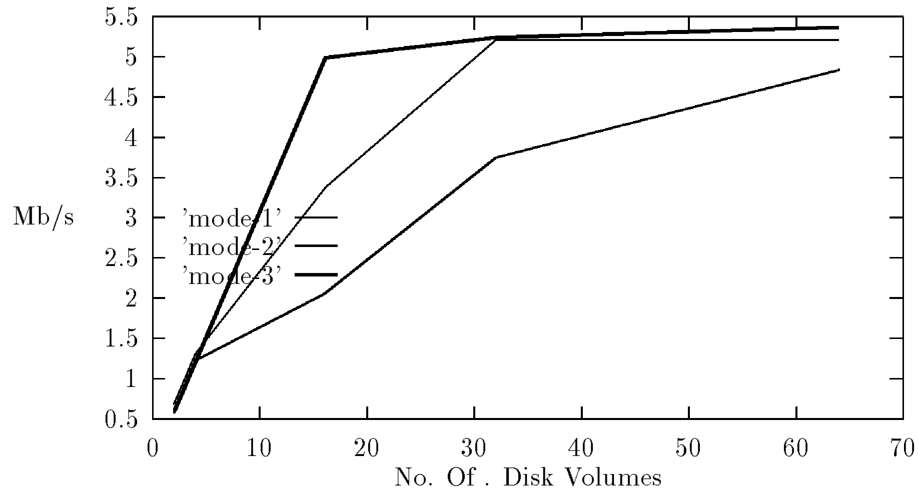


Figure 7: Read for Grid Size 4*4

obtained for mode 3 and with 64 disks. The peak rate in this case is 5.5 Mbytes/sec. For mode 1, the peak speed is 5.12 Mbytes/sec. The lowest file read throughput is observed for mode 2.

Another important point to be noted is that as the number of disks decreases, the read throughput decreases. As the number of disks is decreased below a threshold (in this case 16 disks), the read rate reduces drastically. Hence, the optimal operating point in terms of cost-performance (no. of disks versus the throughput) will be near the knee of the curve.

In the next experiment, we use 64 disks ( maximum available) and vary the number of processors. For this experiment, a 16 M file was read using a 4K buffer.

Figure 8 shows that as the number of processors increases the read throughput increases. The highest read throughput was obtained for 64 processors. Modes 1 and 3 perform comparably. The maximum read rate for mode 1 was 10 Mb/sec,

while for mode 3, it was 9.8 Mb/sec. Due to access ordering and synchronization costs, mode 2 performs worse than the other modes for all cases. As the number of processors increases, the amount of data read per processor decreases. Hence, the individual processors require less time to read. In other words, the available bandwidth at the computational node side increases resulting in an increase in the throughput. However, as the number of processors increases, the rate of increase in the throughput decreases indicating that the bandwidth limitation shifts to the I/O system side.

For studying the performance of each I/O mode in some more detail, we performed three additional experiments in which the number of processors and the number of disks were varied. For these experiments, a 16 MB file was opened and read by varying number of processors. Figures 9, 10, 11 show the effect of varying disk volumes and number of processors for different I/O modes.

Figure 9 shows the performance of the file system for mode 3. The peak read performance is obtained for 64 disk volumes. This figure also shows that the throughput is proportional to the number of processors. For a 64 processor grid, a 16 MB file was read at 10 MB/sec. For the same grid size, if the file is stored on 2 disk volumes, the read rate drops to about 900 KB/sec.

Note that as the number of processor is increased, the knee of the curve is observed at different points for different number of disks. Hence, as indicated earlier, the choice of number of disks depends on how many processors will be involved in an access. More experiments need to be performed to relate this performance to different file sizes. Note that the knee of the curve in these experiments signifies a point indicating the bandwidth limitation shift from the computational nodes to the I/O system.

Figure 10 shows the results for the same experiment for mode 2. The graph shows nearly same trends as observed for mode 3. However, for mode 2, the peak

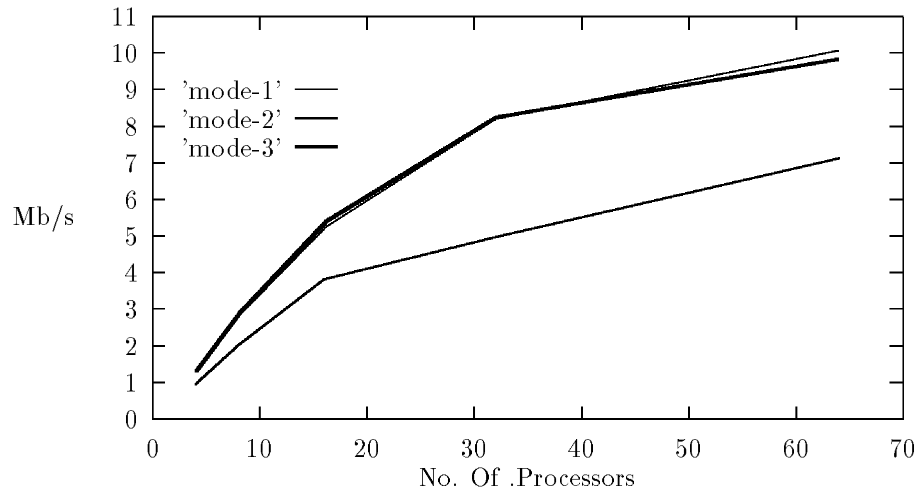


Figure 8: Reading a 16M File using 4K Buffer for 64 Disk Volumes

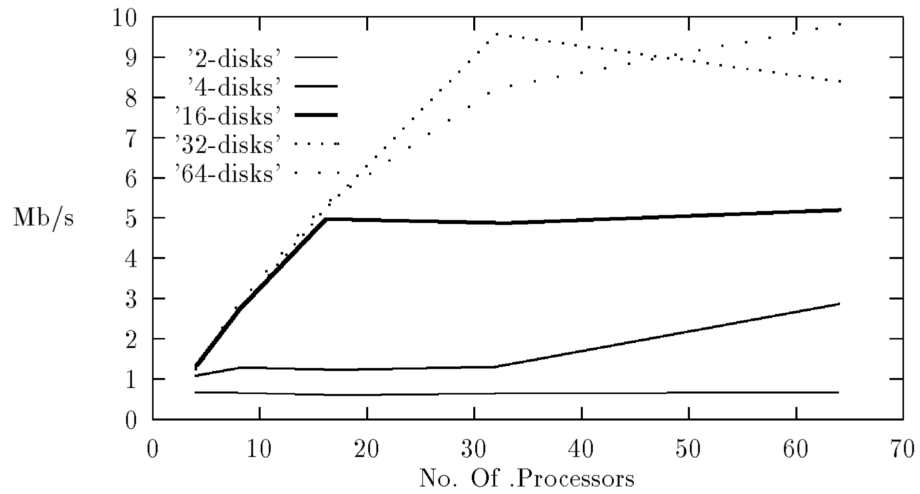


Figure 9: Multicompute Nodes Read (Mode 3 and 16 Mbytes File)

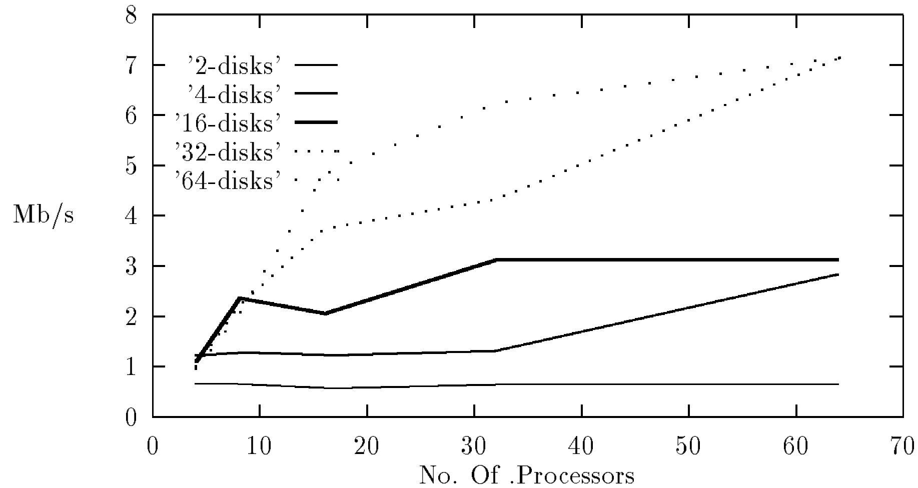


Figure 10: Multicompute Nodes Read (Mode 2 and 16 Mbytes File)

rate obtained was 7MB/sec for 64 processor grid. Note that the performance in mode 2 is sensitive to the order of arrival of the requests because requests must be served in a fixed order. Therefore, we observe a less smooth curve as compared to that for mode 3.

Figure 11 shows the performance of the multicompute nodes for the mode 1. The peak throughput is 10 MB/sec and lowest observed throughput is 900 KB/sec. Note that mode 1 serves requests in the order of arrival and does not require synchronization for each processor to finish before going to the next phase. Therefore, it performs slightly better than mode 3 (which requires synchronization). Therefore, mode 1, is useful for log-structured files or for those computations in which order of accesses does not matter. For example, if the compute nodes perform a search operation in a file, they can access the file in a self-scheduling mode for which mode 1 will provide the best performance.

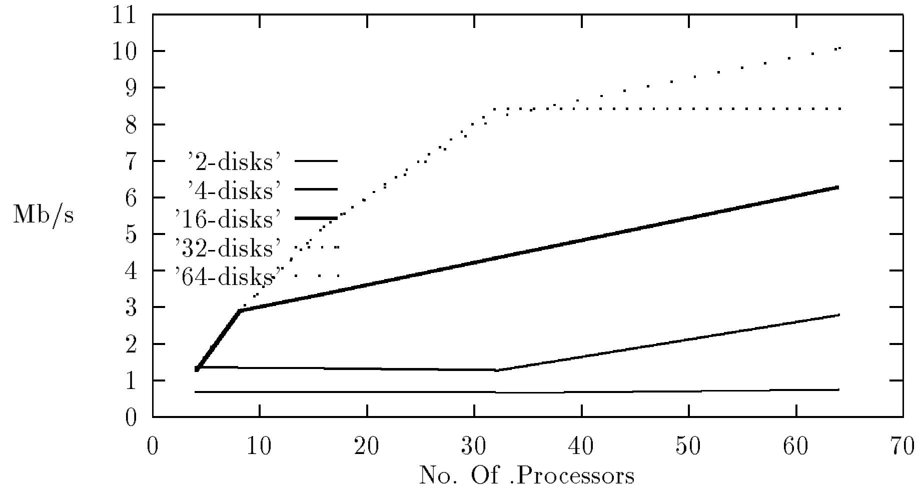


Figure 11: Multicompute Nodes File Read (Mode 1 and 16 Mbytes File)

### Modes 1, 2 and 3: Burst I/O

Tables 6 and 7 present a summary of results obtained for reads and writes using burst-mode I/O for various system configurations. The number of processors was varied form 32 (8*4 mesh) to 512 (16*32 mesh). Each processor accessed 1 Mbytes of data, and therefore, the resulting file size varied from 32 Mbytes to 512 Mbytes. Clearly, burst mode I/O is preferable for large file accesses. Furthermore, a consistent performance is observed over a wide range of processor configurations. However, beyond 256 processors (actually, between 128 and 256) the I/O system becomes a bottleneck resulting in degraded, but still a comparable performance. It should be observed that read rates were normally 2 to 3 times faster than the write rates.

Table 6: Read Throughput in Mbytes/sec (Burst Mode)

| Mesh Size | Mode 1 | Mode 2 | Mode 3 |
|---|---|---|---|
| 8*4 | 8.447 | 8.159 | 8.144 |
| 8*8 | 4.817 | 8.310 | 9.602 |
| 16*8 | 8.715 | 8.6821 | 8.891 |
| 16*16 | 6.519 | 7.18 | 7.169 |
| 16*32 | 6.21 | 6.742 | 6.944 |

Table 7: Write Throughput in Mbytes/sec (Burst Mode)

| Mesh Size | Mode 1 | Mode 2 | Mode 3 |
|---|---|---|---|
| 8*4 | 4.19 | 4.310 | 3.217 |
| 8*8 | 3.622 | 4.28 | 3.907 |
| 16*8 | 2.60 | 3.1545 | 2.862 |
| 16*16 | 2.53 | 2.4255 | 2.479 |
| 16*32 | 2.40 | 1.9845 | 2.286 |

## 3.5 Parallel File System: An Extension of Concurrent File System

Intel Paragon XP/S system uses an advanced version of the Concurrent File System, called Parallel File System or PFS. The parallel file system is built as an OSF/1 mountable file system. The parallel file system is completely compatible both with CFS and OSF/1. PFS allows a global opening call which allows all the nodes to open a same file at the same time. In addition PFS provides an extra I/O mode called mode 4, which is to be used when all the nodes access the same data. PFS also allows variation in the stripe size of the data declustering. However there is very little information regarding the experimental performance of PFS. A lot of experiments need to be carried out for studying the effects of various parameters.

## 3.6 Conclusions

To summarize the results, we conclude that:

- The "declustering" of the files improves the read and write performance of the file system for both single and multiple compute nodes.
- For single compute nodes, using the paged I/O mode, the read rate is higher than the write rate. The file access rates depend on the buffer size used in file access. For the file read, normally, as the buffer size increases, the performance improves to a certain point. The buffer size which provides a reasonably good performance for various configurations is 4 Kbytes which is same as the block size and the stripe size. Currently, user has no control over the block size and the stripe size. Further experiments are needed to study the effect of stripe

sizes.

- Using the burst-mode I/O, the file access rates improve significantly. It is observed that in general the buffer size for burst I/O access should be as large as possible for the best performance.

- For the single compute node case, the position of the node in the Touchstone Delta mesh does not affect the file access time. This shows that the "inter-node" hops between the compute nodes are very small. This shows that there is no possible overhead in using the communication network for the I/O transactions.

- For the multiple nodes case, the data accessed depends on the modes of file access. Mode 0 should be used for reading the shared data, whereas modes 1 to 3 should be used for data distributions. The access throughput depends on the number of processors and it also depends on the total number of disks on which the data is stored. The performance increases initially as the number of processors increase then it remains steady. In general, for small processor grids, the bandwidth is limited on the computational node size, but as the number of processors is increased, it shifts to the I/O system. the point at which this shift occurs depends on the number of processors as well as on the number of disks.

- For the multinode configuration, the performance can be further improved by using the burst mode of operation. Using a large buffer size (2 MBytes) for mode 0, the peak performance of 23.83 MBytes/sec is observed. For the same mode, the peak write rate is about 7 MBytes/sec. For the remaining three file access mode, burst mode gives a better performance than the paged I/O mode.

- The choice between various modes (except mode 0) depends on the type of access pattern and data organization. Mode 2 should be used only when the

size of data access can vary and cannot be determined in advance. Mode 3 should be used if access size is known in advance. Mode 1 should be the choice when order of access is not important because mode 1 does not improve global synchronization.

# Chapter 4

# Cost Analysis of Data Mappings

Chapter 3 presented experimental performance of Touchstone Delta Concurrent File System. In this chapter we analyze the I/O behavior of a similar system which is connected with a parallel file system. Section 1 presents a simple I/O model and discusses the effect of I/O on the overall program performance. Section 2 analyzes the Intel Touchstone Delta CFS, when the processors access the files according to the data distribution on the disks. In this section we calculate the costs associated with I/O and disk requests and compute the overall I/O costs for various data distribution strategies. Section 2 also presents performance results of data distribution. Finally in section 3, we propose an alternative approach for optimal data distribution.

## 4.1 An I/O Model

A lot of work has gone into formulating the computational costs of the parallel machines [Amd67, Gus88, NA91]. In this section we will propose a simple model to explain the I/O characteristics of a parallel machine which lead to the *I/O Bottleneck* [Kun86].

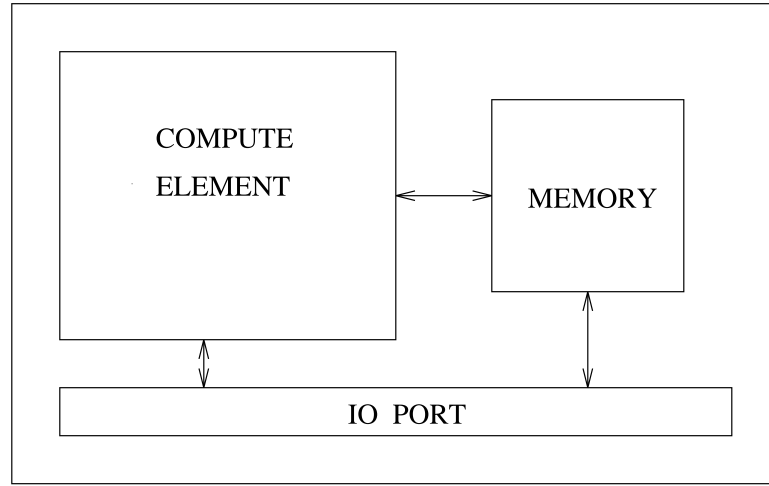


Figure 12: A Processing Element

### 4.1.1 Background

Let us consider a computing element C. C has three main components, a processing element, a local memory and an I/O port (figure 12). Let us imagine a hypothetical parallel machine composed of such computing elements. For our simple analysis, we will not consider the interconnection pattern of the parallel machine. Let $T_s$ be the time required for executing a sequential program using the best serial algorithm and $T_p$ be the time required to execute the same program in parallel on N computing elements using the best parallel algorithm. Initially assume that the program does not require any I/O. Then the speedup $S$ will be given by

$$S = \frac{T_s}{T_p} \tag{1}$$

Let $IO_s$ be the total I/O time required for the sequential program. Similarly, $IO_p$ be the I/O time required for the parallel program. Considering the IO costs, $S_o$ becomes the overall speedup of the parallel program.

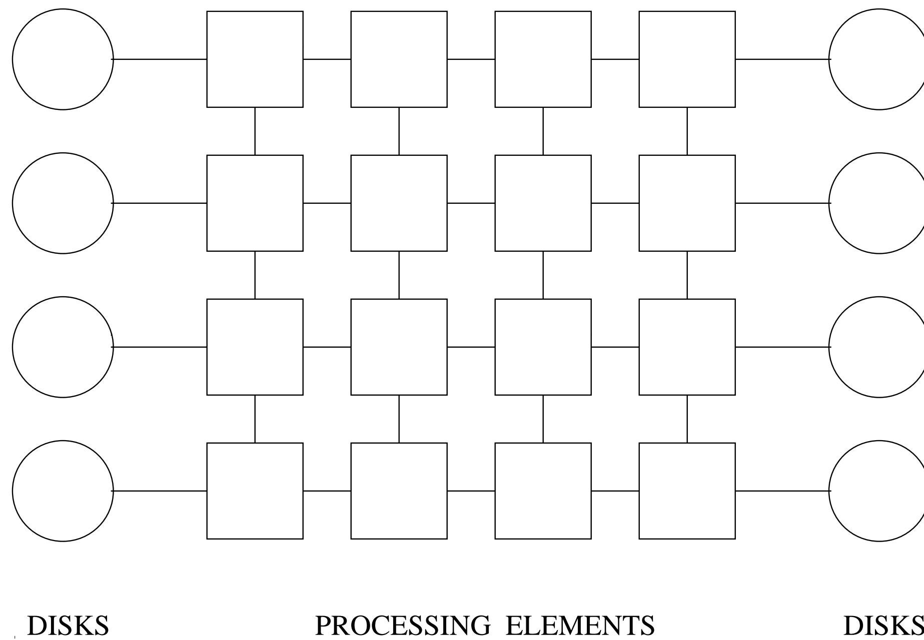


Figure 13: A Mesh Connected Multiprocessor

$$S_o = \frac{T_s + IO_s}{T_p + IO_p} \quad (2)$$

If $IO_P$ does not match the corresponding computational time $T_p$, the resultant speedup $S_o$ will be less than $S$. In other words, the I/O cost will prevent a parallel program to achieve a theoretical speedup. This is called the *I/O bottleneck* [Kun86].

### 4.1.2 An I/O Model for a Multiprocessor

In this section we will analyze I/O characteristics of multiprocessors like Intel Touchstone Delta or Paragon.

For the analysis, let us consider a mesh connected multiprocessor $M$ (figure 13). $M$ consists of $N_p$ processing elements (like C) connected to a parallel file system having $N_D$ disks. For simplicity, we will assume an ideal interconnection network (negligible

contention) between the disks and the processing elements. Each processing element can perform independent I/O with the disks. The cost of the data access depends on the number of disks and the number of processing elements. Assuming a uniform data distribution on the disks, we can define the overall speedup as a two element tuple $S < N_p, N_D >$.

$$S < N_p, N_D >= \frac{T_s + IO_s}{T_p + IO_p} \tag{3}$$

$S < N_p, N_D >$ defines the speedup of a parallel program running in parallel on $N_p$ processors and accessing the data in parallel from $N_D$ disks.

For characterizing the parallel I/O, we formally define the following parameters:

**Definition 1** *Computational Bandwidth* $B^P_{comp}$ *is the total number of computational operations executed by* $N_p$ *processors in a unit time.*

**Definition 2** *IO Bandwidth* $B^D_{io}$ *gives the total amount of data accessed in parallel from* $N_D$ *disks in a unit time.*

Consider a parallel program running on the $N_p$ nodes of the mesh. The program accesses data from the file system having $N_D$ disks. Let $T_p$ be the time required to complete the program execution. Hence $T_p$ can be written as

$$T_p = T_c + T_{io} + T_{misc} \tag{4}$$

where $T_c$ is the time required for computation on the mesh processors. $T_{misc}$ denotes the miscellaneous time that includes the network interference and communication overhead. $T_{io}$ denotes the time required for performing I/O.

Let us consider $T_{io}$. The I/O time is composed of three main factors, I/O request time, disk request time and data access time. The I/O request time depends on

the number of IO requests sent by the computational processors to the file system, whereas, the disk request time depends on the number of disks requests sent by the file system interface to the disks and on the size of each data request. The data access time is nearly constant and depends largely on the hardware specifications of the disks.

The number of IO requests depend on both the amount of distributed data and the type of data mapping. Both the factors are influenced by file access strategies. The data storage policies decide the mapping of the data on the disks, which in turn decides the disk request time.

For simplicity we neglect $T_{misc}$. We can then assume that,

$$B_{io}^{D} = \frac{1}{T_{io}} \qquad (5)$$

$$B_{comp}^{P} = \frac{1}{T_{c}} \qquad (6)$$

The computational bandwidth or $B_{comp}^{P}$ gives the *peak rate of generation of data for performing the I/O (read/write).* The I/O bandwidth $B_{io}^{D}$ is the actual rate of data transfer between the disks and the processors. Hence, even if the data is generated at a higher rate, the data is not transferred to(from) the file system at the same rate. If the program requires accessesing distinct data from disks at every step of the computation, the total execution time will obviously depend on how fast the data is transferred between the disks and the processors.

**Definition 1** *The program is said to be stable iff for all program sizes the I/O bandwidth matches the computational bandwidth.*

*Program size includes the size of the data set used in the program, number of processors and number of disks.*

$$B^P_{comp} \approx B^D_{io} \tag{7}$$

However the term *for all program sizes* brings forward another important issue, i.e., *scalability of the program with respect to the I/O system and how does the scalability affects the I/O bottleneck*. For analyzing the scalability,we define a term *IO Speedup* ($S_{io} < N_p, N_D >$).

Let $T'_{io}$ be the I/O time required to read a particular amount of data when $N_p$ is $p'$ and $N_D$ is $d'$ and $T''_{io}$ be the time required to access the same amount of data when $N_p$ is $p''$ and $N_D$ is $d''$.

**Definition 3** *IO Speedup $S_{io} < N_p, N_d >$ is the ratio of the IO costs for two distinct sets of processor-disk configurations.*

$$S_{io} = \frac{T'_{io}}{T''_{io}} \tag{8}$$

For analysis purposes, we define two conditional parameters $\alpha_p(d)$ and $\alpha_d(p)$. We use following definitions

1. $T^{p'}_{io}$: $T_{io}$ with p' processors and d disks.

2. $T^{p''}_{io}$: $T_{io}$ with p" processors and d disks ($p' < p''$).

3. $T^{d'}_{io}$: $T_{io}$ with d' disks and p processors.

4. $T^{d'}_{io}$: $T_{io}$ with d" disks and p processors ($d' < d''$).

$$\alpha_d(p) = \frac{T^{p'}_{io}}{T^{p''}_{io}} \tag{9}$$

$$\alpha_p(d) = \frac{T_{io}^{d'}}{T_{io}^{d''}} \qquad (10)$$

In other words, $\alpha_d(p)$ is the I/O speedup ($S_{io}$) for a fixed number of disks and variable number of processors p' and p". Correspondingly $\alpha_p(d)$ is the I/O speedup ($S_{io}$) for a fixed number of processors and variable number of disks d' and d".

Now we present the following propositions,

**Definition 2** *The I/O system is called processor bound iff* $\alpha_p(d) \geq 1$ *and* $0 < \alpha_d(p) < 1$.

**Definition 3** *The I/O system is called disk bound iff* $\alpha_d(p) \geq 1$ *and* $0 < \alpha_p(d) < 1$.

**Definition 4** *To maintain the stability of the program for all program sizes, the I/O system should be neither processor bound nor I/O bound.*

Hence, for a program to be truly stable *for program sizes*, the computational bandwidth should match I/O bandwidth and the I/O bandwidth should remain consistent for varying number of processors or disks.

However, the experimental results on the Intel Touchstone Delta have shown that the $B_{io}^D$ varies when either the processor grid size or the number of disks are varied and $B_{io}^D$ is always less than the $B_{comp}^P$.

$$B_{comp}^P > B_{io}^D \qquad (11)$$

The resultant bandwidth $B_p$ ($= \frac{1}{T_p}$) is always decided by the I/O system having the smaller bandwidth $B_{io}^D$. Thus, the resulting bandwidth $B_p$ decreases, thereby increasing the total program execution time $T_p$. As a result, the speedup of the parallel program $S_o$ decreases (section 4.1.1).

$$B_p = min(B^P_{comp}, B^D_{io}) \tag{12}$$

Thus, to improve the resultant bandwidth of the program, $B^D_{io}$ has to be increased. This can be done by reducing both the I/O requests and the disk requests.

## 4.2 Analysis of the I/O costs

In large-scale scientific and engineering applications, parallelism is exploited by decomposing the input domain (representing the physical domain model, normally represented by multi-dimensional arrays). However, for load-balancing, expressing locality of access, reducing communications and other optimizations, several decompositions and data alignment strategies can be used

In order to enable a user to specify the decomposition, Fortran D [FHK+90, Fox90, Tse93, Fox91], and subsequently High-Performance Fortran [For93], have been proposed. The important feature of these extensions is the set of directives that allow a user to decompose, distribute and align arrays in the most appropriate fashion for the underlying computation. The data distribution directives include BLOCK, CYCLIC and BLOCK_CYCLIC distributions (along any dimension of an array). In this section, we study the performance of the Intel Touchstone Delta CFS when the processors access files based on the data distributions. We will concentrate on accessing thet fortran arrays assuming that the arrays are stored in the column-major form on the disks. We will analyze the corresponding bandwidths for each type of data distributions and compute the number of I/O requests and the number of disks requests for each type of distribution.

Table 8: Number of I/O Requests as a Function of Data Distributions

| 1-D Distribution | | | 2-D Distribution | | |
|---|---|---|---|---|---|
| Distr. Type | No. Of Req. | Data per Req. | Distr. Type | No. Of Req. | Data per Req. |
| Column Block | P | $\frac{N^2}{P}$ | Block-Block | $N * \sqrt{P}$ | $\frac{N}{\sqrt{P}}$ |
| Column Cyclic | N | N | Block-Cyclic | $N * \sqrt{P}$ | $\frac{N}{\sqrt{P}}$ |
| Row Block | N*P | $\frac{N}{P}$ | Cyclic-Block | $N^2$ | 1 |
| Row-Cyclic | $N^2$ | 1 | Cyclic-Cyclic | $N^2$ | 1 |

### 4.2.1 Analysis of Array Decompositions

This section presents the cost analysis of data access (read/write) on Touchstone Delta in terms of I/O and disk requests.

Consider a parallel program running on a mesh having $N_p$ processors. Let a $m$ dimensional array to be distributed over the mesh. The array can be distributed in BLOCK, CYCLIC and BLOCK_CYCLIC form. This is the *program mapping* of the array. To obtain the *program mapping*, the processors send I/O requests to the disks on which the array is stored. Table 8 shows the number of I/O requests and the data per requests when an array of size N*N is distributed over P processors.

For a $m$ dimensional array A, let $S_1, S_2, ..Sm$ be the sizes in each dimension. The mapping function $\Re_1^m$ maps the $m$ dimensional array into a one dimensional array of size $\prod_{i=1}^{i=m} S_i$. This is the physical mapping of the multidimensional array on the disks. Hence any array having a certain *program mapping* $M_p$ has a corresponding *disk mapping* $M_d$ defined by the mapping function $\Re_1^m$.

Let $\Psi$ be the interface function that acts as an interface between the *program mapping* and the *disk mapping*. Depending on the *disk mapping*, the interface function translates the number of I/O requests into the corresponding number of disk requests (figure 14). On Touchstone Delta CFS, the I/O function $\Psi$ is implemented

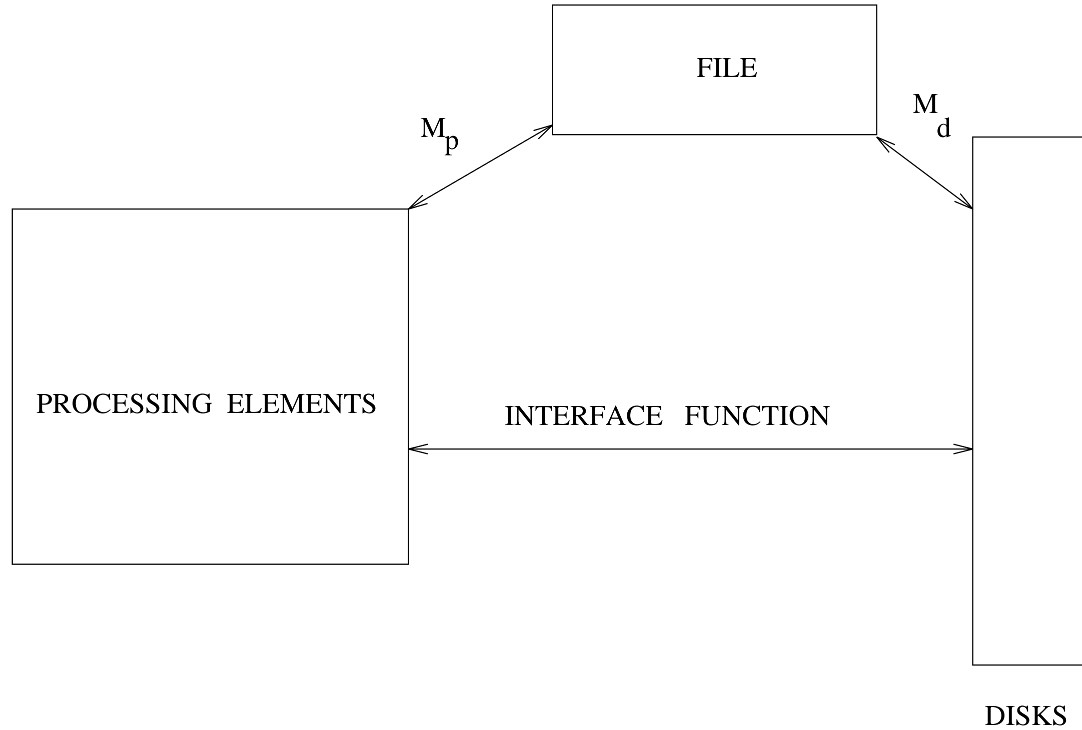


Figure 14: Data Mapping between the Processors and the Disks

as DiskProc and NameProc (section 3.2.2) on the *nameserver* node.

$$M_p \stackrel{\Psi}{\longleftrightarrow} M_d \tag{13}$$

Let us analyze the mapping from program domain to physical domain in a greater detail.

In the *program mapping* $M_p$, the array is visualized as a file distributed over the processing elements. Let $F$ be the total file size of the data to be read (written) by the processors and $F_p$ be the total amount of data to be read per processor.

$$F_p = \frac{F}{N_p} \tag{14}$$

Depending on the program, each processor might access different data. Let $F_{p'}$ be

the amount of data actually read by each processor. $F_{p'} \neq F_p$. Hence

$$F_{p'} = \gamma \times F_p \tag{15}$$

where $\gamma$ is a *file distribution constant.* $\gamma$ varies from processor to processor. $0 < \gamma \leq 1$. When $\gamma = 1$, then $F_{p'} = F_p$.

Let $b_{io}$ be the buffer size used in the program. The buffer size $b_{io}$ and $F_{p'}$ decide the total number of I/O requests $N_{io}$.

$$F_{p'} = N_{io} \times b_{io} \tag{16}$$

In physical mapping $M_d$, the file system visualizes the array as a file distributed over a number of disks. Let $N_D$ be the total number of disks on which file is distributed. Let us assume that it is possible to control the number of the disks from the program. Then let $\alpha$ be the *volume control factor*. If the disk volume is restricted then the total number of disks will be $N_d$, where

$$N_d = \alpha N_D \tag{17}$$

$$0 < \alpha \leq 1 \tag{18}$$

In case of Delta, $N_D$ is 64 and $N_d$ can vary from 1 to 64. The number of disks requests sent from the I/O nodes depends on the number of disks on which the file is stored. Let $B_D$ be the size of a data block. In Touchstone Delta CFS, $B_D$ is equal to 4KBytes. Hence for a file of size $F$,

$$F = N_D \times B_D \times N_b \tag{19}$$

$N_b$ are the total number of data blocks into which the file is declustered. Initially the nameserver finds the number of disks that contain the file data. Let the number of disks be $N_{d'}$. Thus the number of disk requests will be $N_{d'}$. Each disk request prefetches $N_c$ blocks of size $B_D$. In Touchstone Delta, file read prefetches 8 blocks. Hence the total data prefetched is

$$F_{cache} = N_{d'} \times N_c \times B_D \tag{20}$$

Hence, the total number of disk requests per processor is

$$R_{disks} = \lceil \frac{F_{p'}}{F_{cache}} \rceil \times N_{d'} \tag{21}$$

Therefore, the total time required to read the data from the disks is given by

$$T_{io} = R_{io} \times t_{io} + R_{disks} \times t_{disks} + T_{misc} \tag{22}$$

where $t_{io}$ is the time required for each I/O request, $t_{disks}$ is the time required for each disk request. $T_{misc}$ includes all other overheads such as mode synchronization, interconnection network delay and the user interference. The corresponding I/O bandwidth $B_{io}^D$ is

$$B_{io}^D = \frac{1}{T_{io}} \tag{23}$$

### 4.2.2 Analysis of Common Data Distributions

We will now analyze the I/O costs of some commonly used data distributions. Following data distributions are analyzed in this section

1. Column Block

2. Column Cyclic

3. Row Block

4. Row Cyclic

In the following analysis, we consider a two dimensional square array A(Size,Size), where Size is the size of the array per dimension. In addition we will use the following parameters

1. $t_{disk}$. The time required for an individual disk request.

2. $t_{io}$. The time required for individual I/O request.

1. Column Block: In this distribution, the second dimension of the array is distributed in the block fashion. Since the array is stored column-wise on the disks, each processor needs to read the columns allocated to that processor. This can be done easily in one I/O request (per processor) using the required buffer size. The required buffer size will be $\frac{Size \times Size}{N_p}$.
There will be one I/O request per processor. Here, each processor reads the entire data in one request. Hence, buffer size $B_{io}$ is equal to the file size $F_{p'}$. Thus the total disk requests will be

$$\lceil \frac{F_{p'}}{F_{cache}} \rceil \times N_{d'} \tag{24}$$

The number of disk accesses thus depend on the amount of cache data prefetched which in turn depends on the number of disk volumes that are used. Here $R_{io}$ will be 1 and $R_{disks}$ will depend on the number of disks. As the number of disks used is decreased, $T_{disk}$ increases. $T_{misc}$ depends on the mode used. For mode 2, $T_{misc}$ will be larger. The total time for accessing the data in column block form is

$$T_{io}^{cb} = t_{io} + R_{disks} \times t_{disks} + T_{misc} \tag{25}$$

The corresponding bandwidth is

$$B_{io}^{cb} = \frac{1}{T_{io}^{cb}} \tag{26}$$

2. Column Cyclic: In this distribution, the columns of the array are cyclically distributed. First processor getting the first column, second processor the next and so on. Hence each I/O request reads each column in one buffer. Considering a square matrix, the number of I/O requests $R_{io}$ (per processor) will be $\frac{Size}{N_P}$, where Size is the number of columns. rows. The buffer size $B_{io}$ will be equal to Size. The number of disk accesses will be variable as in the previous case. In this case, $T_{io}$ will be much larger than in the first case. In general, time required for column cyclic distribution will be greater than that of column block, resulting in a lower bandwidth $B_{io}^{cc}$.

   The total time $T_{io}^{cc}$ is

$$T_{io}^{cc} = R_{io} \times t_{io} + R_{disks} \times t_{disks} + T_{misc} \tag{27}$$

   The corresponding bandwidth is

$$B_{io}^{cc} = \frac{1}{T_{io}^{cc}} \tag{28}$$

3. Row Block: In this type of distribution, the rows are distributed as a block across the processors. Since the array is stored column-wise on the disks, row block distribution involves the distribution of each column on

the target processors. Thus the buffer size $B_{io}$ will become $\frac{Size}{N_p}$. The number of I/O requests $R_{io}$(per processor) will be equal to the number of columns (which is equal to Size). Hence $T_{io}$ will be larger in this case as compared to the column distributions. The number of disk accesses are independent of the distribution because they will depend on the number of disks used and the number of data prefetched. Since the number of I/O requests increase, the bandwidth $B_{io}^{rb}$ decreases.

$$T_{io}^{rb} = Size \times t_{io} + R_{disks} \times t_{disks} + T_{misc} \tag{29}$$

$$B_{io}^{rb} = \frac{1}{T_{io}^{rb}} \tag{30}$$

4. Row Cyclic: In the row cyclic distribution, all the rows of the array are distributed over the processors. Thus first processor gets first row, second processor second row and so on. This operation becomes very time-consuming in this case, because the arrays are physically mapped column-wise.Hence each processor reads only one array data in each I/O request. The number of I/O requests (per processor) will be much larger and they will be $\frac{Size \times Size}{N_p}$. Hence this distribution will require the largest number of I/O requests. This results in the array distribution becoming very slow (corresponding $B_{io}^{rc}$ will be the lowest).

   Among all the distributions, the column-block distribution is the fastest and the row-block distribution is the slowest.

$$B_{io}^{rc} < B_{io}^{rb} < B_{io}^{cc} < B_{io}^{cb} \tag{31}$$

Hence the program accessing the data in the row cyclic form will have the lowest resultant bandwidth ($B_p$).

### 4.2.3 Experimental Results

This section presents the experimental results of array distributions on Touchstone Delta. The experiments distribute a square array in different patterns. CFS file modes 2 and 3 are used to distribute the array concurrently over the processors.

- Column Block: The column-block distribution implies that the matrix data is distributed along its second dimension onto the processor array. This distribution also conforms with the column-major data distribution over the disk. It requires a single application level I/O request per processor and each processor node can read the entire distributed data in one I/O access. The time required to distribute the data column-wise scales with the number of processors for a portion of the configuration space.

  Table 9 contains the data for a column-block array distribution. The table shows the size of the array, the number of processors participating in the read, the transaction completion time, and the observed bandwidth. For small size arrays and the number of nodes, the bandwidth of the I/O system is underutilized. As the data size and the number of processors increase, the I/O bandwidth is more effectively utilized. However, beyond a certain point, the I/O system becomes a bottleneck due to the large number of processors performing I/O, and the need for synchronization.

  The read rate increases quickly in proportion to the processor grid size, but plateaued at about 64 processors. Degradation in the performance was observed

Table 9: Array Distribution (Column Block) Throughput in MBytes/sec ($N_D = 64$)

| Array Size | Mesh Size | Rate (Mode 2) | Rate (Mode 3) |
|---|---|---|---|
| 1K*1K | 4 | 2.141 | 2.32 |
| 4K*4K | 4 | 4.198 | 7.048 |
| 5K*5K | 16 | 5.098 | 7.44 |
| 4K*4K | 64 | 5.179 | 6.498 |
| 5K*5K | 64 | 6.476 | 7.521 |
| 10K*10K | 256 | 7.038 | 7.65 |
| 20K*20K | 256 | 5.7 | 5.861 |
| 10K*10K | 512 | 5.232 | 5.51 |
| 20K*20K | 512 | 5.517 | 5.63 |

after 256 processors due to a large synchronization overhead. Performance for the small request (1K*1K) case was poor.

- Column Cyclic: Table 10 shows the read access times for the same parameters but with a column-cyclic data distribution on processors. Even though the degree of parallelism in the data access remains the same, the number of I/O requests increases (Table 8) because each processor must make an individual request for each column. This degrades the access time and the bandwidth as illustrated in Table 10. The degradation in the performance in consistent for all configurations and it ranges between a factor of 2 to 10 as compared to that for column-block distribution

- Row Block: Table 10 shows the performance for reading the data array when distributed in a row-block fashion over the processor array. Since the one-dimensional map of the file on the CFS is in column major order, this read operation essentially requires transposing the data while it is being read from disks to nodes. As shown in Table 8, the number of logical request is N*P.

Table 10: Array Distribution (Column Cyclic) Throughput in Kbytes/sec ($N_D = 64$)

| Array Size | Mesh Size | Rate (Mode 2) | Rate (Mode 3) |
|---|---|---|---|
| 1K*1K | 4 | 160.84 | 229.72 |
| 2K*2K | 16 | 1061.85 | 1527.3 |
| 4K*4K | 16 | 1791.31 | 3057.51 |
| 5K*5K | 64 | 1962.16 | 2191.63 |
| 10K*10K | 256 | 846.92 | 856.43 |
| 20K*20K | 512 | 1522.04 | 1581.15 |

Table 11: Array Distribution (Row Block) For Modes 2 and 3 ($N_D = 64$)

| Array Size | Mesh Size | Mode | Rates Kbytes/sec |
|---|---|---|---|
| 1K*1K | 4 | 2 | 56.23 |
| 2K*2K | 4 | 2 | 123.84 |
| 4K*4K | 16 | 2 | 114.09 |
| 5K*5K | 16 | 2 | 142.34 |
| 1K*1K | 4 | 3 | 58.64 |
| 2K*2K | 4 | 3 | 154.04 |
| 4K*4K | 16 | 3 | 224.70 |
| 5K*5K | 16 | 3 | 273.11 |

Hence, as observed from Table 10, the performance degradation due to this distribution is almost two orders of magnitude when compared to the performance of the column-block distribution. We do not present performance figures for larger configurations (i.e. large array and system sizes) since the time it took to complete these experiments exceeded practical limits. Thus, we merely conclude that performance for this distribution was at least more than two orders of magnitude worse than the first two configurations. The peak bandwidth obtained was 0.69 Mbytes/sec. This is only 30% of the slowest case (the 1*1 Kbytes case) for the column-block decomposition of Table 9 above. Further, the 1K*1K case for this distribution is 39 times slower than for the equivalent column-block case.

- Row Cyclic: The row-cyclic distribution involved the largest number of I/O requests. Also the request size was the smallest. It took approximately 15 minutes to distribute 1K*1K character array in row-cyclic order versus the 467 msec it would require in the column-block form. This shows that the direct row distribution of an array is very slow, hence, not possible in practice.

## 4.3 Two Stage Data Mapping

Section 4.2.1 analysed the costs connected with accessing the array data directly from the disks. Though this *direct access* mode is easy to use it has two serious disadvantages.

1. Cost of Data Distribution

   The direct access mapping strongly couples the user mapping of the arrays with the disk mapping on the parallel file system. As a result, the data

distribution results show a large variation in performance. The cost of obtaining a user mapping (say column-cyclic) of an array depends on how it conforms with the disk mapping of the same array on the parallel file system. For example, in Touchstone Delta, distributing a Fortran array in column-block form is the fastest, whereas distributing the same array in the row-cyclic form is extremely costly.

2. Inability to Obtain Complex Two-dimensional Data Distributions.

   Consider a program that needs to access a two dimensional array in Block-Block decomposition. Suppose the program is running on Touchstone Delta in which the data is stored on the disks in the column major form (figure 15). The current version of the Touchstone Delta CFS will not allow this distribution since it does not allow any processor to read the distinct data while the others are sitting idle. The I/O mode 0 permits independent file pointers (section 3.2.3), however each processor needs to read the entire data. Hence it is not possible to obtain complex 2-dimensional array distributions (Block-Block, Block-Cyclic).

Both these problems can be solved using a two-phase data distribution strategy. It involves two phases: (1) Optimal I/O access (2) Redistribution of the data.

1. Optimal I/O access

   Optimal I/O access involves accessing the data from the disks in the *most optimal way*. The optimal form of the data access conforms with data distribution of the disks. Hence the optimal access pattern varies depending on the data mapping on the disks. In Touchstone Delta, where the arrays are assumed to be stored in the column-major form, the optimal data access strategy is the column-block access.

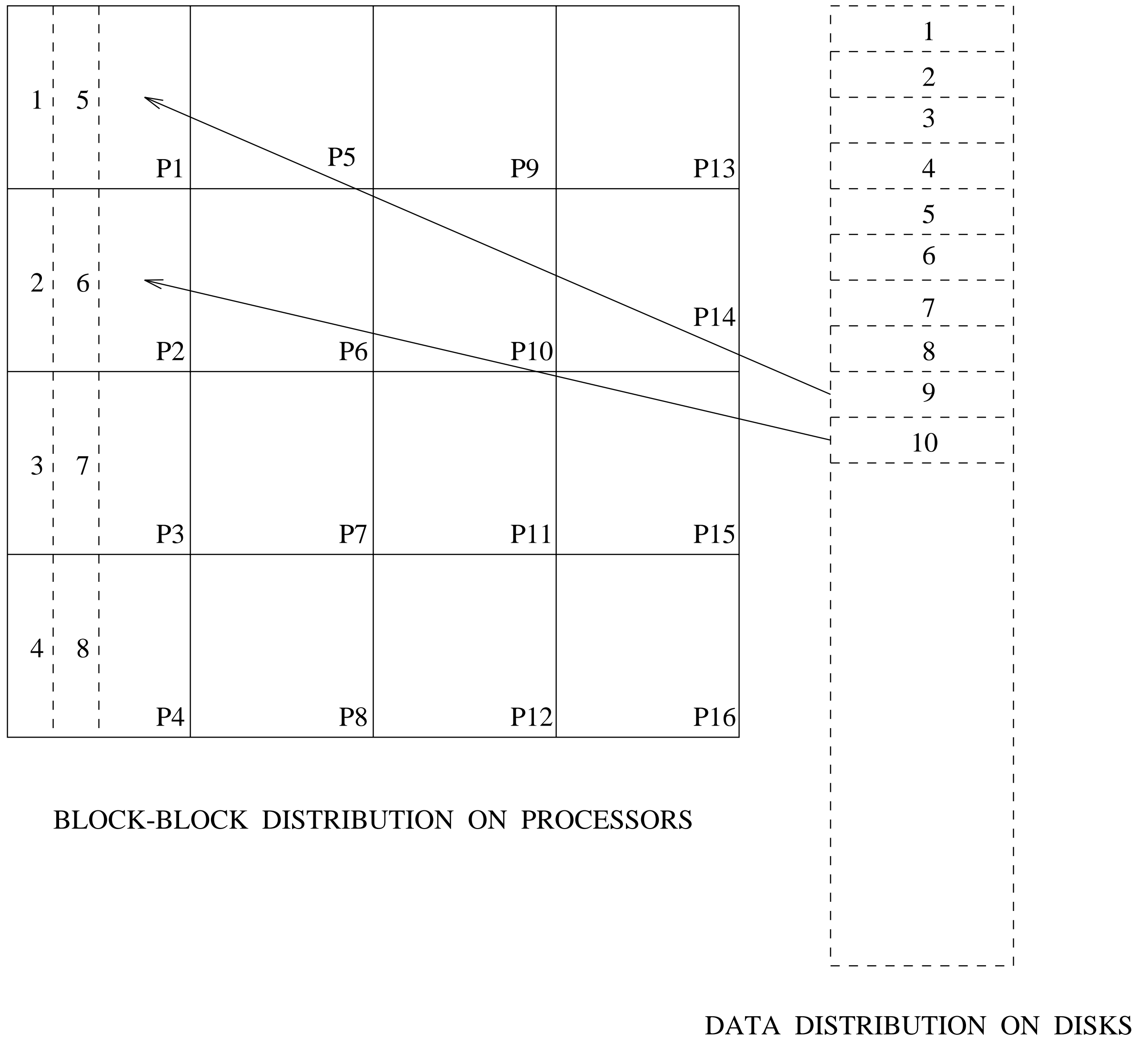


Figure 15: Block-Block Decomposition over 16 Processors

2. Redistribution of the Data

   The second stage of the two-phase data access involves the redistribution of the accessed data (accessed from the disks) into the target distribution. The redistribution cost is negligible as compared to the I/O cost. The generalized algorithm for redistribution is given below

   – Redistribution Algorithm

     (a) Acess the data from the disks in the most optimal form.

     (b) Get information about the target distribution.

     (c) For each processor calculate the data that needs to be communicated.

     (d) Calculate the corresponding source and destination.

     (e) Communicate the data over the required number of processors.

   The redistribution algorithm uses the Pairwise Exchange or (PEX) algorithm [Rav92b, Rav92a], [Zek92a] for data communication. The PEX algorithm requires N-1 steps for N processor system. At each step i, $1 \leq i \leq N-1$, each processor exchanges a message with another processor determined by taking the exclusive-or of the processor number with i. Hence the entire communication schedule is decomposed into a sequence of pair-wise exchanges. The Pairwise Exchange algorithm is given in figure 16.

By using an optimal access strategy for I/O and then redistributing the data according to the target data mapping, the two phase access strategy effectively decouples the user mapping from the disk mapping. Note that if the target data distribution is same as the conforming data distribution (for example, for Touchstone Delta column-block), redistribution is not required.

```
do j =1, nprocs-1
    node = xor(mynumber,j)
        if (mynumber < node)
            receive(node)
            send(node)
        else
            send(node)
            receive(node)
        end if
enddo
```

Figure 16: Pairwise Exchange Algorithm

# Chapter 5

# Runtime I/O Support For Parallel Languages

In recent years, parallel computers have become very popular in the scientific community. One of the main reasons for their success, is the availability of efficient and simple parallel languages.

MIMD languages like Fortran D [Fox90, Fox91, FHK+90, Tse93], Vienna Fortran [CMZ92, BGMZ92a, ZBC+92, BCZ92, BGMZ92b] and Fortran90D/HPF [Zek92c, Zek92b, For93] which use data decompositions to achieve concurrency. Data decomposition often requires the data to be accessed (read/written) from the disks in a distributed manner. In recent past, various attempts were made to develop languages that would support the data distributions from the I/O perspective. We present some prominent MIMD languages which have such capabilities in section 1.

Data decomposition over the processors is often complex and I/O intensive [Jua92c, Raj93], (section 4.2.3). In section 2, we propose an alternative approach of performing parallel I/O from the language platform. We describe the design and implementation of runtime I/O primitives employing two-phase strategy. In the same section,

we present performance results of distributioning data using the runtime primitives. Finally we summarize in section 3.

## 5.1 Languages Supporting Data Distribution

We discuss parallel programs using the Single Program Multiple Data (SPMD) programming paradigm for MIMD machines. This is the most widely used model for large-scale scientific and engineering applications. In such applications, parallelism is exploited by a decomposition of the data domain. To achieve load-balance, express locality of access, reduce communication, and other optimizations, several decompositions and data alignment strategies are often used (e.g., block, cyclic, along rows, columns, etc.). To enable such decompositions to be expressed in a parallel program, several parallel programming languages or language extensions have emerged. These languages provide intrinsics that permit the expression of mappings from the problem domain to the processing domain. These directives are important because they allow a user to decompose, distribute and align arrays in the most appropriate fashion for the underlying computation. An example of parallel languages which support data distribution includes Vienna Fortran [BCZ92], Fortran D [FHK+90] and High Performance Fortran or (HPF) [For93].

In order to address the I/O bottleneck problem, these languages propose to provide some support for parallel I/O operations. We will review the Vienna Fortran and High Performance Fortran proposals for concurrent I/O operations by presenting some simple examples.

```
PROCESSORS P(2,2)
integer A(64,64) DIST(BLOCK,BLOCK) TO P(2,2)
```

Figure 17: Array Mapping Using Vienna Fortran

### 5.1.1 Vienna Fortran

Vienna Fortran (VF) is a data-parallel language which supporting the SPMD model of computation. Vienna Fortran provides explicit expressions for data mapping. Using these expressions, VF allows the user to configure the processor map and the corresponding array distributions. Figure 17 shows an array A(64,64) distributed in Block-Block form over 4 processors arranged in 2*2 mesh.

A detailed explanation of the Vienna Fortran can be found in [BCZ92, CMZ92]. For concurrent file I/O, the language distinguishes between two types of files: standard Fortran files, and array files for which access is only available via concurrent I/O operations. For example, the array, A, above could be written into a file, *f*, by using the following statement

CWRITE(*f*, PROCESSORS='P(2,2)', DIST='(BLOCK,BLOCK) TO P') A

CREAD is used in the similar fashion. Other parallel I/O statements include COPEN, CCLOSE, CSKIP etc.For further detail, the reader is referred to [BGMZ92b].

### 5.1.2 High Performance Fortran

High Performance Fortran or HPF was designed to provide language support for a variety of machines including SIMD, MIMD, and vector machines. The HPF is syntactically similar to Fortran D [FHK+90, Fox90, Fox91], however, HPF incorporates many more features including new run-time library functions. HPF provides an excellent toolset for data mapping. The mapping toolset includes compiler directives such

Figure 18: Data Distributions in Fortran 90D/ HPF

as ALIGN, REALIGN, DISTRIBUTE, REDISTRIBUTE, PROCESSOR etc. Figure 18 provides a simple illustration of some data distributions (for four processors) that can be easily specified in Fortran D, HPF or VF.

HPF has also proposed (but does not currently support) parallel I/O statements. These statements include array mapping and file pointer manipulation statements. Unlike the Vienna Fortran proposal, HPF proposals include common support parallel and sequential files

In the HPF proposal, a FILMAP directive is used to specify a mapping for files. Then, ALIGN and DISTRIBUTE statements are used to map FILEMAPs onto nodes. For example,

```
!HPF$ FILEMAP :: F1(2, 4, *)
!HPF$ DISTRIBUTE F1(*, BLOCK, CYCLIC(2)) ONTO D1
```

READs and WRITEs, or PREADs and PWRITEs are then used to access distributed files. For further information, the reader is referred to [For93, Sni92].

## 5.2 Runtime Primitives for Parallel I/O

A number of high level programming languages have recently introduced intrinsics that support parallel I/O through a runtime library. By using these primitives, I/O operation instructions within applications become portable across various parallel file systems. Further, the primitives are convenient to use; the instructions for carrying outfigure parallel I/O operations don't involve much more than a declaration of the data decomposition mapping and the use of open, close, read, and write routines

Yet, these language supported I/O primitives suffer from a serious drawback. Because they use a direct access mechanism to perform the I/O, the user data distribution mapping remains tightly linked to the file mapping to disks. Thus, they are susceptible to the same performance fluctuations and limitations (e.g., unsupported data distributions) that are observed of the parallel file systems. Also since the design of the parallel file system varies from vendor to vendor, these languages can not effectively exploit various facilities provided in the underlying system.

Motivated by these facts we have implemented a runtime system for parallel I/O. This system will provide the portability and convenience of language supported I/O primitives. In addition, because it makes use of the two-phase access strategy (discussed in section 4.3) to carry out I/O, it effectively decouples user mappings from the file mappings of the parallel file system, and provides consistently high performance independent of the data decompositions used. Further, since these primitives are linked at the compile-time as a runtime library, can be used with any MIMD (node+message passing) program, or from a data parallel program such as one written in HPF.

**Advantages of Runtime I/O Primitives:**

1. The runtime system can be easily ported on various machines which provide parallel file systems. This makes the runtime primitives highly portable and easy to use.

2. By using these primitives, the more complex data distributions (Block-Block or Block-Cyclic) are made available to the user. The only additional information required are the global, local array information and the processor grid information.

3. Primitives allow the user to control the data mapping over the disks. This is a significant advantage since the user can vary the number of disks to optimize the data access time.

4. Under certain conditions, the primitives allow the programmer can change the data distribution on the processors dynamically.

5. The data access time is significantly improved and is made more consistent since the primitives use two-phase access strategy.

### 5.2.1 Approach

Our I/O strategy involves a division of the parallel I/O task into two separate phases according to the two-phase strategy (section 4.3). In the first phase, we perform the parallel data access using a data distribution, stripe size, and set of reading nodes (possibly a subset of the computational array) which conforms with the distribution of data over the disks On Touchstone Delta, the column-block distribution is the conformal mapping. Hence we access (read/write) the data from the disks using this mapping. Subsequently, in phase two, we redistribute the data at run-time to match the application's desired data distribution By employing the two-phase redistribution strategy, the costs inherent in many of the I/O configurations are avoided. Selecting

a single, "good" configuration effectively reduces the bottleneck activity - I/O to the parallel device. Further, the redistribution phase improves performance because it can exploit the higher bandwidths made available by the higher degree of connectivity present within the interconnection network of the computational array.

In the subsections that follows, we discuss the runtime I/O primitives. A brief description of the purpose of each primitive is given followed by a discussion of its functional flow. We provide a detailed description of the syntax of these primitives. We explain the implementation of these primitives on Touchstone Delta by using a simple program. We then present some performance results on the use of these primitives with various data distributions.

### 5.2.2 General Description

The runtime primitives library provides a set of simple I/O routines. These include **popen, pclose, array_map, proc_map, pread** and **pwrite**. Though the exact syntax of these routines varies from C to Fortran, the basic data structures remain the same. This section presents a brief overview of each primitives.

#### popen

The **popen** primitive concurrently opens a file using a specified number of processors $P'$ ($P' \in P$, where P is the number of processors on which the program is executing). The choice of $P'$ is important in the systems like Intel Paragon [Int92] which provide I/O dedicated compute nodes.That is, often the number of processors involved in generating I/O requests must be smaller than the number of processors requiring the data to achieve better performance [Raj93].

The user passes file information to the **popen** primitive which is then stored in a

Table 12: The File Descriptor Array (FDA): Fortran Version

| Unit | Access | Form | Status | No_of_disks | No_of_processors |
|---|---|---|---|---|---|
| 3 | 0 | 0 | 1 | 64 | 4 |
| | | | | | |
| | | | | | |

two dimensional array called File Descriptor Array(FDA) using the file unit number as a key. The file information includes file name, file status, file form, access pattern and the number of disks on which the file will be distributed. For a statement

call **popen**(3,'TEST','SEQUENTIAL,'UNFORMATTED','NEW',-1,-1)

number_of_processors is -1, the file will be opened by default number of processors (P).

### pclose

The **pclose** primitive performs concurrent closing of the parallel files. The **pclose** primitive gets the unit number of the file as an input. Using this as a key, the primitive obtains the number of processors ($P'$). Using the file unit number, these processors close the file.

### array_map

This primitive is semantically similar to the compiler directives in HPF or VF (Figure 17). Only difference is that the directives provide global array information to the compiler, whereas the using the **proc_map** primitive, the user provides the same information to the runtime library. A Fortran D or HPF compiler would directly extract this information from the distribution directives.

Table 13: The Array Description Table (ADT)

| Info | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Global Size | 64 | 64 | -1 | -1 | -1 | -1 | -1 |
| Distribution Code | 1 | 1 | -1 | -1 | -1 | -1 | -1 |
| Block Size | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| nprocs | 2 | 2 | -1 | -1 | -1 | -1 | -1 |

The **array_map** primitive returns an integer called **array descriptor** which will be used by **pread** and **pwrite** routines for acquiring the necessary array information. A table called the Array Description Table or (ADT) is used to store the array information. The user provides the global size of the array, the distribution type, the processor distribution along each dimension and the block size (for CYCLIC distributions).

For example, consider array A(64,64) distributed in BLOCK-BLOCK form over 4 processor arranged in 2*2 mesh. The corresponding Array Description Table is shown in table 13. The value -1 is used to denote don't-care entries.

**proc_map**

The **proc_map** primitive is used for mapping the processors from the physical to the logical domain. The proc_map initializes the logical processor grid according to the user specifications. The dimension of the logical processor grid can vary from 1 to 7. The *proc_map* routine allows two kinds of mappings, one is the system-defined mapping and the second is the user-defined mapping. The user has to pass the number of processors in each dimension, the mapping mode and (or) processor mapping information. **proc_map** initializes a global data structure called P_INFO array, which is used by **pread** and **pwrite** routines. P_INFO array is a two-dimensional array

Table 14: The P_INFO Array for a Two-Dimensional Logical Grid

| x-coord. | 0 | 0 | 0 | 0 |
|---|---|---|---|---|
| y-coord. | 0 | 1 | 2 | 3 |
| Proc.Number | 0 | 1 | 2 | 3 |

of size N+1*P, where N is the dimension of the logical grid and P is the number of processors. For P processors, first N rows store the indices in the corresponding dimensions in the logical grid whereas the (N+1)th row provides the corresponding physical number. For example, if 4 processors are arranged in column-major order in a two-dimensional grid, the corresponding P_INFO array structure is shown in table 14. Using the **proc_map** primitive the programmer can change the logical processor configuration during the execution of the program.

**pread**

The **pread** primitive reads a distributed array from the corresponding file. The **pread** primitive reads the data from the file using $P'$ processors and distributes the data over P processors ($P' \in P$). The **pread** primitive uses the unit number as a key to access the file information from the FDA. The global array information is obtained using the array_descriptor. In general, the runtime system would use a distribution for intermediate access which performs the best, given a specific file distribution. Based on our analysis, such a distribution is the one which requires the minimum number of I/O transactions. However, more work needs to be done to determine a more accurate model which can provide the best distribution to use for I/O access. For our experiments, we use column-block distribution for I/O access because we assume that the files are stored in the column-major fashion on the disk arrays. The two-phase access is used by **pread** to read the data from the file using $P'$ processors. Then

1. Read the input parameters.
2. Get the global array information using ADT.
3. Obtain the logical mapping of the processors
   ($P$) participating in array distribution from proc_map.
4. Use the unit number to acquire information
   on the file such as the number of disks, number of processors ($P'$).
5. If the target data distribution is same as the conformal access distribution
   also $P' = P$ then read the data using the conforming distribution
   and go to 10
6. If the two-phase data access is used, then read the data using
   the conforming distribution. The reading is performed by $P'$ processors
7. From the global array distribution, calculate the
   data that needs to be communicated.
8. Compute the communication schedule for data redistribution.
9. Distribute the data over $P$ processors to obtain the
   target data distribution.
10. Stop

Figure 19: **pread** Algorithm

the data is redistributed over the $P$ processors to obtain the target data distribution. Figure 19 shows the **pread** algorithm.

### pwrite

The **pwrite** primitive is used to write a distributed array using $P'$ processors to the file that was opened (created) by popen. The pwrite uses the array_descriptor to get the array information, the unit number to get the file information and proc_map to obtain the logical grid information. The runtime primitive will choose a distribution for intermediate access which performs the best for a specific file distribution. The chosen distribution will require minimum number of I/O transactions. Since we assume that the array is stored in the column-major form, for Touchstone Delta, the column-block distribution is the conformal distribution. If the processor distribution

1. Read the input parameters.
2. Get the global array information using ADT.
3. Obtain proc_map to obtain the logical mapping of the processors ($P$) participating in array distribution.
4. Use the unit number to acquire information on the file such as number of disks, number of processors ($P'$).
5. If the target data distribution is same as the conformal access distribution and $P' = P$ then write the data using the same access distribution and go to 11.
6. If the two-phase data access is used, then redistribute the data over $P'$ processors using 7,8,9.
7. From the global array information, calculate the data that needs to be communicated.
8. Compute the communication schedule for data distribution.
9. Distribute the data over $P'$ processors in conforming access fashion.
10. Write the data on the disks using the conforming access distribution.
11. Stop.

Figure 20: **pwrite** Algorithm

and the conformal distribution don't match, data is first distributed from $P$ to $P'$ processors. After the distribution, data is written by $P'$ processors using the conforming distribution. Figure 5.2.2 shows the **pwrite** algorithm.

### 5.2.3 Syntax of the Runtime Primitives

This section presents detailed description of the syntax of the runtime primitives designed Fortran MIMD or data-parallel dialects. These primitives will be used as subroutine calls in the source program.

#### popen

The prototype for the **popen** primitive is shown below.

call popen(*unit, file, access, form, status, no_of_disks, no_of_processors*)

where the arguments are defined as follows:

- *unit* : The unit number associated with the file.
- *file* : The path name of the file to be opened.
- *access* : The type of the file access desired, either DIRECT or SEQUENTIAL.
- *form* : The *popen* primitive currently only supports only UNFORMATTED form of data.
- *status* : The creation status of the file; this must be either NEW or OLD.
- *no_of_disks* : The number of disks on which the file is (or is to be) stored. If the file is NEW, the file will be written onto the specified number of disks. If the file is OLD, the *no_of_disks* parameter will take as its value the number of disks on which the OLD file is stored. If the programmer provides a *no_of_disks* value that is greater than the number of disks available in the system, the parameter will take the total number of disks as its value. The programmer can also select the default number of disks by specifying a -1 for the *no_of_disks*.
- *no_of_processors* : The number of processors actually reading or writing the data to the file. These processors represent a subset of the total number of processors on which the program is being executed. The default value for this parameter is just the total number of processors on which the program is running. Selection of the default value is again signified by specifying a -1 for the *no_of_processors*. The no_of_processors primitive allows the user to allocate a certain number of processors as dedicated I/O processors. This is useful in the machines like Intel Paragon XP/S which has dedicated processors for I/O [Int92].

Parameter selections are specified with the following assignment codes :

*access* = 0 for SEQUENTIAL, *access* = 1 for DIRECT.

*form* = 0 for UNFORMATTED, *form* = 1 for FORMATTED.

*status* = 1 for NEW, *status* = 0 for OLD.

*no_of_disks* = -1 for default, *no_of_disks* = number of disks.

*no_of_processors* = -1 for default, *no_of_processors* = number of processors.

For example, the statement

call popen(3, 'TEST', 'SEQUENTIAL', 'UNFORMATTED', 'NEW', -1, -1)

causes the primitive to open a new unformatted parallel file called TEST which will be accessed with a sequentially pattern. The file will be opened by all the processors which are running this program, and it will be stored over all the disks in the file system. This file will be assigned the unit 3. Assuming the source program is executing on 4 processors, the corresponding FDA is given in 12.

### array_map

The prototype of the **array_map** primitive is shown below.

ad = array_map(*array_name, size_info, distr_info, block_size, proc_info*)

The primitive returns an integer called array descriptor.

The parameters are defined as follows:

- *array_name* : The *array_name* is the name of the array which is to be distributed over the processors.
- *size_info* : *size_info* is a one-dimensional array which provides the global size of the array in each dimension.
- *distr_info* : A one-dimensional array giving information about the type of distribution to be used for each dimension. Hence the *distr_info* array will be an

array of size 7.

- *proc_info* : An array which specifies the number of processors in each dimension of the distributed array. The size of the *proc_info* array will be 7, one entry specifying the number of processors in that dimension. The array provides only the number of processors per dimension but does not provide any logical map of the processors. The physical to logical mapping information is provided in the proc_map primitive.
- *block_size* : This is only required for the BLOCK-CYCLIC distribution. It specifies the size of each block.

Consider the same array A of size (64,64) which is distributed in a BLOCK-BLOCK fashion over 4 processors. Hence the array_map statement would have the entries:

*size_info* = [64,64,-1,-1,-1,-1,-1]

*distr_info* = [1,1,-1,-1,-1,-1,-1]

*block_size* = [-1,-1,-1,-1,-1,-1,-1]

*proc_info* = [2,2,-1,-1,-1,-1,-1]

**proc_map**

*proc_map* has the following syntax

call proc_map(*proc_info,map_mode,user_map*)

where the arguments are

- *proc_info*: This is the same array that is passed in the array_map primitive. The proc_info supplies the actual number of processors per the physical grid dimension, which is same as the number of processors on which the program is executing. This is the same array that is passed as a parameter to *array_map*.

- *map_mode* The map mode provides the actual mapping mode. The user can choose the system defined mapping or user defined mapping. If map_mode is -1, then user decides the processor mapping. If the mam_mode is 0,1 or 2 the system mapping will be selected. The system-defined mappings will vary from machine to machine.
- *user_map* If the user chooses to provide the mapping information by himself ( map_mode = -1), the user provides the user_map array. Using the user_map array, corresponding entries in the P_INFO array are filled.

**pread**

*pread* has the following syntax

call pread (*array_name, size1, size2, array_descriptor, unit, io_buffer, io_size*)
where the arguments are

- *array_name* : Name of the array into which the read data is to be placed.
- *array_descriptor* : Pointer into the ADT which is returned by the *array_map* call, and which corresponds to the distribution that the user wishes to have the runtime system apply during the read.
- *size1,size2* : This parameters provide the size of the array which is passed to the pread and pwrite primitive.
- *unit* : The unit number associated with the file to be read. Must have been returned by a *popen* call.
- *io_buffer* : This parameter specifies the local buffer used for storing the data.

- *io_size* : Specifies the size of the local io_buffer that is to be allocated for the I/O transaction.

For example, the following statement reads the data from the file associated with unit 3, into the array A having the descriptor ad. The size of the io_buffer TEMP is given by io_size call.

call pread(A,64,16,ad,3,TEMP,1024)

**pwrite**

The pwrite primitive is similar to that of pread. It has the following syntax

call pwrite (*array_name, size1, size2, array_descriptor, unit, io_buffer, io_size*)

### 5.2.4 A Sample Program

This section provides a sample Touchstone Delta Fortran program using the I/O primitives (Figure 21). The programmer wants to read and write an array in the column-cyclic fashion. The two dimensional array A(64,64) is distributed over 4 processors. Thus the size of the local array is A(64,16). The ADT and the FDA are initiated as the arrays A_INFO and F_INFO respectively. The file TEST is opened by 4 processors using the **popen** primitive. The file TEST will be distributed over the default number of disks (number of disks = -1) The programmer then initializes the processor grid using the **proc_map** primitive. The user passes 0 as the map-mode, thus initiating the system mapping. In this case, the user supplied map (using the mymap array) will be ignored. The **array_map** primitive will be used to obtain the global array information. The array_map returns the array-descriptor "ad" which is used in the **pread** and **pwrite** primitives. The **pread** primitive will read the array A from the file associated with the unit 3 using the conformal access distribution. (e.g.

for Touchstone Delta column-block distribution). Since the resultant distribution is column-cyclic, the data will be redistributed over 4 processors to obtain the resultant column-cyclic distribution. Note the convenience offered to the programmer by the primitive because the user no longer needs to worry about pointer manipulations, file distribution, buffering etc. After computation, the array A will be written into the file associated with the unit 3 using **pwrite**. **pread** and **pwrite** primitives will use the TEMP buffer for performing the burst-mode I/O. Also the I/O transactions will be carried out using the file access mode 3. Since the processor distribution is not same as the conformal distribution, **pwrite** will redistribute the data from column-cyclic to corresponding conformal distribution (column-block) and then write the array to the file using the column-block distribution (conformal distribution for Touchstone Delta).

### 5.2.5 Experimental Results

In this section we present performance results for the runtime primitives when used in conjunction with a variety of data distributions. The tables below contain optimal access, Redistribute, Total access, and Direct Access times for the four 1-dimensional distributions considered in this paper. The **pread** primitive was used to read a 5K*5K and 10K*10K square array.

For a given array size, the optimal access time represents the minimum of the read times of the four distributions; the optimal access time is derived from the distribution that most closely conforms to the disk storage distribution for the given file. The Redistribution time is the time it takes to redistribute data from the conforming distribution to the one desired by the application. The total access time is the sum of the optimal access and Redistribution times; it denotes the time it takes for the data to be read using the optimal Read access and then be redistributed (two-phase

```
            PROGRAM EXAMPLE
            integer size_info(7),distr_info(7),block_size(7),proc_info(7)
            integer A(64,16),ad,array_map,mybuffer,TEMP(1024),mymap(1536)
            COMMON /INFO/ F_INFO,P_INFO,A_INFO
                    size_info(1)=64 ! Global Size in first dimension
                    size_info(2)=64 ! Global Size in second dimension
                    distr_info(1)=0
                    distr_info(2)=2 ! Column Cyclic Distribution
                    block_size(1)=-1 ! Block-size not required
                    block_size(2)=-1
                    proc_info(1)=1
                    proc_info(2)=4 !Four processors along the column
                call proc_map(proc_info,0,mymap)
                call popen(3,'TEST',0,0,0,-1,-1) !Old File
                ad = array_map('A',size_info,distr_info,block_size,proc_info)
                    iosize=1024
                call pread(A,64,16,ad,3,TEMP,iosize)
C Use a temporory buffer called TEMP of size iosize.
                Computation Starts here
            ..................
            ..................

                call pwrite(A,64,16,ad,3,TEMP,iobuffer)
                call pclose(3)
            STOP
            END
```

Figure 21: A Sample Program For Performing Parallel I/O

Table 15: Performance of **pread** for 16 Processors (5K*5K Array)

| Distr. Mode | Optimal Access Time | Redistr. Time | Total Access Time | Direct Access Time | Speedup |
|---|---|---|---|---|---|
| | (a) | (b) | (c)=(a)+(b) | (d) | (d)/(c) |
| Column Block | 3357 | - | **3357** | **3357** | **1** |
| Column Cyclic | 3357 | 1805 | **5162** | **9890** | **1.92** |
| Row Block | 3357 | 673 | **4030** | **69939** | **17.36** |
| Row Cyclic | 3357 | 2603 | **5960** | * | > 604.03 |

Table 16: Performance of **pread** for 16 Processors (10K*10K Array)

| Distr. Mode | Optimal Access Time | Redistr. Time | Total access Time | Direct Access Time | Speedup |
|---|---|---|---|---|---|
| | (a) | (b) | (c)=(a)+(b) | (d) | (d)/(c) |
| Column Block | 10376 | - | **10376** | **10376** | **1** |
| Column Cyclic | 10376 | 7105 | **17481** | **19271** | **1.10** |
| Row Block | 10376 | 2772 | **13148** | **84683** | **6.44** |
| Row Cyclic | 10376 | 10320 | **20696** | * | > 173.95 |

access). The Direct access time is the time it takes to read the data with the selected distribution using direct access. The last row of each table shows the speedup obtained from using the two-phase access strategy over the direct access strategy. Note that the Block-Block distribution is not supported by CFS, hence tables 19 and 20 do not present any performance numbers for direct access.

Tables 15 and 16 show access times for 5Kx5K and 10Kx10K arrays, read and distributed over 16 processors respectively. The Optimal access time occurs for the Column-Block distribution. For all cases below, the '*' symbol denotes an access (read) time on the order of hours. The following observations are made by comparing the direct access times with run-time data redistributions. For all cases, the performance improvement range from a factor of 2 up to several orders of magnitude. For

Table 17: Performance of **pread** for 64 Processors (5K*5K Array)

| Distr. Mode | Optimal Access Time | Redistr. Time | Total Access Time | Direct Access Time | Speedup |
|---|---|---|---|---|---|
| | 5K | 5K | 5K | 5K | 5K |
| | (a) | (b) | (c)=(a)+(b) | (d) | (d)/(c) |
| Column Block | 3324 | - | **3324** | **3357** | **1** |
| Column Cyclic | 3324 | 703 | **4027** | **11407** | **2.83** |
| Row Block | 3324 | 246 | **3570** | **38018** | **10.65** |
| Row Cyclic | 3324 | 768 | **4092** | * | > 879.77 |

Table 18: Performance of **pread** for 64 Processors (10K*10K Array)

| Distr. Mode | Optimal Access Time | Redistr. Time | Total Access Time | Direct Access Time | Speedup |
|---|---|---|---|---|---|
| | (a) | (b) | (c)=(a)+(b) | (d) | (d)/(c) |
| Column Block | 11395 | - | **11395** | **11395** | **1** |
| Column Cyclic | 11395 | 2478 | **13873** | **63400** | **4.57** |
| Row Block | 11395 | 1028 | **11623** | **78767** | **6.78** |
| Row Cyclic | 11395 | 3092 | **14487** | * | > 248.50 |

Table 19: Block-Block Distribution over 16 Processors using **pread** (time in msec)

| Size | Optimal Access Time | Redistr. Time | Total Access Time |
|---|---|---|---|
| 1K*1K | 467 | 112 | 579 |
| 2K*2K | 717 | 416 | 1133 |
| 4K*4K | 2328 | 1253 | 3181 |

example, in table 15 the amount of overhead avoided by using the redistribution strategy (i.e., the difference between the Total Access Time and Direct Access Time) ranges from 1.7 secs, to well over 60 minutes for the 5K Row-Cyclic case. More importantly, the deviation in Total Access time is at most a factor of 1.9 as opposed to the widely varying results produced by the direct access approach.

Tables 17 and 18 shows access times for 5Kx5K and 10Kx10K arrays, read and distributed over 64 processors. The reduction in cost ranged from 7.4 secs, to over 60 minutes for the 5Kx5K Row-Cyclic case. Note that the variation in Total Access time is again very small (at most a factor of 1.27). However, for all the four types of distribution, the total access time is nearly consistent (of the same order). Thus using the two-phase access we are able to get the consistent data distribution performance which is independent of both the disk distribution and the processor distribution.

Tables 19 and 20 show access times for arrays distributed in the Block-Block fashion over 16 and 64 processors respectively. Again, note that the access time is consistent with the times obtained for other distributions.

The results above show that for every case, regardless of the desired data distribution, performance is improved to within a factor of 2 of the Best Access Time performance. Further, the cost of redistribution is small compared with the Total Read Times. This indicates an effective exploitation of the additional degree of connectivity available within the interconnection network of the computational array.

Table 20: Block-Block Distribution over 64 Processors Using **pread** (time in msec)

| Size | Optimal Access Time | Redistr. Time | Total Access Time |
|---|---|---|---|
| 1K*1K | 350 | 82 | 432 |
| 2K*2K | 1100 | 186 | 1286 |
| 4K*4K | 2462 | 577 | 3039 |

Further, the results also show that by using the runtime primitives, the data can be distributed in Block-Block fashion effectively. It is important to note that we obtain consistently good performance, within a factor of 2 of the best performance, for all distributions. Hence, the user now has the freedom to choose the best data distribution on the computational nodes for his/her program without being concerned about how the chosen distribution will affect the I/O performance. This allows independent optimizations for I/O accesses as well as computational algorithm, potentially improving the performance of both.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

This thesis highlighted the importance of I/O in the overall performance of a computer system. The study concentrated on the solving the I/O problem for multiprocessor organizations. We analyzed the software aspect of the I/O problem, i.e., the need to provide scalable and efficient software support to utilize the underlying hardware at the maximum extent. We present some of the prominent results of our work below.

Various issues involved in the design of a parallel I/O system were reviewed. The main intention of this review was to analyze various design factors and to study the interaction between them. The study concentrated on both the software and hardware aspects of the design. This study helped us to gain an understanding of the factors that need to be considered in designing an efficient software support for parallel I/O. We found that the cost associated with accessing the data from the disks is mainly responsible for the low I/O bandwidth. Disk interleaving along with data declustering is commonly used to tackle this problem. However, these hardware advances alone have been unable to reduce these I/O costs. Appropriate software support is essential

to improve the I/O bandwidth. The I/O bandwidth is improved by accessing the files in parallel. It was found that the choice of the file access pattern depends on the scientific workload. It was also observed that data prefetching and disk caching is useful to bring down the I/O response time. We also found that existing parallel file system interfaces do not provide substantial support for parallel I/O operations.

To study the effects of the above discussed parameters, we performed experiments on a existing parallel file system. We used Touchstone Delta Concurrent File System as our experimental test-bed. We found that data declustering dramatically improves the performance of the I/O system. It was found that using the file access modes ( provided to support the parallel file access patterns) files can be distributed easily and efficiently. However, we also observed that overall I/O performance depends on the number of processors and on the number of disks. It was observed that large buffer sizes improve the I/O transaction time. Finally, we observed that use of a common interconnection network for I/O and communication.

An I/O model was presented to explain the effect of I/O performance on the overall bandwidth of a multiprocessor connected with a parallel file system. Using Touchstone Delta CFS as a case study, I/O costs were calculated for common data distributions. Experimental results of data distributions show that the performance is inconsistent and depends on the data mappings on the processors and disks. Moreover, it was observed that there exists a conformal mapping between the processors and disks which gives the best performance. On Touchstone Delta CFS, for the arrays stored in the column-major form, i.e., column-block mapping gives the best performance assuming Fortran program data mapping. Also it was found that complex data distributions (like Block-Block or Block-Cyclic) are not supported in CFS. Based on these results, we proposed a two-phase data access strategy which performs the data access using two stages, first it accesses the data in the conformal manner

(for example, column-block on Touchstone Delta) and then it redistributes the array over the processors to obtain the resultant distribution.

Data distribution is used by several MIMD languages to map the problem domain on the processor domain. Several languages use explicit parallel I/O syntax for distributing the data. However, these languages are susceptible to performance fluctuations and limitations. Motivated by these facts and the design of the two-phase data access strategy, we developed a runtime system for parallel I/O. This system provides the portability and convenience of the language supported data distributions. In addition, it decouples the user mapping from the disk mapping. The runtime system includes primitives for file reading, file writing, array mapping and processor mapping. Further, since these primitives are linked at compile time, they can be used with any MIMD or data parallel language. Experimental results show that these primitives achieve consistent performance across a variety of data distributions, and allow the user to avail of complex data distributions such as Block-Block and Block-Cyclic.

## 6.2 Future Work

This thesis emphasized on the application level software support for parallel I/O. The scope of the runtime primitives can be expanded to include runtime characterization of the data access patterns. Future runtime primitives will *intelligently* schedule I/O requests depending on the data access patterns, number of disks on which the data resides, etc. These primitives will have the ability to choose the number, type of processors and file access modes to provide an optimal I/O performance.

Further modifications in the primitive design are needed to support various data distributions in languages like High Performance Fortran (HPF) or Fortran 90D. This

will finally lead to a common parallel I/O syntax for various languages that can be ported to various target machines. When the parallel I/O support is provided at the language level, we can perform various compiler optimizations. These involve studying I/O dependencies between the statements, scheduling I/O operations to optimize the total I/O transaction time, provide suitable I/O operations for HPF directives like REDISTRIBUTE,ALIGN etc.

# Bibliography


[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *1990 International Conference on Supercomputing*, 1990.

[ACD+91] Anant Agarwal, D. Chiaken, G. D'Souza, Kirk Johnson, and D. Kratz et. al. The MIT Alewife Machine: A large-scale distributed memory multiprocessor. In *Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991.

[Ake91] Akella J. and Siewiorek D. P. Modeling and Measurement of the Impact of Input/Output on System Performance . *Proc. of 18th Ann. Symp. on Computer Architecture*, May 1991.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proc. AFPIPS Spring Joint Computing Conference, Vol. 30*, april 1967.

[And90] Andrew Witkowski, Kumar Chandrakumar, and Greg Macchio. Concurrent I/O system for the Hypercube Multiprocessor. Technical Report C3P-611, California Institute of Technology,Jet Propulsion Laboratory, 1990.

[AS89] Raymond K. Asbury and David S. Scott. FORTRAN I/O on the iPSC/2: Is there read after write? In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 129–132, 1989.

[Bar93] Barnett M., Rick Littlefield, David G. Pyne and Robert van de Geijn. Global Combine on Mesh Architectures with Wormhole Routing. *Proceedings of 7'th International Parallel Processing Symposium*, April 1993.

[BCZ92] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. *Scalable High Performance Computing Conference*, April 1992.

[Ber78] Berbec S., Shibamiya A., Togasaki S., and Yoshida H. Use of direct access storage devices by MVS customers-Guide servey results. *Proc. Guide 47 Conf.*, November 1978.

[BGMZ92a] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent File Operations in a high Performance Fortran. Technical Report ICASE Report 92-46, ICASE, NASA, Hampton VA 23681, September 1992.

[BGMZ92b] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent File Operations in a high Performance Fortran. In *Supercomputing 92*, pages 230–238, November 1992.

[BR89] David K. Bradley and Daniel A. Reed. Performance of the Intel iPSC/2 Input/Output System. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 141–144, 1989.

[CGK+88] Peter Chen, Garth Gibson, Randy Katz, David Patterson, and Martin Schulze. Two papers on RAIDs. Technical Report UCB/CSD 88/479, UC Berkeley, December 1988.

[CGKP90] Peter Chen, Garth Gibson, Randy Katz, and David Patterson. An Evaluation of Redundant Arrays of Disks using an Amdahl 5890. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1990.

[Cho93] Choudhary Alok. Parallel I/O Systems, Guest Editor's Introduction. *Journal of Parallel and Distributed Computing*, January/February 1993.

[CK91] Ann L. Chervenak and Randy H. Katz. Performance of a Disk Array Prototype. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, 1991.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming, Vol.1, No.1*, 1992.

[Cro88] Thomas W. Crockett. Specification of the Operating System Interface for Parallel File Organizations. Publication status unknown (ICASE technical report), 1988.

[Cro89] Thomas W. Crockett. File Concepts for Parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[Dav90] David Kotz and Carla Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 218–230, April 1990.

[Dav92] Dave Minturn. CFS Performance. *Proceedings of the Delta Advanced User Training Class Notes*, pages 75–98, July 1992.

[DCF+86] W.J. Dally, A. Chien, S. Fiske, W. Howart, J. Keen, and M. Larivee. The J Machine: A Fine Grain concurrent Computer. *Information Processing 89, Proceedings of the IFIP Conference*, pages 1147–1153, August 1986.

[DdR92] Erik DeBenedictus and Juan Miguel del Rosario. nCUBE Parallel I/O Software. In *Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (IPCCC)*, pages 0117–0124, April 1992.

[DeB91] DeBenedictis Erik and Juan Miguel del Rosario. Scalable I/O. Technical Report nCUBE-TR001-911015, nCUBE Corporation, October 1991.

[Dem92] G. Demos. High-Performance I/O Systems for Scientific Visualisation. In *Supercomputing 92*, November 1992.

[DJK+92] Lenowski Daniel, Laudon James, Gharachorloo Kourosh, Weber Wolf-Dietrich, Gupta Anoop, Hennessy John, Horowitz Mark, and Lam Monica. The Stanford Dash Multiprocessor. *IEEE Computer*, March 1992.

[DM91] Erik DeBenedictus and Peter Madams. nCUBE's Parallel I/O with Unix Capability. In *Sixth Annual Distributed-Memory Computer Conference*, pages 270–277, 1991.

[DS92] P. C. Dibble and M. L. Scott. The Parallel Interleaved File System: A Solution to the Multiprocessor I/O Bottleneck. *IEEE Transactions on Parallel and Distributed Systems*, 1992. To appear.

[DSE88] Peter Dibble, Michael Scott, and Carla 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.

[EK89] Carla Schlatter Ellis and David Kotz. Prefetching in File Systems for MIMD Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:306–314, August 1989.

[Fal91] Faloutsos Christos and Metaxas Dimitri. Disk Allocation Methods Using Error Correcting Codes. *IEEE Transactions on Computers, Vol. 40, No. 7*, July 1991.

[Fan86] Fang M. F. and Lee R.C.T.and Chang C.C. The idea of de-clustering and its applications. *Proc. 12th Intl. Conf. Very Large Databases, Kyoto, Japan*, August 1986.

[FHK+90] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Uli Kremer, and Chau-Wen Tseng. Fortran D Language Specification. Technical Report Rice COMP TR90-141, Rice University, Houston, Texas, December 1990.

[For93] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report CRPC-TR92225, Center for Research in Parallel Computing,Rice University, January 1993.

[Fox90] Geoffrey C. Fox. Hardware and Software Architectures for Irregular Problem Architecures. Technical Report SCCS-111,CRPC-TR91164, Syracuse University, October 1990.

[Fox91] Geoffrey C. Fox. Fortran D as a Portable Software System for Parallel Computers. Technical Report SCCS-91,CRPC-TR91128, Syracuse University, June 1991.

[FPD91a] James C. French, Terrence W. Pratt, and Mriganka Das. Performance Measurement of a Parallel Input/Output System for the Intel iPSC/2 Hypercube. *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[FPD91b] James C. French, Terrence W. Pratt, and Mriganka Das. Performance Measurement of a Parallel I/O Subsystem for Hypercube Multicomputers. *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[FPD91c] James C. French, Terrence W. Pratt, and Mriganka Das. Performance Measurement of a Parallel Input/Output System for the Intel iPSC/2 Hypercube. Technical Report IPC-TR-91-002, Institute for Parallel Computation, University of Virginia, 1991. Appeared in, Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.

[FPD93] James C. French, Terrence W. Pratt, and Mriganka Das. Performance Measurement of the Concurrent File System of the Intel iPSC/2 Hypercube. *Journal of Parallel and Distributed Computing*, January/February 1993.

[GMG+88] G.Fox, M.Johnson, G.Lyzenga, S.Otto, J. Salmon, and D.Walker. *Solving Problems on Concurrent Processors*. Prentice Hall,Eaglewood Cliffs,NJ, 1988.

[GMS88] Hector Garcia-Molina and Kenneth Salem. The Impact of Disk Striping on Reliability. *IEEE Database Engineering Bulletin*, 11(1):26–39, March 1988.

[Gus88] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM, Vol. 31, Number 5*, pages 532–533, May 1988.

[Int88] iPSC/2 I/O Facilities. Intel Corporation, 1988. Order number 280120-001.

[Int89] Concurrent I/O Application Examples. Intel Corporation Background Information, 1989.

[Int90] Intel. Intel i860 System Description. Technical Report Intel Advanced Information, Intel Corporation, 1990.

[Int91a] Paragon XP/S Product Overview. Intel Corporation, 1991.

[Int91b] Intel. Touchstone Delta System Description. Technical Report Intel Advanced Information, Intel Corporation, 1991.

[Int92] Intel. Paragon XP/S System Description. Technical Report Intel Advanced Information, Intel Corporation, 1992.

[Joy91] Joydeep Ghosh and Bipul Agarawal. Parallel I/O Subsystem for hypercube multicomputers. In *Fifth International Parallel Processing Symposium*, pages 381–4, MAY 1991.

[Joy93] Joydeep Ghosh, Delvin D. Goveas and Jeffrey T. Draper. Performance Evaluation of a Parallel I/O Subsystem for Hypercube Multicomputers. *Journal of Parallel and Distributed Systems*, January 1993.

[Jua92a] Juan Miguel del Rosario. A Guide to Striped Files and Parallel I/O in nCUBE System Software, Release 3.1. Technical Report nCUBE-TR002-920713, nCUBE Corporation, July 1992.

[Jua92b] Juan Miguel del Rosario. High Performance Parallel I/O System. *Institute of Electronics,Information and Communication Engineers Transactions,Japan*, AUGUST 1992.

[Jua92c] Juan Miguel del Rosario, Rajesh Bordawekar and Alok Choudhary. A Two-Phase Strategy for Achieving High-Performance Parallel I/O. Technical Report SCCS-408, NPAC, Syracuse University, October 1992.

[KE91] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 182–189, December 1991. To appear in *Distributed and Parallel Databases*.

[KE92] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Distributed and Parallel Databases*, 1992. To appear.

[KE93] David Kotz and Carla Schlatter Ellis. Caching and Writeback Policies in Parallel File Systems. *Journal of Parallel and Distributed Computing*, January 1993.

[KGP89] Randy H. Katz, Garth A. Gibson, and David A. Patterson. Disk System Architectures for High Performance Computing. *Proceedings of the IEEE*, 77(12):1842–1858, December 1989.

[Kim85] Michelle Y. Kim. Parallel Operation of Magnetic Disk Storage Devices: Synchronised Disk Interleaving. *Proc. of 4th Int. Workshop Database of Database Machines*, 1985.

[Kim86a] Michelle Y. Kim. Synchronized Disk Interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[Kim86b] Michelle Y. Kim. *Synchronously Interleaved Disk Systems with their Application to the Very Large FFT.* PhD thesis, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, 1986. IBM Report number RC12372.

[KOP+89] Randy H. Katz, John K. Ousterhout, David A. Patterson, Peter Chen, Ann Chervenak, Rich Drewes, Garth Gibson, Ed Lee, Ken Lutz, Ethan Miller, and Mendel Rosenblum. A Project on High Performance I/O Subsystems. *Computer Architecture News*, 17(5):24–31, September 1989.

[KOPS88] Randy H. Katz, John K. Ousterhout, David A. Patterson, and Michael R. Stonebraker. A Project on High Performance I/O Subsystems. *IEEE Database Engineering Bulletin*, 11(1):40–47, March 1988.

[Kot91] David Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016.

[Kot92] David Kotz. Multiprocessor File System Interfaces. Technical Report PCS-TR92-179, Dept. of Math and Computer Science, Dartmouth College, March 1992. Abstract appeared in 1992 Usenix Workshop on File Systems.

[KSR92] KSR1 Technology Background. Kendall Square Research, January 1992.

[Kun86] Kung H. T. Memory requirements for balanced computer architecture. *Thirteenth International Symposium on Computer Architecture*, pages 49–54, 1986.

[Lio93] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, February 1993.

[LKB87] 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.

[Mat77] R. E. Matick. Computer Storage Systems and Technology, 1977.

[Mic91] Michelle Y. Kim and Asser N. Tantawi. Asynchronous Disk Interleaving: Approximating Access Delays. *IEEE Transactions on Computers, Vol. 40, No. 7*, July 1991.

[NA91] Daniel Nussbaum and Anant Agarwal. Scalability. *Communications of the ACM, Vol. 34,Number 3*, March 1991.

[nCU92] nCUBE. nCUBE 2 Systems: Technical Overview. Technical report, nCUBE Corporation, 1992.

[Nit92] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[Pie89] Paul Pierce. A Concurrent File System for a Highly Parallel Mass Storage System. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[Pow77] Powell M. L. The DEMOS File System. *Proc. Sixth ACM Symp. Operating System Princ.*, November 1977.

[Raj93] Rajesh Bordawekar, Alok Choudhary and Juan Miguel del Rosario. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. In *International Conference on Supercomputing*, June 1993.

[Rav92a] Ravi Ponnusamy, Alok Choudhary and Geoffrey Fox. Communication Ovearhead on CM-5:An Experimental Performace Evaluation. *Frontiers of Massively Parallel Computation*, pages 108–117, October 1992.

[Rav92b] Ravi Ponnusamy, Rajiv Thakur, Alok Choudhary and Geoffrey Fox. Scheduling Regular and Irregualar Patterns on the CM-5. *Supercomputing'92*, November 1992.

[RB89a] A. Reddy and P. Banerjee. Evaluation of multiple-disk I/O systems. *IEEE Transactions on Computers*, 38:1680–1690, December 1989.

[RB89b] A. Reddy and P. Banerjee. An Evaluation of multiple-disk I/O systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:315–322, 1989.

[RB89c] A. L. Reddy and P. Banerjee. I/O issues for hypercubes. In *International Conference on Supercomputing*, pages 72–81, 1989.

[RB89d] A. L. Reddy and Prithviraj Banerjee. A Study of Parallel Disk Organizations. *Computer Architecture News*, 17(5):40–47, September 1989.

[RB90a] A. L. Narasimha Reddy and Prithviraj Banerjee. Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):140–151, April 1990.

[RB90b] A. L. Narasimha Reddy and Prithviraj Banerjee. A Study of I/O Behavior of Perfect Benchmarks on a Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 312–321, 1990.

[RBA88] A. L. Reddy, P. Banerjee, and Santosh G. Abraham. I/O Embedding in Hypercubes. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 1, pages 331–338, 1988.

[Red92a] A. L. Narasimha Reddy. Reads and Writes: When I/Os Aren't Quite the Same. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, pages 84–92, 1992.

[Red92b] Reddy A. N. A Study of I/O System Organizations. *Proc. Of Int. Symp. on Computer Architecture*, May 1992.

[Rik92] Rik Littlefield. Tuning Communication. *Proceedings of the Delta Advanced User Training Class Notes*, pages 99–120, July 1992.

[Sal86] John Salmon. CUBIX: Programming Hypercubes without Programming Hosts. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 3–9, 1986.

[Sha93] Shahram Ghandeharizadeh and Luis Ramos. Real-Time Display of Multimedia Data Using Parallelism. *To be appear in IEEE Transactions of Knowledge and Data Engineering*, 1993.

[Smi78] Smith A. J. Sequential program prefetching in memory heirarchies. *IEEE Computer*, pages 7–21, December 1978.

[Sni92] Marc Snir. Proposal for IO. Posted to HPFF I/O Forum by Marc Snir, July 7 1992.

[Sto81] Stonebraker M. Operating system support for database management. *Communications of the ACM, Vol.24,No.7*, pages 161–203, JULY 1981.

[Thi91] Thinking Machines Corp. CM-5 System Description. Technical report, Thinking Machines Corporation, 1991.

[TMC87] Connection Machine Model CM-2 Technical Summary, April 1987.

[TMC91] *The Connection Machine CM-5 Technical Summary*, October 1991.

[Tse93] Chau-Wen Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Department of Computer Science, Rice University, P.0. Box 1892, Houston, TX 77251-1892, 1993.

[USP91] U.Nagaraj, U.S. Shukla, and A. Paulraj. Design and Evaluation of a High Performance File System for Message Passing Parallel Computers. *International Parallel Processing Symposium*, pages 549–553, May 1991.

[ZBC+92] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. Technical Report ICASE INTERIM REPORT 21, MS 132c, ICASE, NASA, Hampton VA 23681, 1992.

[Zek92a] Zeki Bozkus, Sanjay Ranka and Geoffrey Fox. Benchmarking the CM-5 Multicomputer. *Frontiers of Massively Parallel Computation*, pages 100–107, October 1992.

[Zek92b] Zeki Bozkus,Alok Choudhary,Geoffrey Fox,Tomasz Haupt and Sanjay Ranka. Compiling Distribution Directives in a Fortran 90D Compiler. Technical Report SCCS-388, NPAC, Syracuse University, July 1992.

[Zek92c] Zeki Bozkus,Alok Choudhary,Geoffrey Fox,Tomasz Haupt and Sanjay Ranka. Compiling the FORALL statement on MIMD parallel computers. Technical Report SCCS-389, NPAC, Syracuse University, July 1992.

# Biographical Data

Rajesh R. Bordawekar received his B.E. degree in Electrical Engineering from the University of Bombay, Bombay, India in 1991. Currently he is pursuing his M.S. in Computer Engineering at the Syracuse University.

He is working as a research assistant at the Northeast Parallel Architectures Center, Syracuse University from fall 1991.

He was awarded the National Talent Search Scholarship by the Government of India during 1985-1991.

His current research interests are in Parallel I/O, Parallel Compilers and Computational Science.