Design and Evaluation of Primitives for Parallel I/O *

Rajesh Bordawekar Juan Miguel del Rosario Alok Choudhary†
Northeast Parallel Architectures Center, 3-201 CST, Syracuse Univ., Syracuse, NY 13244

Abstract

In this paper, we show that the performance of parallel file systems can vary greatly as a function of the selected data distributions, and that some data distributions can not be supported.

We have devised an alternative scheme for conducting parallel I/O - the Two-Phase Access Strategy - which guarantees higher and more consistent performance over a wider spectrum of data distributions. We have designed and implemented runtime primitives that make use of the two-phase access strategy to conduct parallel I/O, and facilitate the programming of parallel I/O operations. We describe these primitives in detail and provide performance results which show that I/O access rates are improved by up to several orders of magnitude. Further, we show that the variation in performance over various data distributions is restricted to within a factor of 2 of the best access rate.

1 Introduction

Parallel computers have become the preferred computational instrument of the scientific community due to their immense processing capacities. Some of the commercially available parallel computers include Intel Paragon [10], nCUBE [13], CM-5 [15].

As scientists expand their models to describe physical phenomena of increasingly large extent, the memory capacity of parallel machines, although immense, become insufficient to contain all the required computational data, and I/O becomes important [1]. Thus, a system with limited I/O capacity can severely limit the performance of the entire program - this is known as the I/O bottleneck problem. This problem has become critical, and the need for high I/O bandwidth has become significant enough that most parallel computers such as the Intel iPSC/2 [8], Intel iPSC/860 [14], Intel Touchstone Delta [9, 4], and the nCUBE [11] now provide some measure of support for parallel I/O.

The goal of parallel I/O is to provide a bottleneck free communication pathway between the processors and I/O devices. This is made possible in hardware by the scalability of the hardware architecture design. For example, as shown in figure 1, the I/O connections between the processor array and the I/O devices, which are a scalable collection of multiple physical paths of fixed bandwidth, are viewed as a single channel of higher bandwidth. In software, a parallel file system provides increased performance by declustering data across the disk array (a technique called striping) thereby distributing the access workload over multiple servers.

Parallel file systems vary in their level of support for data distribution mappings; some provide no support whatsoever. The inconvenience of having to explicitly specify and control file access for a given data distribution has prompted recent proposals for the inclusion of parallel I/O support primitives into parallel programming languages such us HPF Fortran [7] and Vienna Fortran [2]. These primitives could then be implemented as library routines which accept a description of the desired data distribution from the user, and manage data access based upon the correspondence between the user mapping, and the mapping defined by the parallel file system (i.e., the distribution of data across the disks).

1.1 Contributions of the paper

For experiments presented in this paper we limit overselves to the Intel Touchstone Delta file system called Concurrent File System (CFS). We show that the performance of the CFS can vary greatly as a function of the data distribution. Further, that parallel I/O for certain common data decompositions can

*This work was sponsored by ARPA under contract # DABT63-91-C-0028. Alok Choudhary's research is also supported by an NSF Young Investigator Award CCR-9357840. 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 research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Center for Research on Parallel Computation (CRPC).

†Also with ECE Dept.

Figure 1: Scalability in computation and I/O

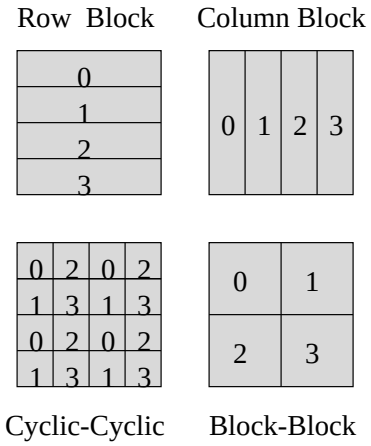

Figure 2: Data Distributions in Fortran 90D/ HPF

not be supported by CFS (i.e., access for these is sequentialized). Based upon these observations, we have devised an alternative scheme for conducting parallel I/O - the two-phase access strategy - which guarantees more consistent performance over a wider spectrum of data distributions [11].

In order to facilitate the programming of parallel I/O operations, we have designed a set of primitives which we have implemented in a runtime library that makes use of the two-phase access strategy. This runtime system supports a number of parallel file systems, thus providing a common I/O interface for parallel programs.

1.2 Organization

The purpose of this paper is to describe the I/O primitives interface design and to present some performance results. The paper has the following organization. In section 2, we overview current parallel language support for data distribution. In section 3, we consider several data decomposition strategies, present performance results for the CFS based upon direct access (i.e., the access strategy used by a typical parallel program on the basis of programmer specified data distribution), and analyze the costs associated with this type of access. Performance results were obtained on the Intel Touchstone Delta [9, 3]. In section 4, we describe the design and implementation of the runtime system employing the two-phase strategy. In section 5, we present experimental performance results for the runtime primitives. Finally, we summarize in section 6.

2 Languages Supporting Data Distribution

We concentrate on parallel programs which use 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.) (figure 2). 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, 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 [2], Fortran D [5] and High Performance Fortran or (HPF) [7, 6].

In order to address the I/O bottleneck problem, these languages propose to provide some support for parallel I/O operations. Important examples include Vienna Fortran [2] and High Performance Fortran [7, 6].

3 Analysis of Data Distributions for Parallel I/O

In this section, we will analyze the data mapping from the disks (distributed files) to the compute nodes.

We discuss the I/O costs associated with various data distributions and present experimental results.

3.1 Mapping Problem

In order to perform a mapping from distributed file to processor array, we note that two mappings have to be considered. The organization of the file data over the set of disks represents the first mapping, M1. The second mapping, M2, involves the (more familiar) mapping of data over the set of processing elements. For parallel I/O to take place efficiently, both these mappings must be resolved into a data transfer strategy. Current parallel file systems on the nCUBE/2 [11] and the Intel Touchstone Delta resolve these mappings into a single data transfer mapping which is used to compute proper source and destination addresses during file data access - we call this *direct access*. Problems arise from this approach in cases where the first and second mappings resolve into a data transfer mapping (representing an access strategy) that performs poorly. In succeeding sections, we will show that such problematic mapping pairs are quite common.

The enormous costs associated with such a direct access strategy mapping is illustrated in figure 3. In section 3.3, using the experimental results, we show that this type of access strategy gives very poor performance.

To illustrate a mapping that can not be supported by existing systems, consider a program that has to read data into a distributed array in a Block-Block decomposition (see Figure 2). Suppose that the data is stored over the distributed disks in column-major order. The current Intel CFS (Concurrent File System) could not support this requirement because it does not allow any processor to read data while others idle, this is illustrated in figure 4. The exception to this is mode 0 (independent file pointers to a shared file); this mode would require the programmer to manage file pointer adjustment throughout the program.

3.2 Direct Access Cost Analysis

Since the cost of data access is dominated by per message startup latency, and seek time, the cost of data movement can be evaluated on the basis of the total number of requests needed to complete a transaction (e.g., process of reading in a 4Kx4K matrix into the computational array). Figure 3 illustrates the dependence of the number of requests on data distribution; we see that a Row-Cyclic distribution generates many more requests than a Row-Block distribution.

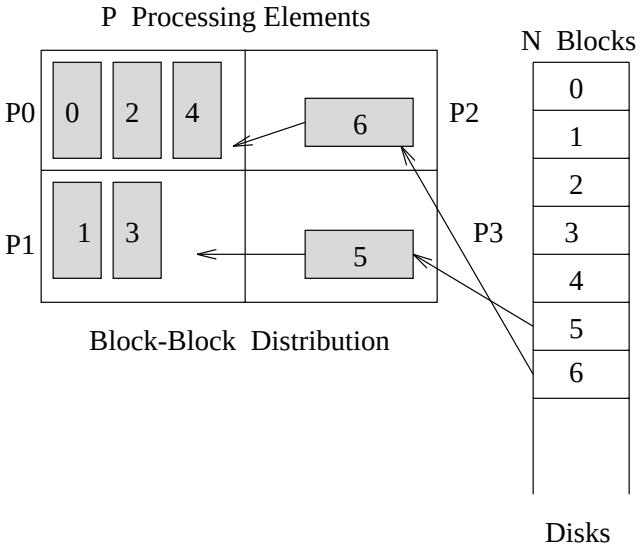

Figure 4: CFS Access for Block-Block Distribution with Column-major Disk Storage Pattern

Table 1: Number of I/O Requests as a Function of Data Distributions for 2-D Arrays.(R_{dist})

Distr. Type	R_{dist}	S_{dist}
Block-Block	$N * \sqrt{P}$	$\frac{N}{\sqrt{P}}$
Block-Cyclic	$N * \sqrt{P}$	$\frac{N}{\sqrt{P}}$
Cyclic-Block	N^2	1
Cyclic-Cyclic	N^2	1

Table 1 shows R_{dist} and S_{dist} for an N*N array distributed over P processors, where R_{dist} is the number of requests per transaction when considering only the data distribution and ignoring contributions from the stripe size; and S_{dist} is the size of the largest contiguous block of data that can be transferred between a processor and an I/O device per request(i.e., request size). In generating the table, it is assumed that the data is stored in a column-major one-dimensional map over the disks.

Thus, the total number of requests for a given transaction, R_{trans}, as a function of both data distribution and stripe size S_{stripe}, can be expressed as

$$R_{trans} = R_{dist} \times \frac{S_{dist}}{min(S_{dist}, S_{stripe})} \qquad (1)$$

Note that the assumption that S_{stripe} equals S_{dist} is equivalent to ignoring stripe size contributions (i.e., assumption in table 1) and that we do obtain the

Figure 3: Effects of Distribution upon Number of Requests (M2 Map)

results in the table.

For the following discussion, it is assumed for simplicity that the either S_{stripe} divides S_{dist} or vice versa (i.e., GCD(S_{stripe}, S_{dist}) = MIN(S_{stripe}, S_{dist})).

3.3 Direct Access Performance

In this section we present performance results for direct access using various data distributions. The experiments were conducted on an Intel Touchstone Delta. The Delta is a 16x32 mesh structured, 512 processor multicomputer with two disks connected on either side of each row; thus, it has 64 disks.

In these experiments, the mesh size was varied from 4 processors to 512 processors and all 64 disks were used. For each mesh size the data array size was varied; a square two-dimensional array was distributed across the processors. The smallest array used was 1Kx1K (1MByte), and the largest was 20Kx20K (400 MBytes). For each mesh size, the array was distributed in four ways: Row-Block, Row-Cyclic, Column-Block, Column-Cyclic. The larger arrays were distributed over larger mesh sizes such as 256 and 512.

An input file was distributed over 64 disks in a round-robin fashion (32 I/O nodes, 2 disks per I/O node) with a stripe size of 4 Kbytes in column major fashion.

The Concurrent File System (CFS) on the Delta supports several modes of operation, each one determining a degree of synchronization and sharing of file pointers.

For our experiments, we restrict ourselves exclusively to mode 3 (shared file pointer, synchronized access) since this gives the best performance for direct access [4].

3.3.1 Column-Block Distribution

This distribution conforms with the column-major data distribution over the disks. 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.

Table 2 shows the performance for the Column-Block array distribution. The table shows the size of the array (file on disks), the number of processors participating in the read, the transaction completion time, and the observed bandwidth. For small arrays and number of nodes, the bandwidth of the I/O system is under-utilized. As the data size and 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.

The read rate increased quickly in proportion to the processor grid size, but plateaued at about 64 processors. Degradation in the performance was observed after 256 processors due to a large synchronization overhead.

3.3.2 Column-Cyclic Distribution

Table 3 shows the read access times for the same parameters but with a Column-Cyclic data distribution. Even though the degree of parallelism in the data access remains the same, the number of I/O requests increases because each processor must make an indi-

Table 2: Column Block Distribution (Time in msec)

Array Size	No. of Procs.	Time
1K*1K	4	431
4K*4K	4	2277
5K*5K	16	3357
5K*5K	64	3324
10K*10K	256	13707
20K*20K	512	70953

Table 3: Column Cyclic Distribution (Time in msec)

Array Size	No. of Procs.	Time
1K*1K	4	4353
4K*4K	16	5233
5K*5K	64	11407
10K*10K	256	116763
20K*20K	512	252980

vidual request for each column. This increases the access time as illustrated in table 3. The drop in performance versus the Column-Block distribution is consistent for all configurations and it ranges between a factor of 2 to 10.

3.3.3 Row-Block Distributions

Table 4 shows the performance for a Row-Block distribution. This read operation essentially involves a transposition of the data as it is being read from the disks. Table 1 shows that the number of logical requests is NxP for this decomposition. We observe from table 4 that the performance degradation due to the decomposition is almost two orders of magnitude when compared to that of the Column-Block distribution.

We do not present performance figures for larger configurations (i.e., large array and system sizes) since the time to complete these experiments exceeded practical limits.

Table 4: Row Block Distribution (Time in msec)

Array Size	No. of Procs.	Time
1K*1K	4	17051
2K*2K	4	25966
4K*4K	16	71205
5K*5K	16	91536
5K*5K	64	38018

3.3.4 Row-Cyclic Distribution

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 a 1Kx1K character array in Row-Cyclic order versus the 467 msec. it would require in Column-Block form. This shows that the direct row distribution of an array is very slow, hence, not possible in practice.

The large variation in performance observed above motivated the design of the two-phase access strategy for the parallel I/O primitives as described in the following section.

4 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 out 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.

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 below) 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.

Advantages of Runtime I/O Primitives:

1. The runtime system can be easily ported on various machines which provide parallel file systems.
2. Complex data distributions (Block-Block or Block-Cyclic) are made available to the user.
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. 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.

4.1 Approach

Our I/O strategy involves a division of the parallel I/O task into two separate phases. 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 (i.e, we introduce an intermediate mapping $M2'$, and access data with $M2' = M1$). 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 (i.e., from $M2'$ to M2).

By employing the two-phase redistribution strategy, the costs inherent in many of the I/O configurations are avoided. 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 subsection that follows, we discuss the runtime I/O primitives. A brief description of the purpose of each primitive, its functional flow, and syntax is provided followed by some performance results.

4.2 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. Syntax description and implementation details are presented in [3].

4.2.1 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 [10] 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 [4].

Table 5: The File Descriptor Array (FDA): Fortran Version

Unit	Access	Form	Status	No. of disks	No. of procs
3	0	0	1	64	4

The user passes file information to the **popen** primitive which is then stored in a 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',-1,4)

opens a file called TEST over 4 processors. The corresponding FDA is shown in table 5. The file will be distributed over all the disks (number_of_disks = -1). If the number_of_processors is -1, the file will be opened by default number of processors (P).

4.2.2 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.

4.2.3 array_map

This primitive is semantically similar to the compiler directives in HPF or VF. A Fortran D or HPF compiler would directly extract this information from the distribution directives.

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

Table 6: The Array Description Table (ADT)

Info	1	2	3	4	5	6	7
Global Size	64	64	-1	-1	-1	-1	-1
Distr. 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

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 6. The value -1 is used to denote don't-care entries.

4.2.4 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. Using the **proc_map** primitive the programmer can change the logical processor configuration during the execution of the program.

4.2.5 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. 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 the data is redistributed over the P processors to obtain the target data distribution. Figure 5 shows the **pread** algorithm.

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 and $P' = P$ then read the data using the same distribution, 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 5: pread Algorithm

4.2.6 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. If the processor distribution 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 6 shows the **pwrite** algorithm.

4.3 A Sample Program

This section provides a sample Touchstone Delta Fortran program using the I/O primitives (Figure 7). 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

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, 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 6: pwrite Algorithm

```
PROGRAM EXAMPLE
size_info(7),distr_info(7),block_size(7),proc_info(7)
A(64,16),ad,array_map,mybuffer
TEMP(1024),mymap(1536)
COMMON /INFO/ F_INFO,P_INFO,A_INFO
size_info(1)=64 size_info(2)=64
distr_info(1)=0 distr_info(2)=2
block_size(1)=-1 block_size(2)=-1
proc_info(1)=1 proc_info(2)=4
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)
call pread(A,64,16,ad,3,TEMP,iobuffer)
Use a temporary buffer called TEMP of size iobuffer.
Computation Starts here
.....
call pwrite(A,64,16,ad,3,TEMP,iobuffer)
call pclose(3)
END
```

Figure 7: A Sample Program For Performing Parallel I/O

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**. 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 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 Best Read, Redistribute, Total Read, and Direct Read times for the four 1-dimensional distributions considered in this paper.

For a given array size, the Best Read time represents the minimum of the read times of the four distributions; the Best Read 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 Read time is the sum of the Best Read 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 access). The Direct Read 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 11 and 12 do not present any performance numbers for direct access. (1 denotes Column Block access, 2 column

Table 7: Comparing Direct Access with Two-phase Access (16 Processors, 5K*5K Array, time in msec)

Distr Mode	Best Read	Re Distr.	Total Read	Direct Read	Speedup
1	3357	-	**3357**	**3357**	**1**
2	3357	1805	**5162**	**9890**	**1.92**
3	3357	673	**4030**	**69939**	**17.36**
4	3357	2603	**5960**	*	> 604

Table 8: Comparing Direct Access with Two-phase Access (16 Processors, 10K*10K Array, time in msec)

Distr. Mode	Best Read	Re Distr.	Total Read	Direct Read	Speedup
1	10376	-	**10376**	**10376**	**1**
2	10376	7105	**17481**	**19271**	**1.10**
3	10376	2772	**13148**	**84683**	**6.44**
4	10376	10320	**20696**	*	> 173

cyclic access, 3 denotes Row Block and 4 represents Row cyclic access.)

Tables 7 and 8 show access times for 5Kx5K and 10Kx10K arrays, read and distributed over 16 processors respectively. The Best Read time occurs for the Column-Block distribution. For all cases below, the '*' symbol denotes a read time on the order of hours. The following observations are made by comparing the direct access read 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 example, in table 7 the amount of overhead avoided by using the redistribution strategy (i.e., the difference between the Total Read Time and Direct Read Time) ranges from 1.7 secs, to well over 60 minutes for the 5K Row-Cyclic case. More importantly, the deviation in Total Read time is at most a factor of 1.9 as opposed to the widely varying results produced by the direct access approach.

Tables 9 and 10 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 Read time is again very small (at most a factor of 1.27). However, for all the four types of distribution, the total read time is nearly consistent (of the same order). Thus using the two-phase access we are able to get the data distribution performance which is independent of both the disk distribution and the processor distribution.

Table 9: Comparing Direct Access with Two-phase Access (64 Processors, 5K*5K Array, time in msec)

Distr. Mode	Best Read	Re Distr.	Total Read	Direct Read	Speedup
1	3324	-	**3324**	**3357**	**1**
2	3324	703	**4027**	**11407**	**2.83**
3	3324	246	**3570**	**38018**	**10.65**
4	3324	768	**4092**	*	> 879

Table 10: Comparing Direct Access with Two-phase Access (64 Processors, 10K*10K Array, time in msec)

Distr. Mode	Best Read	Re Distr.	Total Read	Direct Read	Speedup
1	11395	-	**11395**	**11395**	**1**
2	11395	2478	**13873**	**63400**	**4.57**
3	11395	1028	**11623**	**78767**	**6.78**
4	11395	3092	**14487**	*	> 248

Tables 11 and 12 show access times for arrays distributed in the Block-Block fashion over 16 and 64 processors respectively. Again, note that the read time is consistent with the times obtained for other distributions.

5.1 Discussion

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 Read Time performance for all distributions. 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. Further, the results also show that by using the run-time primitives, the data can be distributed in Block-Block fashion effectively.

Table 11: Block-Block Distribution over 16 Processors using the Runtime Primitives (time in msec)

Size	Best Read	Redistr.	Total Read
1K*1K	467	112	579
2K*2K	717	416	1133
4K*4K	2328	1253	3181

Table 12: Block-Block Distribution over 64 Processors using the Runtime Primitives(time in msec)

Size	Best Read	Redistr.	Total Read
1K*1K	350	82	432
2K*2K	1100	186	1286
4K*4K	2462	577	3039

6 Conclusions

The need for high performance parallel I/O has become critical enough that most manufacturer's have provided some support for parallel I/O within their file systems. Recently, several high performance languages have proposed the inclusion of primitives to support parallel I/O.

We have shown that, using the direct access approach made available by production file systems, performance of existing file systems are inconsistent and depends upon both the data distribution and the file mapping to disk. We provide an example of how support for some of the more complex data distributions may not be provided. Also, we describe how programming language extensions, although simplifying the programming, do not alleviate the problems that exist within the file system.

We presented a set of runtime primitives which make use of the two-phase access strategy, and showed that the runtime primitives achieve consistent performance across a variety of data distributions, and allows the user to avail of complex data distributions such as Block-Block and Block-Cyclic. Further, the primitives allow the user to choose a subset of processors for performing I/O in order to optimize access on the basis of the selected data distribution.

References

[1] Alok Choudhary, Parallel I/O Systems, *Journal of Parallel and Distributed Computing*, January/February 1993.

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

[3] Rajesh Bordawekar. Issues in Software Support for Parallel I/O. Master's Thesis, ECE. Dept., Syracuse University, May 1993.

[4] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. An Experimental Performance Evaluation of Touchstone Delta Concurrent File System. *ICS'93*, pages 367-377, July 1993.

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

[6] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. *Supercomputing'93* (to appear), November 1993.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Technical Report CRPC-TR92225. CRPC, Rice University, January 1993.

[8] 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.

[9] Intel. Touchstone Delta System Description. Intel Advanced Information, Intel Corporation, 1991.

[10] Intel. Paragon XP/S System Description. Intel Advanced Information, Intel Corporation, 1992.

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

[12] 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, December 1992.

[13] nCUBE. nCUBE-2 Systems: Technical Overview. Technical Report, nCUBE Corporation, 1992.

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

[15] Thinking Machines Corp. CM-5 System Description. Technical Report, Thinking Machines Corporation, 1991.