Improved Parallel I/O via a Two-phase Run-time Access Strategy[*]

(Presented in: IPPS '93 Parallel I/O Workshop)

Juan Miguel del Rosario   Rajesh Bordawekar   Alok Choudhary

Northeast Parallel Architectures Center
111 College Place, Room 3-201
Syracuse University
Syracuse, NY 13244-4100

February 9, 1993

Abstract

As scientists expand their models to describe physical phenomena of increasingly large extent, the memory capacity of parallel machines become insufficient to contain all the required computational data, and I/O becomes crucial. Thus, a system with limited I/O capacity can severely constrain the performance of the entire program. This problem has become critical enough that most parallel computers now provide some measure of support for parallel I/O.

We provide experimental results, performed on an Intel Touchstone Delta and nCUBE 2 I/O system, to show that the performance of existing parallel I/O systems can vary by several orders of magnitude as a function of the data access pattern of the parallel program. We then propose a two-phase access strategy, to be implemented in a runtime system, in which the data distribution on computational nodes is decoupled from storage distribution. In the first phase, data is accessed based on the conforming distribution from the I/O system; subsequently, in phase 2, it is redistributed at run-time. Our experimental results show that performance improvements of several orders of magnitude over direct access based data distribution methods can be obtained, and that performance for most data access patterns can be improved to within a factor of 2 of the best performance. Further,the cost of redistribution is a very small fraction of the overall access cost.

Keywords: parallel I/O, parallel architecture, data distribution, operating system support, performance evaluation.

1. Introduction

In recent years, two important events have occurred in the area of high-performance computing: the development of very high speed processors, and the development of massively parallel computers based on these processors. The processing capacities of these parallel computers have made them the computational instrument of choice in the scientific community and they can be found in one form or another in almost every major academic and research institution. Some of the commercially available parallel computers include Intel Paragon [INTE92], nCUBE [NCUB92], CM-5 [THIN91], and Kendall Square [KSR92]. It is anticipated that within the next few years, parallel computers will be capable of computational rates in the Teraflops range [HPCC91].

[*] This work was supported in part by NSF under CCR-9110812, DARPA under contract # DABT63-91-C-0028 and by NSF MIP-9110810. The contents of this paper do not necessarily reflect the position or the policy of the Government and no official endorsement shoud be inferred.

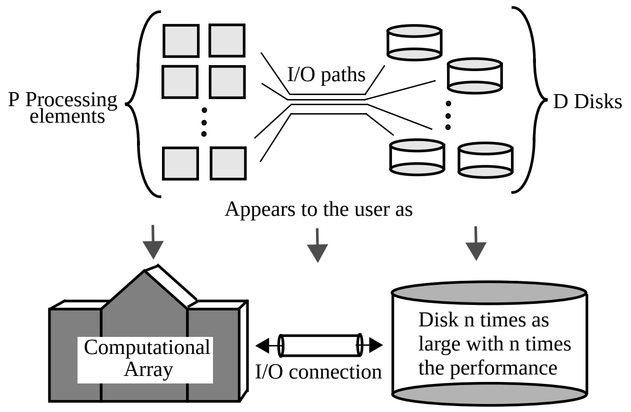

Figure 1: Scalability in computation and I/O

1.1 The I/O Bottleneck

Compared with the attention accorded to processors, interconnection networks and memories of these parallel systems, very little attention has been paid to the scalability and performance of the I/O system. However, 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 crucial. Thus, a system with limited I/O capacity can severely constrain the performance of the entire program - this is known as the I/O bottleneck problem. This problem has become critical enough that most parallel computers such as the Intel iPSC/2 [FREN93], Intel iPSC/860 [PIER89], Intel Touchstone Delta [INTE92], and the nCUBE [DEBE91] now provide some measure of support for parallel I/O.

The goal of parallel I/O is to provide a bottle-neck 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 must exploit the scalability of the hardware and provide increased performance by declustering data across the disk array (a technique called striping) thereby distributing the access workload over multiple servers.

In scientific programs, users commonly exploit parallelism by using data decomposition to specify a partitioning of the data across the set of processors. This establishes a mapping between the logical file data and each processor in the processor array. The data access pattern for parallel I/O, and its complexity, is defined by the level of correspondence between the data distribution and data declustering over the disks. The extent to which the I/O bottleneck exists for a given program is greatly determined by the complexity of this data access pattern.[†]

1.2 Contribution of the Paper

In this paper we limit ourselves to the Intel Touchstone Delta file system, called Concurrent File System (CFS),

[†]. Parallel file systems vary in their level of support for data distribution mappings; some provide no support whatsoever. In these cases, a programmer must be aware of the file mapping to the disk array and must explicitly remap the data from the data distribution to the disk declustering mapping. That is, the programmer must manage the access pattern of each processor through an indexing of the file pointer.

and the nCUBE-2 parallel file system. We show that file system performance can vary greatly as a function of the selected data distribution. Further, that parallel I/O for certain common data decomposition patterns can 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. The basic idea behind the two-phase access strategy is to first perform the I/O with a processor mapping which conforms with the file mapping over the disks. In a subsequent phase, the data is redistributed to match the user selected data distribution mapping. This strategy effectively decouples the user selected data distribution mapping from the file mapping to the disks (i.e., the declustering mapping). The effects of this are that the actual data access phase always gives good performance independent of the user selected mapping, and that the higher degree of connectivity available in the processor array is exploited more effectively for the distribution of data.

1.3 Outline

We first discuss a parallel programming model for a typical parallel computer, the parallel I/O mapping problem, and the system performance associated with a parallel I/O system; this is presented in section 2. In section 3, we consider several data decomposition strategies and evaluate the performance of the I/O system based upon direct access (i.e., the access strategy used by a typical parallel program on the basis of programmer specified data distribution). The performance is obtained for the Intel Touchstone Delta [INTE92] and the nCUBE-2 systems [NCUB92], and we observe a great disparity in performance as a function of data decomposition. In section 4, we propose a parallel I/O strategy that tremendously improves the overall performance for a large number of data distributions. Again, using experiments on the Intel Touchstone Delta and nCUBE-2, we illustrate the obtained improvements in performance. Finally, conclusions and future work are presented in section 5.

2. Programming and System Performance Model

In this section, we briefly discuss our program, system, and performance models for parallel I/O. We describe the problems that arise from the formulation of a data access strategy based upon the mappings for data distribution and data declustering over the disk array - the mapping problem.

2.1 Programming Model

For this paper, we focus on parallel I/O for parallel programs using the Single Program Multiple Data (SPMD) programming paradigm for MIMD machines. The SPMD programming model is the most widely used model for large-scale scientific and engineering applications. In such applications, parallelism is exploited by decomposing the input data. To perform load-balancing, express locality of access, reduce communications, and other optimizations, several decompositions and data alignment strategies are often used (e.g., block, cyclic, along rows, columns, etc.).

In order to enable a user to specify such decompositions, Fortran D [FOX90], and subsequently High-Performance Fortran (HPF) [HPFF92], 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. It is important to note that the directives for specifying domain decomposition may be used with any other language (e.g., C, C++, Ada, etc.). Figure 2 provides a simple illustration of some data distributions (for four processors) that can be easily specified in Fortran D or HPF.

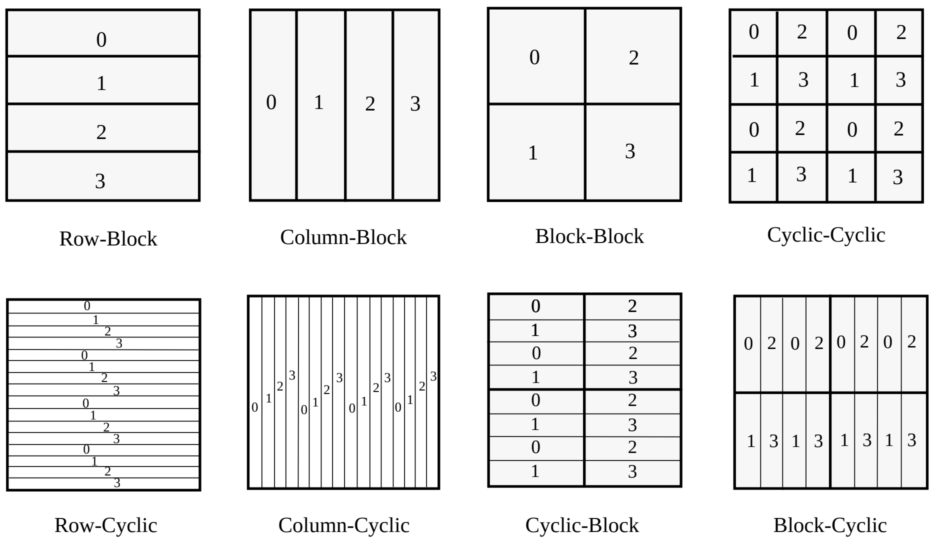

Figure 2: Fortran D/HPF data distribution examples

2.2 The Mapping Problem

Language extensions provide us with a means of representing data distribution information in a way that closely matches the underlying computation thereby simplifying the programming. However, provisions for language support which will allow similar specifications to be made with I/O expressions - mapping a distributed file to the computational array - have not been sufficiently addressed. As a result, it is difficult and sometimes impossible to perform such a parallel I/O mapping in a manner that results in optimal performance.

In order to perform such a mapping from distributed file to processor array, we note that two mappings have to be considered. Most parallel I/O subsystems today provide some sort of data declustering over sets of disks [INTE92, NCUB92, THIN91]. The distribution organization of the file data over the set of disks represents the first mapping, M1. For any system, this mapping is determined by the system configuration setup. 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 [DELR92] 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.

As a simple example, consider a weather modeling program which reads satellite information from a file that is distributed on a set of disks. Assume that the satellite information is stored as position associated data on a rectangular grid region covering the geographical area of interest. Further, assume that the data is distributed across the disk array according to grid column. Thus the M1 mapping is a column mapping from logical data grid positions to the physical disks. Suppose that the weather modeling program requires that the data be read in according to rows of the grid region. This represents the M2 mapping, which is a row mapping from the data to the computational array. On existing I/O systems, these mappings will be resolved into an access strategy which requires that each computational element read numerous small sections of data from the disks in order to construct its allocated row in local memory.

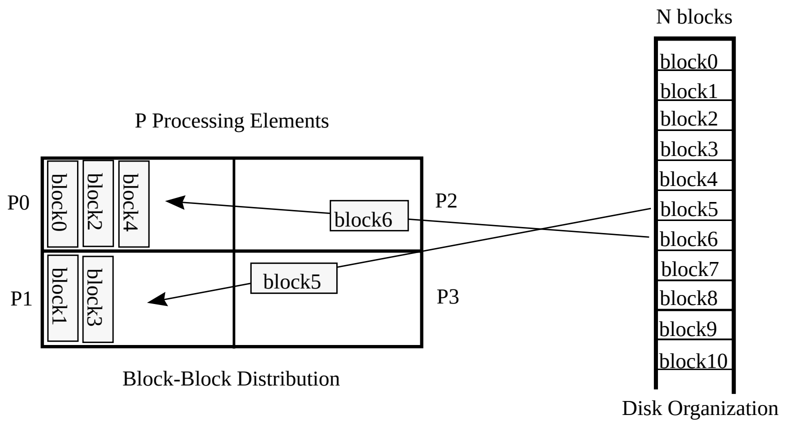

Figure 3: CFS access for Block-Block distribution with column-major disk storage pattern

The enormous costs associated with such a direct access strategy mapping is discussed in the next section and is illustrated in figure 4. Later, 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 3. 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 application making it tedious to program.

2.3 Overall System Performance

The performance measure of a particular configuration for parallel I/O must be based upon an evaluation of its potential for data movement. This data transfer rate is dependent upon the communication bandwidth of each processing element and I/O device, and on the aggregate start-up latency for the complete transfer. Since the cost of data access is dominated by per message start-up 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). Further, the number of requests per transaction can be computed given the data distribution, the number of processors, and the stripe size (where the stripe is the unit of declustering of data across the disks, and stripe size is just its size in bytes). Figure 4 illustrates the dependence of the number of requests on data distribution; we see that a Row-Cyclic distribution generates much many more requests than a Row-Block distribution.

Table 1 shows R_{dist} and S_{dist} for an NxN 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. This can be easily generalized to n-dimensional arrays, but for purposes of this paper, we will only consider up to two dimensions. If data is stored in a row-major one-dimensional map over the disks, then the same results apply by switching column with row and vice-versa.

If we let R_{trans} represent the number of requests per transaction when contributions from the stripe size, S_{stripe}, are considered, then the difference between R_{trans} and R_{dist} arises because the stripe size affects the request size. Whereas in table 1 we assumed that the request size was equal to S_{dist}, now it is represented by the term on the right of R_{dist} in

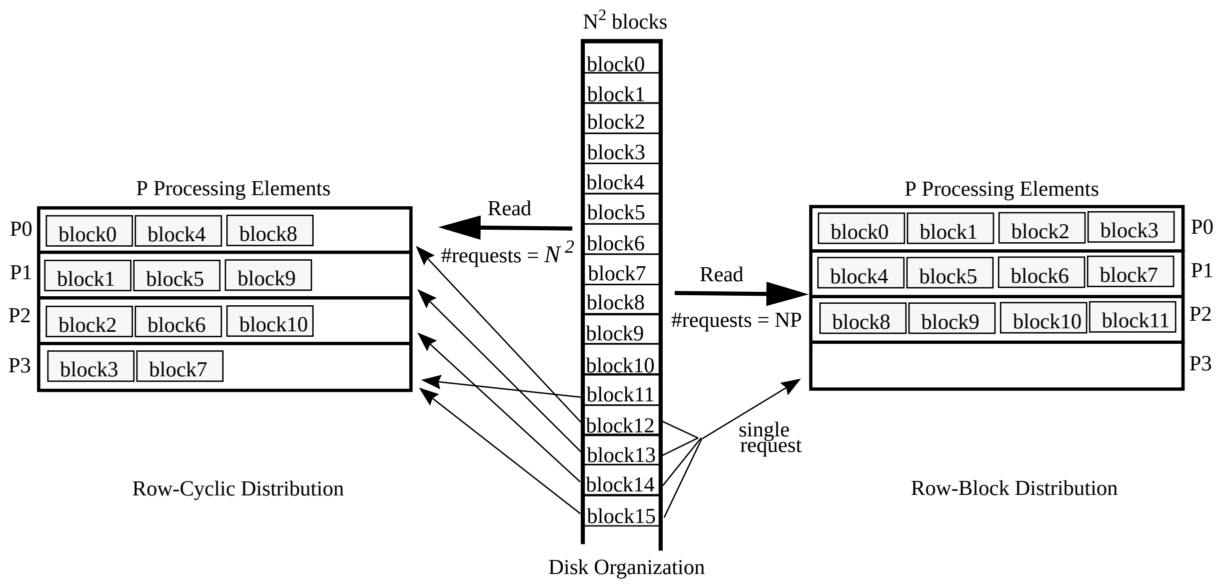

Figure 4: Effects of distribution upon number of requests

equation (1) below. What this term states is that the request size is determined by the minimum of the request size, when only distribution is considered (i.e., S_{dist}), and the stripe size S_{stripe}.

Table 1: Number of I/O requests as a function of data distribution: R_{dist}

1-D distribution			**2-D distribution**		
Distribution type	R_{dist}	S_{dist}	R_{dist}	S_{dist}	Distribution type
Column-Block	P	N^2 / P	$N x \sqrt{P}$	$N / \sqrt{P}$	Block-Block
Column-Cyclic	N	N	$N x \sqrt{P}$	$N / \sqrt{P}$	Block-Cyclic
Row-Block	$N x P$	N / P	N^2	1	Cyclic-Block
Row-Cyclic	N^2	1	N^2	1	Cyclic-Cyclic

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

$$R_{trans} = R_{dist} \times (S_{dist} / \text{MIN} (S_{dist}, S_{stripe})) \tag{1}$$

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

For the preceding discussion, caching is assumed to be absent within the computational array. Also, it is assumed for simplicity that either S_{stripe} divides S_{dist} or vice-versa (i.e., GCD(S_{stripe}, S_{dist}) = MIN(S_{stripe}, S_{dist})).

Consider as an example a 4 processor program reading a 1MB file from a set of disks. Assume that the selected distribution is one-dimensional Column-Block and that the file is striped across the disks in column-major fashion. We see from the first row of table 1 that, as a function of this decomposition, the number of requests will be R_{dist} = P = 4, and the size of each request will be S_{dist} = 1MB / 4 = 256K. With a stripe size of S_{stripe} = 512K, equation (1) shows that $R_{trans} = R_{dist}$ and each processor will require only 1 request (nodes 0 & 1 from the first disk, and nodes 2 & 3 from the second disk). However, if the stripe size was S_{stripe} = 128K, then we see that $R_{trans} = R_{dist}$ x (256K/128K) = 2 x Rdist. Thus, each processor will need 2 requests (e.g., node 0 will take its first 128K from disk 0 and its second 128K from disk1).

3. Performance of Distribution based Direct Access

In this section we present performance results for direct data access based on the data distribution specified within the computational array. Experimental results are presented for the Intel Touchstone Delta and nCUBE-2.

3.1 Intel Touchstone Delta

In this experiment, the mesh size was varied from 4 processors to 512 processors; 64 disks were used. While the mesh size was held constant, the 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[†]. The 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. Note that on the Intel Touchstone delta, the stripe size and storage distribution over the disks (i.e., round-robin) cannot be controlled from a user level program. Since a file is a one-dimensional map of the array, the file is stored in column-major order. The following subsections describe the performance of data access by the computational array for various data distributions.

The CFS on the Touchstone Delta supports several modes of operation, each one determining a degree of synchronization and sharing of file pointers. For our experiments, we restrict ourselves to mode 3 since this gives the best results for direct access. Mode 3 can be characterized as providing a shared file pointer, and synchronized access with fixed length records per access. The synchronization requires that all the processors access data at once.

3.1.1 Column-Block Distribution

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 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. The time required to distribute the data column-wise scales with the number of processors for a portion of the configuration space.

Table 2 contains the data for a 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 size 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, and the need for synchronization.

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. Performance

[†]. large files cannot be read into smaller computational arrays because the total memory required to store the data on nodes is smaller than the file size.

for the small request case (i.e., 1Kx1K) was poor.

Table 2: Column-Block distribution

Array size	Mesh Size	Time (ms)	Rate MBytes/sec
1Kx1K	4	431	2.32
4Kx4K	4	2277	7.05
5Kx5K	16	3357	7.44
5Kx5K	64	3324	7.52
10Kx10K	256	13707	7.65
20Kx20K	512	70953	5.63

3.1.2 Column-Cyclic Distribution

Table 3 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 1) because each processor must make an individual request for each column. This degrades the access time and the bandwidth as illustrated in table 3. The degradation in performance is consistent for all configurations and it ranges between a factor of 2 to 10 as compared to that for Column-Block distribution.

Table 3: Column-Cyclic distribution

Array size	Mesh Size	Time (ms)	Rate MBytes/sec
1Kx1K	4	4353	0.23
4Kx4K	16	5233	3.06
5Kx5K	64	11407	2.19
10Kx10K	256	116763	0.86
20Kx20K	512	252980	1.58

3.1.3 Row-Block Distributions

Table 4 shows the performance for reading the data array and decomposing it in Row-Block fashion over the processor array. Since the one-dimensional map of the file on the concurrent file system is in column major order, this read operation essentially requires a transposition of the data while it is being read. As shown in table 1, the number of logical requests is NxP. Hence, as observed from table 4, the performance degradation due to the decomposition 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 1Kx1K case) for Column-Block decomposition of table 2 above. Further, the 1Kx1K case for this distribution is 39 times slower than for the equivalent Column-Block case.

Table 4: Row-Block distribution

Array size	Mesh Size	Time (ms)	Rate MBytes/sec
1Kx1K	4	17051	0.06
2Kx2K	4	25966	0.15
4Kx4K	16	71205	0.22
5Kx5K	16	91536	0.28
5Kx5K	64	38018	0.69

3.1.4 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.

3.2 nCUBE

For this experiment, the nCUBE-2 Model 10 was used. The nCUBE-2 Model 10 architecture has a hypercube interconnection network backplane which can be populated by up to 1024 computational processors. Up to 8 16-channel I/O boards can be attached to each side of the computational array, each channel containing an I/O node with a connection for a string of disks. The system used for this experiment had 256 compute processors and 8 disks, each on a separate I/O node.

The effects of distribution on the performance of the nCUBE parallel I/O system was measured by running a series of experiments using 16 and 64 processor subcubes, and 8 disks. Read times for 16MB and 64MB arrays were measured. Four array distributions were used: Row-Block, Row-Cyclic, Column-Block, Column-Cyclic. Also, the effects of stripe size on data access were measured by varying the stripe size for read configurations where subcube size, file size, and array distribution remained fixed. On the nCUBE, the stripe size and declustering strategy (i.e., data distribution over the I/O disks) can be manipulated at the application level. However, to limit the complexity of the experiments, a row-major striping was selected. Therefore, all the results of table 1 apply here by switching “row” to “column” and vice-versa.

For all distribution measurements, the best results were used regardless of stripe size (i.e., the stripe size is not fixed for a given distribution, and the stripe size that afforded the best performance was used in each case). To a large extent, this enabled us to decouple the effects that arise due to stripe size, thus allowing the study of the effects of data decomposition strategies on the performance of the I/O system.

3.2.1 Row-Block Distribution

On the nCUBE, the Row-Block distribution often provides the best performance because, compared with other data distributions, it coincides best with the distribution strategy of data over the disks. The data in table 5 shows the transfer rate increasing and read time decreasing with subcube size for a given data set.

Table 5: Row-Block distribution

Array size	Subcube size	Time (ms)	Rate MBytes/sec
4Kx4K	16	5140	3.26
4Kx4K	64	4290	3.91
8Kx8K	64	17700	3.79

3.2.2 Column-Block Distribution

The Column-Block distribution requests, in effect, that a transpose of the array be performed as it is read. The data size per request is smaller and so the number of requests larger than for the Row-Block case; this fact is made evident by the much slower read times shown in table 6. Here, read times increase by over a factor of 36, 50, and 33 times respectively over the Row-Block distribution cases.

Table 6: Column-Block distribution

Array size	Subcube size	Time (ms)	Rate MBytes/sec
4Kx4K	16	190000	0.09
4Kx4K	64	215000	0.08
8Kx8K	64	593000	0.11

3.2.3 Row-Cyclic Distribution

The Row-Cyclic distribution is a median between Column-Block and Row-Block with respect to the number of requests required to complete a transaction; this results in correspondingly median read times as reflected in table 7.

Table 7: Row-Cyclic distribution

Array size	Subcube size	Time (ms)	Rate MBytes/sec
4Kx4K	16	8290	2.02
4Kx4K	64	9080	1.85
8Kx8K	64	24400	2.75

The increase in read time in going from 16 to 64 processors for a 4Kx4K file size (see tables 6 & 7) is further evidence of this behavior. However, the transfer time for each processor has also decreased since the processor data size has decreased. This indicates that, for the present configuration, the cost of an increase in the number of requests has overridden the expected gains from increased bandwidth. This is a significant result in that, for certain cases, it may be more efficient to allow a subset of the computational array to perform the I/O, subsequently redistributing the data over the entire set of processors in the array.

```
/*
 * Code to read satellite data and distribute it over the computational array.
 * /

 void ReadSatelliteData () {
        char *buf;
        int nbytes;
        . . .
        read (fd, buf, nbytes, ROW_BLOCK);
 }
```

Figure 5: Read algorithm example

4. Two-Phase Access Strategy

In this section, we present a two-phase access strategy for conducting parallel I/O. We are motivated by the following observations from our experimental results: for all cases, performance fluctuates greatly with varying data distributions (i.e., logical request size), performance degradation by a factor of 20 or more is commonly observed; the bandwidth for a given configuration is highly dependent upon the file size; lastly, stripe size dependent factors (e.g., load-balance, request size) cause widely divergent read times for most distributions.

4.1 Strategy Description

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). Subsequently, in phase two, we redistribute the data at runtime to match the application's desired data distribution (i.e., from M2' to M2). Figure 5 illustrates what a read segment might look like.

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.

4.2 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; this is derived by using 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

We show that the Total Read times are less than the Direct Read times for all the cases tested. Furthermore, the variation in the Total Read time across distributions is at most a factor of 2 of the Best Read times for the cases tested.

4.2.1 Intel Touchstone Delta

Table 8 shows timing data for 5Kx5K and 10Kx10K files read and distributed over 16 processors. The Best Read time is for the Column-Block distribution. For all cases below, the '*' symbol denotes a read time on the order of

Table 8: Redistribution on 16 processors

Distribution		Column Block	Column Cyclic	Row Block	Row Cyclic
Best Read (a)	5Kx5K	3357	3357	3357	3357
	10Kx10K	10376	10376	10376	10376
Redistribute (b)	5Kx5K	-	1805	673	2603
	10Kx10K	-	7105	2772	10320
Total Read (c) = (a)+(b)	5Kx5K	3357	5162	4030	5960
	10Kx10K	10376	17481	13148	20696
Direct Read (d)	5Kx5K	3357	9890	69939	*
	10Kx10K	10376	19271	84683	*
Speedup (d) / (c)	5Kx5K	1	1.92	17.36	> 604.03
	10Kx10K	1	1.10	6.44	>173.95

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, the amount of overhead avoided by using the redistribution strategy ranges from 1.7 secs, to well over 60 minutes for the 5K Row-Cyclic case; the deviation in Total Read time is at most a factor of 1.9.

Table 9 shows timing data for 5Kx5K and 10Kx10K files read and distributed over 64 processors. The cost of reading is reduced from 7.4 secs, to over 60 minutes for the 5Kx5K Row-Cyclic case. The variation in Total Read time is at most a factor of 1.27.

Table 9: Redistribution on 64 processors

Distribution		Column Block	Column Cyclic	Row Block	Row Cyclic
Best Read (a)	5Kx5K	3324	3324	3324	3324
	10Kx10K	11395	11395	11395	11395
Redistribute (b)	5Kx5K	-	703	246	768
	10Kx10K	-	2478	1028	3092
Total Read (a)+(b)	5Kx5K	3324	4027	3570	4092
	10Kx10K	11395	13873	11623	14487
Direct Read (d)	5Kx5K	3324	11407	38018	*
	10Kx10K	11395	63400	78767	*
Speedup (d) / (c)	5Kx5K	1	2.83	10.65	>879.77
	10Kx10K	1	4.57	6.78	>248.50

4.2.2 nCUBE

Table 10 shows timing data for 4Kx4K files read and distributed over 16 processors with varying decompositions. The Best Read time is for the Row-Block distribution which corresponds with the row-major striping over the disks.

For all cases below, the ‘*’ symbol denotes a read time on the order of an hour. A comparison of the Total Read times with Direct Read times show that the performance improvements obtained by using the redistribution strategy range from 2.5 secs, to up to 35 minutes for the 4Kx4K Column-Cyclic case, and the deviation in Total Read time is at most a factor of 2.8.

Table 10: Redistribution on 16 processors

Distribution		Row Block	Row Cyclic	Column Block	Column Cyclic
Best Read (a)	4Kx4K	5140	5140	5140	5140
Redistribute (b)	4Kx4K	-	603	615	9700
Total Read (c) = (a)+(b)	4Kx4K	5140	5743	5755	14840
Direct Read (d)	4Kx4K	5140	8290	190000	*
Speedup (d) / (c)	4Kx4K	1	1.44	33.02	>242.59

Table 11 presents timing data for 4Kx4K and 8Kx8K files read and distributed over 64 processors for varying decompositions. The Best Read time is again for the Row-Block distribution. By comparing the Total Read times with Direct Read times, we see that the redistribution strategy improves performance be 4.6 secs, to over 59 minutes for the 4Kx4K Column-Cyclic case, and the Total Read time deviates by at most a factor of 1.8

Table 11: Redistribution on 64 processors

Distribution		Row Block	Row Cyclic	Column Block	Column Cyclic
Best Read (a)	4Kx4K 8Kx8K	4290 17700	4290 17700	4290 17700	4290 17700
Redistribute (b)	4Kx4K 8Kx8K	- -	203 780	242 877	2500 9980
Total Read (c) = (a)+(b)	4Kx4K 8Kx8K	4290 17700	4493 15480	4532 18577	6790 27680
Direct Read (d)	4Kx4K 8Kx8K	4290 17700	9080 24400	215000 593000	* *
Speedup (d) / (c)	4Kx4K 8Kx8K	1 1	2.02 1.58	47.44 31.92	>530.19 >130.06

Table 12 presents timing data for 4Kx4K and 8Kx8K arrays redistributed from Row-Block distribution on 16 processors to 16, 32, and 64 processors with varying distributions, and its percentage of the cost of Total Read and Direct Read times. We see that the redistribution costs, with few exceptions and although non-optimized algorithms are used, are extremely small relative to Total Read times presented in previous tables; they are even much smaller percentages of the Direct Read times.

Table 12: Redistribution for Row-Block from 16 to 16, 32, and 64 processors

Row-Block to:		Column-Block			Column-Cyclic			Row-Cyclic		
		t msec	% of Tot.	% of Dir.	t msec	% of Tot.	% of Dir.	t msec	% of Tot.	% of Dir.
16 to 16	4Kx4K	615	10.69	3.24	9700	65.36	0.02	603	10.50	7.27
16 to 32	4Kx4K	678	11.65	-NA-	9670	65.29	-NA-	605	10.53	-NA-
16 to 64	4Kx4K	722	12.32	0.36	9640	65.22	0.02	584	10.20	6.43
64 to 64	4Kx4K	242	5.33	0.11	2500	38.52	0.01	203	4.52	2.23
64 to 64	8Kx8K	877	4.72	0.15	9980	36.05	0.02	780	5.03	3.19

4.2.3 Discussion

The results above show that for every case but one, performance is improved to within a factor of 2 of the Total Read performance of the “best” distribution. Upon examination of the one exception (table 10: Column-Cyclic), we see that the predominant cost arises from redistribution. This result is not alarming because the redistribution algorithms used for the experiment were implemented without optimizations which would otherwise have been incorporated into a real system. Thus, optimizing the redistribution algorithms would improve performance for these exceptions so that they lie within the expected range. Further, although redistribution costs for these exceptions were high relative to Total Read times, they were still orders of magnitude less than Direct Read times for all cases, thereby guaranteeing significant performance improvements.

The results also show that speedups range from about 2 for Block distributions to on the order of 100 for Cyclic distributions. This result is really equivalent to the previous result (that performance is within a factor of 2 of the best case). However, it does emphasize that the use of more complex distributions have a high probability of producing performance comparable to that of Block distributions. Two-phase access has effectively decoupled data and storage distributions by introducing a level of indirection into the mapping.

Although write cases have not been addressed in this paper, it is expected that results for writes will be analogous to those for reads.

5. Conclusion and Future Work

The goal of this research has been to develop a strategy for optimizing parallel I/O over the wide variety of possible access patterns. We have demonstrated that, with the two-phase access strategy for parallel I/O, it is possible to obtain significant improvements in performance over previously used methods. Data distribution and storage distribution have been decoupled, enabling the most effective configuration to be used for parallel I/O. Speedups obtained ranged from about 2 for simple Block distributions to orders of 100 for more complex distributions. Among others, worst case times were improved to within a factor of 2 of the best read times. Therefore, performance of the I/O system can be made more consistent over a wide variety of data distributions. Optimizing redistribution algorithms should result in further gains.

Our current task involves the integration of this strategy into a parallel I/O run-time system that will implement this strategy automatically and transparently. Future work includes a more complete characterization of the parallel I/O parameter space, and compiler participation through language extensions and preprocessing of information providing selection assistance.

References

[BORD93] R. Bordawekar, A. Choudhary, J.M. del Rosario, “An Experimental Performance Evaluation of Touchstone Delta Concurrent File System”, *International Conference on Supercomputing*, June, 1993 (submitted). Also available as NPAC Technical Report SCCS-420.

[DEBE91] E. DeBenedictis, P. Madams, “nCUBE’s Parallel I/O with Unix Compatibility”, *Proceedings of the Sixth Distributed Memory Computing Conference*, Portland, Oregon, April, 1991, pp.270-277.

[DELR92] J.M. del Rosario, “High Performance Parallel I/O on the nCUBE 2”, *IEICE transactions*, Japan, August, 1992.

[FOX90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng, M. Y. Wu, “ Fortran D Language Specification”, *Technical Report Rice COMP TR90-141*, Department of Computer Science, Rice University, December 1990.

[FREN93] J.C. French, T.W. Pratt, M. Das, “Performance Measurement of the Concurrent File System of the Intel iPSC/2 Hypercube”, *Journal of Parallel and Distributed Computing*, Jan./Feb. 1993.

[HPCC91] “Grand Challenges: High Performance Computing and Communications”, A Report by the Committee on Physical, Mathematical, and Engineering Sciences, Office of Science and Technology Policy, 1991.

[HPFF92] K. Kennedy (chair), “High Performance Fortran Language Specification, version 0.3”, *CRPC Tech. Report*, High Performance Fortran Forum, Rice University, 1992.

[INTE92] Intel, “A Touchstone Delta System Description”, Intel, Portland, Oregon,1992. Intel Advanced Information.

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

[NCUB92] nCUBE, “nCUBE 2 Systems: Technical Overview”, nCUBE, Foster City, California, 1992.

[PIER89] P. Pierce, “A concurrent file system for a highly parallel mass storage subsystem”, *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 161-6, 1989.

[THIN91] Thinking Machines Corporation, “A CM-5 System Description”, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.