COLLECTIVE PARALLEL I/O

by

WILLIAM J. NITZBERG

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 1995

We attack this problem in three steps: we evaluate an early parallel I/O system, the Intel iPSC/860 Concurrent File System; we design and analyze the performance of two classes of algorithms taking advantage of collective parallel I/O; and we design *MPI-IO*, a collective parallel I/O interface likely to become the standard for portable parallel I/O.

The collective I/O algorithms fall into two broad categories: *data block scheduling* and *collective buffering*. Data block scheduling algorithms attempt to schedule the individual data transfers to minimize resource contention and to optimize for particular hardware characteristics. We develop and evaluate three data block scheduling algorithms: *Grouping*, *Random*, and *Sliding Window*. The data block scheduling algorithms improved performance by as much as a factor of eight. The collective buffering algorithms permute the data before writing or after reading in order to combine small file accesses into large blocks. We design and test a series of four collective buffering algorithms and demonstrate improvement in performance by two orders of magnitude over naive file I/O for the hardest, three-dimensional distributions.

Software Engineer, Perfect Software, Eugene, 1984-85

System Administrator, Department of Physics, University of Oregon, Eugene, 1982-84

AWARD AND HONORS:

Eighth Place, International ACM Scholastic Programming Contest, 1990
Mortar Board/Druids Junior Honor Society Award, 1983-84

GRANTS:

Tektronix Research Fellowship, 1989
Coca Cola Scholarship, 1986-87
William W. Stout Scholarship, 1984-85

PUBLICATIONS:

P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, JP. Prost, M. Snir, B. Traversat and P. Wong. Overview of the MPI-IO parallel I/O interface. In *Proceedings of the IPPS '95 Workshop on I/O in Parallel and Distributed Systems*, pages 1-15, April 1995.

D. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 215-227, April 1995.

J. Krystynak and B. Nitzberg. Performance characteristics of the iPSC/860 and CM-2 I/O systems. In *Proceedings of the 7th IEEE International Parallel Processing Symposium*, pages 837-841, April 1993.

W. Liu, V. Lo, B. Nitzberg, and K. Windisch. Non-contiguous processor allocation algorithms for distributed memory multicomputers. In *Proceedings of Supercomputing '94*, pages 227-236, November 1994.

B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. In *IEEE Computer*, pages 52-60, August 1991.

This would never have happened without the help of Betty Lockwood and Jan Saunders in the front office who got things signed, figured out deadlines, and generally dealt with the administrative details, going far and beyond the call of duty.

Finally, I must give credit to my friends and family who suffered with me through this process. My parents, Jerry and Esther, were unfailingly supportive. My brother Mark offered more help than I was able to take advantage of. Randy Goodall got me my first job with UNIX, running 2.8 BSD on a PDP-11/23. Michelle Koblas, Dave Koblas and Mark VandeWettering gave friendship, support, and entertainment. Chris Sauer was extremely supportive, and gave me an incentive to finish by engaging in her own quest for a higher degree (I win). And, lastly, Anne Urban, whom I owe a debt which must be repaid in large amounts of raw fish, not only encouraged and supported my endeavor, but also read and edited late into the night.

LIST OF FIGURES

the main factors inhibiting broad acceptance of distributed memory parallel machines for scientific computing are I/O performance and portability.

The major factor limiting parallel I/O performance is that the existing parallel I/O systems evolved directly from I/O systems for serial machines. Serial I/O systems are heavily tuned for sequential, large accesses, limited file sharing between processes, and a high degree of both spatial and temporal locality. Recent experience [88, 107, 115] has shown there are significant differences between file I/O on serial and parallel machines, presenting a new arena of challenges for the designers of distributed memory parallel systems.

In the scientific world, a program is not considered truly portable unless it not only compiles, but also runs efficiently. Thus, portability and performance are inseparable. There are two emerging standards for parallel programming: High Performance Fortran (HPF) and the Message Passing Interface (MPI). HPF, a data parallel extension of Fortran 90, includes the Fortran intrinsics for file access, but due to its immaturity, HPF is far from high performance. Implementors of HPF are currently searching for a standard parallel file interface on which to build efficient implementations of these intrinsics [80]. MPI provides a standard interface for message passing, but lacks any support for file operations.

In this work, we propose a solution to both of these problems: a collective parallel I/O interface with efficient algorithms to implement it. The proposed standard collective I/O interface not only provides source code portability, but also supports the unique file access patterns which occur in scientific applications.

These systems vary in size, complexity, software integration, and performance. Distributed memory parallel systems fall into two general classes: workstation clusters and "highly parallel systems". Workstation clusters usually consist of a few tens of UNIX workstations or PCs connected by a commodity network such as Ethernet or FDDI. Each node of a workstation cluster runs its own independent operating system, and add-on software provides facilities for running parallel applications and managing the system as a single computer. Workstation clusters of PCs [26], and Sun and SGI systems [28] have been used for parallel scientific computing with good results. Highly parallel systems are typically larger, with hundreds to thousands of nodes, and consist of more specialized hardware as well as a more integrated operating system which provides the abstraction of a single system image. The use of specialized network hardware on highly parallel systems achieves very low latency and high bandwidth compared with the commodity networking technology available for workstation clusters. Examples of highly parallel systems available today include: the Convex Exemplar [29], the Cray Research T3D [110], the IBM SP series [74], the Intel Paragon [76], the MasPar MP-2 [103], and the Meiko CS-2 [95]. As technology improves, the distinction between the different classes of parallel machine is blurring, with systems such as the IBM SP2 a cross between a workstation cluster and a fully integrated highly parallel system.

Processing nodes are further designated as compute nodes and I/O nodes (see Figure 1). Applications run on the compute nodes. Compute nodes are char-

the speed of the I/O interface. On some systems, such as the Intel Paragon, I/O nodes are simply compute nodes with disks attached, while others, such as the Thinking Machines CM-5, have special purpose I/O nodes, while yet others, such as the IBM SP2, blur the distinction between compute nodes and I/O nodes, and allow any node to act as either a compute node or an I/O node, or both simultaneously.

Storage devices, usually disks, hold quantities of data too large to fit in physical memory for periods of time longer than a single application run. Storage devices are not limited to simple disks. The Intel Paragon uses hardware RAIDs. A storage device might even consist of an entire mass storage system such as UniTree or NAStore [140]. From the parallel system perspective, all storage devices can be adequately characterized by the amount of storage provided, the block size (the smallest amount of data which can be efficiently read or written), the latency (time to transfer a block), and the bandwidth (maximum bytes per second the device can transfer). As most parallel systems simply use disks, the term "disk" is used interchangeably with storage device. A typical parallel system uses SCSI attached disks with a 1-2 gigabyte capacity and a few megabytes per second bandwidth (similar to what one would find on today's PCs).

Interconnection networks provide the only means of transferring information between the nodes of a parallel system. Network technology and performance vary greatly from one parallel machine to the next, from simple commodity (1 megabyte per second Ethernet), to proprietary, buffered, wormhole

for compatibility with existing applications which expect serial files, for ease of migrating files between systems, and because parallel filesystems are often built on top of serial filesystems.

In a parallel I/O system, individual files are partitioned among I/O nodes for performance (see Figure 2). This improves the performance of a single access by allowing multiple disks to operate in parallel. The distribution of file data among I/O nodes and disks is called the "file layout". A file is broken down into "file chunks," each assigned to one I/O node. The most common file layout is a round-robin distribution of the canonical file (in 4 - 64 kilobyte blocks) onto the I/O nodes. This layout, also called "striping" maximizes disk parallelism by allowing large accesses to span multiple I/O nodes, while minimizing the possibility that a set of smaller accesses will contend for a single I/O node.

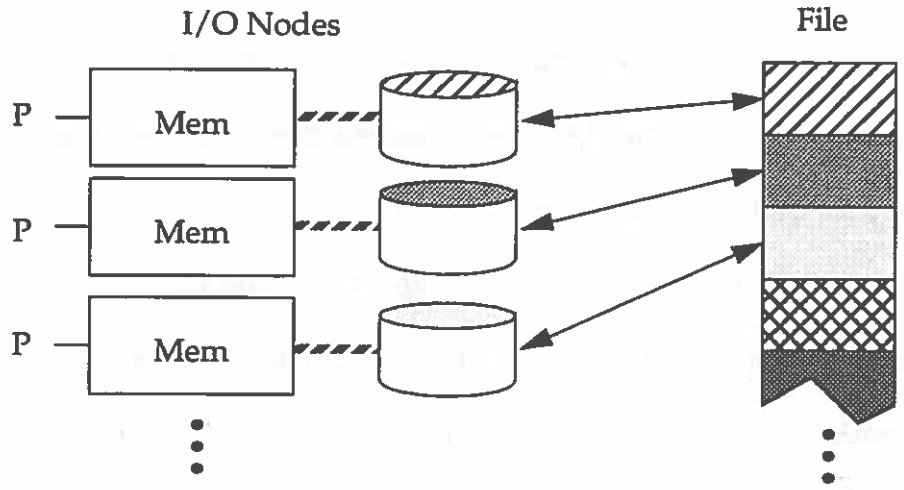

FIGURE 2. Files are partitioned among I/O nodes for improved performance.

mize performance while taking into consideration communication overhead, processor cache utilization, load balance, and other algorithm specific factors.

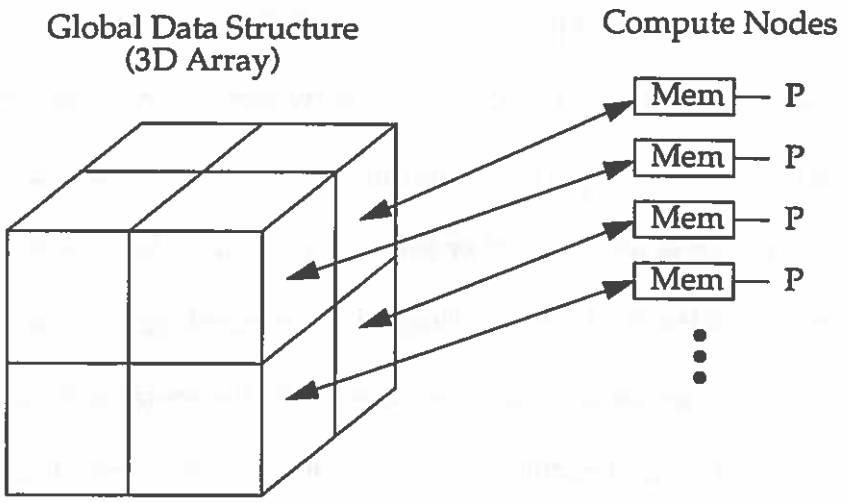

FIGURE 3. Global data structures are distributed among compute nodes for improved performance.

Collective Parallel I/O

Parallel I/O is the process of transferring a global data structure, distributed among compute nodes, to a file striped across storage devices. At the lowest level, a parallel I/O operation can be viewed as a mapping between compute node memories and disk addresses. Since this mapping can be quite complicated and involve a significant amount of data movement, optimization of this mapping with respect to data distribution, file layout, and machine architecture is critical for parallel I/O performance. The programmer has two choices: to hand code

approach the full bandwidth. Disks are notoriously inefficient when accessing less than a full disk block. A write may require a costly read-modify-write cycle, incurring two disk accesses to transfer a small amount of data, and a read cache may be ineffectual when numerous small uncoordinated read requests are issued, requiring each small access to read an entire disk block.

The major problem with the naive algorithm is its uncoordinated nature. However, scientific applications are highly synchronized. The temporal behavior of a generic scientific application is described in Figure 4.

```
Initialize
Repeat:
      Compute
      Communicate
      Perform I/O (read, write, read-write)
      Checkpoint (every few iterations)
Finalize
```

FIGURE 4. Temporal behavior of a generic scientific application.

During each cycle, the individual processes of the parallel application are nearly synchronized, and all processes start performing their respective I/O accesses more or less at the same time. The uncoordinated nature of the I/O operations in these applications is not inherent to the application, nor is it inherent to parallel I/O—instead, it is an artifact of the poor file I/O interface.

A collective I/O interface has the potential to solve both performance and portability problems. *Collective I/O* is that in which both compute nodes and I/O nodes participate in a coordinated way utilizing global knowledge of the I/O

scientific world requires source code which both compiles and runs fast. The interface and algorithms developed in this work satisfy both needs simultaneously.

The Parallel I/O Behavior of Scientific Applications

Supporting high performance I/O on traditional computing systems used for scientific processing (from workstations to vector supercomputers) is largely a solved problem. I/O behavior on these systems has been well studied [97]: files are large (megabytes to gigabytes); file accesses are large; files are accessed completely; and files are accessed sequentially. These characteristics lead to efficient caching and prefetching, allowing the I/O system to run at near peak performance. Until recently it was believed that the I/O behavior of parallel scientific applications was identical to that of "serial" scientific applications, but no studies existed to verify this belief.

CHARISMA

The CHARISMA project was initiated in June 1993 to CHARacterize I/O in Scientific Multiprocessor Applications. CHARISMA is the first (and to date, the only) project to look at real scientific workloads on parallel systems. Two studies have been so far been published: one on the Intel iPSC/860 at NASA Ames Research Center [88], and one on the Thinking Machines CM-5 at the National Center for Supercomputing Applications [115].

The CHARISMA studies show two significant differences between traditional supercomputer scientific applications and parallel scientific applications:

analysis, write mostly for data generation, and read/write for out-of-core solvers. The details of the I/O behavior are determined by how data structures are stored in memory, and how I/O accesses are specified using a particular interface or library. We restrict our attention to the large class of data generation applications, typified by computational fluid dynamics (CFD) applications. Using an example algorithm, we show how the data structures for a parallel application are inherently different from the data structures used for the serial version of the same algorithm.

The NAS Parallel Benchmarks have become the de-facto standard for measuring the performance of parallel computers [5]. The benchmarks are representative of the algorithms used for computational fluid dynamics (e.g. simulating airflow over a wing). They are "pencil and paper" benchmarks, specifying algorithms only, and not implementations. In this way, implementors are free to tune for architectural idiosyncracies, and the performance measured will reflect the "best" performance of the algorithm on a machine, not the performance of a prespecified generic implementation. This eliminates the problem of the benchmark inadvertently favoring a particular architecture.

The Scalar Pentadiagonal (SP) and Block Tridiagonal (BT) pseudo-application benchmarks represent the two common methods of numerically simulating high speed compressible airflow. The area to be simulated is represented as a three-dimensional grid of five components (velocity in x, y, and z, density, and temperature). Time is simulated in discrete steps—one iteration of the algorithm

On a distributed memory parallel computer, there are three common methods of implementing these algorithms, differentiated by how each method distributes the three-dimensional grid among processes [135, 136]:

1. Pipelined Gaussian Elimination
2. Dynamic Block-Cartesian Decomposition
3. Multi-partition

The data is distributed to maximize load balance and minimize communication overhead—hopefully this results in maximum performance. However, it is the distribution of the data structure which makes the I/O inherently more difficult. We will see in "Parallel I/O: The Low-level Perspective" on page 34 why these data distributions lead to the small non-sequential accesses found in the CHARISMA studies.

In the following discussion of these algorithms, let P be the number of processes.

Pipelined Gaussian Elimination

The Pipelined Gaussian Elimination method breaks the three-dimensional grid into P chunks, one for each processor. Consider Figure 5. In this example, there are 8 processors, and the grid size is N x N x N. The three-dimensional grid is cut in half in each dimension, yielding 8 smaller cubes. During a traversal in the x dimension, nodes 1, 3, 5, and 8 start computing the information for their grid points, while nodes 2, 4, 6, and 7 wait. As soon as node 1 finishes solving a line of

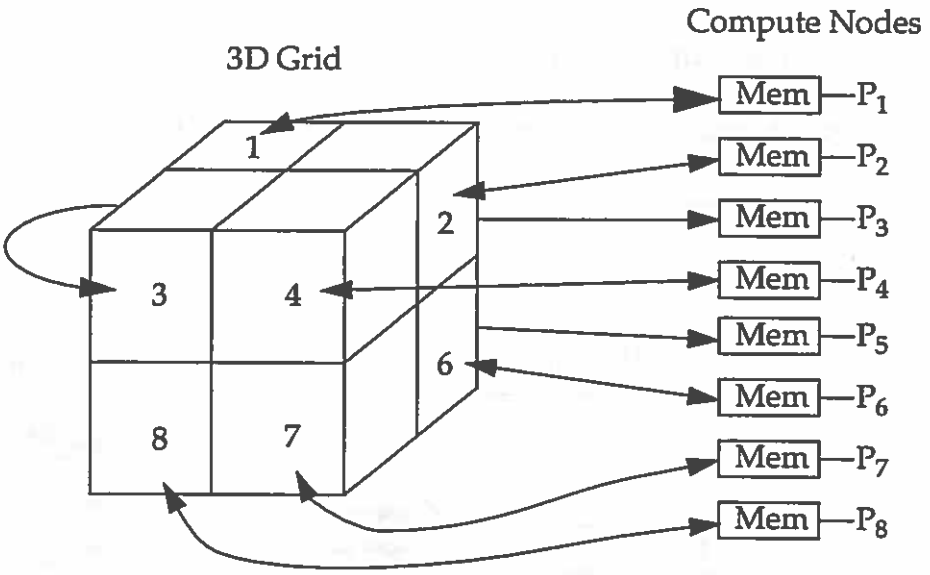

FIGURE 5. Pipelined Gaussian Elimination method on 8 processes: the 3D grid is partitioned into 8 cubes and one cube is assigned to each process.

Dynamic Block-Cartesian Decomposition

The Dynamic Block-Cartesian method slices the three-dimensional grid into P planes, one for each compute node (see Figure 6). Rather than pass boundary values, the Dynamic Block-Cartesian method ensures that every traversal can proceed without any communication. Initially, the grid is sliced into x-y planes, allowing the first two traversals to be performed without any communication. After these two traversals, the entire grid is redistributed into x-z planes, allowing the z dimension to be traversed, followed by the x dimension of the next time step. Finally, another redistribution into y-z planes allows the last two traversals of the next time step to be completed. A total of three complete transpose operations must be performed for every 6 traversals (2 time steps). Each transposition

each compute node is given $\sqrt{P}$ sub-cubes, arranged so that for any plane of sub-cubes, each compute node gets exactly one sub-cube of data from that plane (see Figure 7). This means that all compute nodes operate on each plane of sub-cubes. Although similar to pipelined Gaussian elimination, this scheme enables all nodes to be active at all times, without waiting for a pipeline fill. Each traversal requires $(\sqrt{P}-1)N^2$ grid points to be transferred between nodes in a total of $(\sqrt{P}-1)P$ messages. The multi-partition method requires P to be a perfect square.

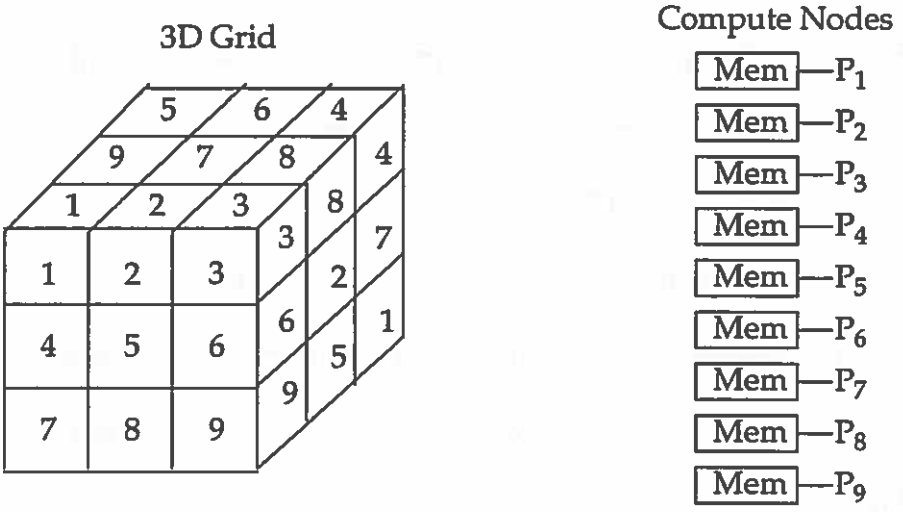

FIGURE 7. Multi-partition method onto 9 compute nodes; the three-dimensional grid is broken into 27 ($\sqrt{9}^3$) sub-cubes arranged for optimal load balance (all compute nodes operate on each plane of sub-cubes, maximizing parallelism).

The Multi-partition scheme provides good load balancing and coarse-grain communication, and performs perhaps the best across a wide range of parallel architectures [3, 136].

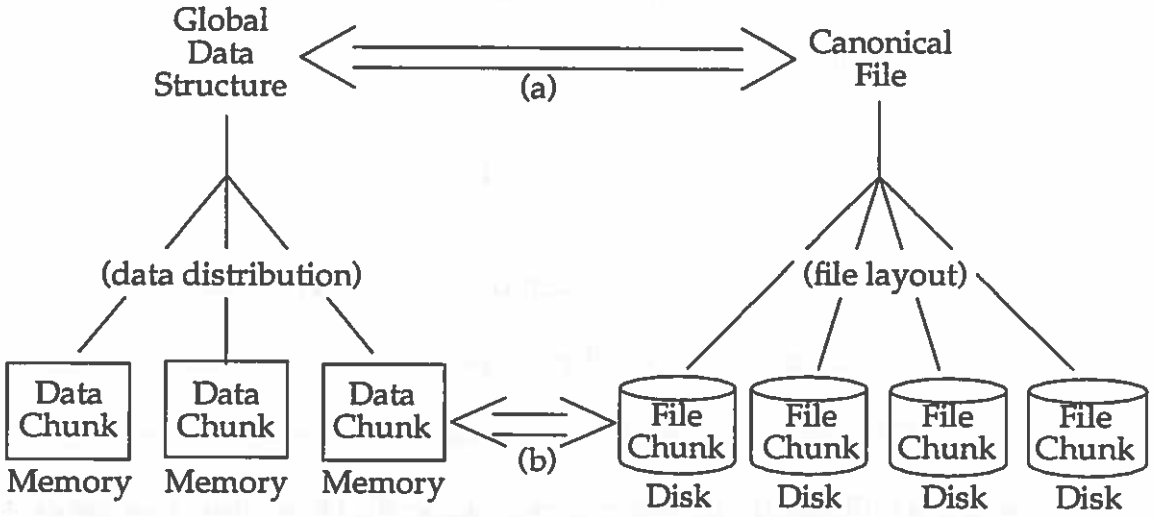

FIGURE 8. Parallel I/O is a mapping problem which can be viewed from (a) the global perspective and (b) the low-level physical perspective.

Global Perspective

At the highest level, a read (or write) operation simply copies a global data structure from a file into memory (or from memory into a file). Both the global data structure and the file are logical objects. We define the following terms which will be used throughout this dissertation. The *global data structure* is the logical view of the data from the application's point of view. The global data structure is distributed (*data distribution*) among compute node memories by cutting it into *data chunks* and storing each chunk in a separate compute node's memory. The file represents a linearization of the global data structure, such as the row-major ordering of a three-dimensional array. We call this linearization the *canonical file.* The canonical file is distributed among the storage devices attached to I/O nodes. We use the term *file layout* to describe the distributing or partitioning of a file into

ture into many small pieces (specified as the distribution size or block size) and deals these pieces out to the P compute nodes in a round-robin fashion until all pieces have been dealt. Both HPF BLOCK and HPF CYCLIC specify one-dimensional distributions (Figure 9). To express more than a one-dimensional distribution, each dimension is independently distributed using one of the HPF distribution directives (Figure 10).

Figure 10 shows how data from an 8x8 array is distributed onto 4 processes. The figure shows which data elements are assigned to which processes by process number. For example, the upper left 4x4 square is assigned to process 1 for the HPF (BLOCK, BLOCK) distribution. We use "*" to designate that a particular dimension is not distributed.

uses a combination of (BLOCK, BLOCK, BLOCK) and CYCLIC. In addition, there are a few simple distribution patterns which are inexpressible using HPF directives. For example, a random pattern, where data is distributed to nodes randomly, and the broadcast read/write reduce patterns, where all data is distributed to every node.

ARC3D is a parallelized scalar pentadiagonal time-stepping flow solver, which uses pipelined Gaussian elimination. The ARC3D distribution is hand-tuned to avoid a load balancing problem exhibited by the HPF BLOCK distribution. A straight HPF BLOCK distribution forces all nodes (except the last few) to receive the same amount of data. Although this makes the job of an HPF compiler easier, when the number of compute nodes does not divide the data size evenly, a serious load imbalance may result. For example, a 25 element array distributed via HPF BLOCK onto eight nodes leaves two nodes nearly empty (Figure 11). For a three-dimensional array, this load imbalance is further magnified: a 25 x 25 x 25 element array distributed using HPF (BLOCK, BLOCK, BLOCK) onto 8 x 8 x 8 nodes leaves 296 out of 512 empty or nearly empty. ARC3D's distribution algorithm avoids this problem, giving each node approximately the same number of grid points.

distribution is hard for an I/O system to optimize, because it lacks any regular pattern.

File Layout

The key to performance is maximizing parallelism. In a scientific application, better computational performance is attained when clever data distributions minimize communication overhead and lead to good load balance and increased parallelism. For application I/O, parallelism is exploited by distributing files across multiple storage devices located throughout the system. File layout is the partitioning of a file into file chunks, and the distribution of these chunks to storage devices. A file chunk is the portion of a file assigned to any single storage node.

File layout is another form of data distribution. Here, the global data structure is the file, and the file is distributed among storage devices rather than compute node memories. Clever file layouts lead to good load balance within the I/O subsystem and increased I/O performance. Spreading file chunks out among storage devices improves performance by allowing multiple chunks to be accessed from different storage devices simultaneously. Distributing the file chunks can also reduce contention for specific storage devices.

The canonical file, a linearization of the global data structure, is a one-dimensional sequence of bytes (the standard UNIX byte stream). This definition of the canonical file stems from the fact that current parallel I/O systems evolved

by multiple I/O nodes. In general, file block size is chosen to be a small multiple of disk block size, large enough to minimize seek overhead somewhat, but small enough to provide ample parallelism.

All currently available parallel file systems use an HPF CYCLIC distribution of the canonical file (in file blocks) onto disks (see Figure 12). This file layout is also called *striping*, and the file block size is referred to as the *striping unit*. For example, the default file layout on the IBM SP2 PIOFS is an HPF CYCLIC distribution of 32 kilobyte file blocks.

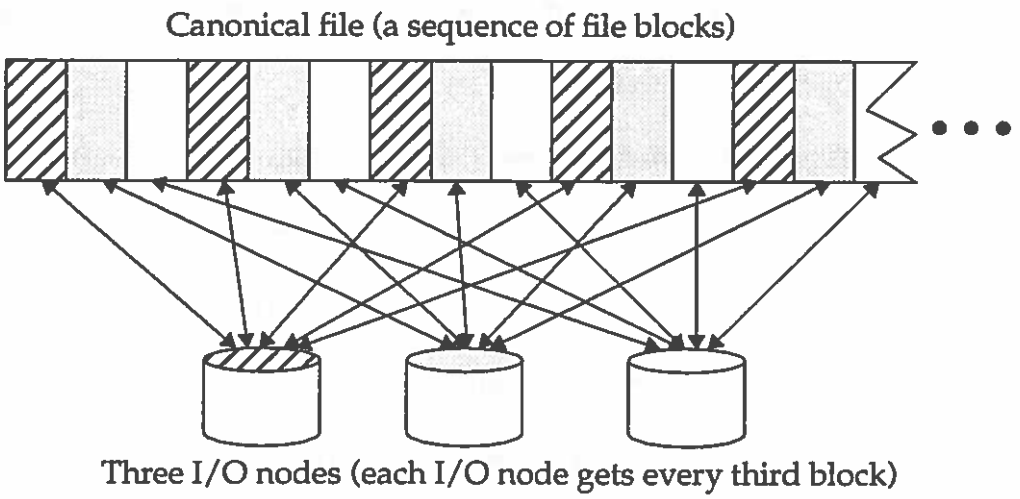

FIGURE 12. File layout: the canonical file is distributed in an HPF CYCLIC distribution onto 3 I/O nodes.

Parallel I/O: The Low-level Perspective

At the lowest level, i.e. at the level of physical data transfer, parallel I/O can be seen as a complex mapping problem from compute node memories to disk

N^3 array onto P compute nodes using an HPF BLOCK distribution yields a data block size of $\frac{N^3}{P}$ elements, while an HPF (BLOCK, BLOCK, BLOCK) distribution yields a much smaller $\frac{N}{\sqrt[3]{P}}$ data block size. Thus, the compute nodes are concurrently writing multiple streams of data blocks to the disks. On the receiving end, the I/O nodes must buffer the data blocks of a certain size as they are received from compute nodes and transfer them to disk. As discussed earlier, the optimal unit of data transfer with respect to the disks is the file block size, which typically bears no relation to the data block size. Furthermore, the order in which data blocks arrive at a given I/O node bears no relationship to their locations on disk, potentially leading to highly inefficient disk I/O.

Without some kind of coordination of activities, parallel I/O performance can be seriously degraded by the mismatch in spatial layout, mismatch in the sizes of the units of data to be transferred, and temporal mismatch in the order in which writes are issued by compute nodes and the ideal order in which they should be received at I/O nodes. The challenges faced by the designers of parallel I/O systems lie in the specification of the mapping of data in compute node memory to blocks on disk and the efficient transfer of these blocks given the spatial, temporal, and size mismatches.

We address these problems through the development of a coordinated approach to parallel I/O. We attack the parallel I/O problem through two avenues. First, we address the low-level problem of efficient data transfer through the design of algorithms that perform coordinated data movement based on knowl-

was first described in 1984 by Salem and Garcia-Molina [121, 122]. Disk striping speeds up file access by partitioning a file and distributing file chunks among multiple disks---servicing a file access with multiple disks simultaneously speeds file access. Disk striping, and its successor, RAID technology [30, 65, 111], focus on hooking together multiple disks to create one logical disk which is both faster and more reliable. In [70] Herbst provides a good review of striping and RAID technology as of 1991.

In 1987, Ellis, Dibble, and Scott extended the idea of disk striping to apply to supplying fast file access to a parallel application rather than a single process. This extension was called "file interleaving". They demonstrated the first parallel file system based on file interleaving: Bridge [52, 55]. Bridge was implemented on the BBN Butterfly. Disks were simulated by reserving memory and having the disk driver insert the appropriate delays a real disk would exhibit. The file layout pattern was HPF CYCLIC, distributing file blocks in a round-robin manner. Shortly thereafter, several commercial parallel file systems (based on file interleaving) were introduced. Thinking Machines Corporation introduced the Connection machine CM-2 DataVault [132] which interleaved files at both the individual bit and file block level, and Intel Corporation introduced the iPSC/2 Concurrent File System (CFS) [75, 112], which interleaved files in 4 kilobyte file blocks. On all of these systems, file interleaving, what we call file layout, was done in an HPF CYCLIC pattern.

ing in a single cell. A productized version of Vesta is available for the IBM SP2 as PIOFS [6, 39].

These new parallel file systems focused on providing a high-level interface to the hardware, but their implementations were based on the commonly held assumption that the I/O behavior of scientific applications did not change between serial and parallel versions. Miller and Katz [97] analyzed scientific applications on a Cray vector supercomputer and found that file access were large and sequential. However, recent studies of parallel scientific applications show that scientific applications for distributed memory parallel machines behave quite differently from their sequential Cray equivalents. We dissected a representative scientific application, the NAS BT benchmark, from the algorithm perspective. Rather than focus on a specific implementation, we investigated all of the common methods of implementing the algorithm and showed that the small file accesses, which are the heart of the parallel I/O performance problem, are an inherent part of the parallelization process, and not an idiosyncracy of current implementation practices. Crandall, et. al. [43] analyzed the access patterns of five scientific applications via run-time instrumentation. These applications exhibited the small accesses which the BT analysis illustrates. In the CHARISMA project [88, 107, 115], Kotz et. al. studied the scientific file access patterns by instrumenting the systems software on two systems with real scientific workloads (at NASA Ames and NCSA). The CHARISMA project supports our analysis that scientific applications perform small file accesses, and our choice of using a three-dimen-

They demonstrate an order of magnitude performance improvement reading an out-of-core two dimensional array on the Intel Delta and nCUBE systems. Two-phase I/O is essentially collective buffering with HPF BLOCK target (abbreviated CB-B). We demonstrate similar performance improvements writing 3D arrays.

The "Extended Two-Phase" algorithm of Thakur and Choudhary [128, 130] extends the two-phase algorithm to allow dynamic assignment of I/O work to compute nodes. Thakur and Choudhary test the extended two-phase algorithm on the Intel Delta, but only compare its performance to the naive method of performing I/O (rather than to the two-phase algorithm). The extended two-phase algorithm is essentially collective buffering with file layout target (abbreviated CB-FL). Finally, Bennett, et. al., describe "Local Collective I/O Optimization" [10], which uses auxiliary nodes to perform the permutation operation. However, compute nodes are limited to accessing whole file blocks from the auxiliary nodes, which could lead to significant communication overhead (due to receiving more data than is needed from auxiliary nodes). Our collective buffering provides a single uniform framework flexible enough to cover two-phase I/O, extended two-phase, local collective I/O optimization, and others.

An approach taken by Seamons et. al. in Panda [125] is to eliminate the grain size mismatch entirely, by changing the file layout to match the data distribution. Preliminary performance writing 3D arrays in 3D chunks on the iPSC/860 is promising, but format conversion back to a one-dimensional file (e.g. for visual-

tributions for the pipelined Gaussian elimination, and a balanced three-dimensional distribution with multiple blocks assigned to each process for the multi-partition method. The only interface capable of describing the multi-partition distribution so that access to the 3D grid can be done in a collective call is MPI-IO (see Appendix for details). PIOUS [99, 100] is only capable of handling one-dimensional distributions, and is essentially a portable version of the Intel PFS built on top of PVM. Vesta and PASSION [19, 31, 32, 129] provide mechanisms to describe two dimensional decompositions (2D arrays). Panda [124] supports three-dimensional distributions of 3D arrays, but only in HPF (BLOCK, BLOCK, BLOCK) and balanced 3D block distributions (where every process gets a single subcube). PPFS [72, 73] is a new parallel file system which may be able to represent the multi-partition distribution, but it is unclear from the documentation how this would be accomplished.

These interfaces/libraries also differ in their target application domain: MPI-IO is a library interface targeted at both in-core and out-of-core message passing applications, Panda is a library targeting data generation applications. PASSION and Jovian target out-of-core algorithms and were designed with the intention integrating them into a parallel compiler. PPFS focuses on configurable generality to support parallel file system research rather than user applications.

and software design of the CFS, and suggest several alternative hypothesis to explain unexpected performance results. Finally, we present a method of obtaining near peak performance in the face of the unexpected (negative scaling) performance.

CFS Hardware

All of the performance tests were run on the iPSC/860 at the Numerical Aerodynamic Simulation (NAS) facility at NASA Ames Research Center. The iPSC/860 system is a hypercube-based MIMD parallel computer. The system at the NAS facility consists of 128 compute nodes, 10 I/O nodes, 1 Ethernet node, and an IBM PC-class front end computer. The CFS consists of the 10 I/O nodes, 10 SCSI disks, an Exabyte 8mm tape drive, and various library routines and servers (see Figure 13).

travel through the hypercube network (shaded area in Figure 13) to the destination compute node. The peak rate of the I/O system (also the peak rate of the CFS) is limited by the slowest link in the data path (see Table 1).

TABLE 1. Throughput Speeds

Device	Mbytes/sec
Hard Disk (Maxtor 8760S, 760 MB)	1
SCSI	4
I/O Node memory bandwidth (16 Mhz 80386)	64
Hypercube Interconnect (per link)	2.8
Compute Node memory bandwidth (40 Mhz i860 XR)	160

In this case, the slowest link is the disk itself, with an estimated throughput of approximately 1 megabyte/second. This implies that the peak throughput the hardware could sustain is 10 megabytes/second.

CFS Software

The software is divided into four parts: node libraries linked into user applications, NX operating system subroutines which are replicated on every compute node, disk block servers running on each I/O node, and a name server running on one I/O node. The iPSC/860 was running the NX operating system, version 3.3.1, 3/92 update.

A user application running on the compute nodes can access data on the CFS by first performing an open() system call. The call causes a request to be sent to the name server process which converts the file name into an absolute disk

Peak Software Rates

The performance of the system was examined in the following manner (see Table 2). Performance was measured for:

1. I/O node memory bandwidth,
2. compute node memory bandwidth,
3. requesting and sending 4 kilobytes worth of data through the hypercube interconnect,
4. reading and writing a single disk from a node.

TABLE 2. Measured Software Rates

Operation	MB/sec
I/O Node Memory Bandwidth (using `memset()`)	11.5
Compute Node Memory Bandwidth (using `memset()`)	48.7
Hypercube Network (Request/Reply with 4 kilobytes)	
nearest neighbor	2.2
across the whole machine	2.0
Disk to Compute Node (reading or writing)	0.82

These measurements imply that the maximum sustained I/O throughput of the CFS is 8.2 megabytes/second (= 10 disks * 0.82 MB/sec per disk).

Performance Tests

Five different tests were performed: broadcast reading, reading and writing an HPF BLOCK distribution, and reading and writing an HPF CYCLIC distribution. Broadcast reading simulates loading an initial data set onto every node.

Wall clock time was independently measured on each node. The time reported for a test is the maximum of these times. The global synchronization ensures that this maximum corresponds to the wall clock time.

Consider the example in Figure 15. The shaded areas represent the individually measured running times. Without the global synchronization, the maximum running time would be 11 time units (node 1). However, the total wall clock time required to complete the job on all nodes is 12 time units. The global synchronization forces wall clock time to be measured from the beginning of the first node's execution to the end of the last node's execution. Otherwise, a node with maximum individual running time which finished before another node would cause the running time to be under reported.

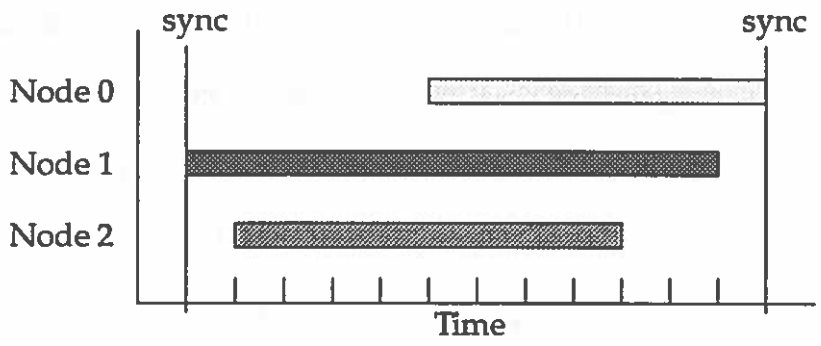

FIGURE 15. Example Runtime Chart.

Selected tests were re-run on a fully dedicated system (with no users on the iPSC/860 or the front-end, and the system isolated from the network) to verify that the NQS runs were true reflections of the performance of the CFS. The performance difference between the fully dedicated and semi-dedicated system runs was insignificant.

Performance Results

The performance results should be viewed in light of the fact that timings varied by more than 20% from one run to the next. For example, a write test using an HPF CYCLIC distribution on 32 nodes, a 32 megabyte file, and a 32 kilobyte block size was repeated six times on a dedicated machine. The performance ranged from 5.5 to 7.6 seconds (4.2 to 5.8 megabytes/second), which is more than a 20% difference. Small timing variations that occur can be magnified by the asynchronous nature of the iPSC/860 to cause large overall timing variations.

Results are reported in megabytes per second, and are plotted against the number of nodes used in the test (Figure 17 - Figure 25). In general, the performance of a perfectly scaling I/O system should increase linearly with the number of nodes up to the peak I/O rate. Once the peak rate is reached, the performance should remain at the peak rate up to the maximum number of nodes in the system (Figure 16).

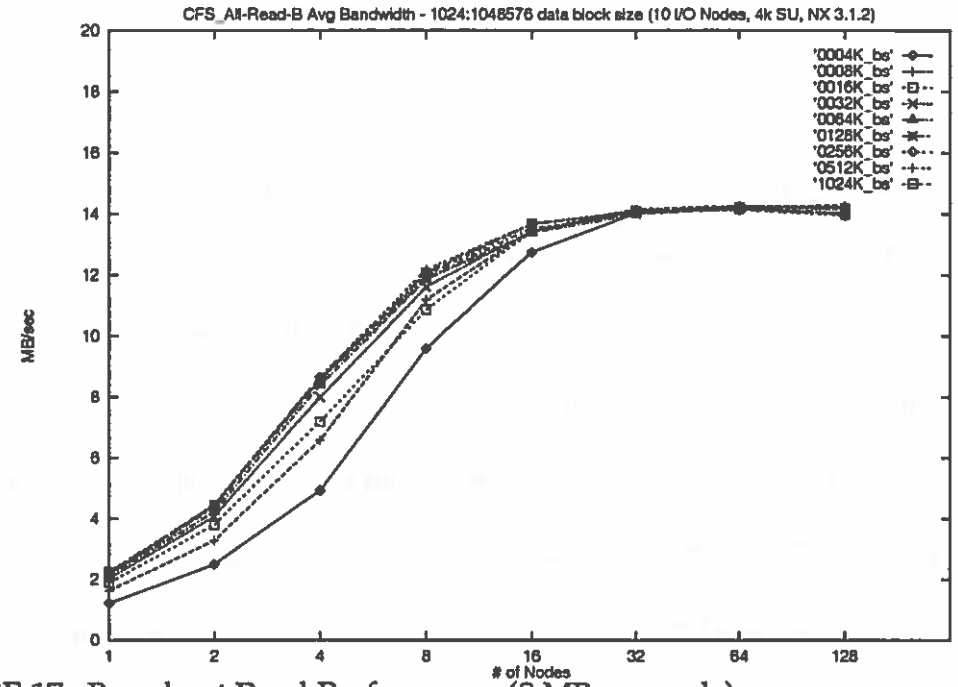

FIGURE 17. Broadcast Read Performance (8 MB per node).

Note that the peak rate (14 MB/sec) is higher than the stated theoretical peak rate for the installed hardware (10 MB/sec). This increased throughput is the result of caching on the I/O nodes, as the same data is being read by all nodes.

In this example, disk bandwidth is the limiting factor. Broadcasting data from a file on disk can be performed much faster by reading the data into a few nodes' memories, then performing a tree broadcast using the hypercube interconnect of the iPSC/860 system. A better estimate for peak throughput in this case would be approximately 60 MB/sec; eight nodes read the file, then perform a tree broadcast using the hypercube interconnect.

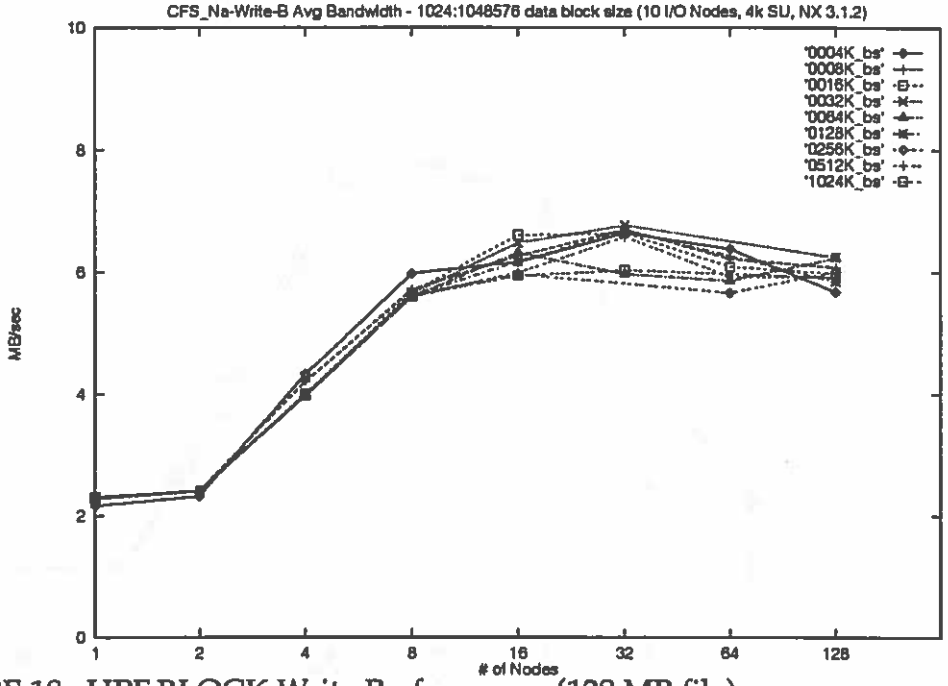

FIGURE 18. HPF BLOCK Write Performance (128 MB file).

The HPF BLOCK read tests (Figure 19), however, exhibits a surprising sharp drop in performance. The performance scales nicely from 2 megabytes per second on one node through 8 megabytes per second on 16 nodes, but then drops to below 1 megabyte per second on 64 and 128 nodes. Not only does this read test not scale, but the performance is worse at 128 nodes than it is on a single node. This performance anomaly is due to resource contention for the read cache, and is examined in "Investigating Disk Cache Behavior" on page 64.

The read tests (Figure 21) perform identically to the HPF BLOCK read tests through 8 nodes (increasing from about 2 megabytes per second to about 7-8 megabytes per second). When run on more than 8 nodes, the performance degrades. Except for the 32 and 64 kilobyte block sizes, the larger the block size, the worse the performance on more than 8 nodes. This is expected (given the HPF BLOCK test results) as the 1 megabyte block size, 128 node tests have identical data distributions. The 512 kilobyte block size tests, although not identical, are very similar, etc. It is unclear why the 64 kilobyte block size performed so much better than the others.

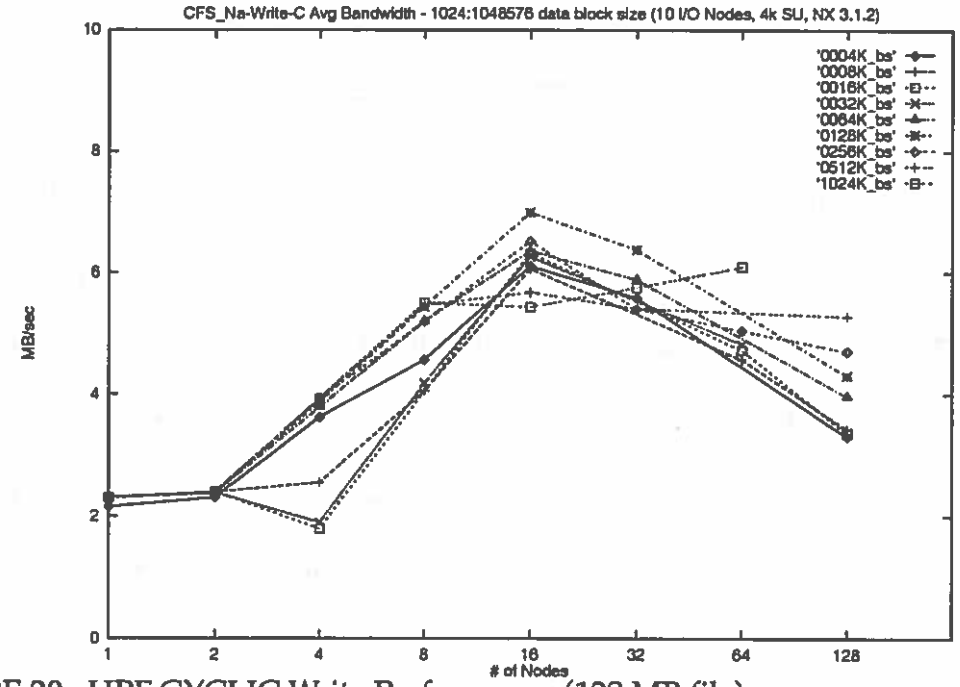

FIGURE 20. HPF CYCLIC Write Performance (128 MB file).

connected in a hypercube, each I/O node is connected over a single link to one compute node. It is standard manufacturing practice to anchor (connect) the I/O nodes to the low numbered compute nodes. In our system, the I/O nodes are connected to compute nodes 2, 6, 10, 14, 18, 26, 30, 34, 38, and 42. Further, all of the CFS tests allocated nodes starting from node 0. So, a 16 node test would use nodes 0-15, while a 64 node test would use nodes 0-63. Although all of our tests used all 10 I/O nodes, the placement of the I/O nodes within the system combined with the placement of the CFS tests among the compute nodes may have led to significant network contention, and contributed to the performance drop.

First, we investigated message latency as a possible cause. Since the speed at which disk requests can be made decreases as distance increases (see Table 2 on page 50), we re-ran selected 16 node tests, varying the location of the 16 compute nodes used within the system. If latency is a major factor in performance, one would expect the tests run on low numbered nodes (close to the I/O nodes) to perform the best, and the tests run on high numbered nodes (far from the I/O nodes) to perform the worst. However, performance was not affected by location, implying that internal network latency is not causing the problem.

Next, we investigated contention. If multiple messages are sent across a network link simultaneously, they must contend for access to the link; typically, the message transfers are serialized. For example, if a network link runs at 1 megabyte per second, and two messages, each 1 megabyte in size, are sent over the link, it will take at least two seconds to complete the transfer. The iPSC/860

Investigating Disk Cache Behavior

The most likely explanation for the drop in read performance is thrashing. The I/O system on the iPSC/860 uses disk block caching and pre-fetching to improve performance. However, in this case, it backfires. In this section, we describe why the combination of caching and pre-fetching on the iPSC/860 yields greatly reduced performance for the HPF BLOCK read tests.

Caching improves performance by exploiting data re-use. If a disk block is read multiple times, it can be read from disk once (an expensive operation), stored in cache, and then read from cache for succeeding operations. However, since cache size is limited, typically, only recently accessed blocks are stored in cache. The HPF BLOCK and CYCLIC reading and writing tests did not re-use any data. In fact, for reading and writing large solution files, it is doubtful that caching would ever be used.

However, the I/O system also uses pre-fetching to improve performance. Pre-fetching depends on locality of reference and requires caching to be effective. When a disk block is accessed, it and several successive blocks are read into cache. It is typically more efficient to read multiple blocks from disk in a single operation than to read the same blocks, one disk operation at a time. When the successive blocks are accessed by the application, they can simply be returned from cache. If, however, the successive blocks are not accessed, or are purged from cache before they are accessed, then the extra disk I/O to pre-fetch them was wasted. All of the

Now consider running the HPF BLOCK read test using 64 compute nodes. Restrict attention to one I/O node for simplicity. The first 31 nodes read a block, causing 31 pre-fetch operations, using 248 of the 250 cache blocks available. When the next 31 nodes read a block, they also cause pre-fetch operations, but since there is no more free space in cache, each pre-fetch must purge the data from a previous pre-fetch operation. Most likely, data is purged before anything but the first block is read by the requesting node. This continues for the entire file. For every read operation, 8 blocks are pre-fetched, 1 block is returned, and the other 7 are purged from cache before they can be read. This should result in an 8X slow-down from 16 to 64 nodes, but no performance difference between 64 and 128 nodes—exactly what was measured (Figure 19).

HPF CYCLIC reading exhibits this problem too. For large block sizes, the results are the same as for the HPF BLOCK tests. As the block size is decreased, performance improves. A smaller block size allows better use of cache. As the block size decreases, the reads are closer together in the file, and there is more chance that each read will be able to use pre-fetched data. Since there is no way to use pre-fetching for writes, the writing does not suffer from this problem.

Note that a second performance drop can be seen for both the HPF CYCLIC read and write tests. Performance drops for small block sizes (4, 8, and 16 kilobytes) above 16 nodes. Further investigation is required to determine the cause of this performance drop.

that this method would scale above 128 nodes. The slight drop (from 8 to 7 megabytes per second) can be attributed to synchronization overhead.

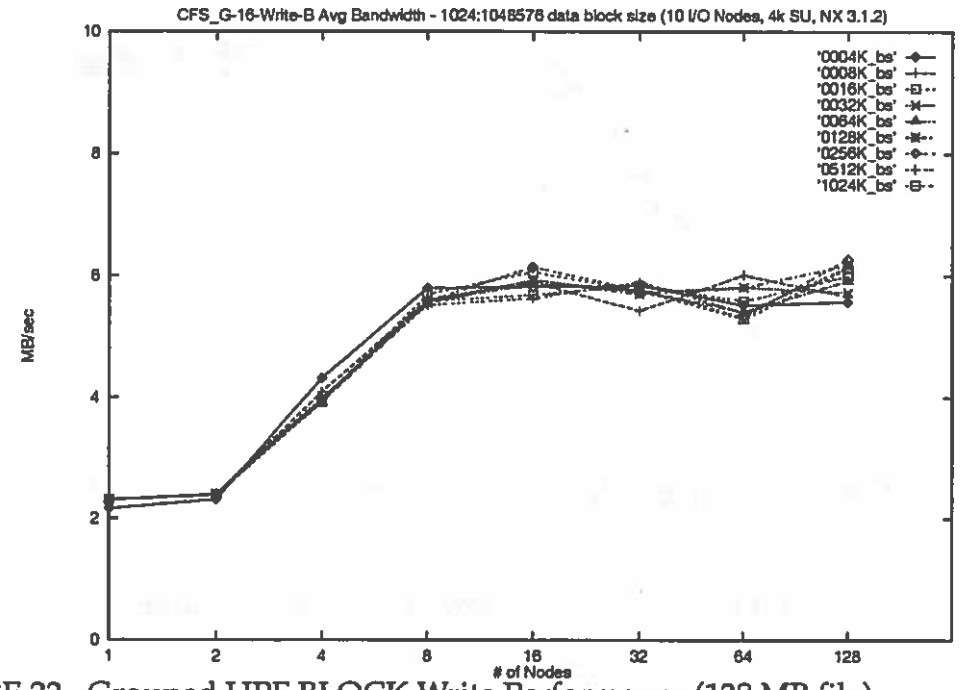

FIGURE 22. Grouped HPF BLOCK Write Performance (128 MB file).

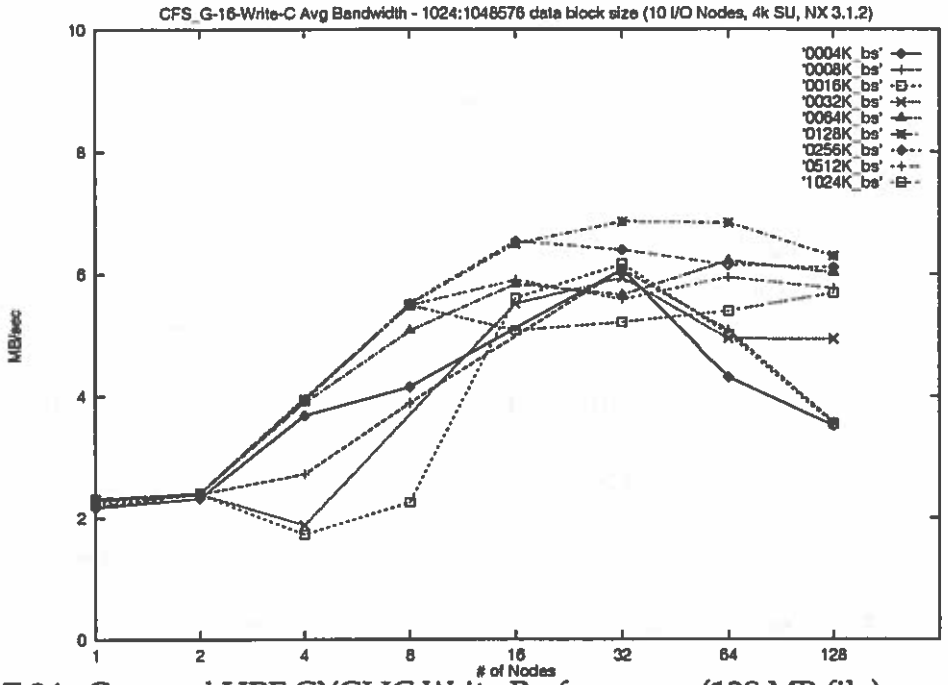

FIGURE 24. Grouped HPF CYCLIC Write Performance (128 MB file).

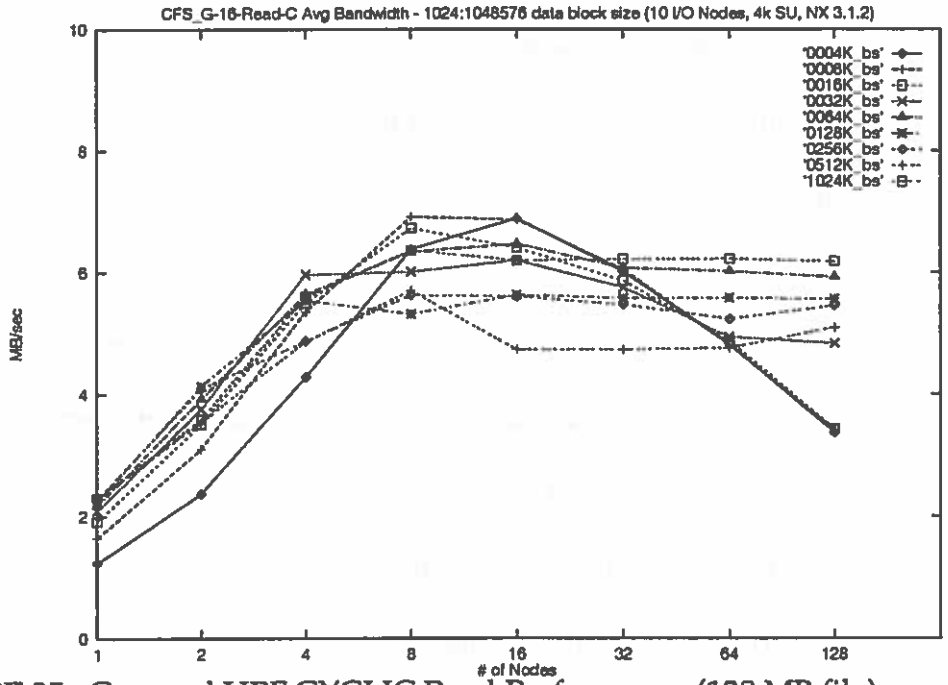

FIGURE 25. Grouped HPF CYCLIC Read Performance (128 MB file).

We propose a better approach is to fix the problem directly: develop algorithms and techniques for optimizing parallel access, such as grouping, and implement these optimizations as part of a portable parallel I/O library. It is beyond the scope of this work to fix filesystem design flaws, although we attempt to reveal flaws and suggest fixes.

Parallel I/O Library

Highly parallel systems are difficult to program, and in general, obtaining peak performance from the I/O system requires a significant effort. Further, this effort must be repeated for each highly parallel system one wishes to use. A library of global I/O routines would not only allow portable programs to be written with minimal effort, but it would also allow programmers to concentrate on algorithm development and execution speed instead of I/O performance. A collective parallel I/O library provides the framework for machine dependent I/O algorithms, such as "grouping".

The library should include routines for reading and distributing, as well as collecting and writing, one-, two-, and three-dimensional decompositions of arrays. On the iPSC/860, the library could incorporate both the grouping of nodes and the "transpose" operation necessary to ensure that block sizes are above 4 kilobytes. This would greatly increase throughput for the average user.

tion remains essentially the same. Fineberg [60] measured the performance of the BTIO benchmark [61], which uses the multi-partition distribution method, on the Paragon and SP2. Both systems were capable of providing 40 megabytes per second, yet BTIO achieved only 90 kilobytes per second on the Paragon and 1 megabyte per second on the SP2---still slow by about two orders of magnitude. Other parallel file systems studied [17, 23, 90, 114] show similar results: performance on small accesses is abysmal. These studies, including our iPSC/860 study, motivate the need for improved parallel I/O performance.

applications at NAS which use three-dimensional data sets for aircraft performance simulations, and by the use of 3D meshes in a wide range of other scientific applications. These are described thoroughly in Chapter II, but we review them here for convenience.

The *global data structure* used in these experiments is a three-dimensional array. This data structure is distributed among compute nodes in a manner which maximizes load balance and which minimizes communication overhead for the computation. For the class of scientific algorithms involved, communication is necessary to exchange boundary elements, so the communication overhead of an application is proportional to the number of boundary elements. For this reason, arrays are generally distributed in all three dimensions, minimizing the surface to volume ratio, thus minimizing the number of boundary values to be exchanged (see Figure 5 - Figure 7). In this case, the *data chunks* assigned to individual compute nodes are sub-cubes of the global, three-dimensional array.

The *canonical file* is the logical view of the data from the perspective of the file system. In this study, we assume that the canonical file is a single sequential file resulting from a linearization of the multi-dimensional mesh, based on some ordering of the elements such as row-major order. This definition of the canonical file stems from the fact that current parallel I/O systems evolved directly from Unix-based file systems for sequential machines. All commercially available systems (e.g. Intel iPSC/860 CFS, Intel Paragon PFS, IBM SP2 PIOFS, and Meiko CS-2) use the canonical view of a file as a sequential byte-stream. The file layout pat-

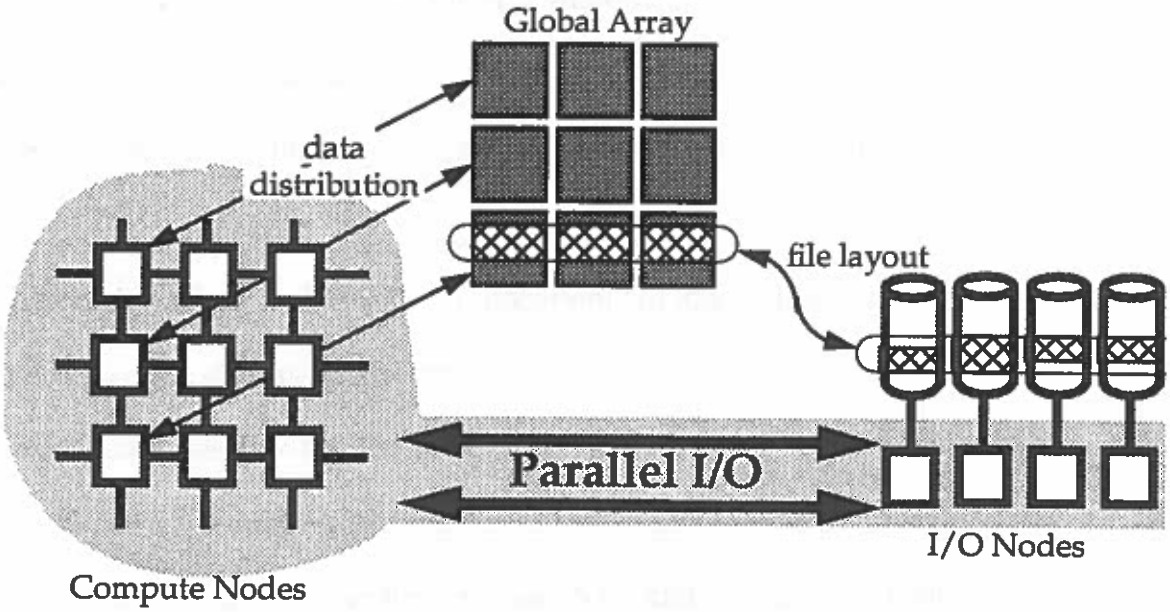

FIGURE 26. Parallel I/O Complexity: two-dimensional sub-blocks of the global array are mapped to compute nodes, but the array is mapped in slices to disks.

The challenges faced by designers of I/O transfer algorithms are best illustrated by taking a close look at the current method used for performing parallel I/O. We refer to this transfer method as the Naive Algorithm.

Naive Algorithm

The Naive Algorithm basically treats parallel I/O the same as workstation I/O. It allows the programmer to treat her application as a collection of standard Unix processes accessing a shared file. Until recently, programmers have been forced to use the Naive algorithm, as parallel I/O interfaces were incapable of describing the complex three-dimensional data distributions common to scientific applications. Today, there are only three interfaces capable of supporting these

requires that the previously flushed contents be read back from disk, the new data block(s) added to the buffer, and then the updated disk block re-written.

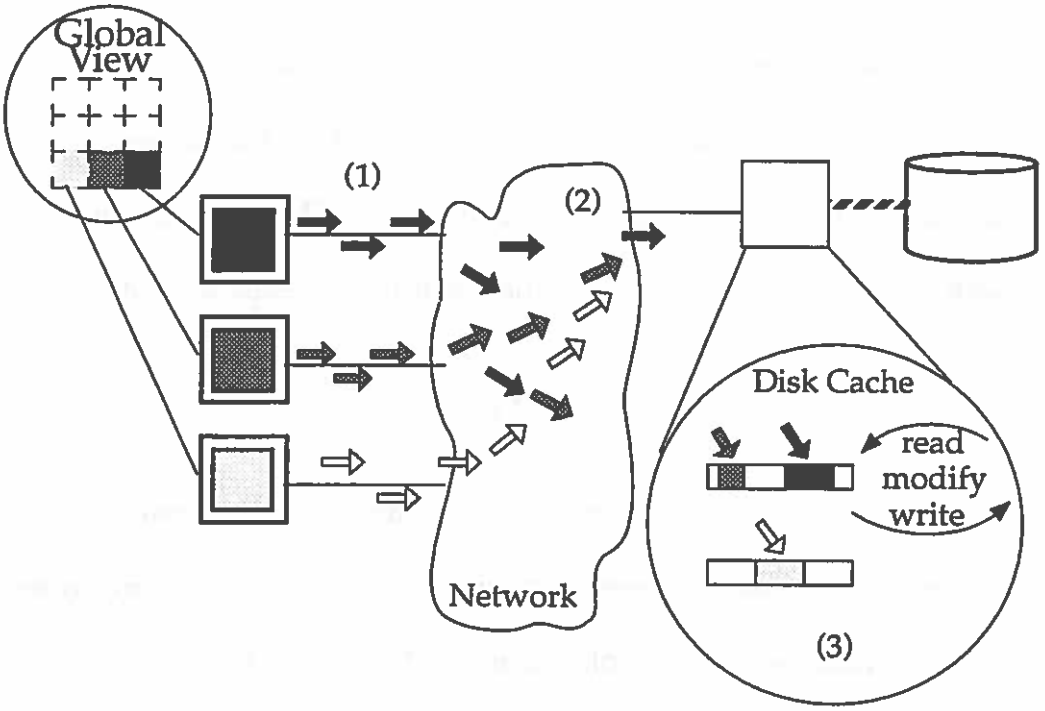

FIGURE 27. Naive Algorithm: (1) compute nodes independently send small requests to the I/O nodes; (2) data travels through the network and arrives at I/O nodes in an uncoordinated fashion; (3) I/O nodes must perform read/modify/write operations due to inefficient caching.

The Naive Algorithm exposes the major problems involved in parallel I/O: resource contention for both the network and OS services; poor locality (both temporal and spatial); arrival of data blocks in arbitrary order, resulting in excessive disk access that cannot be remedied by caching; and load imbalance in which one or more disks is overloaded while others remain idle.

Data Block Scheduling

On a serial system, I/O transfer operations are often reordered to minimize disk head seeks, but there is little other benefit from scheduling I/O accesses. In addition to scheduling to maximize the efficient use of resources (e.g. disk head seeks), the many-to-one relationship imposed by the parallel I/O architecture (many compute nodes to a few I/O nodes) coupled with the volumes of data which must be transferred for a global I/O operation, combine to create the potential for significant contention for resources. In an ideal system, contention effects would merely place an upper bound on performance, but in real systems, contention effects can result in thrashing—causing aggregate performance to drop off sharply. This performance reduction can result from optimistic network protocols (e.g. with back-off and retry algorithms), from misguided caching schemes, and from architectural limits of the hardware (e.g. small number of hardware thread contexts). The simplest way to reduce the effects of this contention is to schedule transfers intelligently, allowing sufficient parallelism to reach peak performance, but not enough to induce thrashing.

Data block scheduling consists of two approaches: the multiple streams of I/O requests are reordered (spatial scheduling) and sets of I/O requests are performed in phases (temporal scheduling). Spatial scheduling performs the same optimization as in a serial machine where it supports efficient use of the underlying system. However, on a parallel machine, it is greatly complicated by the need

also perform the function of coalescing a given I/O request from multiple nodes. Finally, the grain size mismatch is more complex due to the fact that data distribution and file layout are completely independent of each other.

Collective buffering uses global knowledge of the data distribution, file layout, and machine architecture characteristics to perform a permutation step which maps the data distribution on compute nodes to the file layout on disks. In other words, the data is rearranged by the compute nodes prior to the issuance of I/O operations, in order to minimize the number of disk operations needed on each disk node (see Figure 28). The permutation can be performed "in-place," where the compute nodes transpose the data among themselves, or "on auxiliary nodes," where the compute nodes transpose the data by sending it to a set of auxiliary buffering nodes. The target distribution of the permutation, the number of nodes used, and the amount of memory used to for buffering are chosen to optimize performance. The "two-phase I/O" of del Rosario, et. al. [49] is a particular instance of the "in-place" permutation while the "local collective I/O optimization" of Bennett, et. al. [10] is a particular instance of the "on auxiliary nodes" permutation; both were independently developed. Note that the two permutation techniques perform the identical operation, but "in-place" requires extra memory on each compute node, while "on auxiliary nodes" requires extra compute nodes to do the buffering. The latter has the advantage of permitting true asynchronous I/O with little interference to the work of the original compute nodes.

In collective buffering, data to be written to disk is first organized into relatively large sequential pieces. Each piece resides on a single node, which then writes to disk. This organize-write operation can happen simultaneously on many nodes. Permutation of the data prior to disk I/O takes advantage of the low latency times of the communication network. Although permutation increases network traffic, it avoids the high disk latencies that would result from data accesses that have not been optimized for disk I/O. The permutation operation is particularly efficient on machines with hardware support for scatter/gather operations.

Experiments with Collective I/O Transfer Algorithms

In this section we describe experiments we conducted to test the performance of our data block scheduling and collective buffering transfer algorithms. These algorithms were implemented and tested on two widely used parallel machines: the IBM SP/2 and the Intel Paragon, both housed at NASA Ames NAS Division. First, we describe the basic hardware and software architecture of the two target machines. We then discuss the architectural and file-related parameters we have chosen. Finally, we describe the algorithms and the algorithm parameters selected for these experiments, and give results for each transfer algorithm.

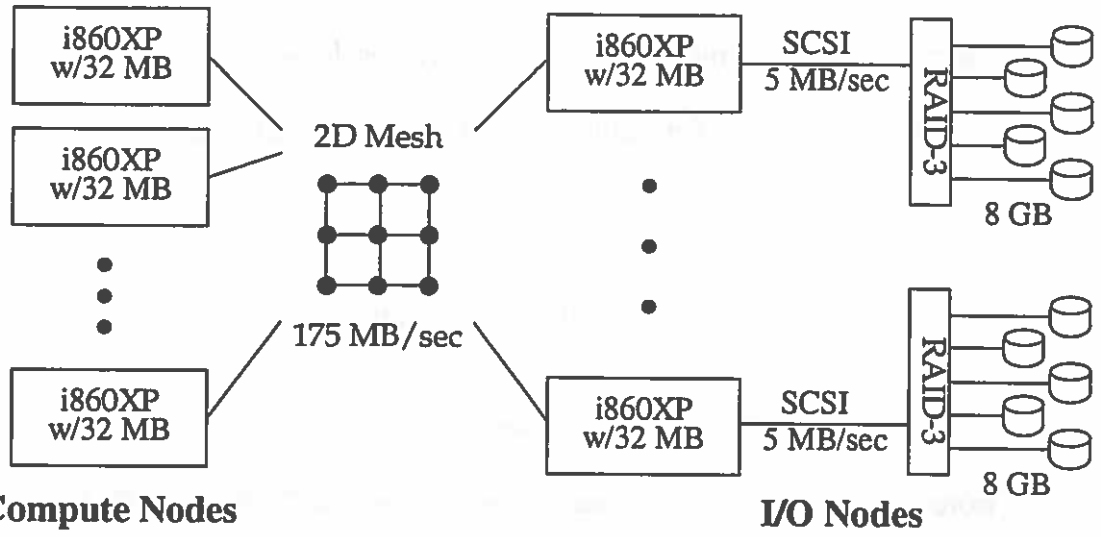

FIGURE 29. Intel Paragon at NAS: 224 i860XP processors, each with 32 MB memory; the 9 I/O nodes each have one 8 GB RAID-3 disk array.

Various architectural performance parameters for the Paragon are given in Table 4.

TABLE 4. Intel Paragon Performance

Performance		Measured	Peak
Memory	Bandwidth	140 MB/sec	200 MB/sec
Network	Latency	60 usec	
	Bandwidth (per link)	70 MB/sec	175 MB/sec
RAID-3 (Local)	Write Bandwidth (per RAID)	2.5 MB/sec	5 MB/sec

Paragon PFS stripes file blocks across I/O nodes in an HPF CYCLIC pattern. The stripe block size is set at filesystem creation time. All experiments were performed with the PFS striped across 6 I/O nodes, using a stripe block size of 64 kilobytes. Although the PFS interface supports "file modes" for defining data distribution patterns, they were not powerful enough to specify more than a simple

space sharing, so all tests ran on dedicated compute nodes. However, the 8 I/O nodes were used for development work in addition to being PIOFS servers, which may have contributed to the timing variations from one run to the next.

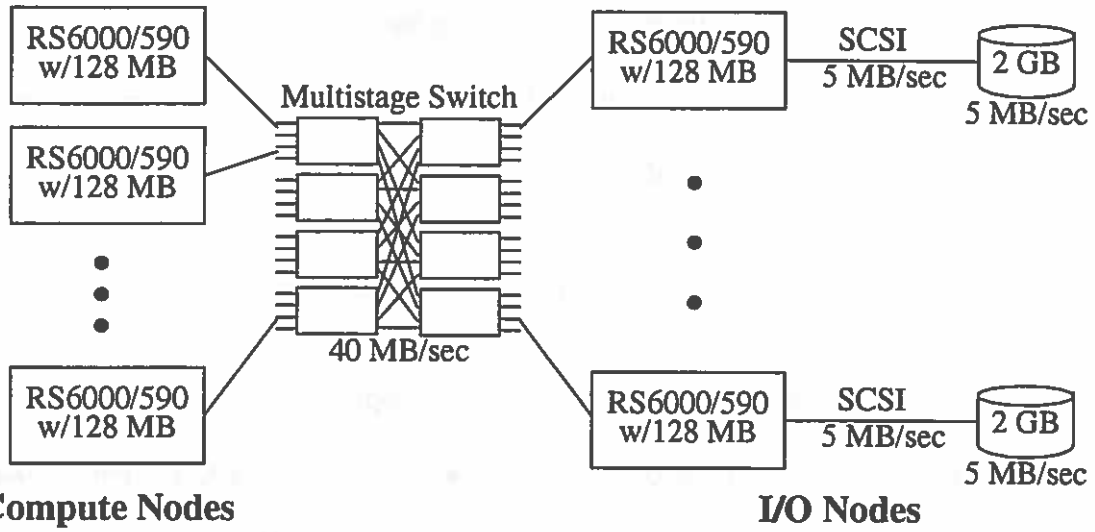

FIGURE 30. IBM SP2 at NAS: 160 RS6000/590 processors, each with 128 MB memory and 2 GB local disk; software partitioned into compute and I/O nodes.

Various architectural performance parameters for the SP2 are given in Table 5.

TABLE 5. IBM SP2 Performance

Performance		Measured	Peak
Memory	Bandwidth	1.2 GB/sec	2 GB/sec
Network	Latency	45 usec	
	Bandwidth (per link)	35 MB/sec	40 MB/sec
Disk (Local)	Write Bandwidth (per disk)	2.7 MB/sec	5 MB/sec

PIOBENCH ran on top of the highly portable MPI message passing library. All of the transfer algorithms were hand-coded into the PIOBENCH program using MPI calls for synchronization and communication (pseudocode for these algorithms is given later). All data access was via the standard UNIX read and write system calls.

In order to realistically test the performance of our I/O transfer algorithms, we ran experiments over a wide range of parallel I/O parameters, including data distribution, file size, and number of compute nodes for both hardware platforms described above. We ran PIOBENCH on 1, 2, 4, 8, 16, 32, 64, and 128 compute nodes, with small (16 megabyte) and medium (128-192 megabyte) sized files. We used the standard HPF data distributions—BLOCK, CYCLIC, (BLOCK, BLOCK, BLOCK), as well as a Random distribution. Under HPF, both BLOCK and (BLOCK, BLOCK, BLOCK) have fixed data block sizes dependent on the file size, number of compute nodes, and compute node distribution. For CYCLIC and Random, we varied the data block size from 128 bytes to 1 megabyte. The file layouts used were those of the underlying parallel file systems: HFP CYCLIC(64 kilobytes) for Paragon PFS, and HPF CYCLIC(32 kilobytes) for SP2 PIOFS.

In all experiments, we measured total I/O time. Timing runs were repeated, and results reported are the average over a minimum of three trials, and represent at most a 20% error (10% error for the Paragon PFS) with a 90% confidence level. Selected runs which required more than 10 minutes were run only

Data block size is a function of distribution, file size, and the number of compute nodes. The data block size for the HPF CYCLIC and random distributions is a parameter of the distribution, and can take any value from one byte to the entire file size. For the HPF BLOCK distributions, however, the data block size is the file size divided by the number of nodes (in the first dimension). Given a file size and number of compute nodes, there is exactly one data block size for an HPF BLOCK distribution; this is reflected as a single large data block size on the performance graphs. The three-dimensional (BLOCK, BLOCK, BLOCK) distribution has some leeway: the size of each of the three dimensions of the global data structure, and the size of each of the three dimensions for distribution onto the nodes can be chosen. We fix the size of the three dimensions for distribution onto the nodes to be equal (or as equal as possible, assigning leftover nodes to the first dimensions). The sizes of the three dimensions of the global data structure range from all equal to having slightly more data in the first dimension; this allows us to measure across several data block sizes while retaining the (BLOCK, BLOCK, BLOCK) distribution pattern. All of the (BLOCK, BLOCK, BLOCK) distributions result in small data blocks (less than 2 kilobytes for even the larger file sizes).

For small data blocks, the absolute performance of both the Paragon and the SP2 was abysmal. On Paragon PFS (Figure 32 - Figure 35), performance peaked at approximately 0.025 megabytes per second for 128 byte data blocks (the smallest data block size tested). Performance increased linearly with an increase in the data block size doubling to 0.05 megabytes per second for 256 byte data

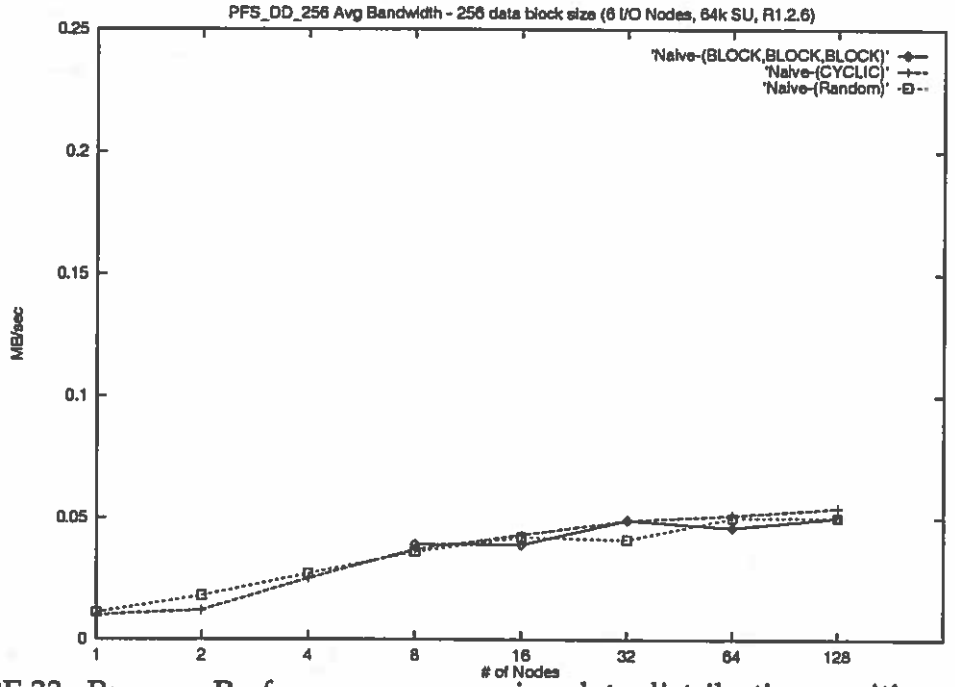

FIGURE 33. Paragon Performance comparing data distributions writing a 16 MB file using 256 byte data blocks.

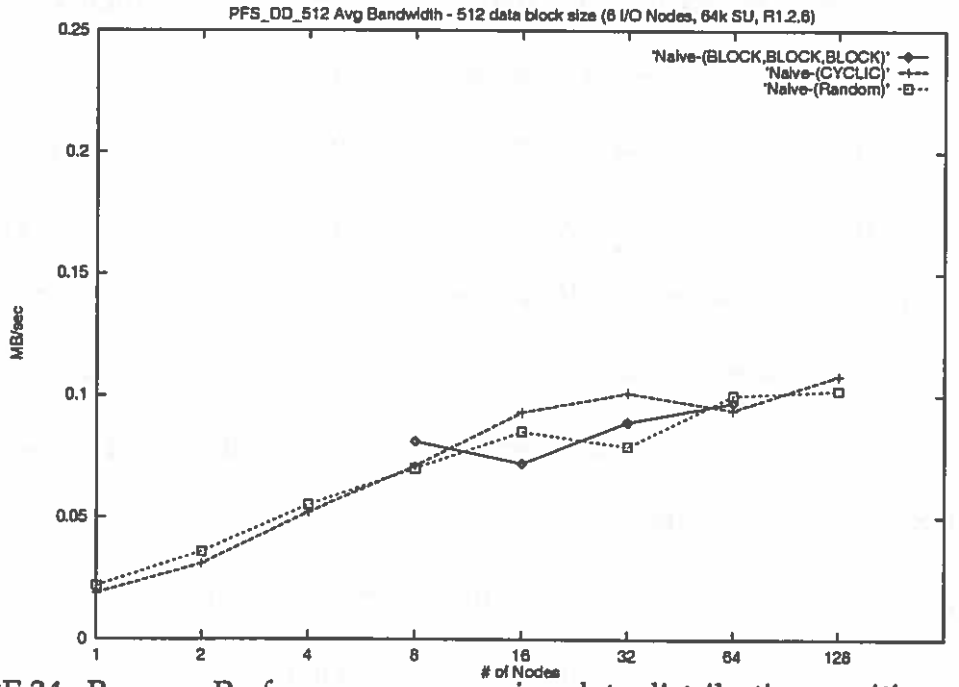

FIGURE 34. Paragon Performance comparing data distributions writing a 16 MB file using 512 byte data blocks.

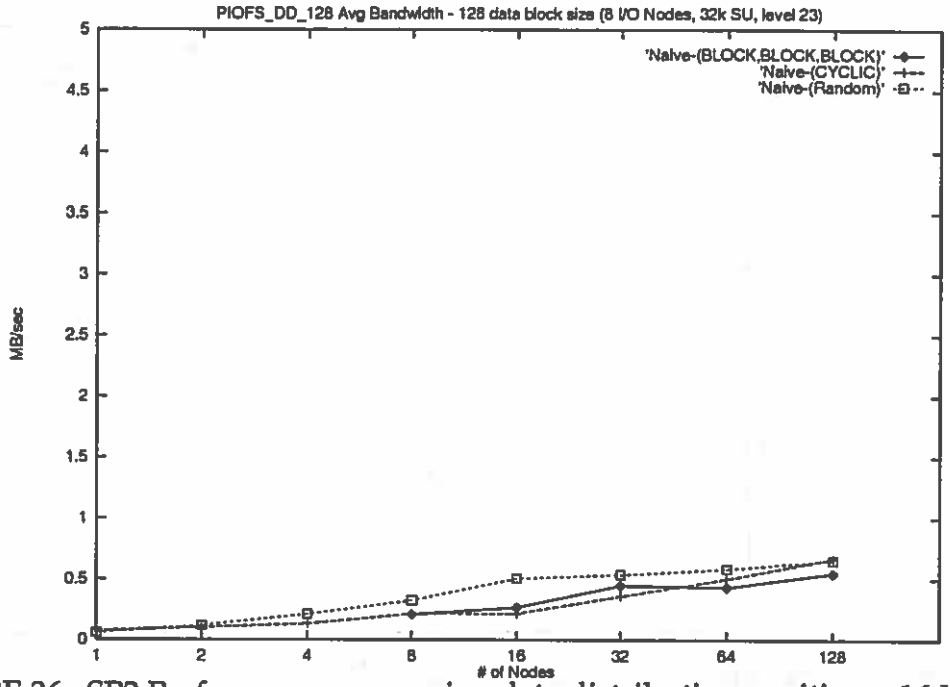

FIGURE 36. SP2 Performance comparing data distributions writing a 16 MB file using 128 byte data blocks.

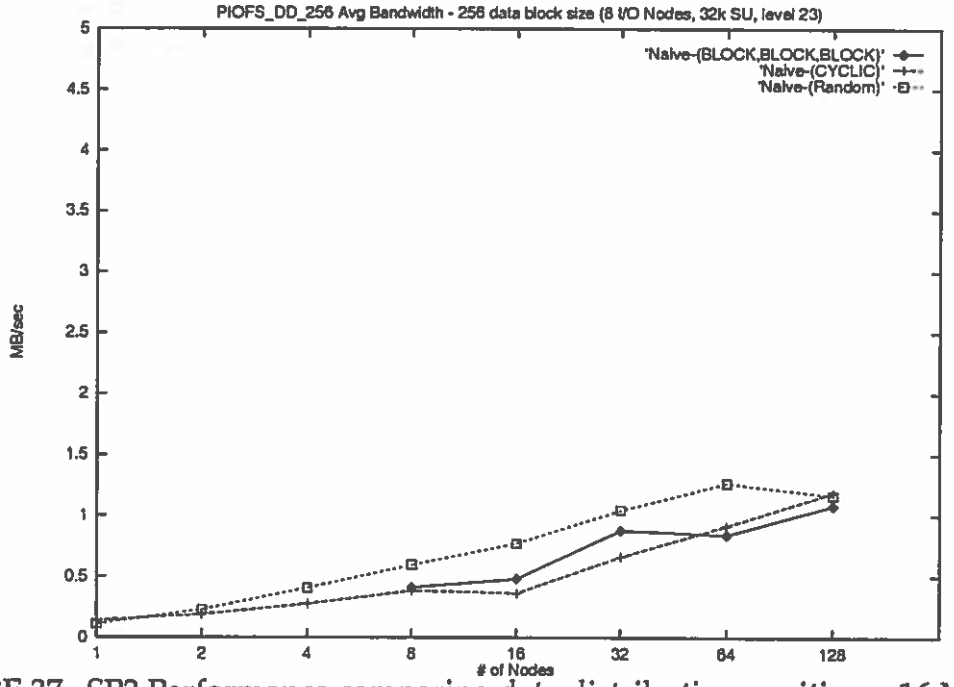

FIGURE 37. SP2 Performance comparing data distributions writing a 16 MB file using 256 byte data blocks.

For large block sizes, performance of both the Paragon and the SP2 was good, peaking at just under 20 megabytes per second on both systems. On the Paragon (Figure 40 - Figure 42), a distinct performance jump can be seen at, and above, 64 kilobyte data blocks which is both the striping unit size and file system block size. Performance for data blocks of 64 kilobytes and greater is nearly identical, with larger data blocks yielding no greater performance. Again, the performance for the different data distributions is similar, with the performance of the random distribution exceeding HPF CYCLIC for data block sizes greater than 64 kilobytes (recall that the HPF (BLOCK, BLOCK, BLOCK) distributions do not produce large data block sizes). The apparent drop in performance as the number of compute nodes increases between graphs is an artifact of the PIOBENCH program—a small file size coupled with a large data block size results in a significant percentage of execution time being devoted to open/close overhead rather than performance measurement.

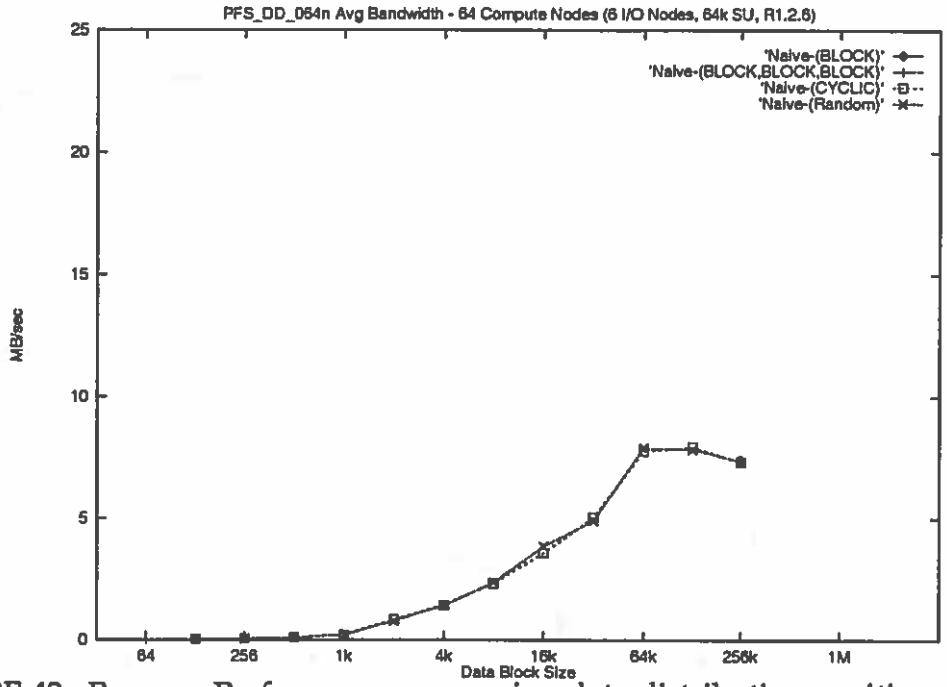

FIGURE 42. Paragon Performance comparing data distributions writing a 16 MB file from 64 compute nodes.

SP2 performance (Figure 43 - Figure 45) increases more smoothly with increasing data block size, noticeably rising at 4 kilobyte data blocks (the file system block size). There is no distinct performance jump, as there is on the Paragon, at the striping unit (32 kilobytes). In fact, except for a peak at 512 kilobytes, performance is relatively level across data block sizes in the range 16 kilobytes to 1 megabyte. As with Paragon performance, data distribution is not as important as data block size for performance, and although the random data distribution outperforms the others, performance of all distributions is similar.

```
/* Make elementary datatype (x,y,z,temp,pres) */
MPI_Type_contiguous(5, MPI_DOUBLE, &etype);

/* Make datatype for each subcube */
for (each subcube assigned to this process) {
      MPIO_Type_make_nd_datatype(
            global_array_dimensions[0..2],
            dimensions_of_local_subcube[0..2],
            location_within_global_array[0..2],
            &subcube_type[i]);
}

/* Combine subcubes into one filetype */
MPI_Type_struct(nsubcubes,
      subcube_type[0..nsubcubes],
      subcube_file_positions[0..nsubcubes],
      &filetype);

/* Make MPI datatype for subcubes in memory */
MPI_Type_struct(nsubsubes,
      subcube_type[0..nsubcubes],
      subcube_memory_locations[0..nsubcubes],
      &memtype);

/* Open the file using filetype created above */
MPIO_Open("output-file",
      MPIO_CREATE|MPIO_WRONLY,
      etype,
      filetype,
      &fp);

/* . . . Compute . . . */

/* Write global data structure to output-file */
MPIO_Write_all(fp, &local_data, memtype);
```

FIGURE 89. MPI-IO pseudocode for saving the 3D solution array for the NAS BT benchmark using the multi-partition distribution.

MPI-IO is a proposed extension to the MPI standard. The MPI-IO interface is equally suited for use with all proposed parallel I/O algorithms, such as the data block scheduling and collective buffering algorithms proposed in this work, the disk-directed I/O proposed by Kotz, and the new file layouts proposed by

of all file access operations, and defines filetypes, an extension of MPI datatypes, to allow the programmer to specify the distribution of the global data structure. It is expected that an implementation of MPI-IO for a particular system would target I/O optimizations to the file layout and machine architecture of that system. For example, PMPIO [138], the portable implementation of MPI-IO, implements collective buffering with an HPF CYCLIC target. It is expected that the system administrator tune the target block size and number of nodes for the particular configuration at hand. Recall that all currently available commercial systems use an HPF CYCLIC file layout.

In order to illustrate how the MPI-IO interface is used, consider implementing the multi-partition version of the BT benchmark (see "Multi-partition" on page 23). In the example (Figure 7), the 3D array is distributed among nine compute nodes, and each compute node is assigned three subcubes of the global array.

An MPI-IO type constructor function is used to create each subcube, giving the dimensions of the global 3D array, and the dimensions and placement of the local subcube within that array. The three subcube types are combined to create the filetype. The filetypes of each compute node are passed into the MPIO_Open call, and together, specify the global data distribution (see Figure 88).

1. Data block size dominates data distribution as a factor affecting performance.
2. Grouping, which minimizes contention by reducing parallelism, can be an effective technique for eliminating performance bottlenecks caused by thrashing, independent of its cause.
3. The Random and Sliding Window algorithms showed promise for small data block sizes only, but for these sizes, collective buffering techniques were orders of magnitude better.
4. Collective buffering, which combines small data blocks together to allow larger data accesses, significantly improves parallel I/O performance by up to two orders of magnitude, requiring only modest buffer space (1 megabyte per I/O node).

The performance potential of these collective data transfer algorithms is clear. The next step is to design a high-level collective parallel I/O interface, providing the framework under which to implement these algorithms, thereby shielding users from the prospect of hand coding intricate, non-portable optimizations to improve I/O performance.

Discussion

Near peak parallel I/O performance can be achieved by using the collective I/O algorithms described in this chapter. However, it is not feasible to expect individual programmers to implement these algorithms by hand. Our intention is

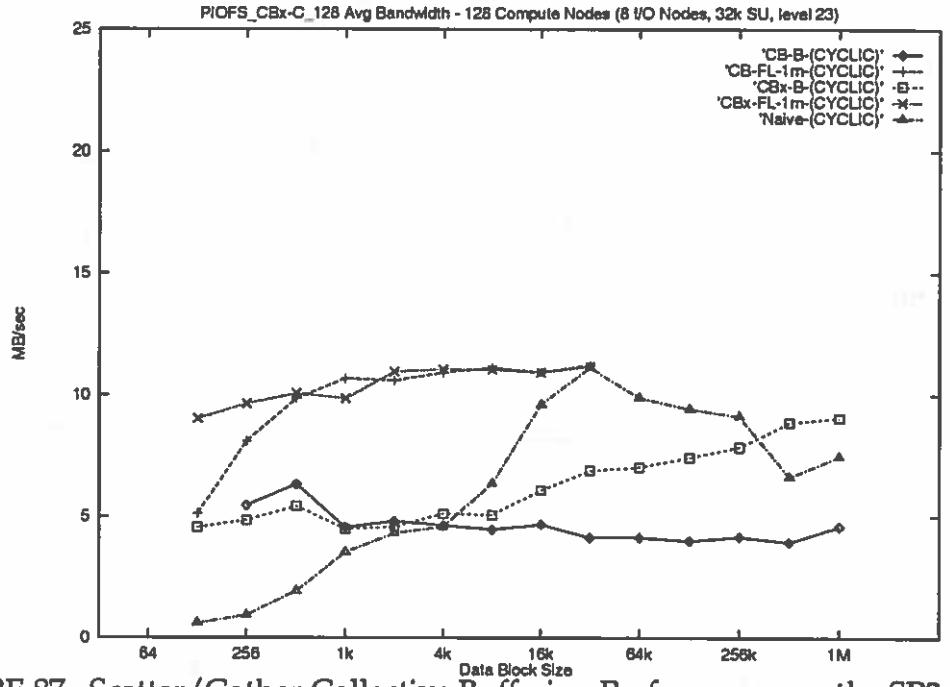

FIGURE 87. Scatter/Gather Collective Buffering Performance on the SP2 writing an HPF CYCLIC distributed 128 MB file on 128 nodes.

Collective Buffering Summary

We have shown:

1. Collective buffering significantly improves Naive parallel I/O performance by two orders of magnitude for small data block sizes.
2. Peak performance can be obtained with minimal buffer space (approximately 1 megabyte per I/O node).
3. Performance is dependent on intermediate distribution (up to a factor of 2)
4. There is no single intermediate distribution which provides the best performance for all cases, but a few come close

of rounds times the number of target compute nodes. Using scatter/gather message passing, each round requires only one message to be sent from each compute node to each target compute node.

```
/* N nodes, Pt target compute nodes */
k = process_rank();
if (k < Pt)
      allocate buffer space
for (r = 0; r < nrounds; r++) {
      if (k < Pt)
            /* setup to receive one big block */
      barrier_synchronize();
      foreach local data block d in this round {
            dt = target node for d
            add d to dt scatter/gather block
      }
      for (n = 0; n < Pt; n++)
            send n's scatter/gather block to n
      if (k < Pt) {
            wait for the message to arrive
            write out the buffer
      }
}
```

FIGURE 85. Collective Buffering with Scatter/Gather Pseudocode.

We were unable to fully implement collective buffering with scatter/gather (CBx) due to limitations of the message passing library used by PIOBENCH. Instead, we simulate the scatter/gather operation by sending messages of size identical to the messages which a real scatter/gather would send.

Simulated CBx performed as expected (Figure 86 - Figure 87), providing peak performance down to the smallest data block sizes tested, on both the Paragon and SP2. Although the method we used to simulate CBx was slightly optimistic, as it did not account for the time taken building scatter/gather buffers, we

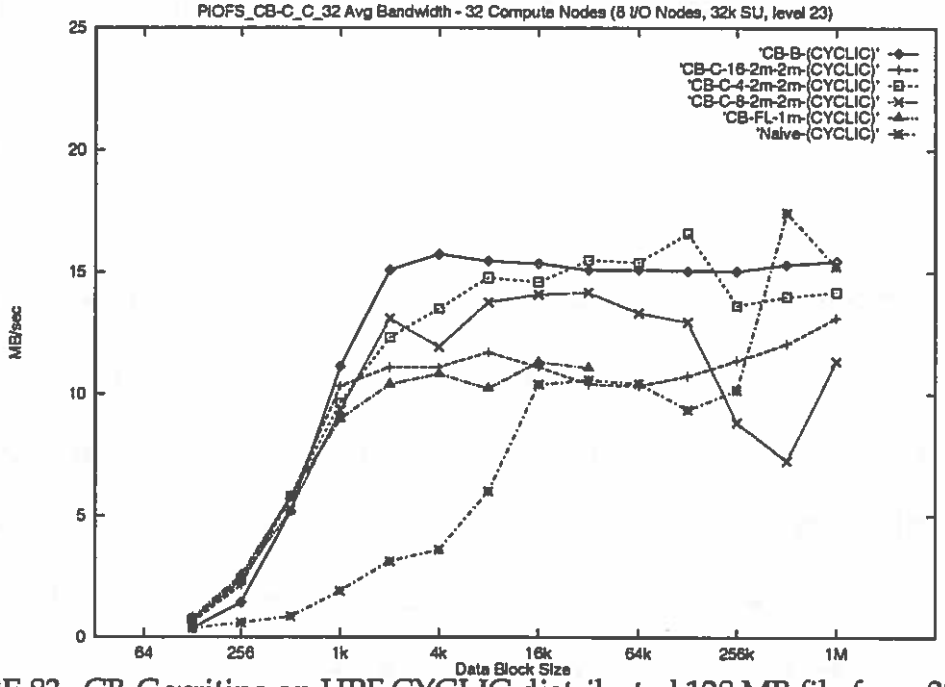

FIGURE 83. CB-C writing an HPF CYCLIC distributed 128 MB file from 32 nodes on the SP2.

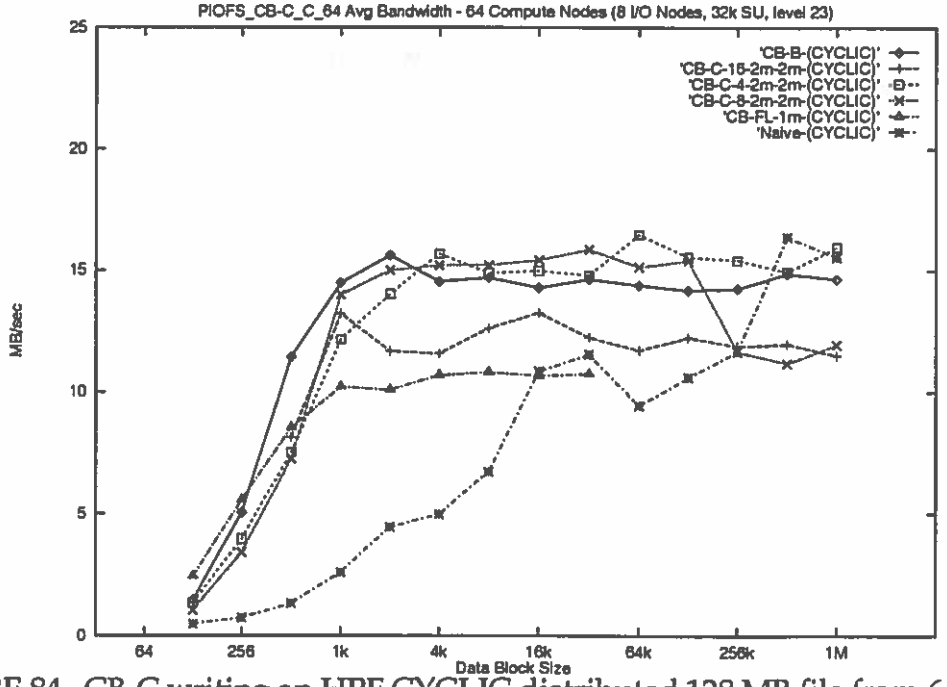

FIGURE 84. CB-C writing an HPF CYCLIC distributed 128 MB file from 64 nodes on the SP2.

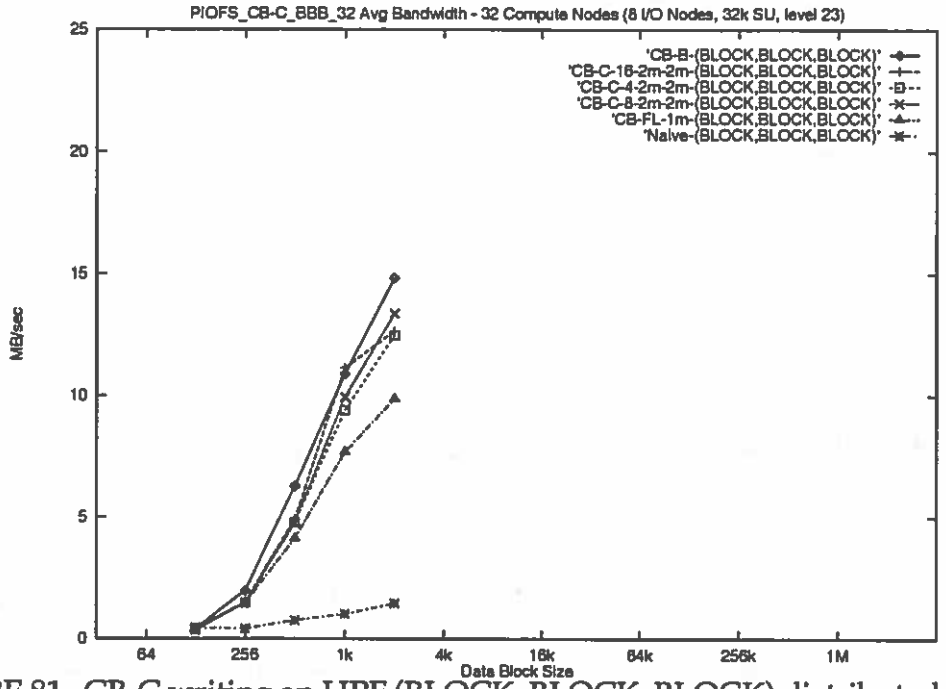

FIGURE 81. CB-C writing an HPF (BLOCK, BLOCK, BLOCK) distributed 128 MB file from 32 nodes on the SP2.

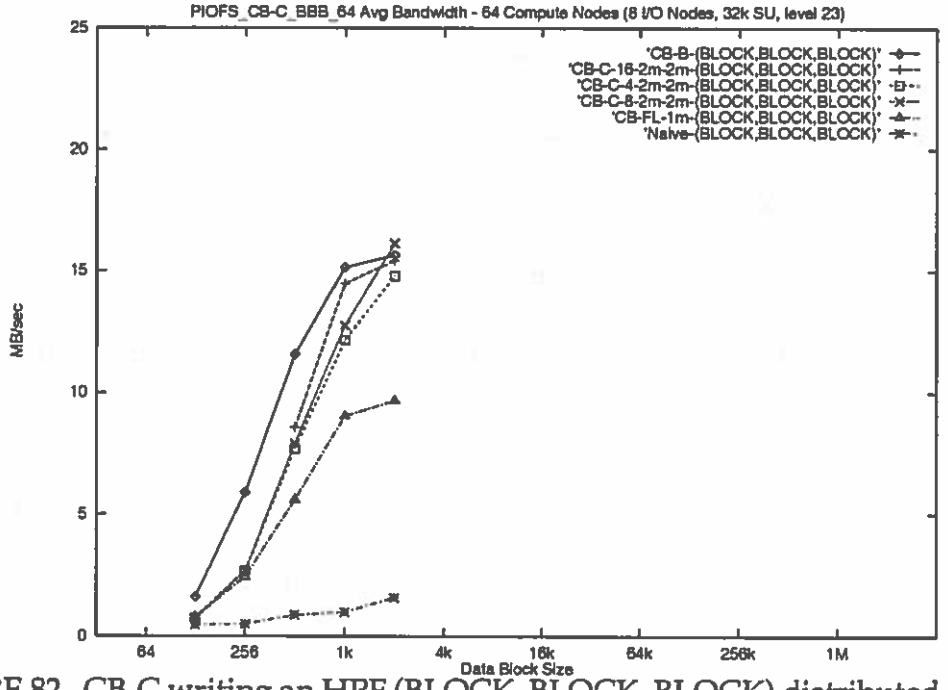

FIGURE 82. CB-C writing an HPF (BLOCK, BLOCK, BLOCK) distributed 128 MB file from 64 nodes on the SP2.

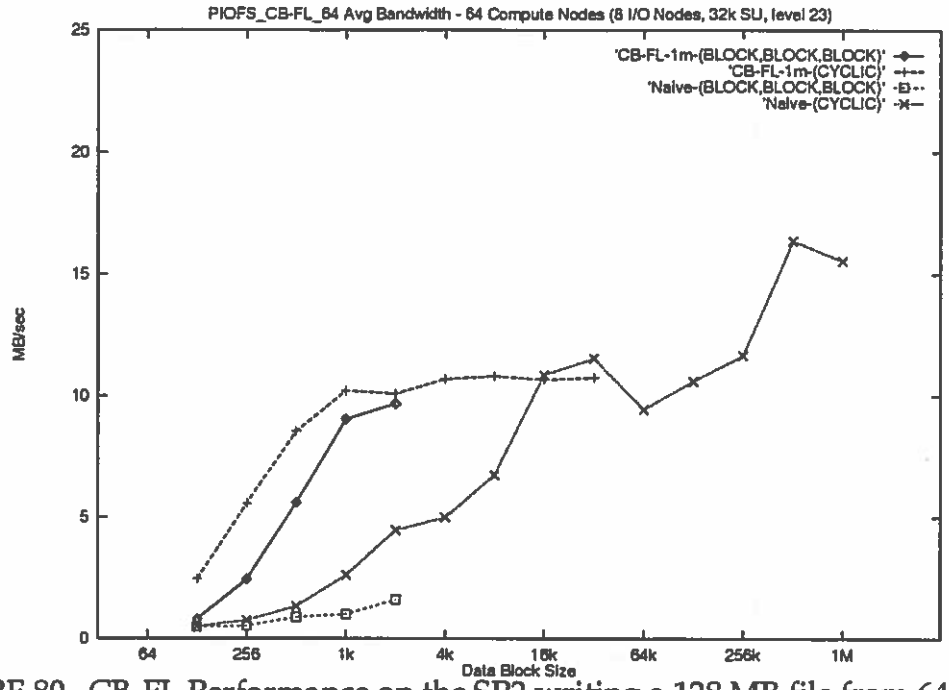

FIGURE 80. CB-FL Performance on the SP2 writing a 128 MB file from 64 nodes.

Collective Buffering with HPF CYCLIC Intermediate Distribution (CB-C)

CB-FL is an ideal alternative to CB-B on the Paragon, able to sustain peak performance for all medium data block sizes, while minimizing the required buffer space. On the SP2, however, CB-FL reached only a little more than half of peak performance. Peak performance was measured writing much larger sized data blocks (greater than 256 kilobytes). This suggests that an alternative intermediate distribution for the SP2 should be used which has larger intermediate blocks than the 32 kilobyte sized intermediate blocks of CB-FL. We generalize CB-FL to allow any HPF CYCLIC intermediate distribution, abbreviated CB-C. Note that both CB-B and CB-FL are special cases of CB-C. CB-B is an instance of CB-C using all compute nodes as target nodes, and a target block size equal to the global data

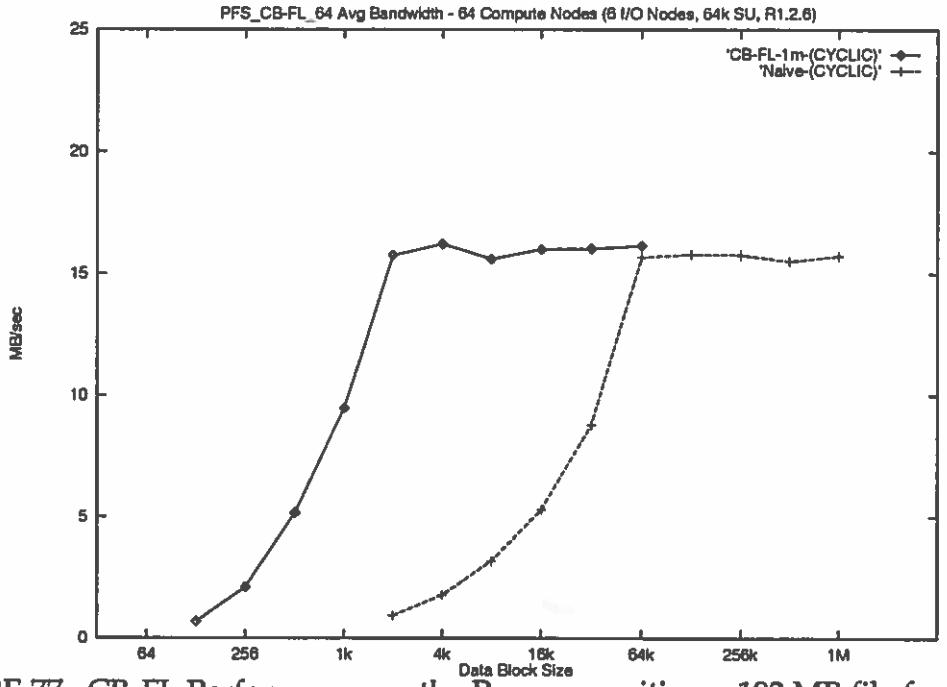

FIGURE 77. CB-FL Performance on the Paragon writing a 192 MB file from 64 nodes.

On the IBM SP2, CB-FL used 8 compute nodes as the target of the permutation, and an HPF CYCLIC(32 kilobyte) intermediate distribution. Each target node reserved a one megabyte buffer, yielding an aggregate buffer space of 8 megabytes. On medium sized data blocks, i.e. 4 kilobytes - 32 kilobytes, using CB-FL on the SP2 (Figure 78 - Figure 80) attained the performance of the Naive algorithm operating on 32 kilobyte data blocks—this is approximately half of peak performance. Again, performance drops off below 4 kilobyte data blocks as seen before.

As with CB-B, CB-FL significantly outperforms the Naive algorithm across a wide range of small and medium data block sizes with greatly reduced buffer space.

On the Intel Paragon, CB-FL used 6 compute nodes as the target of the permutation, and an HPF CYCLIC(64 kilobyte) intermediate distribution. Each target node reserved a one megabyte buffer, for an aggregate buffer space of 6 megabytes. CB-FL on the Paragon (Figure 75 - Figure 77) sustained peak performance on data block sizes as low as 4 kilobytes. Below 4 kilobytes, the performance drops off as seen in CB-B measurements. Buffer space was greatly reduced compared to CB-B, from 192 megabytes, for these tests, to 6 megabytes, with no apparent lack of performance. In addition, running CB-B on a file this large was prevented by the message passing system software which did not support the number of simultaneous messages required for a one-step permutation.

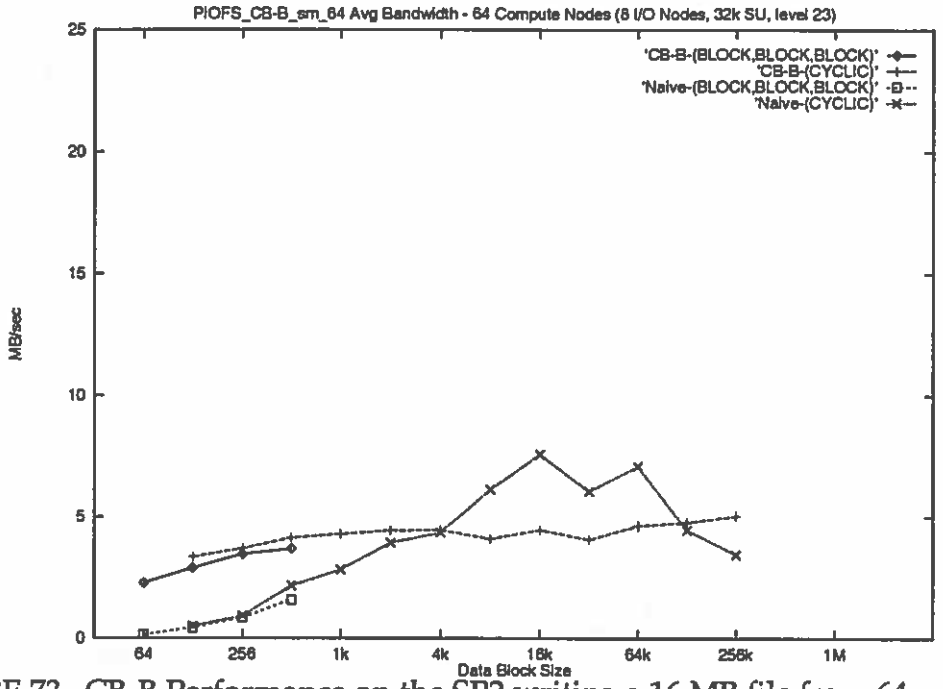

FIGURE 73. CB-B Performance on the SP2 writing a 16 MB file from 64 nodes.

CB-B greatly improves performance for all small block sizes on both the Paragon (for less than 64 kilobyte data blocks) and on the SP2 (for less than 8 kilobyte data blocks). The performance of CB-B is unaffected by the original data distribution, performing nearly identically on both HPF (BLOCK, BLOCK, BLOCK) and the HPF CYCLIC data distributions. Performance when writing data blocks as small as 1 kilobyte in size attains 75% of measured peak performance. However, performance for data blocks smaller than 1 kilobyte is still poor.

Collective Buffering with File Layout Intermediate Distribution (CB-FL)

CB-B works, but the BLOCK distribution requires a significant amount of buffer memory, an amount equal to the size of the global data structure. For the

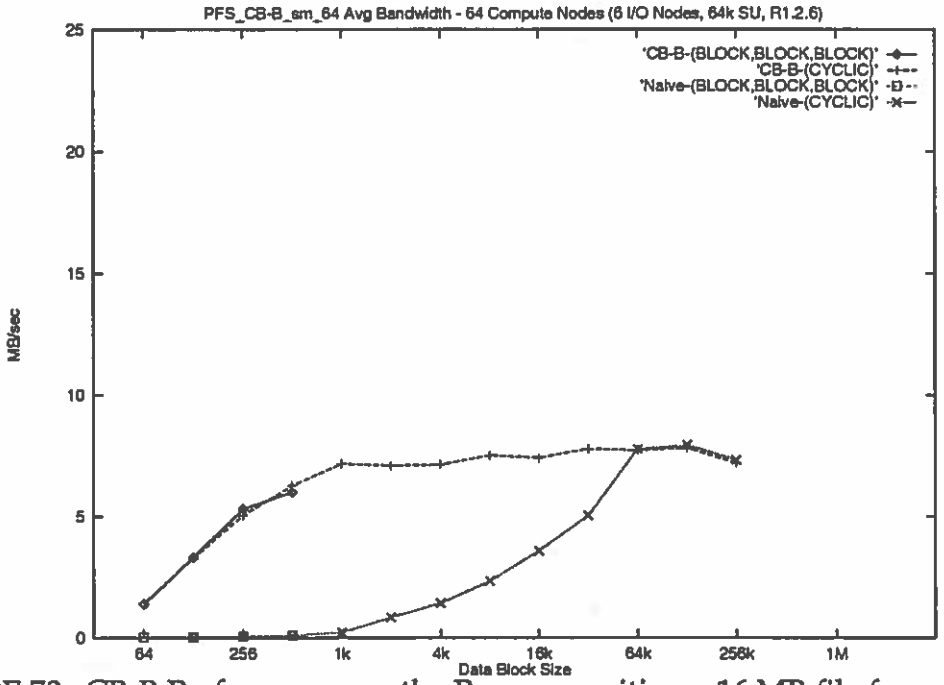

FIGURE 70. CB-B Performance on the Paragon writing a 16 MB file from 64 nodes.

On the IBM SP2 (Figure 71 - Figure 73), CB-B significantly outperforms the Naive algorithm for data block sizes between 64 bytes and 4 kilobytes. Above 4 kilobytes, which is the underlying AIX file system block size, performance is more varied. On 16 compute nodes, CB-B is almost always superior to Naive; on 32 compute nodes, it is comparable to Naive through 64 kilobyte data blocks (the PIOFS stripe unit), then the Naive algorithm is superior; on 64 compute nodes, Naive is superior for data blocks sized from 4 kilobytes to 64 kilobytes, but not above. Generally, CB-B works well for small data block sizes, but for larger data blocks, the overhead of performing the permutation slows CB-B performance. Finally, as on the Paragon, the overhead of message latency causes performance to drop off below one kilobyte data blocks.

On the Intel Paragon (Figure 68 - Figure 70), CB-B significantly outperforms the Naive algorithm for all data block sizes from 64 bytes to 64 kilobytes. Above 64 kilobytes, the data block size is large enough to take full advantage of the PFS file system (which has a 64 kilobyte stripe unit), but even with the added overhead of permuting the data, CB-B still outperforms the Naive algorithm by a small margin for all but two of the data points shown. The performance of CB-B on large data block sizes demonstrates that the larger data access size, the better the performance. Performance using data blocks less than one kilobyte falls off sharply, although it is still superior to the performance of the Naive algorithm. This performance drop is most likely a result of the bandwidth/latency characteristics of message passing on the Paragon. The overhead of message latency begins to dominate as the number of messages sent increases and the amount of data per message decreases.

main computation. Using auxiliary nodes permits true asynchronous operation. The computation can progress as soon as the permutation step has finished without waiting for the I/O to complete. In addition, the ideal amount of buffer space and the ideal number of nodes to use for the permutation can be chosen independent of the application's resource use. However, this approach requires the use of a different, more valuable resource, additional compute nodes, which could have been used for the current or some other application. As the trade-off between using compute nodes or auxiliary nodes does not affect the algorithm, we have restricted our attention to using compute nodes to support the permutation.

We developed and evaluated four collective buffering techniques. First, we look at the simplest approach: using all the compute nodes to permute the data to a simple HPF BLOCK intermediate distribution in a single step. Second, we refine this approach by (a) realistically limiting the amount of buffer space used on compute nodes and (b) using an intermediate distribution which matches the file layout. Third, we consider the case of using a more general HPF CYCLIC intermediate distribution. Finally, we investigate a method for eliminating the latency dominated overhead of the permutation phase by taking advantage of scatter/gather hardware, greatly increasing the applicability of collective buffering to applications which use small data blocks.

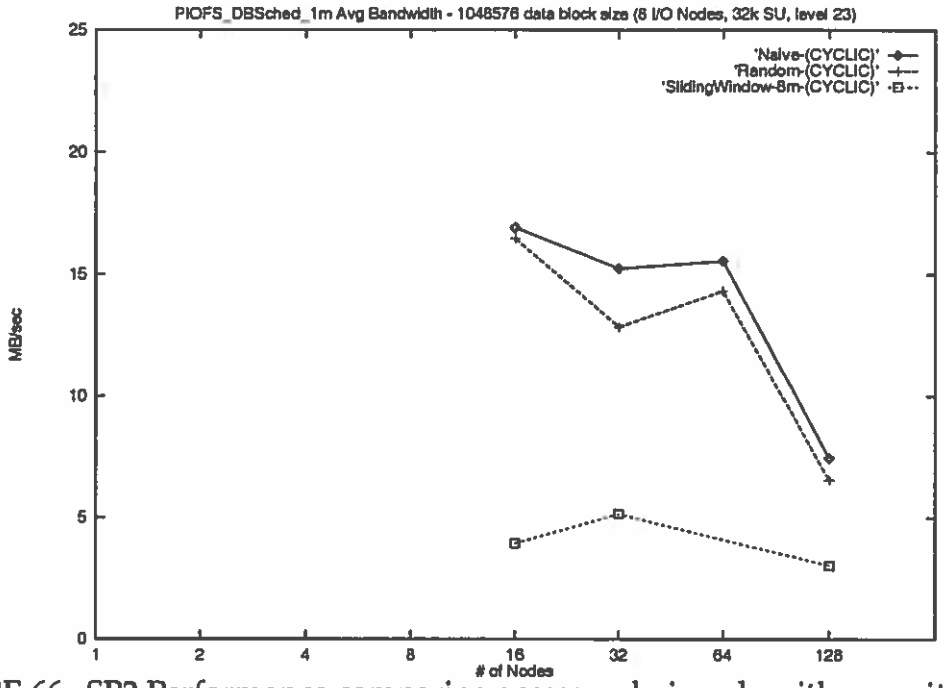

FIGURE 66. SP2 Performance comparing access ordering algorithms writing a 128 MB file using 1 megabyte data blocks.

Data Block Scheduling Summary

Performance of all of the data block scheduling techniques are disappointing. The only performance gains measured were for small data block sizes. The abysmal absolute performance for these data block sizes outweighs the few performance gains. Implementing these transfer algorithms would be unwarranted. As we shall see in the next section, combining data blocks, and increasing the effective data block size is much more promising.

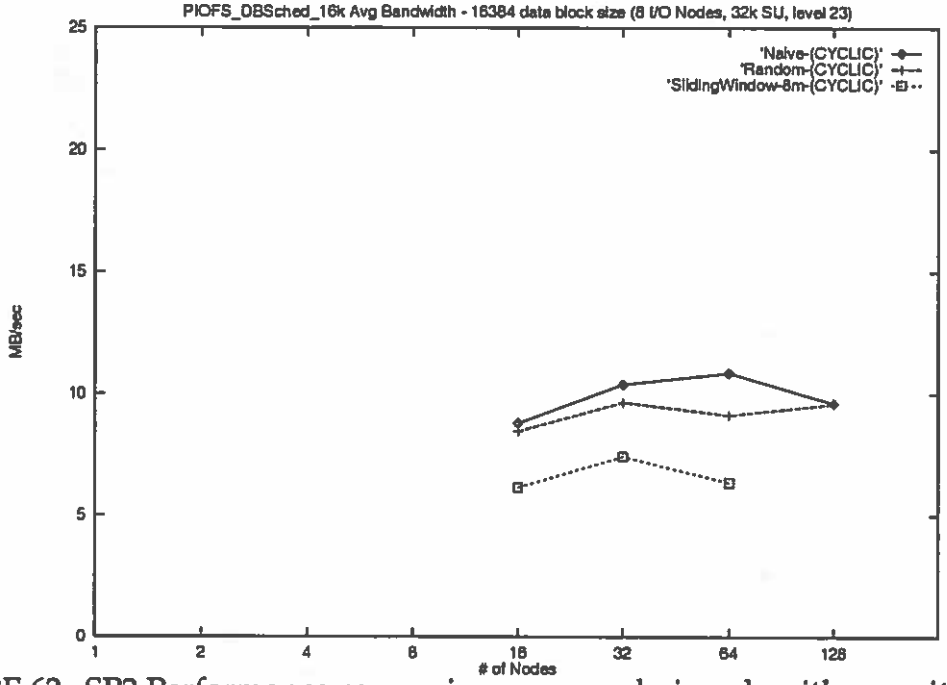

FIGURE 62. SP2 Performance comparing access ordering algorithms writing a 128 MB file using 16 kilobyte data blocks.

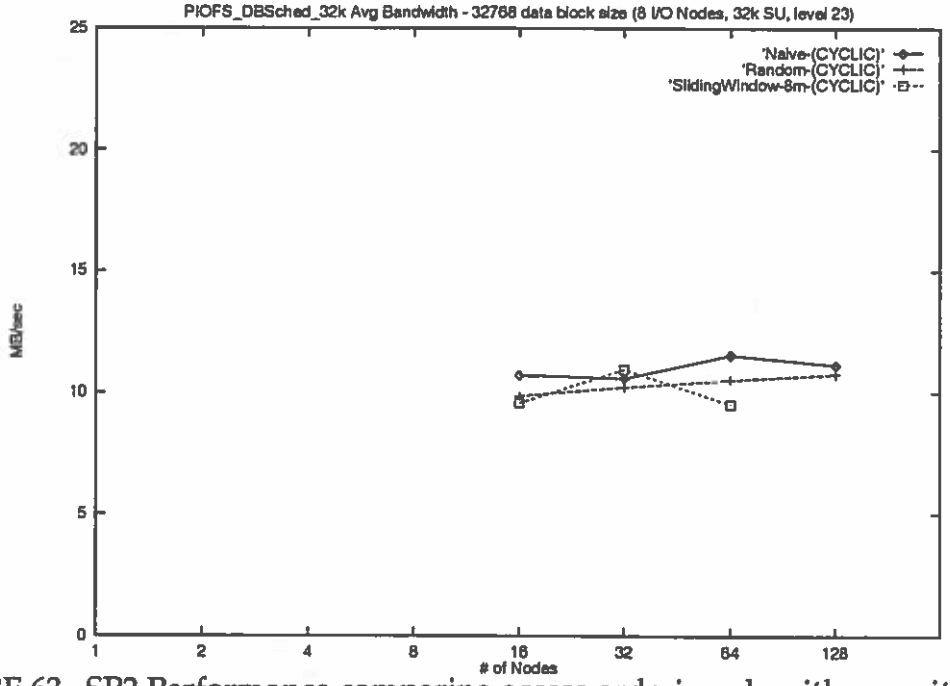

FIGURE 63. SP2 Performance comparing access ordering algorithms writing a 128 MB file using 32 kilobyte data blocks.

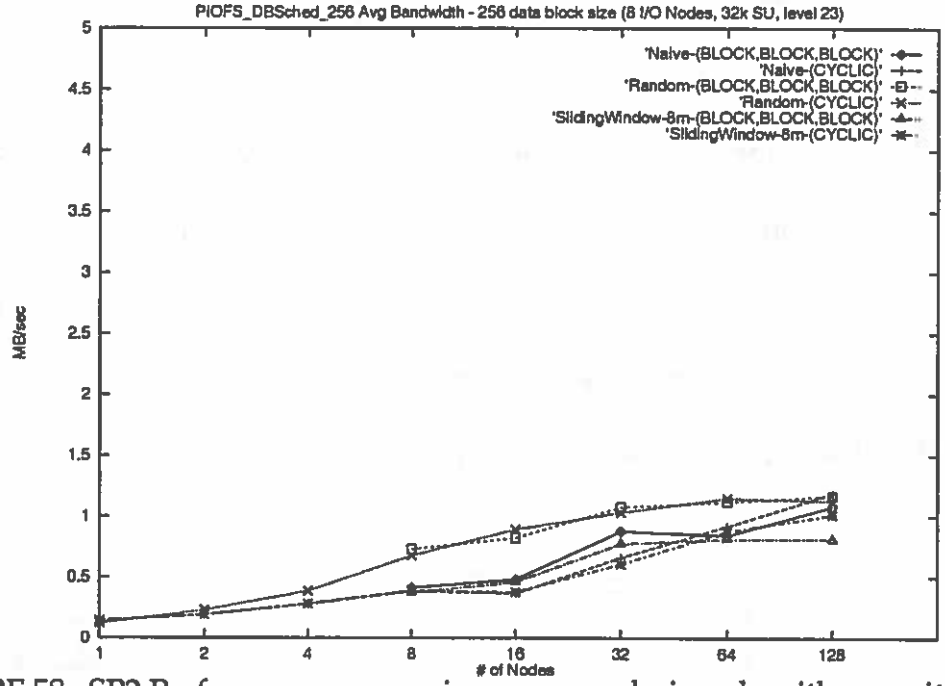

FIGURE 58. SP2 Performance comparing access ordering algorithms writing a 16 MB file using 256 byte data blocks.

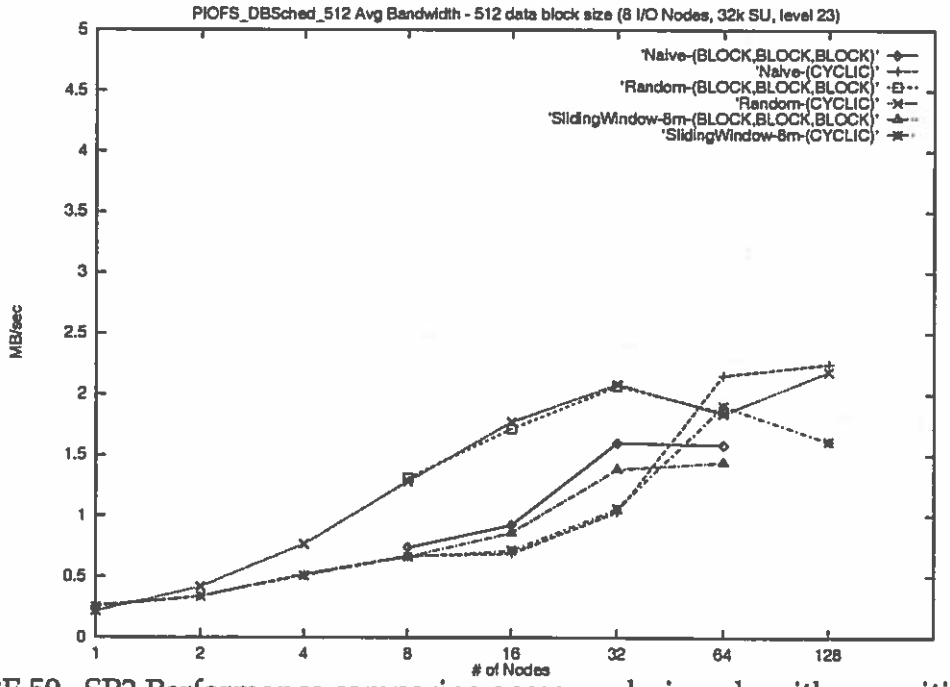

FIGURE 59. SP2 Performance comparing access ordering algorithms writing a 16 MB file using 512 byte data blocks.

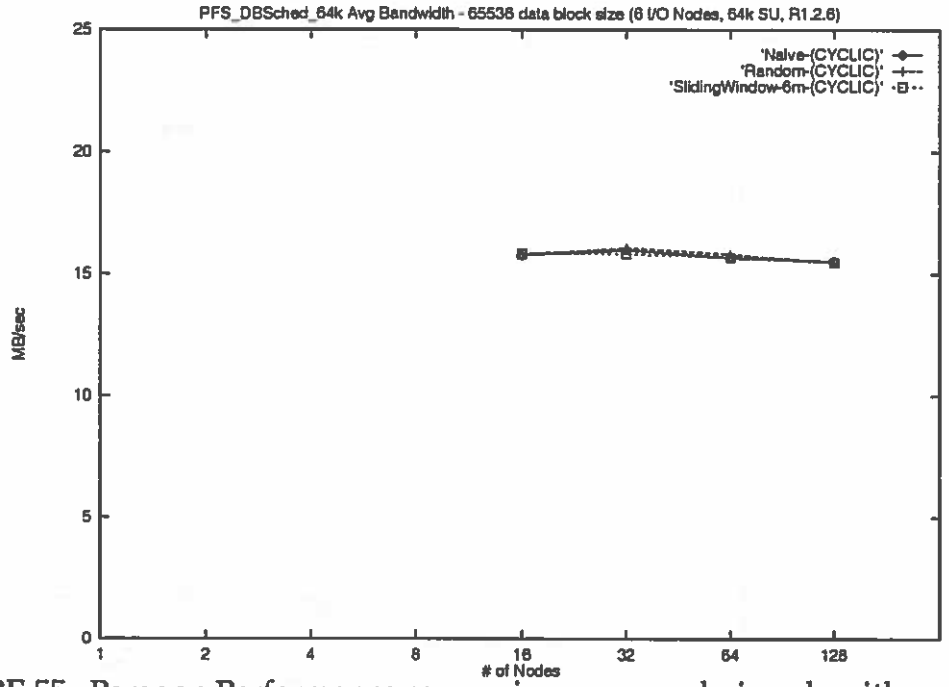

FIGURE 55. Paragon Performance comparing access ordering algorithms writing a 192 MB file using 64 kilobyte data blocks.

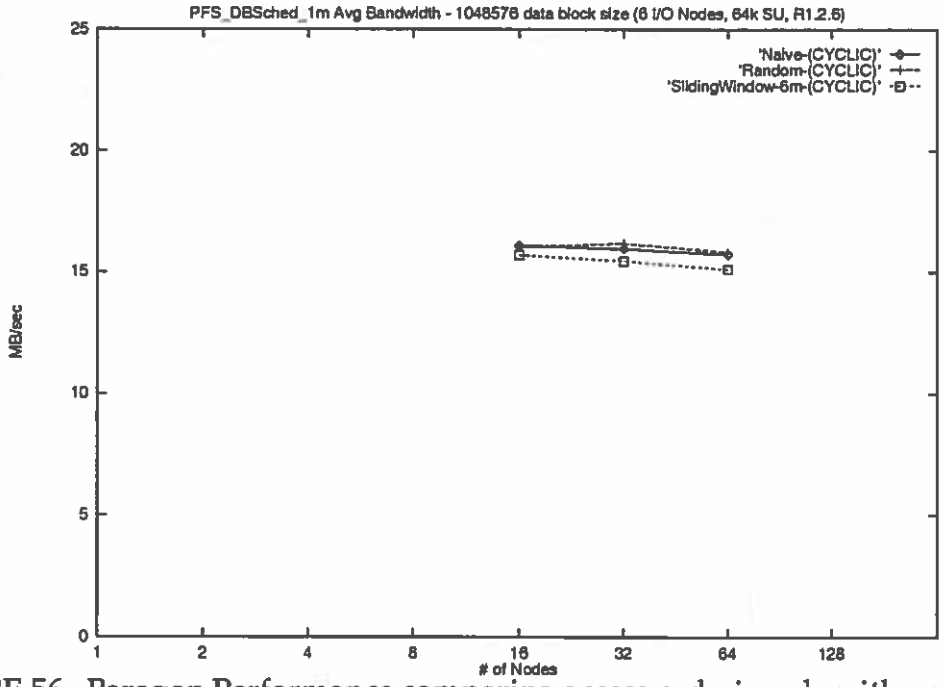

FIGURE 56. Paragon Performance comparing access ordering algorithms writing a 192 MB file using 1 megabyte data blocks.

any file system block caching avoiding costly read/modify/write operations. For large data block sizes, the Sliding Window algorithm should minimize seek time by keeping accesses near each other. Sliding Window, however, may limit I/O parallelism, as its behavior is nearly opposite to the Random transfer algorithm. A Generalized Sliding Window algorithm is proposed in "Future Work" on page 153. This new algorithm slides multiple windows, one for each storage device. This allows maximal parallelism within the I/O subsystem while still taking advantage of locality.

Access Ordering Algorithm Performance

Generally, access ordering, i.e. spatially reorganizing the data blocks, has little effect. Across the full range of data block sizes, Paragon PFS performance is not improved by either of the Random or Sliding Window algorithms. For data block sizes smaller than 64 kilobytes (Figure 53 - Figure 54), the Naive algorithm is best, and for larger block sizes (Figure 55 - Figure 56), performance of the Naive, Random, and Sliding Window algorithms are identical.

Access Ordering Algorithms

Access ordering algorithms schedule access operations on data blocks. The Random algorithm performs file accesses in random order. The Sliding Window algorithm schedules file accesses so that data closely positioned in the canonical file is accessed at roughly the same time.

Random Algorithm

The Random transfer algorithm probabilistically maximizes access parallelism to the I/O subsystem. The Random Algorithm issues writes from each compute node in random order (see Figure 51). The unit of data transfer is the data block.

```
/* N nodes writing */
randomize the list of data blocks
foreach data block rd in randomized list {
      Perform I/O on rd;
}
```

FIGURE 51. Random Algorithm Pseudocode.

The Random transfer algorithm probabilistically eliminates contention for I/O resources (in a similar manner to randomized message passing algorithms which avoid message contention). A highly structured data distribution can lead to all compute nodes accessing a single storage device. For example, an HPF CYCLIC distribution with size equal to the half the file layout stripe unit would have the compute nodes accessing only half of the I/O nodes at the beginning of the I/O

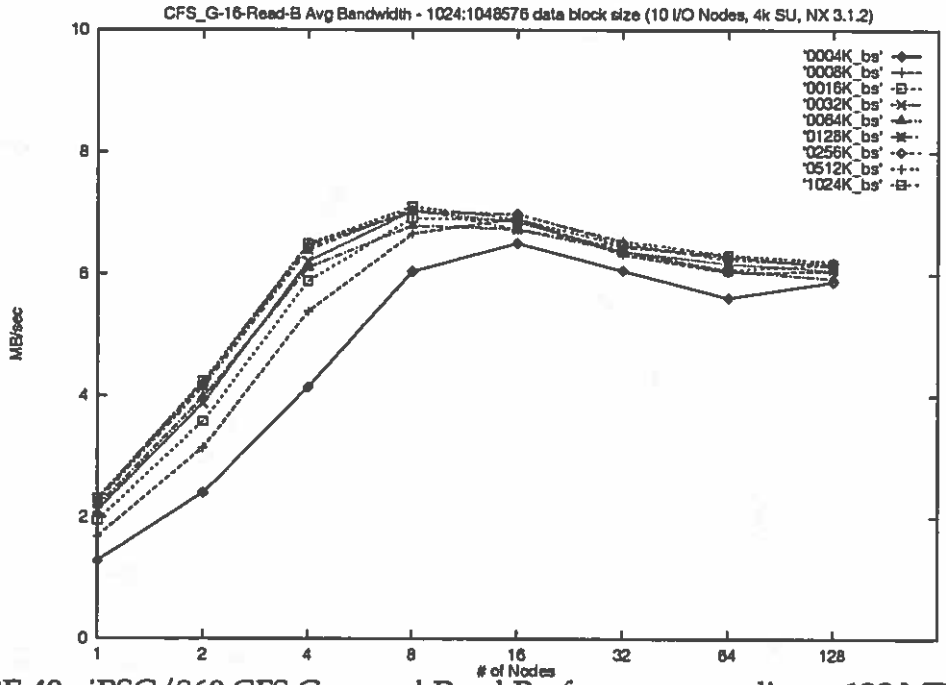

FIGURE 49. iPSC/860 CFS Grouped Read Performance reading a 128 MB file distributed HPF BLOCK.

This type of performance anomaly is not limited to the older technology of the iPSC/860. When first installed at NAS, the performance of the SP2 PIOFS exhibited a similar performance drop for small numbers of nodes when scaling up the data block size above 256 kilobytes (Figure 50).

Grouping is indicated whenever negative scaling occurs: when adding more nodes or accessing larger data blocks causes the aggregate performance to drop. This phenomenon is a tell-tale sign of thrashing, which usually indicates a system design flaw. In essence, grouping is a hack to mask a bug in the I/O system. Grouping may, however, be the easiest way to solve the performance anomaly, and grouping can be easily implemented as part of a collective I/O library without changing the operating system or hardware.

We give two examples where grouping benefits parallel I/O performance: on the iPSC/860 CFS and on the SP2 PIOFS. As seen in Chapter III, the limited memory available on I/O nodes combined with the poor caching policy of the CFS caused a significant aggregate performance drop for all data block sizes using the Naive algorithm and more than 16 compute nodes (Figure 48).

Data Block Scheduling Algorithms

We consider two aspects of scheduling the transfer of the individual data blocks to the I/O nodes:

1. *grouping*—"temporally" coordinating among compute nodes to modify the speed and time at which requests are sent to the I/O nodes, and
2. *access ordering*—"spatially" rearranging the order in which a compute node sends the individual transfer requests to an I/O node.

We developed three algorithms which take advantage of the collective interface via data block scheduling; Grouping (node and data), Random, which employs access ordering, and Sliding Window, which combines both coordination and access ordering.

Grouping Algorithm

Thrashing, induced by resource contention, can be a major source of performance degradation. A simple technique which eliminates the contention, and hence the thrashing, is to limit the parallelism for I/O operations so that limited resources in the file system are not oversubscribed. We call this "grouping". Grouping can limit the amount of control parallelism (node grouping) or limit the amount of data parallelism (data grouping).

Instead of a free-for-all, node grouping works by "coloring" each process and ensuring that processes of at most one color perform I/O at one time (see

Seamons, et. al. For a more detailed description of the MPI-IO interface, see the appendix.

Contributions

We evaluated an early parallel I/O system, the Intel iPSC/860 Concurrent File System, revealing the inherent inefficiencies in the way the parallel I/O problem was attacked. This study led to our development of new interfaces and algorithms for parallel I/O.

1. We devised a coordinated approach to performing global parallel I/O operations—*Collective I/O*—which permits significant optimizations to be performed.
2. We formulated a new model to describe the parallel I/O problem providing insights into the inherent difficulties of efficiently supporting parallel I/O. The key concept introduced in this model is the *data block*.

We designed two classes of algorithms taking advantage of collective parallel I/O: *data block scheduling* and *collective buffering*. In order to measure performance, we performed real tests on real machines (the Intel Paragon and the IBM SP2), and we designed algorithms to work with existing hardware and operating systems. The key results from this work show that:

1. Data distribution is secondary to the amount of data accessed in individual file operations: the larger the accesses, the better the performance.
2. Parallelism, normally the key to performance, can lead to thrashing caused by resource contention. Limiting the parallelism (via *Grouping*) can improve performance by as much as a factor of eight.

but do not guarantee disk parallelism, and disk orderings maximize disk parallelism, but may still bottleneck by sending to a single compute node at a time (contention effects). The ideal algorithm would ensure maximal parallelism at both the compute nodes and I/O nodes simultaneously. (Note that Random probabilistically comes close.)

Over the life of a parallel application, the system environment (number of running jobs, network usage, contention for disks, etc.) can change dramatically. It should be possible to design dynamically self tuning algorithms which adjust to this changing environment. For example, one could modify the Grouping algorithm to change the group size based on current resource contention. Self tuning algorithms would also be particularly useful for new architectures or new configurations of existing machines. Rather than re-running a rack of experiments to determine the optimal algorithm for the new architecture, the algorithms could converge to the best solution on-the-fly.

The Sliding Window algorithm performed poorly, despite our intuition that it would improve performance. We conjecture that the algorithm did not perform well because we slid a single window over the canonical file, which may have limited the amount of parallelism available at both the compute nodes and the I/O nodes! This unintentional parallelism reduction could be eliminated by a generalization of our original Sliding Window algorithm. Rather than slide a single window, the generalized algorithm would have D separate windows, one for each I/O node. The windows would slide over the file chunks rather than the

nodes. The synchronized nature of collective I/O may make it easier to use RAID technology efficiently.

The processing nodes of parallel systems are usually segregated into compute nodes and I/O nodes. It is unknown whether the best use of resources is to dedicate nodes to perform I/O only, or to allow nodes to share compute and I/O responsibilities. The tradeoffs between both approaches needs to be investigated. Perhaps a dynamic approach in which the system responds to application requirements for compute and I/O requirements by reconfiguring would be the best solution. Some preliminary work in this area has been started by Kotz [87].

All of this work deals with supporting a single parallel application at a time. The real key to making parallel systems ready for prime time is supporting not just a single application, but supporting all of the applications within a workload simultaneously. The collective interface defines ideal boundaries for I/O activity. These boundaries present the opportunity to schedule I/O activity across a whole system.

part of a parallel I/O environment if it supports a high-level interface to describe the partitioning of file data among processes and a collective interface describing complete transfers of global data structures between process memories and the file. Further efficiencies can be gained via support for asynchronous I/O, allowing computation to be overlapped with I/O, and control over physical file layout on storage devices (disks).

Parallel file systems and programming environments have typically solved the problems of data partitioning and collective access by introducing file modes. The different modes specify the semantics of simultaneous operations by multiple processes. Once a mode is defined, conventional read and write operations are used to access the data, and their semantics are determined by the mode. The most common modes are described in Table 6 [12, 75, 119].

TABLE 6. Common "File Modes"

File Mode	Description	Examples
broadcast reduce	all processes collectively access the same data	PFS global CMMD sync-broadcast
scatter gather	all processes collectively access a sequence of data blocks, in rank order	CFS modes 2 and 3 PFS sync &record CMMD sync-sequential
shared offset	processes operate independently, but share a common file pointer	CFS mode 1 PFS log mode
independent	allows programmer complete freedom	CFS mode 0 PFS UNIX mode CMMD local & independent

be plugged-in, or defined procedurally, in order to evaluate (and optimize) parallel I/O performance.

Vesta [36, 37, 40, 59] is a library-based parallel file system which runs on the IBM SP2. It provides user-defined parallel views of files for data partitioning, collective operations for data access, and asynchronous operations. Vesta is designed to scale to hundreds of compute nodes, with no sequential bottlenecks in the data-access path.

Jovian [10], PASSION [31, 129], and VIP-FS [51] target out-of-core algorithms. Panda [124, 125, 126] supports a collective global array interface, and optimizes file access by making the file layout correspond to the global array distribution—a 3D array is stored in 3D "chunks" in the file.

Overview of MPI-IO

The goal of the MPI-IO interface is to provide a widely used standard for describing parallel I/O operations within an MPI message-passing application. The interface establishes a flexible, portable, and efficient standard for describing independent and collective file I/O operations by processes in a parallel application. In a nutshell, MPI-IO is based on the idea that I/O can be modeled as message passing: writing to a file is like sending a message, and reading from a file is like receiving a message. MPI-IO intends to leverage the relatively wide acceptance of the MPI interface in order to create a similar I/O interface. The MPI-IO interface is intended to be submitted as a proposal for an extension of the MPI

Emphasis has been put in keeping MPI-IO as MPI-friendly as possible. When opening a file, a communicator is specified to determine which group of processes can get access to the file in subsequent I/O operations. Accesses to a file can be independent (no coordination between processes takes place) or collective (each process of the group associated with the communicator must participate to the collective access). MPI derived datatypes are used for expressing the data layout in the file as well as the partitioning of the file data among the communicator processes. In addition, each read/write access operates on a number of MPI objects which can be of any MPI basic or derived datatype.

Data Partitioning in MPI-IO

Instead of defining file access modes in MPI-IO to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), we chose another approach which consists of expressing the data partitioning via MPI derived datatypes. Compared to a limited set of pre-defined access patterns, this approach has the advantage of added flexibility and expressiveness.

MPI derived datatypes are used in MPI to describe how data is laid out in the user's buffer. We extend this use to describe how the data is laid out in the file as well. Thus we distinguish between two (potentially different) derived datatypes that are used: the filetype, which describes the layout in the file, and the buftype, which describes the layout in the user's buffer. In addition, both filetype and buftype are derived from a third MPI datatype, referred to as the elementary

MPI-IO provides filetype constructors to help the user create complementary filetypes for common distribution patterns, such as broadcast/reduce, scatter/gather, and HPF distributions. In general, we expect most MPI-IO programs will use filetype constructors exclusively, never needing to generate a complicated MPI derived datatype by hand.

FIGURE 91. Partitioning a file among parallel processes.

In MPI-IO, the filetype and etype are specified at file open time. This is the middle ground between specifying the data layout during file creation (or file-system creation) and during data access (read/write). The former is too restrictive, as it prohibits accessing a file using multiple patterns simultaneously. In addition, static data layout information must be stored as file meta-data, inhibiting file portability between different systems. Specifying the filetype at data access time is cumbersome, and it is expected that filetypes will not be changed too often.

In order to better illustrate these concepts, consider a 2-D matrix, stored in row major order in a file, that is to be transposed and distributed among a group of three processes in a row cyclic manner (see Figure 92). The filetypes implement

Note that using MPI derived datatypes leads to the possibility of very flexible patterns. For example, the filetypes need not distribute the data in rank order. In addition, there can be overlaps between the data items that are accessed by different processes. The extreme case of full overlap is the broadcast/reduce pattern.

Using the filetype allows a certain access pattern to be established. But it is conceivable that a single pattern would not be suitable for the whole file. The MPI-IO solution is to define a displacement from the beginning of the file, and have the access pattern start from that displacement. Thus if a file has two or more segments that need to be accessed in different patterns, the displacement for each pattern will skip over the preceding segment(s). This mechanism is also particularly useful for handling files with some header information at the beginning (see Figure 93). Use of file headers could allow the support of heterogeneous environments by storing a "standard" codification of the file data.

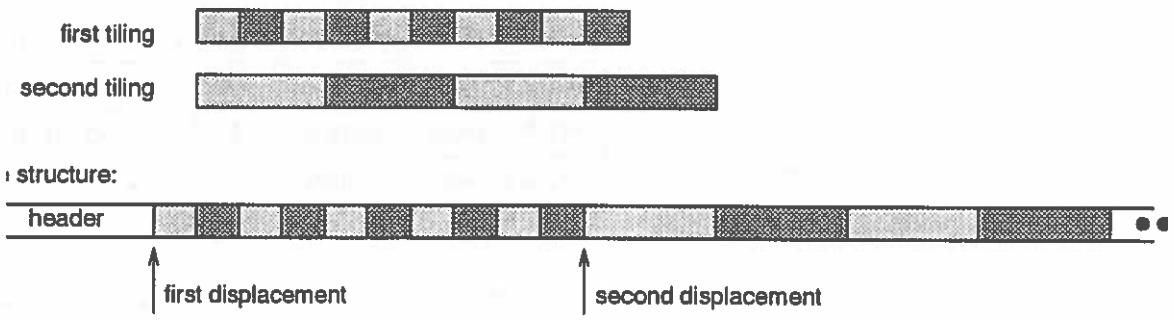

FIGURE 93. Displacements.

Positioning

UNIX file systems traditionally maintain a system file pointer specifying what offset will be used for the read or write operation. The problem with this interface is that it was primarily designed for files being accessed by a single process. In a parallel environment, we must decide whether a file pointer is shared by multiple processes or if an individual file pointer will be maintained by each process. In addition, parallel programs do not generally exhibit locality of reference within a file [88]. Instead, they tend to move between distinct non-contiguous regions of a file. This means that the process must seek on almost every read or write operation. In addition, in multi-threaded environments or when performing I/O asynchronously, it is difficult to ensure that the file pointer will be in the correct position when the read or write occurs.

MPI-IO provides separate functions for positioning with explicit offsets, individual file pointers, and a shared file pointer. The explicit offset operations require the user to specify an offset, and act as atomic seek and read or seek and write operations. The individual and shared file pointer operations use the implicit system maintained offsets for positioning. The different positioning methods are orthogonal; they may be mixed within the same program, and they do not affect each other. In other words, an individual file pointer's value will be unchanged by executing explicit offset operations or shared file pointer operations. The MPI-IO data access functions which accept explicit offsets have no

where in the file, so they can also point to an item that is inaccessible by this process. In this case, the offset will be advanced automatically to the next accessible item. Therefore specifying any offset in a hole is functionally equivalent to specifying the offset of the first item after the hole. A relative offset is one that only includes the parts of a file accessible by this process, excluding the holes of the filetype associated with the process.

Absolute offsets may be easier to understand if accesses to arbitrary random locations are combined with partitioning the file among processes using filetypes. If such random accesses are not used, relative offsets may be preferable. If the file is not partitioned (all filetypes are identical), absolute and relative offsets are the same.

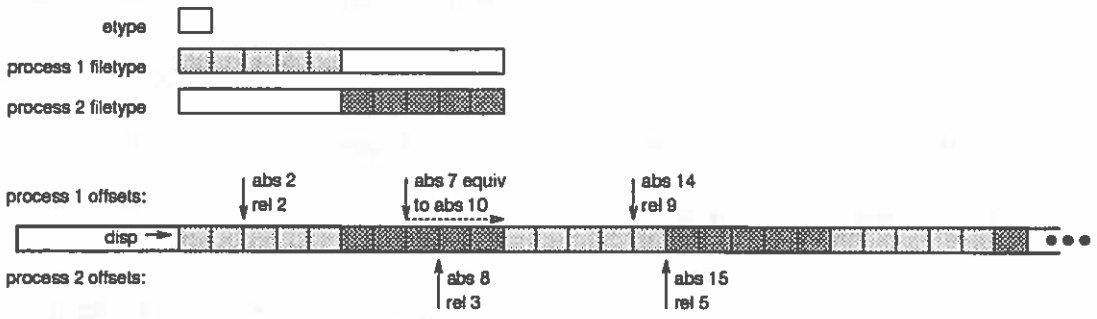

FIGURE 94. Absolute and relative offsets.

File Pointers

When a file is opened in MPI-IO, the system creates a set of file pointers to keep track of the current file position. One is a global file pointer, "shared" by all

be different from the amount requested), and the file pointer is updated by that amount. When MPI-IO non-blocking accesses are made using an individual or the shared file pointer, the update cannot be delayed until the operation completes, because additional accesses can be initiated before that time by the same process (for both types of file pointers) or by other processes (for the shared file pointer). Therefore, the file pointer must be updated at the outset, by the amount of data requested.

Similarly, when blocking accesses are made using the shared file pointer, updating the file pointer at the completion of each access would have the same effect as serializing all blocking accesses to the file. In order to prevent this, the shared file pointer for blocking accesses is updated at the beginning of each access by the amount of data requested. For blocking accesses using an individual file pointer, updating the file pointer at the completion of each access would be perfectly valid. However, in order to maintain the same semantics for all types of accesses using file pointers, the update of the file pointer in this case is also made at the beginning of the access by the amount of data requested.

Although consistent, and semantically cleaner, updating the file pointer at the initiation of all I/O operations differs from accepted UNIX practice, and may lead to unexpected results. Consider the scenario in Figure 95.

safe for the user to reuse the buffer. With suitable hardware, the transfer of data out/in the user's buffer may proceed concurrently with computation.

Note that just because a non-blocking (or blocking) data access function completes does not mean that the data is actually written to "permanent" storage. All of the data access functions may buffer data to improve performance. The only way to guarantee data is actually written to storage is by using the MPIO_File_sync call. However, one need not be concerned with the converse problem—once a read operation completes, the data is always available in the user's buffer.

Coordination

Global data accesses have significant potential for automatic optimization, provided the I/O system can recognize an operation as a global access. Collective operations are used for this purpose. MPI-IO provides both independent and collective versions of all data access operations. Every independent data access function MPIO_xxx, has a collective counterpart MPIO_xxx_all, where "_all" means that all processes in the communicator group which opened the file must participate.

Independent calls do not imply any coordination among processes. Collective calls imply that all processes belonging to the communicator associated with the opened file must participate. However, as in MPI, no synchronization pattern between those processes is enforced by the MPI-IO definition. Any required syn-

Miscellaneous Features

File Layout in MPI-IO

MPI-IO is intended as an interface that maps between data stored in memory and a file. Therefore, the basic access functions only specify how the data should be laid out in a virtual file structure (the file type), not how that file structure is to be stored on one or more disk. This was avoided because it is expected that the mapping of files to disks will be system specific, and any specific control over file layout would therefore restrict program portability. However, there are still cases where some information will be necessary in order to optimize disk layout. MPI-IO allows a user to provide this information as hints specified when a file is created. These hints do not change the semantics of any of the MPI-IO interfaces, instead they are provided to allow a specific implementation to increase I/O throughput. However, the MPI-IO standard does not enforce that any of the hints will be used by any particular implementation.

Read/Write Atomic Semantics

When concurrent data accesses involve overlapping data blocks, it is desirable to guarantee consistent interleaving of the accesses. For example, the UNIX read/write interface provides atomic access to files. Suppose process A writes a 64K block starting at offset 0, and process B writes a 32K block starting at offset 32K (see Figure 96). The resulting file will have the 32K overlapping block (start-

Note that the cautious mode only guarantees atomicity of accesses within an MPI application, not between two different MPI applications accessing the same file data. Therefore, its effect is limited to the confines of the MPI_COMM_WORLD communicator group of the processes that opened the file, typically all the processes in the job.

Current Status and Future Developments

Currently, four implementations of MPI-IO are in progress. NASA Ames is working on a portable implementation, primarily targeted at workstation clusters, IBM Research is working on an implementation for the SP2 built on top of IBM's PIOFS file system, and Lawrence Livermore National Laboratory is working on implementations for the Cray T3D and Meiko CS-2. Prototypes for the IBM and NASA implementations are targeted for completion by May 15, 1995, will full versions by the end of the year.

General information, copies of the latest draft, and an archive of the MPI-IO mailing list, can be obtained via the WWW at:

http://lovelace.nas.nasa.gov/MPI-IO/

To join the MPI-IO mailing list, send your request to mpi-io-request@nas.nasa.gov (see the Web page for details).

[9] S. J. Baylor and C. E. Wu. Parallel I/O workload characteristics using Vesta. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[10] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. Saltz. Jovian: A framework for optimizing parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–20. IEEE Computer Society Press, October 1994.

[11] B. Bershad, D. Black, D. DeWitt, G. Gibson, K. Li, L. Peterson, and M. Snir. Operating system support for high-performance parallel I/O systems. Technical Report CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[12] M. L. Best, A. Greenberg, C. Stanfill, and L. W. Tucker. CMMD I/O: A parallel Unix I/O. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 489–495, 1993.

[13] R. Bordawekar and A. Choudhary. HPF with parallel I/O extensions. Technical Report SCCS-613, NPAC, Syracuse University, 1993.

[14] R. Bordawekar and A. Choudhary. Compiler and runtime support for parallel I/O. In *Proceedings of IFIP Working Conference (WG10.3) on Programming Environments for Massively Parallel Distributed Systems*, Monte Verita, Ascona, Switzerland, April 1994. Birkhaeuser Verlag AG, Basel, Switzerland.

[15] R. Bordawekar and A. Choudhary. Communication strategies for out-of-core programs on distributed memory machines. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 395–403, Barcelona, July 1995.

[16] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995. Also available as the following technical reports: NPAC Technical Report SCCS-0696, CRPC Technical Report CRPC-TR94507-S, SIO Technical Report CACR SIO-104.

[17] R. Bordawekar, A. Choudhary, and J. M. D. Rosario. An experimental performance evaluation of Touchstone Delta Concurrent File System.

[27] R. Carter, B. Ciotti, S. Fineberg, and B. Nitzberg. NHT-1 I/O benchmarks. Technical Report RND-92-016, NAS Systems Division, NASA Ames, November 1992.

[28] K. Castagnera, D. Cheng, R. Fatoohi, E. Hook, W. T. Kramer, C. Manning, J. Musch, C. Niggley, W. Saphir, D. Sheppard, M. Smith, I. Stockdale, S. Welch, R. Williams, and D. Yip. NAS experience with a prototype cluster of workstations. In *Proceedings of Supercomputing '94*, pages 410–419, November 1994.

[29] *Convex Exemplar Scalable Parallel Processing System*. Convex Computer Corporation, 1994. Order number 080-002293-000.

[30] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[31] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[32] A. Choudhary, R. Bordawekar, S. More, K. Sivaram, and R. Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.

[33] A. Choudhary, I. Foster, G. Fox, K. Kennedy, C. Kesselman, C. Koelbel, J. Saltz, and M. Snir. Languages, compilers, and runtime systems support for parallel input-output. Technical Report CCSF-39, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[34] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, April 1995.

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

[46] E. DeBenedictis and P. Madams. nCUBE's parallel I/O with Unix capability. In *Proceedings of the Sixth Annual Distributed-Memory Computer Conference*, pages 270–277, 1991.

[47] E. P. DeBenedictis and S. C. Johnson. Extending Unix for scalable computing. *IEEE Computer*, 26(11):43–53, November 1993.

[48] J. M. del Rosario. High performance parallel I/O on the nCUBE 2. *Transactions of the Institute of Electronics, Information and Communications Engineers*, J75D-I(8):626–636, August 1992.

[49] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[50] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[51] J. M. del Rosario, M. Harry, and A. Choudhary. The design of VIP-FS: A virtual, parallel file system for high performance parallel and distributed computing. Technical Report SCCS-628, NPAC, Syracuse, NY 13244, May 1994.

[52] P. Dibble, M. Scott, and C. Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[53] B. Duzett and R. Buck. An overview of the nCUBE 3 supercomputer. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 458–464, 1992.

[54] T. A. El-Ghazawi. Characteristics of the MasPar parallel I/O system. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 265–272, 1995.

[55] C. Ellis and P. Dibble. An interleaved file system for the Butterfly. Technical Report CS-1987-4, Dept. of Computer Science, Duke University, January 1987.

[65] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, April 1989.

[66] A. S. Grimshaw and E. C. Loyot, Jr. ELFS: object-oriented extensible file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, page 177, 1991.

[67] A. S. Grimshaw and J. Prem. High performance parallel file objects. In *Proceedings of the Sixth Annual Distributed-Memory Computer Conference*, pages 720–723, 1991.

[68] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.

[69] M. Harry, J. M. del Rosario, and A. Choudhary. VIP-FS: A VIrtual, Parallel File System for high performance parallel and distributed computing. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 159–164, April 1995. Also appeared in ACM Operating Systems Review 29(3), July 1995 pages 35–48.

[70] K. Herbst. Trends in mass storage: vendors seek solutions to growing I/O bottleneck. *Supercomputing Review*, pages 46–49, March 1991.

[71] W. D. Hillis and L. W. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[72] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995.

[73] J. V. Huber, Jr. PPFS: An experimental file system for high performance parallel input/output. Master's thesis, Department of Computer Science, University of Illinois at Urbana Champaign, February 1995.

[74] IBM 9076 Scalable POWERparallel 1: General information. IBM brochure GH26-7219-00, February 1993.

[86] D. Kotz. Expanding the potential for disk-directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, October 1995. To appear. Currently available as Dartmouth PCS-TR95-254.

[87] D. Kotz and T. Cai. Exploring the use of I/O nodes for computation in a MIMD multiprocessor. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–89, April 1995.

[88] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proceedings of Supercomputing '94*, pages 640–649, November 1994.

[89] D. Kotz and N. Nieuwejaar. File-system workload on a scientific multiprocessor. *IEEE Parallel and Distributed Technology*, pages 51–60, Spring 1995.

[90] J. Krystynak. I/O performance on the Connection Machine DataVault system. Technical Report RND-92-011, NAS Systems Division, NASA Ames, May 1992.

[91] J. Krystynak and B. Nitzberg. Performance characteristics of the iPSC/860 and CM-2 I/O systems. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 837–841, 1993.

[92] T. T. Kwan and D. A. Reed. Performance of the CM-5 scalable file system. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 156–165, July 1994.

[93] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[94] Parallel file I/O routines. MasPar Computer Corporation, 1992.

[95] Computing surface CS-2: Technical overview. Meiko brochure S1002-10M115.01A, 1993.

[96] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, May 1994.

[97] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

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

[109] W. J. Nitzberg. *Collective Parallel I/O*. PhD thesis, University of Oregon, December 1995.

[110] W. Oed. The Cray Research massively parallel processor system CRAY T3D. Technical report, Cray Research GmbH, M"unchen, Germany, November 15 1993.

[111] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[112] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160. Golden Gate Enterprises, Los Altos, CA, March 1989.

[113] J. T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[114] T. W. Pratt, J. C. French, P. M. Dickens, and S. A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166. Golden Gate Enterprises, Los Altos, CA, 1989.

[115] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

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

[117] D. A. Reed, C. Catlett, A. Choudhary, D. Kotz, and M. Snir. Parallel I/O: Getting ready for prime time. *IEEE Parallel and Distributed Technology*, Summer 1995. Edited transcript of panel discussion at the 1994 International Conference on Parallel Processing.

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

[130] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. Technical Report CACR-103, Scalable I/O Initiative, Center for Advanced Computing Research, Caltech, June 1995. Submitted to the Special issue of *Scientific Programming/* on Implementations of HPF.

[131] Thinking Machines Corporation. *Programming the CM I/O System*, November 1990.

[132] Connection Machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines, April 1987.

[133] *The Connection Machine CM-5 Technical Summary.* Thinking Machines Corporation, October 1991.

[134] CM-5 scalable disk array. Thinking Machines Corporation glossy, November 1992.

[135] R. F. Van der Wijngaart. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. In *Proceedings of Supercomputing '93*, pages 102–111, November 1993.

[136] R. F. Van der Wijngaart, T. Phung, and E. Barszcz. Three implementations of the NAS scalar penta-diagonal benchmark. submitted for presentation at Supercomputing '94, November 1994.

[137] S. Weeratunga. personal communiation, 1991. Applied Algorithms Group, NASA Ames Research Center.

[138] P. Wong, B. Nitzberg, S. Fineberg, and B. Traversat. PMPIO: A portable MPI-IO library. Technical report, NAS, NASA Ames Research Center, Moffett Field, CA 94035-1000, 1995. to appear.

[139] A. Woo. personal communiation, September 1995. Applied Algorithms Group, NASA Ames Research Center.

[140] T. S. Woodrow. Heirarchical storage management system evaluation. Technical Report RND-93-014, NAS, NASA Ames Research Center, Moffett Field, CA 94035-1000, August 1993.