User-Controllable Input and Output for Parallel Computers

by

Jang Sun Lee
B.E., Kyungpook National University, 1983
M.S., Korea Advanced Institute of Science and Technology, 1985

Abstract of Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate School of Syracuse University
June, 1997

Approved ____________________

Date ____________________

ABSTRACT

The control of I/O by traditional file systems and virtual memory systems can result in substantial performance degradation in the parallel computing environments of many scientific applications. This is because the access patterns are substantially different from those of a sequential computing environment. We believe that programmers are better able than systems to reduce I/O time through explicit I/O management for such environments. Thus, to achieve high performance, file systems or I/O mechanisms for parallel machines should be user-controllable in the sense that programmers should be able by considering the access patterns to determine the file structure, control the distribution of data blocks on physical disks, and easily present a variety of access patterns with a minimum number of I/O requests. However, the majority of current file systems for parallel computers are based on a linear file model and do not support user-controllable features.

In this dissertation we have designed a software, `UPIO` (`U`ser-controllable software for `P`arallel `I`nput and `O`utput), that is intended to efficiently support user controllability and also maximize I/O performance for scientific applications on MIMD multicomputers. An important part of UPIO is "logical disk," which can be defined by programmers based on data domain. The logical disk extends the abstraction of a linear file model into an n-dimensional file model and removes the limitations of the linear file model. Programmers have total control over the logical disk, making it possible to control the layout of data blocks across disks and aggregating disk bandwidth through UPIO's interfaces. These interfaces are based on a logical disk, which allows programmers to present a variety of access patterns with a minimum number of I/O operations.

UPIO is extensively evaluated with some synthetic access patterns and also with some scientific computations, for which we have also designed new external

algorithms. The new algorithms are based on a logical disk and demonstrate that programmers can use UPIO to produce high-performance external algorithms with relative ease.

User-Controllable Input and Output for Parallel Computers

by

Jang Sun Lee
B.E., Kyungpook National University, 1983
M.S., Korea Advanced Institute of Science and Technology, 1985

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate School of Syracuse University
June, 1997

Approved ______________________________

Date ________________________________

ACKNOWLEDGMENT

First, I would like to thank my advisor, Dr. Sanjay Ranka, who helped me find an interesting and challenging research area, and who gave me direction and guidance when I needed it for the successful completion of my studies at Syracuse University. I also appreciate the interest shown in my research by my thesis committee members, Drs. P. Bruce Berra, Salim Hariri, Shiu-Kai Chin, Kishan Mehrotra, and Chao-Wei Ou. I especially thank Dr. Berra, who has been a constant source of encouragement and support during the course of this work. I am also very thankful to Electronics and Telecommunications Research Institute in Korea for its support during my doctoral studies.

During this study, there have been a number of people with whom I have had extensive discussions that helped me refine my ideas and thesis. Those people are Jung Min Kim, Sung-Hoon Ko, Sang-Gue Oh, Sung-Young Park, and Kyoungro Yoon.

I would like to thank Mrs. Elaine Weinman, who spent many hours reading this thesis. Her exceptional vocabulary greatly enhanced the quality of this dissertation. I also thank Mr. Peter Crockett for his excellent computer system support.

I acknowledge the support my family provided throughout my years at Syracuse University. Most important, I thank my dear wife, Hyun Sook, and my wonderful daughter, Hae Nim, for their endurance and patience. I also thank my American "parents," Owen and Jean Roberts, for being instrumental in helping my family find a wonderful life in America and for their prayers for my family.

Finally, I would like to thank my mother for everything she has done for me throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the first generation of multicomputers the host was the I/O provider for the processors on the parallel processing computer, which made it difficult to provide sufficient I/O bandwidth to keep powerful computers running at full speed for large problems. Furthermore, the throughput and capacity of I/O subsystems lags behind the corresponding increase in the speed and capacity of the main memory and processor. It becomes increasingly difficult to provide sufficient I/O bandwidth to keep powerful computers running at full speed for large problems that may consume immense amounts of data.

With increasing processor power, multimedia systems and many scientific applications such as weather forecasting, aircraft simulation, seismic exploration, and molecular modeling are increasingly being implemented on multicomputers. These applications have intense I/O demands and are becoming increasingly I/O bound. This trend is likely to continue, causing more and more applications to become I/O bound [7, 77].

In the applications mentioned above, data is transfered as needed between memory and disk by the program. It is therefore very important to provide an efficient I/O architecture. The newer generation of parallel processing computers introduce parallelism into the I/O subsystem in order to lessen the impact of this problem. One such approach is to connect many disks to the computer in parallel, spreading individual files across all disks, or use separate servers connected by a high bandwidth channel [38, 87, 50, 49, 67, 3]. Some of the currently available massively parallel, message-passing multicomputers incorporate caching and prefetching as a means of delivering the benefits of a parallel I/O architecture to user programs [78, 80, 35, 27, 57, 55]. However, parallel hardware alone cannot guarantee a high performance. Effective parallel file systems that manage parallel disks as well as external algorithms for the I/O intensive applications on multicomputers should be built to increase performance. Consequently, I/O problems should be considered at several levels ranging from external algorithms and applications design through system architectures and file systems to physical storage devices. In this dissertation we consider the I/O problems at the levels of file system and external algorithms design. We have designed a software, `UPIO` (User-controllable software for `P`arallel `I`nput/`O`utput), to attack the problems at the file system level, and some important external algorithms for scientific applications based on the software.

The primary goal of UPIO is to maximize I/O performance for scientific applications on MIMD multicomputers. The most important features of UPIO for achieving this goal are as follows:

- Flexible file model: UPIO supports a domain-specific file model and a variety of application interfaces in order to present numerous access patterns that will minimize the effort needed to design a new external algorithm or modify an optimal existing internal algorithm for external computation.

- User controllability: File structure, data distribution, and data access can be controlled totally by programmers.

- Efficiency: The domain-specific file model and user controllability give low I/O overhead and allow programmers to exploit the aggregate bandwidth of parallel disks.

1.1 Need for a Domain-Specific File Model

The majority of conventional file systems for parallel computers are based on a Unix-like file system in which files are treated as a stream of bytes growing linearly [71, 2, 68, 78]. This model has provided sufficient functionality and performance for uniprocessor and supercomputer applications that access files in a simple, sequential fashion [71, 75]. However, the I/O behavior of a number of jobs on parallel computers traced and analyzed by Kotz and Nieuwejaar [58] shows different characteristics. The I/O accesses of these applications are much less sequential to consecutive portions of a file.

We believe that the main causes of the I/O characteristics on parallel computers are from mapping (data) domain to the file abstraction and vice versa. As an example of this problem, consider an implementation of Cannon's matrix multiplication [62] on data that will not fit in memory. In Cannon's algorithm for matrix multiplication, each matrix is partitioned and assigned to each compute node if the data is small enough to fit into memory. One of the best ways to design efficient external algorithms for parallel machines may be to modify the efficient internal algorithms. So, we assume that a modified Cannon's algorithm for external computation reads submatrix and computes matrix multiplication on the data, one after another, and stores the resulting submatrix back to the disks. Typically, a matrix is stored on disks in row- or column-major order under the abstraction of the linear file model.

That is, the matrix is mapped into the linear file abstraction in order to store the data on disks row-by-row or column-by-column. The accesses to submatrices will make a series of requests to the files on disks with the existing file interfaces.

File modes are an addition to the Unix-like file system interface to enhance the interface. They regulate access to a file shared by multiple processors. However, some of the file modes are not used often and are not able to express certain data partitioning [29, 58]. In addition to that, they do not support a domain-specific file model.

Recently, new file system interfaces have appeared in the literature [71, 2, 34] that improve the interface to a linear file model. However, they do not remove the programmers' burden of mapping and remapping between the data domain and the linear file model. We believe that providing a domain-specific file model is crucial to developing applications that match the programmer's semantic model of the data domain in a parallel computing environment.

1.2 Need for User Controllability

The parallel algorithm community has requested certain capabilities for optimal external algorithms for parallel machines. Corman and Kotz [15] presented a list of capabilities that must be provided by the system in order to support optimal external algorithms, the most important of which is to control the declustering of data.

The most common programming model for MIMD multicomputers is the SPMD (Single Program Multiple Data) model. In this programming model, parallelism is achieved by partitioning data among processors. Programmers have to decide how data should be partitioned and manage communications between processors. When the partitioned data for each processor is so immense that all the data cannot fit into the memory at the same time, the overall computation should be performed iteratively in phases. Each phase reads data from disks into memory, performs computation on the data in memory, and writes the results back to the disks. Thus, in each iteration a large number of data transfers between the main memory and the disks, as well as a large temporary storage space for data staging are required. If the data transfer is not handled efficiently, it can be a serious problem.

An operating system such as virtual memory is one possible approach toward solving the I/O problem. However, I/O controlled by an operating system such as "paging systems" degrades I/O performance for many specialized scientific applications, even though the operating system makes programming simple, because the access patterns in a parallel computing environment are different from those of a sequential computing environment [91, 86] and, in this case, programmers are better able than systems to predict which data will be needed. Thus it is important to support hardware and/or software according to the individual I/O requirements of applications. In order to exploit a high-performance I/O in parallel computers, the best way may be to enable programmers to determine the file structure and data distribution that match the access patterns of particular applications. Actually, it has been observed that the data distribution policy should be matched with the access patterns of applications for a good I/O performance. Otherwise, an inferior performance may result, because a number of concurrent requests for different data can be directed to a single disk [61, 60] and many small I/O requests.

File systems or I/O mechanisms for parallel machines should be user-controllable in the sense that programmers should be able to determine the file structure by considering the access patterns, control the distribution/redistribution of data blocks on physical disks, present a variety of access patterns easily, and also prefetch arbitrary data blocks without prefetching mistakes. However, the majority of the existing file systems for parallel computers do not support user-controllable features, and they distribute data in a simple round-robin fashion across a number of disks hiding the pattern of distribution.

1.3 Thesis

I/O problems in parallel computing environments should be considered at several levels, ranging from external algorithms and applications design through system architectures and file systems to physical storage devices. The primary focus of this dissertation is to consider I/O problems at file system level and design efficient external algorithms with the following thesis:

> It is feasible to support efficient external algorithms for scientific and engineering applications and improve system performance by providing user-controllable, parallel I/O software. An important object with which to provide user-controllable, parallel I/O mechanisms is the "logical disk," which can be defined, based on data domain, by application programmers. The logical disk extends the abstraction of a linear file model into an n-dimensional file model and removes the limitations of the linear file model. Application programmers have total control over the logical disk and are able to control the layout of data blocks on disks, aggregating disk bandwidth by distributing or redistributing data blocks across disks, partitioning data blocks to the processors, accessing multiple disjoint chunks of data with a single I/O operation, and prefetching data blocks without prefetching mistakes. Programmers do not have to use a file pointer to access data in a file, which eliminates the overhead

of managing the offset of the file, while providing required file modes, thus making it possible to do I/O collectively at user level by combining many small I/O requests into one large I/O request, and also making it easy to share data in parallel system environments.

To validate this thesis theoretically and practically, we design a user-controllable, parallel I/O software, test the software with some synthetic access patterns, incorporate the basic object, *logical disk*, of the software into the design, and analyze external algorithms for some key scientific applications. Some points of particular interest are:

- File abstraction: domain-specific file model that matches the semantic of data domain in a parallel computing environment.
- Interface for parallel I/O: flexibility and expressiveness of high-level information about I/O to the system.
- The effects of the user-controllable, parallel I/O.
- Designing external algorithms based on internal algorithms and logical disks.
- Support for scientific applications, although the user-controllable, parallel I/O may be useful for other applications as well.

1.4 Contributions

The primary objective of this dissertation study is to support necessary I/O requirements in order to maximize I/O performance for scientific applications on MIMD multicomputers. To meet this objective, we focused our research on two important

I/O requirements: a flexible file model and user controllability. For our purposes we developed UPIO, an important part of which is the *logical disk*.

This dissertation study shows that a logical disk that is defined by programmers can extend the abstraction of a linear file model into an n-dimensional file model and give programmers total control over file data. That is, a logical disk allows programmers to define file structures and explicitly control the way of distributing data across I/O subsystems by considering the basic computational structure of the applications.

A large portion of this dissertation deals with the design, implementation, and evaluation of UPIO's data-access interfaces. UPIO supports two different models of data storage and access for which there are separate interfaces that can be used together in an application and that allow programmers to represent complex data access patterns collectively with a minimum number of I/O operations. The interfaces deliver a high I/O performance by providing enough information to optimize disk-head movements and efficiently use the space for a buffer cache. They do not use a file pointer to access data in a file, which eliminates the overhead of managing the offset of the file.

Collectively represented I/O requests in each processor can be served separately or together. The interfaces allow programmers to specify how the system serves I/O requests according to access patterns. If I/O requests are specified to be served collectively, processors cooperate to combine the requests into a larger request. The request is served in an efficient way by overlapping permutations, data transfers, and disk accesses, thus increasing message-passing bandwidth utilization.

The heart of a large class of scientific and engineering applications is an iterative algorithm. Each iteration in the algorithm generally consists of computation phases, communication phases, and I/O phases. In each iteration, temporary values must be read for computation and the result of the computation should be written repeatedly. Thus, if the read-and-write operations can be done collectively with a single I/O operation (we call it *atomic I/O*), the I/O latency can be reduced and the I/O performance can be increased. UPIO allows users to do read and write operations with a single I/O request by combining the operations in various patterns. Since blocking and nonblocking I/O operations are applied to the atomic I/O operations, atomic I/O operations can overlap I/O operations and computations.

UPIO is extensively evaluated with some synthetic access patterns and is also evaluated with some important scientific computations, for which we have designed new external algorithms. We have a strong belief that there is no difference between internal and external computations except for data size. We designed the algorithms based on internal algorithms that are efficient in terms of computations, communications, or usage of memory. Therefore, if we modify the algorithm to be I/O efficient, then the new external algorithm can be optimal in terms of I/O, computation, and communication. This dissertation study shows how programmers can produce high-performance external algorithms using UPIO based on efficient internal algorithms. This implies that a logical disk can be the base of a computational model for designing and analyzing new external algorithms.

1.5 Organization of Dissertation

Providing a flexible file model and user controllability in I/O is the primary focus of our research. These issues are closely related to the file structures, ways of data partitioning/distribution, and presenting access patterns. In Chapter 2 we examine the main features of some related parallel file systems and consider the issues involved.

An overview of a high-performance software called *UPIO* for user-controllable, parallel I/O that will provide a flexible file model and user controllability is presented in Chapter 3. *Logical disk*, an important object of UPIO, is introduced and a general procedure to access data on physical disks through a logical disk is described in this chapter.

UPIO supports two different models of data storage and access for which there are separate interfaces, but these interfaces can be used together in an application. Chapter 4 describes the basic data-access interfaces of UPIO, which represent complex access patterns collectively with a minimum number of I/O requests. Evaluation results for the interfaces are discussed in Chapter 5. More complex access patterns can be presented using the advanced data-access interfaces of UPIO presented in Chapter 6. We evaluate the performance of the interfaces with some synthetic access patterns and describe the evaluation results in Chapter 7.

Collectively represented I/O requests with the interfaces in each processor can be served collectively. To perform the I/O requests collectively, processors cooperate to combine the requests into one larger request. Chapter 8 describes an efficient way of performing combined I/O requests. We analyze the method in terms of number of communications compared with other methods.

Chapter 9 describes the performance of UPIO with some important scientific computations, such as matrix multiplication and a linear equation solver. Designing external algorithms for the computations based on logical disks is also included in this chapter. In addition, we show how programmers can produce high-performance external algorithms using UPIO based on efficient internal algorithms.

Finally, in Chapter 10, we end this dissertation with our conclusions and some ideas for future work.

Chapter 2

Related Work

The primary goal of UPIO is to maximize I/O performance for scientific applications on MIMD multicomputers that distinguish between I/O nodes and compute nodes. In these systems the I/O nodes are designated to do I/O processing, and the remaining compute nodes are designated to do primarily application processing. Many commercial parallel machines, such as the Intel iPSC, Intel Paragon, nCUBE, and Thinking Machines' CM-5, are based on the architecture.

Providing a flexible file model and user controllability within the systems is the primary focus of our research. These issues are closely related to the file structures of the existing parallel file systems, ways of data partitioning/distribution, and presenting access patterns. In this chapter we examine the main features of some related parallel file systems.

2.1 I/O Workload Studies

File access patterns have been studied and some common characteristics of the patterns identified for a uniprocessor [30, 75]. Files usually are accessed sequentially and contiguously, but it seems that parallel file systems and the applications that use them are not mature enough to know what forms might be typical. Most parallel file systems have been optimized to support large-file data accesses.

Miller and Katz traced and analyzed the I/O behavior of supercomputer applications on a Cray Y-MP [69]. The types of application I/O are compulsory requests for initialization and completion, periodic large requests to checkpoint the application state, and staging I/O in each iteration because of memory limitations. Those in the last category were the dominating I/O operations in the applications. I/Os are the equivalent of paging under a paging virtual memory operating system, but they are done under program control, because the program is better able than the operating system to predict which data it will need. All I/Os were bursty, with bursts coming in cycles. Pasquale and Polyzos [76] studied I/O intensive supercomputing applications and found the characteristics similar to those founded by Miller and Katz.

Crocket [17] defines six file access patterns for parallel file systems: *sequential* (no parallel access); *partitioned sequential*, in which each process accesses a separate portion of the file; *interleaved sequential*, in which each process accesses noncontiguous blocks of the file separated by constant stride; *self-scheduled sequential*, in which each process accesses the next record in the file; *global random*, in which a process accesses any records in the file in any order; and *partitioned random*, in which a process accesses records randomly within a separate portion of the file. Kotz and Ellis [51, 56] analyzed different cache management prefetching policies with a synthetic workload, which are also classified roughly into sequential and random.

The tracing study done on an Intel iPSC/860 at NASA Ames by Nieuwejaar and Kotz [58] shows that there are many applications that issue thousands or millions of small (< 200 bytes) requests, while some applications issue a relatively small number of large requests. This study also shows that many parallel applications access files in regular patterns: almost 93% of all I/Os were strided. Another tracing study done on a Thinking Machines CM-5 at the National Center for Supercomputing Applications by Purakayastha [81] shows similar workload characteristics.

Intel's CFS (Concurrent File System) [78, 35, 80] and Thinking Machines sfs (scalable file system) are based on the Unix file system with additional access modes to support parallel I/O, which will be described in the next section. Generally, CFS stripes each file across all I/O nodes in a round-robin fashion. Thus, the access patterns observed on the systems may be caused by the limitation of conventional file systems or the characteristics of computations.

2.2 I/O Modes

The majority of existing parallel file systems are based on the conventional Unix-like file system decluster files across I/O nodes, or disk arrays attached to different nodes to provide higher bandwidth. The Concurrent File System (CFS) [78, 35, 80] for the Intel iPSC, Touchstone Delta, and Paragon parallel computer systems [44, 46], the file system for the nCUBE parallel computer [80, 22, 24], and the file system for the Kendel Square Research KSR-1 [83] are the main examples of parallel file systems that decluster files.

So far, however, programmers cannot define the patterns for declustering across I/O nodes even if some systems allow programmers to define the unit of declustering. They only provide programmers with the mechanism to place each open file in a certain access mode, allowing different ways of sharing a common file pointer [17, 78, 84, 2, 52]. I/O modes are just an addition to the normal Unix-like file system interface. The most common I/O modes and the systems that support the modes are summarized in Table 2.1 [2, 46, 35].

Table 2.1: Most I/O modes used in various parallel I/O systems

MODE	DESCRIPTION	EXAMPLES
Broadcast/ Reduce	All nodes collectively access the same data	Express single PFS global mode CMMD system-broadcast
Scatter/ Gather	All nodes collectively access a sequence of data blocks, in rank order	Express multi CFS modes 2 and 3 PFS sync and record modes CMMD sync-sequential
Shared offset	Nodes operate independently but share a common file pointer	CFS mode 1 PFS log mode
Independent	Each node operates independently with its own file pointer	Express async CFS mode 0 PFS Unix mode CMMD local and independent

In the *broadcast/reduce* mode, all nodes access the same data in a file. If the access is a read, the same data is broadcast to all readers. When writing, only one of the nodes writes data to the file. The node can always be the same node or an arbitrary one.

In the *scatter/gather* mode, all nodes share a single file pointer and synchronize each I/O operation. Each node accesses a distinct chunk of data contiguously, using the file pointer in rank order. The chunks can be of the same size, or not.

In the *shared offset* mode, all nodes share a common file pointer, as in the scatter/gather mode, but the nodes operate independently. The access order is not predetermined in this mode; rather, the order is determined on the fly: I/O operations are performed in a first-come, first-serve fashion.

In the *independent* mode, each node has a private file pointer with no implied interactions with other nodes.

Various systems provide different combinations or variants of the I/O modes. The different I/O modes use various forms of communication to synchronize the I/O operations.

Express [16] supports three I/O modes: *single, multi*, and *async* mode. In the *single* mode, all nodes behave as if they were one; they perform common I/O operations with a single copy of the data in the file, regardless of how many nodes are in use. In the *multi* mode all nodes operate together, but their data is interleaved in deterministic order. In the *async* mode, every node performs I/O operations independently.

Intel CFS, which runs on the Intel iPSC, the Touchstone Delta, and the Paragon, stripes file data across I/O nodes in 4-KB blocks. The CFS provides four access modes: modes 0, 1, 2, and 3. In mode 0 each node has its own private file pointer and performs operations independently; mode 1 provides a common file pointer, and all nodes share the file pointer. Each node operates independently, but only one node may access the file at a time. In mode 2 all nodes share a common file pointer and synchronize on each I/O operation according to the node IDs; mode 3 is the same as mode 2, except that all accesses deal with the same amount of data on each turn.

The Intel Parallel File System (PFS) [28] that evolved from the Intel CFS stripes file data across multiple disks or RAID devices. New access modes and control over the striping parameters (at the file system level by the system administrator) are added. Modes *M_UNIX, M_LOG,* and *M_SYNC* correspond to CFS modes 0, 1, and 2, respectively. The *M_RECORD* mode is similar to CFS mode 3, except that the amount of data is known in advance in order to allow asynchronous access. The *M_GLOBAL* mode allows synchronized access like the Express *single*, with only one copy of the data in the file.

Thinking Machines sfs (scalable file system) supports UNIX functionality while simultaneously extending it for the CM-5's scalable disk array [2, 68] and introduces new I/O modes: *local independent, global independent, global synchronous broadcast,* and *global synchronous sequential*. In the *local independent* mode, each node performs I/O operations completely independently; the *global independent* mode allows all nodes to perform I/O operations independently like the *local independent* mode. However, each node has its own local file pointer. In the *global synchronous broadcast* mode, nodes can read or write the same file data, which is logically equivalent to one node doing the I/O for all the nodes. On read operations, one node reads file data and broadcasts the file data to the other nodes. On write operations, only one node writes data to the file as if a node is doing write. The *local independent* mode allows nodes to access file data in an interleaved manner according to the node IDs.

SPIFFI [34] partitions files in a round-robin fashion across disks. In this system programmers can specify a set of disks and the data size to be distributed on disks. SPIFFI provides four types of file pointers: one type of local file pointer and three types of shared file pointers (global, synchronized-global, and distributed). A local file pointer is identical to a Unix file pointer. A global file pointer corresponds to PFS's *M_LOG* mode. A special node is dedicated to serving this type of file pointer. A synchronized-global file pointer is comparable to PFS's *M_SYNC* mode, but this file pointer permits programmers to specify a subset of compute nodes that share a file pointer. A distributed file pointer is similar to the other two shared file pointers, but each I/O node maintains a file pointer for its file fragment.

As we explained above, major systems based on Unix-like file systems provide various I/O modes to support parallel I/O. However, the tracing study done by Nieuwejaar and Purakayastha [81, 58] shows that shared file pointers are not used often in practice, providing a poor performance because of a failure to match the needs of applications.

2.3 File Partitioning

Partitioning file data among the nodes is a very useful feature in large-scale scientific and engineering applications, because parallelization is done by data partitioning. Load-balance, locality of access, and other optimizations can also be achieved.

Figure 2.1[1] shows a simple example of data partitioning that is relatively common for partitioning multidimensional arrays. It is done by a set of directives of programming languages, such as HPF [31], Fortran D [33], or Vienna Fortran [6], one for each dimension of the array. There are four options: `BLOCK`, `CYCLIC`, `BLOCK_CYCLIC` and $*$. BLOCK divides the array into equal-sized blocks, assigning each block to each node successively. CYCLIC specifies a round-robin division, assigning every block to nodes in a round-robin manner. BLOCK_CYCLIC is similar to CYCLIC, but it first divides the dimension into contiguous chunks of a certain size, then assigns these chunks in the same fashion.

[1] In this figure the attributes of data partitioning are written from the horizontal to the vertical dimension.

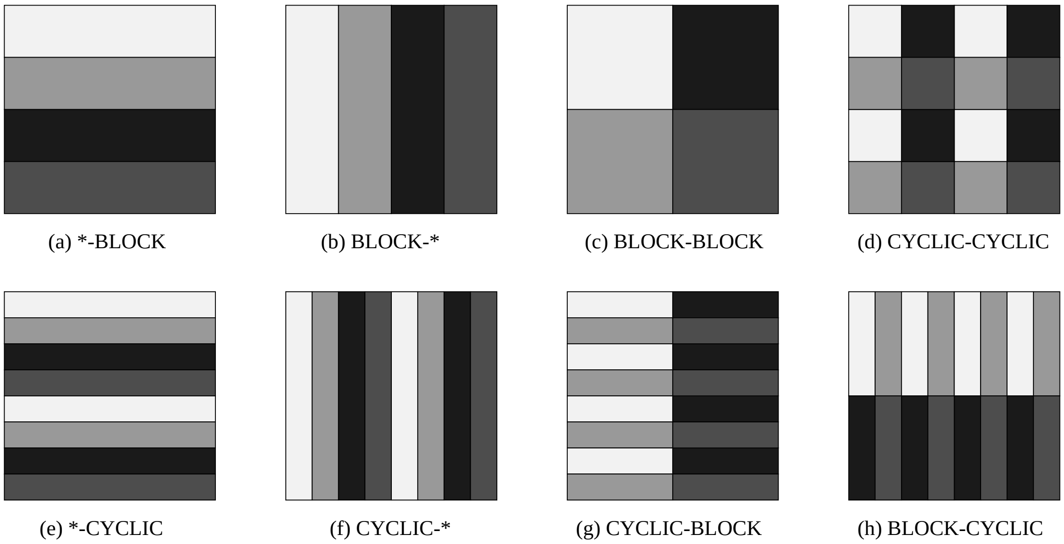


Figure 2.1: Data Partitioning

The various I/O modes are closely related to file partitioning, because one important function of I/O modes is to define the part of the data to be accessed by a certain node. However, it is not always true that the I/O modes can express the data partitioning. The I/O mode, especially the scatter/gather mode, can express some data partitioning, but not other. Partitioning a file in a row-cyclic manner, as shown in Figure 2.1 (e), means that node i accesses every ith row in round-robin fashion. Let's assume that an $N \times N$ 2-D matrix is stored in row-major order in a file and the file is partitioned on four nodes. The partitioning then can be expressed using the scatter/gather mode by accessing a full row at a time. All nodes synchronize at each access. The first node accesses the first row, the second node accesses the second row, and so on. In the same way, the access patterns of Figure 2.1 (a), (b), and (f) can be expressed by using the scatter/gather mode.

Other access patterns (Figure 2.1 (c), (d), (g), and (h)) cannot be expressed by using the scatter/gather mode. In these patterns, only two nodes can access that part of each row. The other two nodes have to participate in accessing data even though they are not actually accessing any data, because all nodes collectively access file data in rank order in the scatter/gather mode. This causes serialized access and much redundant synchronization. Programmers solve the problem by using an independent mode instead, which is why other modes are not often used in practice [58]. However, using an independent mode may also be a cause of inefficiencies in disk access, because the current file-system interface makes the programmer break down large parallel I/O activities into small, noncontiguous requests.

To solve the deficiencies, new I/O techniques and file system interfaces have been proposed that represent data requests collectively. Two-phase [23] and disk-directed I/O [53] approaches are representative I/O techniques that have been introduced recently. They show that I/O performance can be improved by supporting collective I/O that conforms to the layout of the file even if there are some drawbacks such as extra buffering or an increase in network traffic and congestion. Some of the file systems will be described in the following section.

2.4 Layout of Data on the Physical Location

In order to achieve optimal performance, control over the layout of data is required. However, the partitioning schemes described in Section 2.3 do not provide any control over the actual layout of data, but merely a method of distribution that specifies the mapping of data to physical parallel machines without considering the layout of the data on the physical locations. This may not guarantee an optimal performance.

Almost all current parallel file systems simply distribute data in a simple round-robin fashion across a number of disks. However, it is important to distribute data across disks matching the application access pattern [10, 11, 48, 91]. Jensen [48] analyzed a series of distribution functions that lay out file data to disks and showed the effectiveness of custom mapping for application-specific access patterns.

Recently proposed unconventional file systems provide some features that exert limited control over the layout of data. These allow file partitioning or distribution of data across disks to be expressed explicitly. In this section we briefly describe the methods of data distribution and file system interfaces to present the distributions.

2.4.1 nCUBE

The framework of the nCUBE parallel I/O is based on two mapping functions [20, 21]. The first mapping function specifies how data is to be partitioned among processors, and the second specifies the way that data is to be distributed on parallel I/O devices. The first mapping function is specified by the programmer, and the second is defined during system configuration by the system programmer. At runtime, the system software composes another mapping function with the second function and the inverse of the first.

The abstract dataset in an nCUBE is considered as a stream of bytes, and the input of the mapping functions is the index of a byte in the abstract dataset. The mapping functions permute the index bits of the byte and output processor or I/O device number and the index of the byte in the processor or I/O device.

The nCUBE interface using the mapping functions is conceptually elegant; however, the mapping functions have one important limitation: all sizes must be to the power of two. This includes the number of processors, the number of I/O devices, the number of partitions, and the array dimensions. The nCUBE interface does not also provide the user with control over data layout, thus the user may not be able to partition data effectively on processors based on the layout even if the mapping functions provide flexible partitioning patterns by using arbitrary subsets of index bits.

2.4.2 Vesta

The Vesta file system [14, 10, 11, 12, 13] developed by IBM allows users to define logical partitions and gives some control over the data distribution.

Files in the Vesta file system have an explicit two-dimensional structure defined in terms of structural parameters: *cells* and *basic striping unit* (BSU). In the horizontal dimension the file is composed of an ordered set of cells that logically connects the different chunks of the file's data. Cells are related to the physical layout of the file, and each cell must be entirely located on an I/O device. Thus the number of cells may specify the maximum parallelism in terms of I/O devices. BSU is the basic unit of data to be spread across cells. The number of cells and the BSU size is defined when a file is created.

A user can partition a file into disjoint subfiles in several ways by using the following four partitioning parameters: *Vbs* specifies how many consecutive BSUs are to be grouped into a subfile from each cell; *Vn* specifies how many subfiles are interleaved in each cell; *Hbs* specifies how many contiguous cells are spanned when subfiles are interleaved; and *Hn* specifies how many subfiles are interleaved across the cells of the file.

The product $Vbs \times Hbs$ defines a basic block of a subfile. The product $Vn \times Hn$ defines a template of subfiles, and the template is repeated in a file. Figure 2.2 shows an example of how subfiles are defined using the partitioning parameters.

Each subfile is processed by different processes simultaneously, which improves performance since it guarantees that each process accesses only its own portion of a file and reduces the need for communication and synchronization among processors. However, the Vesta interface does not provide a way to access overlapping regions of a file, which could be a serious limitation because, in many physical computations,

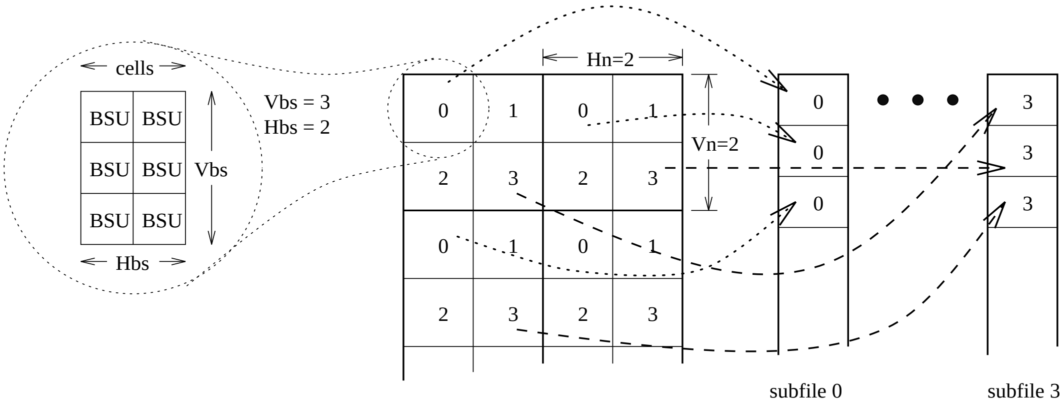


Figure 2.2: How to make subfiles using partitioning parameters

logically adjacent processors share boundary data.

The two-dimensional file structure of the Vesta allows programmers to express the layout information, expecting performance gains. However, this could result in a significant performance degradation, because the horizontal dimension is heavily related to hardware and the hardware extent will be much smaller than that of the data structures. The two-dimensional file abstraction provides a flexible way of partitioning, but it does not support the partitioning, multidimensional data structures.

2.4.3 MPI-IO

The MPI-IO [8, 9] is a draft standard for describing parallel I/O operations within an MPI message-passing standard [32]. In MPI-IO, the I/O is modeled as message passing: writing to a file is similar to sending a message, and reading from a file is similar to receiving a message.

The notion of MPI-derived datatypes is used in MPI-IO to describe the data layout in the file as well as the partitioning of the file data; *etype* (elementary datatype) and *filetype* are the basic structures of the files in MPI-IO. The etype can be a simple base type such as an integer or a float and can be constructed from a simple base type like *struct* in the C programming language. The filetype is derived from the etype and defines a data pattern that is tiling the file data throughout the file. The filetype consists of fields of data that are located at specified displacements, and it may contain *holes* between the fields. A process can only access the file data that corresponds to the data fields, and it skips over the hole.

Complementary file types allow other processes to access file data that resides in holes. Thus through the proper definitions and combinations of filetypes, common access patterns can be generated.

2.4.4 PPFS

PPFS [41, 40] is an experimental file system being designed by the University of Illinois for parallel scientific applications. Files in PPFS are linear streams of records, rather than bytes. PPFS includes a number of predefined data distributions and allows users to define a mapping function that maps a linear stream of records into disks.

2.4.5 The Galley Parallel File System

The Galley [72, 73] file system being designed at Dartmouth allows programmers to control declustering data across disks by defining *subfiles*. That is, files in the Galley are structured as a set of *subfiles*; each subfile resides on a single disk as a cell in the Vesta file system and the number of subfiles can be defined by the programmers. This structure provides programmers with the ability to control how data is distributed across disks.

Each subfile is more structured by a collection of one or more independent *forks*. Each fork of a subfile is similar to a traditional Unix file: it is a named, linear, addressable stream of bytes. While the number of subfiles is fixed at file creation time, the number of forks in a subfile is not fixed.

Galley provides strided read/write operations as well as general read/write operations to support the data access patterns they studied [58], simple- and nested-strided access patterns. The interfaces allow programmers to represent complex access patterns with a minimum number of I/O operations. In Galley, however, there is no notion of file-level read/write operations. Each read/write operation is applied to only a single fork of a subfile. Programmers have to keep track of the offset of each fork, because each fork is similar to a traditional Unix file. It can be a burden to programmers when each compute node accesses data in forks across all subfiles, because they have to keep track of the offsets of n files if the number of subfiles is n. It can also further limit the upgrading of I/O performance, because each compute node must issue a separate I/O request to each I/O node (each compute node can issue a single I/O request with the strided interface to each I/O node), and the independent accesses may be optimized if they are combined.

Chapter 3

UPIO: User-controllable Parallel I/O

3.1 Overview of UPIO

In general, file systems and virtual memory systems control I/O, providing facilities to use I/O easily. However, these can be a cause of degrading I/O performance of many scientific and engineering applications on parallel computers, because many of the applications exhibit access patterns that are predictable and the access patterns in a parallel computing environment are different from those of a sequential computing environment [58, 91]. In this case, programmers are better able than systems to reduce I/O time through an explicit management of I/O [10, 12, 92].

Intuitively, it may seem that the best course is to allow application programmers to manage the disk directly for explicit I/O. However, recent I/O subsystems are so complicated that it is, in fact, impossible for application programmers to manage the disk directly. Furthermore, today's typical parallel file systems [12, 74, 28, 25] have been derived from Unix [79] and use several modes of file pointer and offset to access data on disks hiding the configuration of I/O subsystems. Therefore, it is apparently impossible to allow application programmers to manage I/O explicitly.

Then how would it be possible for application programmers to manage I/O explicitly? Needless to say, if a system were to provide programmers with disks that can be used exclusively and simple I/O facilities to manage disks directly, programmers could manage I/O directly. We satisfy the requirements with a *logical disk*. A *logical disk* is the disk that is defined by an application programmer. That is, we allow application programmers to define their own disks. Thus an application programmer can have his/her own disk with exclusive control over the disk. Application programmers can define any type of disk they want to create: a one-dimensional array of disk blocks, a two-dimensional array of disk blocks, or an n-dimensional array of disk blocks.

In this chapter we describe the important features of UPIO based on *logical disk*. The main goals of UPIO are the following:

- Flexible file model: UPIO should support a domain-specific file model and a variety of interfaces with which to present the file model.
- User controllability: File structure, data distribution, and data access should be controlled by programmers.
- Efficiency: The domain-specific file model and user-controllability give low I/O overhead and allow programmers to exploit the aggregate bandwidth of parallel disks.

The domain-specific file model is achieved through logical disks that are defined by programmers based on data domain. The logical disk allows programmers to determine a file structure that matches the application access pattern or presents the basic structure of computation. It also allows programmers to control the layout and distribution of data blocks on physical disks, reduce I/O latency by minimizing the number of I/O requests to noncontiguous data and prefetch arbitrary data blocks without prefetching mistakes, and minimize the effort needed to design a new external algorithm or modify an optimal internal algorithm for external computation.

3.1.1 Logical Disk

In scientific and engineering applications, parallelization is done by partitioning data among the processors. Load balance, locality of access, and other optimization is also done based on the partitioned data. In other words, the partitioned data is the basic structure of computation.

If the amount of data is too huge for it all to fit into memory at the same time, it should be stored in a file on disks. However, the abstraction of a one-dimensional file forces the parallel I/O subsystem to store data in a file without considering the structure of computation and the access patterns of the application. For example, a multidimensional array is considered as a series of planes of 2-D array under the abstraction of a one-dimensional file. Since the most common method stores array data on the disk in row- or column-major order, the planes will be stored one-by-one in the same way without considering the structure of computation and access patterns.

This may be a hurdle for programmers in managing I/O explicitly and in improving performance, because programmers have to map data domain into the linear file abstraction, and vice versa. To remove the burden of mapping and remapping between the data domain and the liner file model, file systems and virtual memory systems control I/O, thus providing facilities for using I/O easily. However, these can be a cause of degrading I/O performance for many data parallel applications [91, 86].

We believe that programmers can represent the semantics of data domain on disks as it is by extending the abstraction of a one-dimensional file to an n-dimensional file. For example, if a two-dimensional matrix is in internal memory, application programs access an element of the matrix through the index of the element. This method can be applied to another two-dimensional matrix that is too huge to fit into the internal memory all at one time. If we divide the matrix into submatrices that are small enough to fit into the internal memory, we can get another matrix of the submatrices and map each element of the newly created matrix into one of the submatrices.

If we define the newly created matrix and each element of the matrix as a *logical disk* and a *logical disk block*, respectively, we can have a two-dimensional logical disk. Programmers can access any submatrix through the index of the logical disk, because each element of the logical disk is mapped into one of the submatrices. Therefore, a logical disk and the index of the logical disk correspond to a file and its file pointer, respectively.

We can also define a one-dimensional array as a logical disk to map each element of the array with one of the submatrices. In other words, a *logical disk* is an n-dimensional array that is defined by programmers to map each element of the array into some data elements that correspond to a logical disk block, which is also defined by programmers. Hence, programmers can define a flexible file abstraction according to the needs of applications by defining the structure of a logical disk.

A logical disk is just an abstract block-index domain on which programmers control I/O explicitly. The logical disk does not allocate space for it, either in memory or on a physical disk. Programmers are therefore able to map the basic structure of computation into a logical disk block, represent the semantic of data domain by defining the shape of a logical disk, and control data distribution by distributing the abstract index across I/O or compute nodes.

Figure 3.1 illustrates the I/O procedure using a logical disk. Programmers can access data through the logical disk they defined by using the high-level I/O primitives of UPIO. In these primitives the only thing programmers have to do is to specify the indices of the logical disk blocks they want to access.

The mapping between the logical disk blocks and physical disk blocks is done automatically by the simple mapping algorithms of UPIO when data is accessed through the logical disk. The mapping operations will be described later in detail. The primitives in UPIO do not use a file pointer and an offset of the file to map the logical disk blocks into the physical disk blocks. They use only indices of the logical disk blocks specified by programmers, which eliminates the overhead of keeping track of the file pointer and offset, something that should be done in other file systems. Therefore, the system using the logical disk should be as easy to use as conventional file systems and also efficient.

3.1.2 Logical Disk Block

UPIO allows programmers to specify the structure of a file by defining the structure of a logical disk and its logical disk block. A logical disk block is defined by programmers based on the data types they use in their applications, considering the basic structure of computation. That is, a logical disk block is a set of contiguous elementary data in a physical data domain.

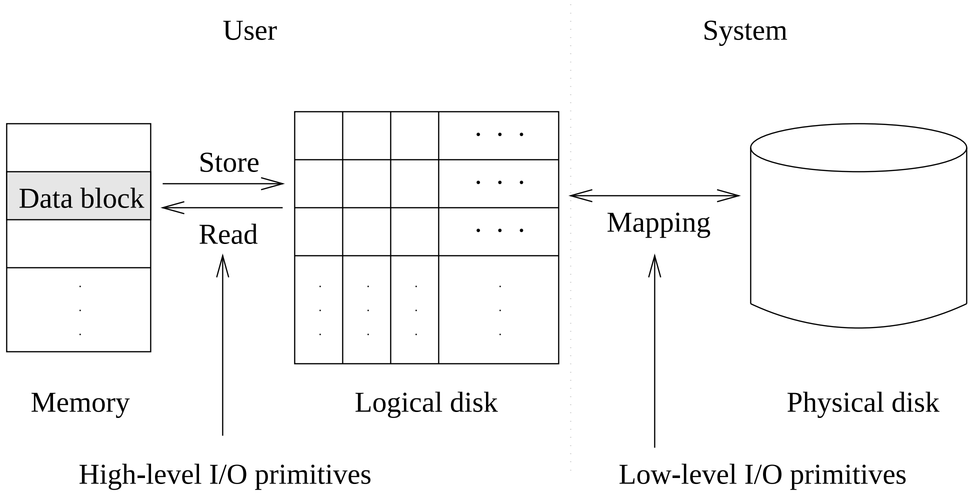


Figure 3.1: Procedure of user controllable I/O

Currently, UPIO supports three types of logical disk blocks for array data: *column-wise, row-wise*, and *block-wise* logical disk blocks. In all cases, a logical disk block is defined by specifying the number of data elements in each dimension. Let's consider the $N \times N$ two-dimensional array data shown in Figure 3.2.

The array data can be partitioned columnwise and then mapped into logical disk blocks as shown in Figure 3.2 (a). That is, c columns of array data can be mapped into a logical disk block, where $c \geq 1$. We call it a *column-wise logical disk block*. If the array data are partitioned rowwise and each partition is mapped into a logical disk block, the logical disk block is called a *row-wise logical disk block*. In other words, r rows of array data are mapped into a logical disk block, where $r \geq 1$ (see Figure 3.2 (b)). The array data can be partitioned in rectangles of $r \times c$ ($1 \leq r$ and $c \leq N$) and then each rectangle can be mapped into a logical disk block (see Figure 3.2 (c)), which is called a *block-wise logical disk block*; in an extreme case, only one data element could be a logical disk block.

The logical disk blocks can be components of a one-dimensional logical disk.

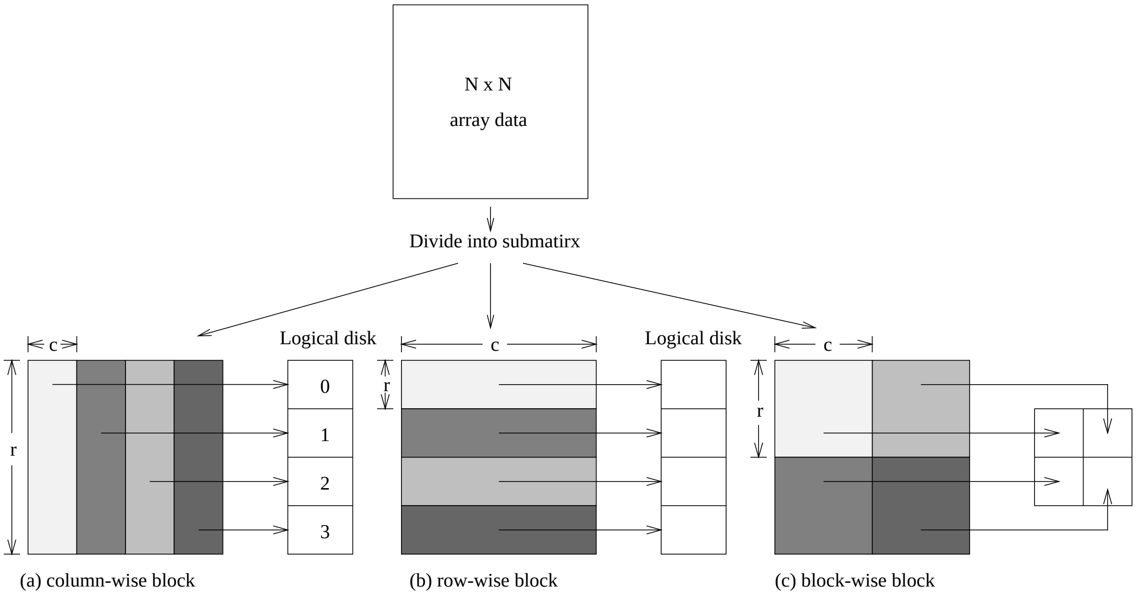


Figure 3.2: Logical disk block

In this case, the array data is mapped into a one-dimensional disk. Figure 3.2 (a) and (b) show how a one-dimensional logical disk is associated with the array data. Figure 3.2 (c) shows how a two-dimensional logical disk is associated with the array data.

Of course, row- or column-wise blocks can be associated with a two- or an n-dimensional logical disk, and block-wise blocks can also be associated with a one- or an n-dimensional logical disk. Therefore, a logical disk allows the programmer to do domain-specific I/O.

Programmers can define any size and type of logical disk block based on the physical data domain by considering the access patterns of applications or the basic structure of computation. Figure 3.3 shows a simple example of some mapping between a 3-D array and a logical disk. This figure illustrates that programmers can define a 1-, 2-, or 3-D logical disk based on the 3-D array. In Figure 3.3(a), a logical disk block consists of the whole data elements in y-dimension, and some data

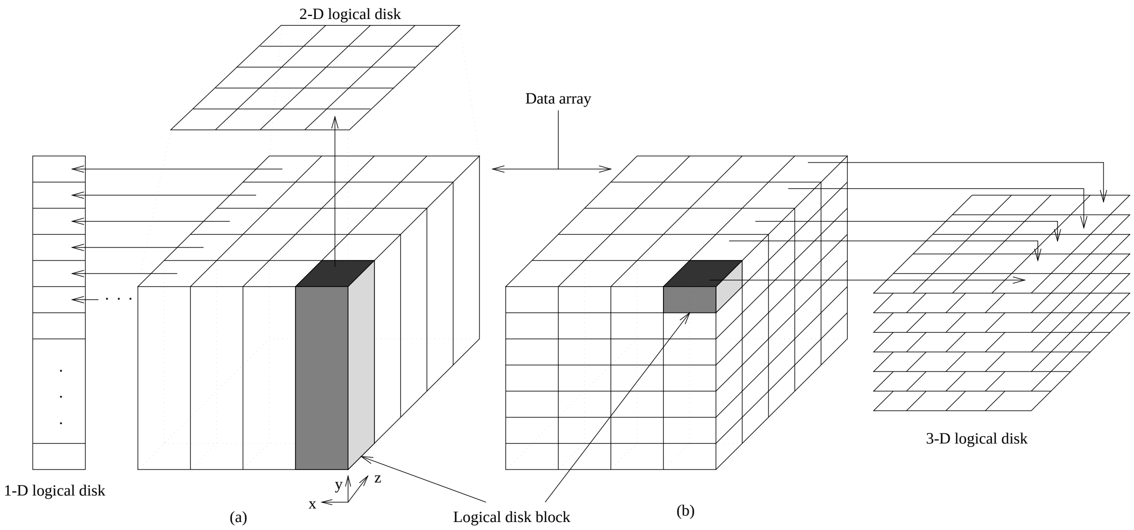


Figure 3.3: Example of logical disks based on 3D array

elements in x- and z-dimension, respectively, and the logical disk blocks are mapped into 1-D and 2-D logical disks. In Figure 3.3(b), a logical disk block consists of some data elements in x-, y-, and z-dimension, respectively, and the logical disk blocks are mapped into a 3-D logical disk.

This logical disk allows programmers to take advantage of the characteristics of physical data structure and to manage I/O explicitly by using the index of the logical disk blocks.

3.1.3 Mapping Between Logical and Physical Disk Blocks

In general, file systems have pointers to the file data blocks on the disk. The UNIX operating system [1] contains the table of contents of disk blocks in the *inode* for locating a file's data on the disk. The Intel Touchstone Delta file system called Concurrent File System [74] also contains pointers to the data blocks of a file on the disk in the file header. In this dissertation we call the pointers to the file data blocks on the disk the *table of contents of disk blocks*, like the inode in UNIX that is used to map logical disk blocks to physical disk blocks.

Figure 3.4 illustrates the general mapping procedure. In this figure, a 3×3 two-dimensional logical disk is used to access the data on the disk. Each logical disk block is conceptually mapped to the entries in the table of contents of disk blocks.[1] (For a while, we assume that the size of a logical disk block is equal to that of a physical disk.) The system can locate the required data blocks through the entry in the table (i.e., the index of a physical disk block) as in existing (concurrent) file systems. The index of a physical disk block is very simply determined by using the index of the logical disk that is specified by programmers. The index of a physical disk block corresponding to a logical disk block $(i,\ j)$ is calculated as follows:

$$\textit{index of physical disk block} = i \times no\_col + j$$

where no_col is the number of columns in a logical disk.

In Figure 3.4 the first element, (0,0), of a logical disk is mapped to the first entry in the table of contents of disk blocks. If the logical disk is ordered in row-major order, the logical disk block, (0,1), will be mapped into the second entry, $(1 = 0 \times 3 + 1)$. The next entry will be associated with the logical disk block, (0,2), and so on. The logical disk can be ordered in column-major order. In that case, the second entry in the table of contents of disk blocks will be associated with the logical disk block, (1,0). The next entry will be associated with the logical disk block (2,0).

The logical disk block size can be different from the physical disk block size. In this case, one logical disk block may be mapped into several physical disk blocks, or several logical disk blocks may be mapped into a single physical disk block. There-

[1] For simplicity, we adopted Unix's multi-level scheme for the table: direct, indirect, and doubly-indirect. The table contains 1,000 direct mapping entries, two indirect mapping entries, and one doubly-indirect mapping entry.

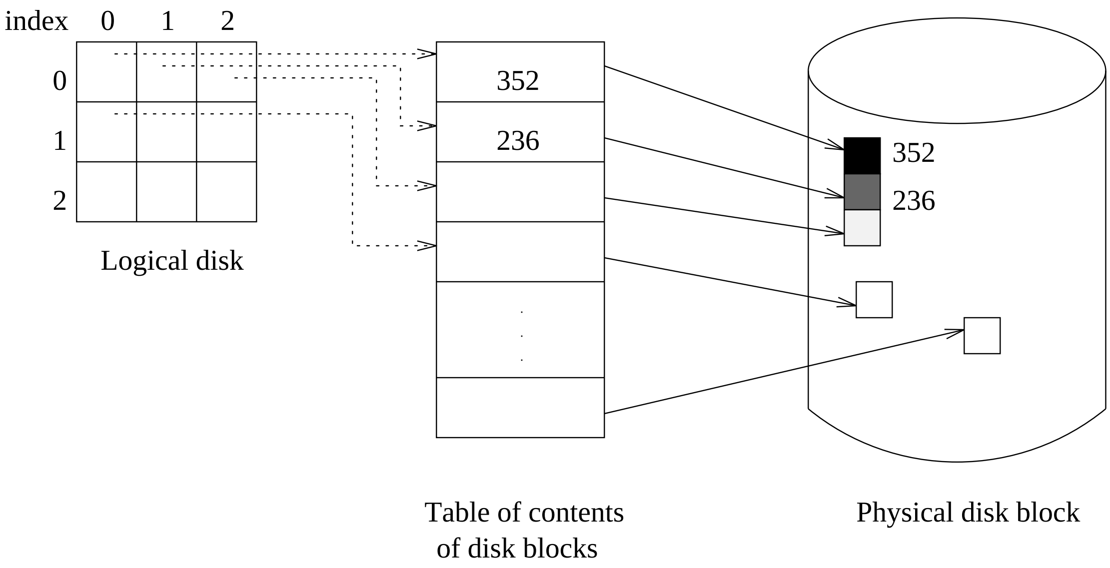


Figure 3.4: Accessing data blocks on disk through logical disk

fore, it is not sufficient to calculate only one physical disk block entry corresponding to a logical disk block by the method given above. The first and last index of the physical disk blocks should be calculated to map a logical disk block into the physical disk block(s). Following is a simple way to calculate the first and last index of the physical disk blocks corresponding to a logical disk block (i, j):

$$\textit{index of the 1st block} = (i \times no\_col + j) \times lblk/pblk$$

$$\textit{index of the last block} = [(i \times no\_col + j + 1) \times lblk - sz\_of\_ele]/pblk$$

where *sz_of_ele* is the size of data element; *lblk* and *pblk* are the logical and physical disk block sizes, respectively.

If the indices of the first and the last block are the same, the logical disk block (i, j) is mapped into only one physical disk block. If they are different, the logical disk block (i, j) is mapped into *index of the 1st block*, *index of the 1st block*+1, ..., *index of the last block*.

The byte offset into a physical disk block should also be calculated if a logical disk block size is different from the physical disk block size. The byte offset into a physical disk block can be simply calculated by using the logical disk block, its size and physical disk block size. Following is a simple way to calculate the byte offset of a logical disk block, (i, j), into a physical disk block:

$$\textit{byte offset} = (i \times no\_col + j) \times lblk \bmod pblk$$

The data size in the last physical disk block should be calculated if the indices of the first and the last block are different; following is a simple way to calculate the data size:

data size in the last block = *lblk* − [(*pblk*− *offset in the 1st block*) + (*index of the last block* − *index of the 1st block* −1) × *pblk*]

In Figure 3.4 let's assume that a physical disk block contains 1024 bytes and a logical disk block size is 512 bytes. If programmers want to access a logical disk block, (0, 1), the index of the first and last block will be 0 ($(0 \times 3 + 1) \times 512/1024 = 0$ and $[(0 \times 3 + 1 + 1) \times 512 - sz\_of\_ele]/1024 = 0$). Thus, the logical disk block (0, 1) will be mapped into the first block (block number 352); $(0 \times 3 + 1) \times 512 \bmod 1024$ $(= 512)$ will be the byte offset into the block number 352. That is, the 512th byte (starting from 0) in block number 352 is the first byte of the logical disk block, (0, 1). Logical disk block (0, 2) is mapped into the index of disk block 236. The first byte of the logical disk block will be the 0th byte in block number 236 ($0 = (0 \times 3 + 2) \times 512$ mod 1024).

3.2 Target Architecture Model

Our architecture model assumes a multiple instruction stream, multiple data stream (MIMD) multicomputer in which the processor nodes are connected through an interconnection network in a certain fashion and each processor node has its own memory (see Figure 3.5). A set of disks are attached to some processor nodes called

I/O nodes. We call other nodes compute nodes.

We configured our target architecture model in Proteus [5], which is an execution-driven simulator for parallel architectures; it has been validated against real message-passing machines. It performs various activities in a simulated machine by multiplexing a single processor to provide accurate information about the timing and behavior of an application and the underlying simulated architecture.

Proteus itself is actually a simulation engine that combines user applications with architecture modules to create a simulator; it interleaves the execution of an application program with the simulation of the underlying simulated architecture. Proteus simulates MIMD multiprocessors in which independent processor nodes are connected through an interconnection network. The interconnection network can be a bus, a direct network, or an indirect network. Each processor node is a generic sequential processor, which mainly consists of a processor, memory, and modules for network or shared memory access.

For I/O nodes in our architecture model we added IOPs (I/O Processor) and Ruemmler and Wilkes' HP97560 disk model [85] to Proteus. The disk model is re-implemented by Kotz [59] and has been validated against disk traces provided by HP. It simulates one or more disk drives attached to one or more SCSI (Small Computer Systems Interconnect) buses.

In our architecture model each I/O node serves one disk using one I/O bus and manages a cache that uses an LRU-replacement and write-back strategy. We assume that each node has a direct-memory access (DMA) capability, and I/O is modeled as message passing: I/O requests and replies are sent and received between a compute node and an I/O node as messages using DMA. We assumed that I/O nodes are dedicated to doing I/O processing, including the calculation of physical disk block index using logical disk block index, while compute nodes do application

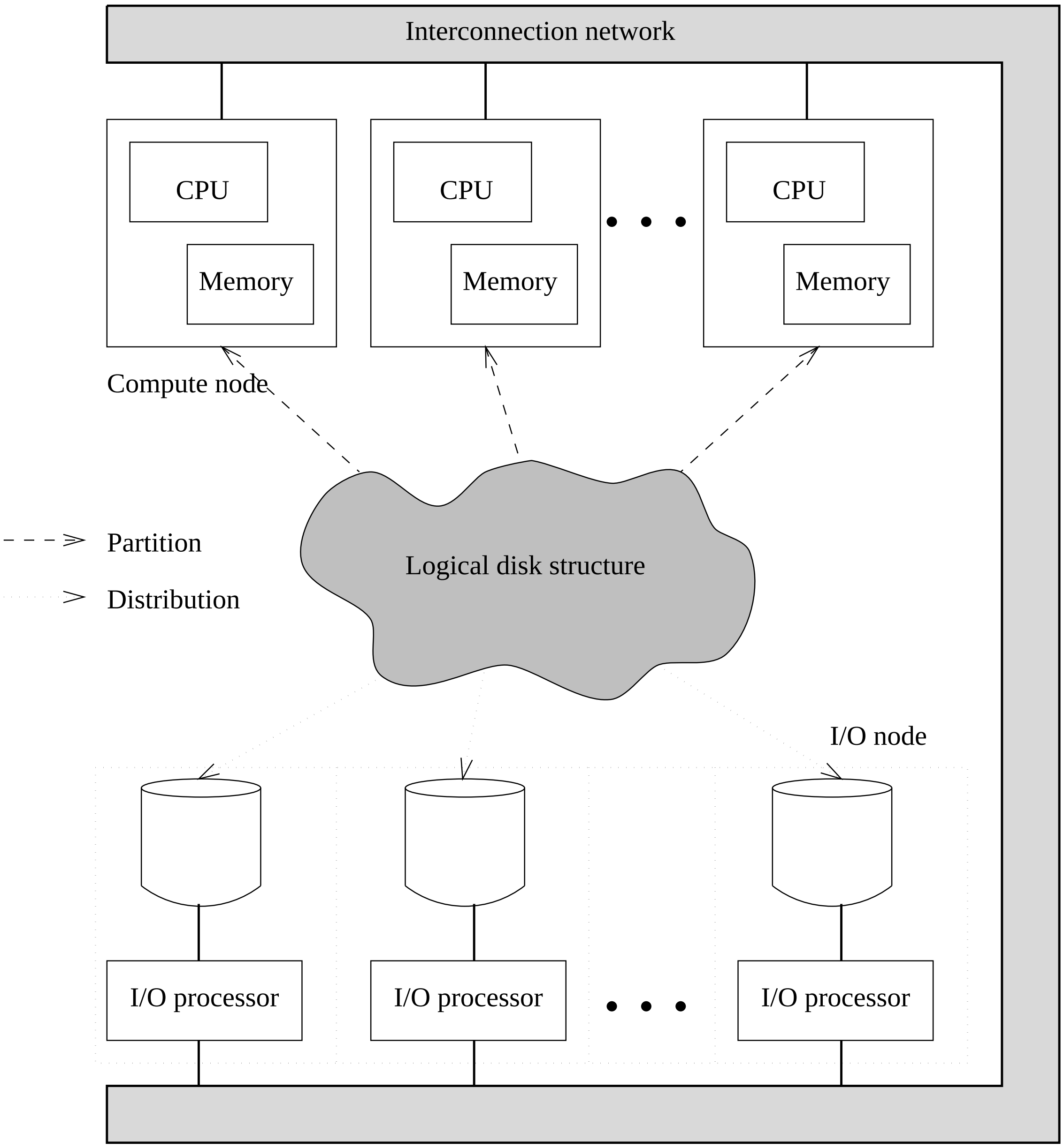


Figure 3.5: Architecture model

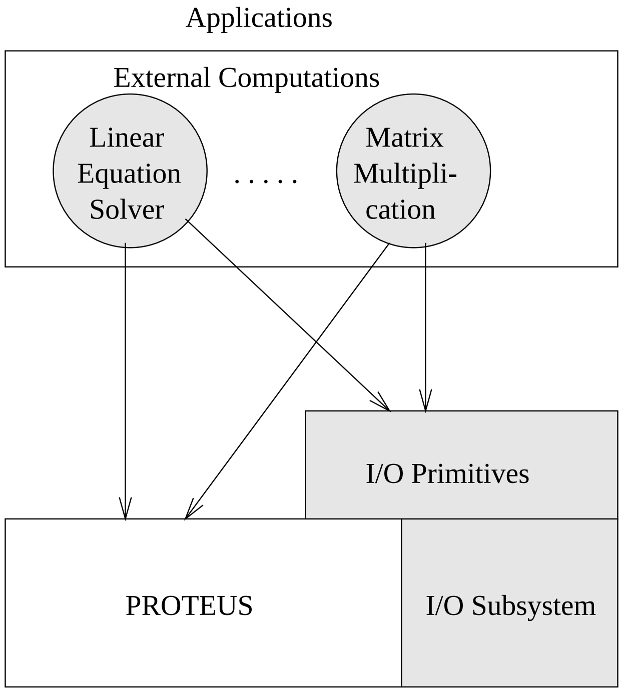


Figure 3.6: Overview of testbed

processing. Figure 3.6 represents the overall view of our testbed.

In the test bed, rather than make specific assumptions about the underlying network, we assume (in order to analyze the external algorithms we designed with UPIO) that the cost for an access to a remote node is independent of the distance between the communicating nodes. Communication between nodes has a start-up overhead of τ, while the data transfer rate is $1/\mu$. The time taken to send a message from one node to another is modeled as $\tau + \mu m$, where m is the size of the message. For our complexity analysis we assume that τ and μ are constant, independent of the link congestion and distance between two nodes. With new techniques such as wormhole routing and randomized routing [19, 62, 65, 70], the distance between communicating processors seems to be less of a determining factor on the amount of time needed to complete the communication. Further, the effect of link contention (due to several messages traversing common links along their routes) is limited due to the presence of virtual channels and the fact that link bandwidths are much larger than node interface bandwidths. This permits us to view the underlying interconnection network as a virtual crossbar network connecting the nodes. Although our external algorithms are analyzed under these assumptions, they can be efficiently implemented on a wide variety of interconnection networks, including meshes and hypercubes, on our simulator.

3.3 Storage Models for UPIO

The user-controllable parallel I/O operations can be applied to two different storage models for MIMD parallel machines: *local logical disk model* (LLDM) and *global logical disk model* (GLDM). Figure 3.7 specifies how the data is mapped into logical disk(s), how the logical disk blocks are distributed across physical disks, and how the data is accessed through the logical disk by each compute node.

In the local logical disk model (Figure 3.7(a)), the (input) data is divided into local data belonging to each compute node. Each group of local data is mapped into a separate logical disk and stored in a separate file called *local file*, where each logical disk is the abstraction of each local file. In other words, each compute node has its own logical disk and local file. The compute node has sole control of this data and accesses the data through the logical disk. Any sharing with other compute nodes will be done through explicit message passing. In this model the logical disk blocks of each logical disk can be distributed over many I/O nodes. However, in our study we assume that the logical disk blocks are not distributed over as many I/O nodes as in a PSS (partitioned secondary storage) system [91].

In the global logical disk model (Figure 3.7(b)), data is mapped into a global logical disk and stored in a single file across the physical disks under the control

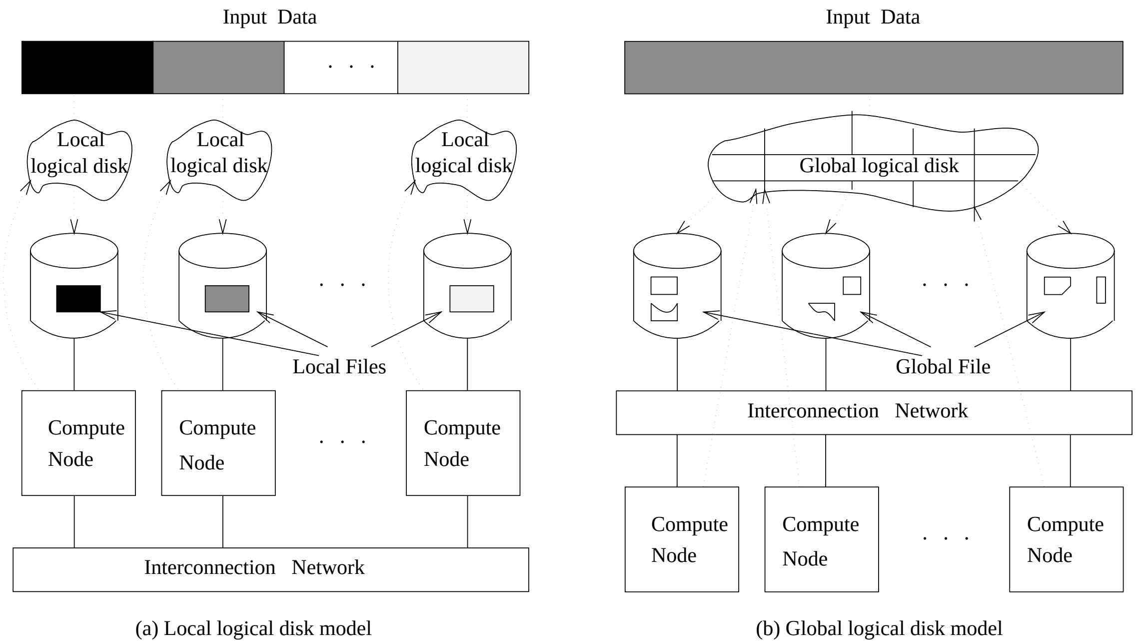


Figure 3.7: Storage model for external computation

of programmers, where the global logical disk is the abstraction of the global file. The global logical disk blocks can be accessed directly by applications and can be logically partitioned and assigned to each compute node for collective I/O.

Each logical disk block of a global logical disk is the basic distribution unit, and the logical disk blocks of a global logical disk are distributed over I/O nodes by programmers. This ability can extract the maximum parallelism available by matching the pattern of partitioning to compute nodes or by reducing the access conflicts to the same I/O node. The distribution scheme, which is described later in detail, is similar to that used to decompose an array in HPF [31], because a logical disk is an array of logical disk blocks and data is mapped into logical disk blocks. But our approach is different from most other majority file systems for parallel machines that distribute data in a simple round-robin fashion across a number of physical disks.

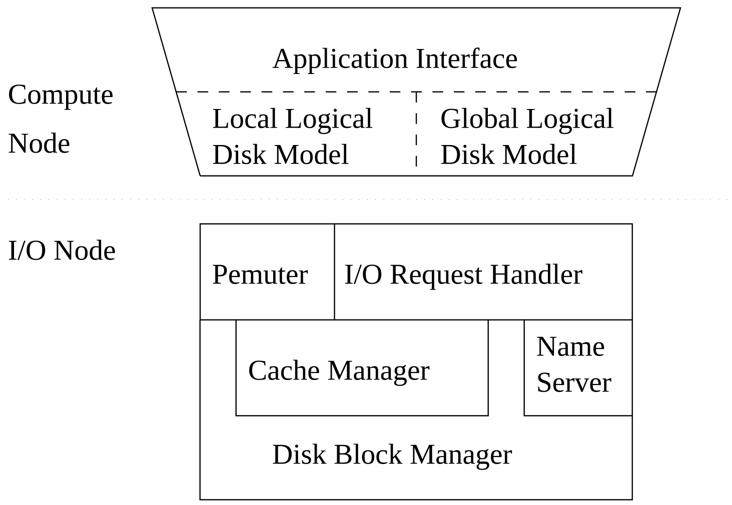


Figure 3.8: Main components of UPIO

UPIO allows programmers to use any storage model in describing algorithms for external computations, because each model has its advantages and areas of applicability. Some algorithms or some phases of an algorithm may be optimal under a certain storage model. These storage models may also be theoretical works for designing algorithms for external computations and for analyzing I/O complexities of the algorithms.

3.4 The Structure of UPIO

The major components of UPIO are the *application interface*, *I/O request handler, name server, cache manager, permuter*, and *disk block manager* (see Figure 3.8).

In its functionality, the application interface of UPIO is closely related to the file system interface of the existing parallel file systems. However, the interface of UPIO provides a flexible file model (structure) and user controllability. The application interface directs requests for file data to the destination I/O nodes as messages. UPIO supports two kinds of storage models, and there are corresponding application interfaces that are described in the next chapter. The code for the application interface is in compute nodes and the others are in I/O nodes.

3.4.1 I/O Request Handler

When an I/O node receives a request from a compute node, the I/O request handler decodes requests and directs the request to other components. To simplify the implementation and to focus our research on the application interface, we assumed that only one request is served at a time and designed the I/O request handler as a separate thread in I/O nodes. The I/O request handler remains idle until a request arrives from a compute node; I/O requests that arrive while the I/O request handler is serving another request are queued.

The I/O request handler translates requested logical disk block information into physical disk block(s) information one after another in the way we explained in the previous section. While mapping LDBs into physical disk blocks, it generates *data segment request(s)* for the mapped physical disk block(s). A compute node that requests the disk block, memory address of the compute node to put or get data, data size to access to the disk block, offset to the disk block, number of data segments, and offset between data segments are the main information for a data segment request (see Figure 3.9).

For each mapped physical disk block the request handler searches a hash table called *table of mapped blocks* in which each entry is associated with a physical disk block and contains the data segment requests for the disk block (see Figure 3.9). If the block is found, it adds data-segment requests to the table entry and increases the number of requests for the block. It also creates a thread, called a *permuter*, to

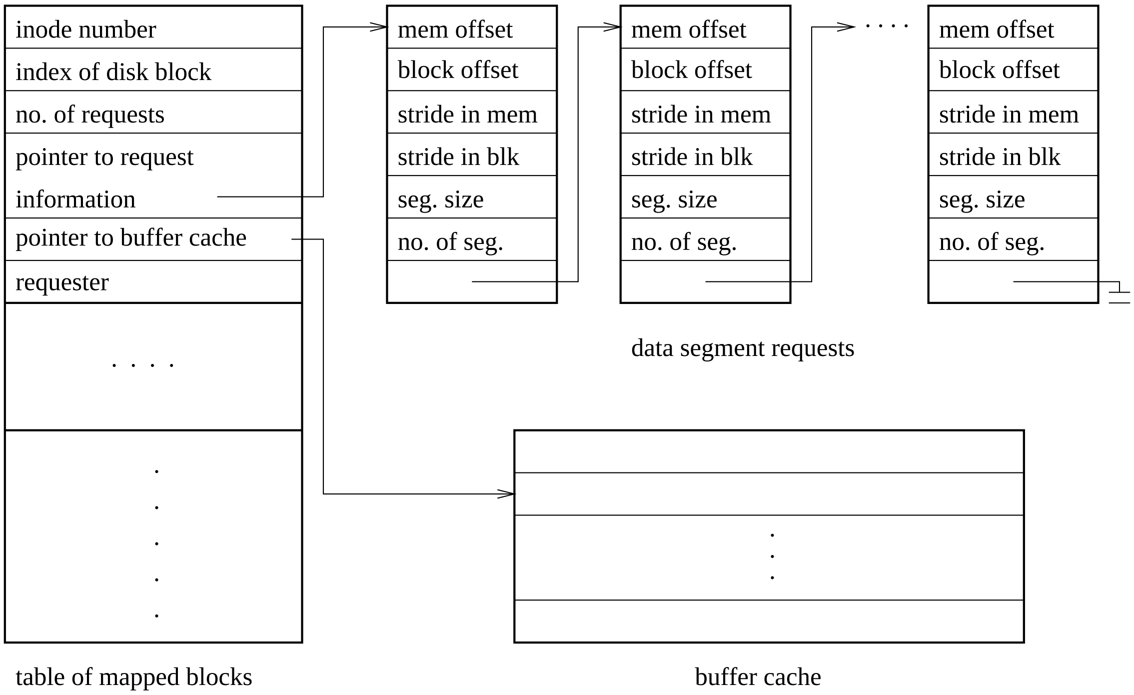


Figure 3.9: Main data structures in I/O node

serve data requests for the block (one permuter is associated with one block) if one has not been created.

Otherwise, it reserves a table entry, adds data request information, and passes the index of the mapped physical disk block(s) to the *cache manager*. It adds the pointer to the buffer, which is assigned (or found) and returned by the cache manager to the hash table; if the block is found in the buffer cache, the I/O request handler creates a permuter to serve data requests for the block. Otherwise, the I/O request handler passes the index of the mapped block and the pointer to the table entry to the *disk block manager*. If there is no free slot in the table of mapped blocks,[2] the I/O request handler remains idle until a table entry is released by a permuter.

[2]The number of table entries is the same as the number of blocks in the I/O node's buffer cache.

3.4.2 Cache Manager

Each I/O node has a thirty-two 4-KB-buffer cache that uses LRU-replacement and write-back strategy.[3] The cache manager maintains a hash table that contains the pointers to all buffers of the cache. For each disk block requested by the I/O request handler, the cache manager searches the block in the cache through the hash table. If the block is found, it marks the buffer *reserved clean* (the buffer was marked *clean*) or *reserved dirty* (the buffer was marked *dirty*) to keep the block from evacuating before using the block and then it returns that buffer's pointer to the I/O request handler.

Otherwise, it reserves a slot of the buffer cache, marks it *reserved*, and passes that buffer's pointer to the I/O request handler. An LRU list of all blocks in the cache is maintained on the hash table. If there are no more free blocks in the cache, the cache manager uses the list to determine a block to be replaced; the victim is the first block in the LRU list marked *clean* or *dirty*. If the replaced block is marked *dirty*, the cache manager makes the I/O request handler write out the block before a new block is brought.

3.4.3 Disk Block Manager, Permuter, and Name Server

The *disk block manager* logically partitions a disk into 4-KB blocks. It maintains a sorted list of disk blocks that are requested by the I/O request handler, rather than using a specific disk-scheduling algorithm, manages physical disk blocks, accesses the disk, and creates threads for the permuter.

If the status of the buffer cache that is passed by the I/O request handler

[3]We do not implement any specific cache so that our cache can be thrashed.

is *dirty*, it writes the data in the buffer cache back to the disk. The disk block manager brings a disk block that passed by the I/O request handler into the buffer, and creates a new thread for the block.

The permuter created by the I/O request handler or disk block manager puts (read operation) or gets (write operation) data segments to or from compute nodes according to the data segment request information on the table of mapped blocks. That is, in a read operation it sends a data segment to a compute node, while in a write operation it sends a request for a data segment to a compute node. After completing a data segment request, it decreases the number of requests. If the number reaches 0, it changes the status of the block in the cache: *clean* or *dirty* for read or write operation, respectively. It also moves the block to the end of the LRU list on the hash table for the buffer cache.

The main functions of the name server are naming and authentication. The name server is placed in each I/O node and there is a master name server among the name servers for the global logical disk storage model. The master name server is accessed only once — when a global logical disk is newly created or open.

In the next chapter we describe the application interface in detail, including the performance evaluation for the functionality of the interface. We minimized our efforts to design other components because we believe that we can use the software of other I/O subsystems for the other components after mapping LDBs into physical disk blocks, and we focused our research on the application interface. Thus, we have briefly described the functionality of the other components in this chapter; we will explain them more in succeeding chapters.

Chapter 4

Basic Application Interface of UPIO

The UPIO application interface is closely related in its functionality to the file system interface of existing parallel file systems. However, the interface of UPIO provides a flexible file model (structure) and user controllability.

UPIO provides two storage models with two application interfaces for each model; both interfaces can be used in an application at the same time. The interface functions for the storage models are classified into functions that deal with data domain as logical disk (e.g., logical disk creation and deletion), functions that deal with data access through logical disks, and functions that deal with logical disks (e.g., distribution of logical disk blocks across I/O nodes).

4.1 Logical Disk Handling Functions

In conventional file systems for parallel machines, files are structureless — that is, they are just a stream of bytes as in the Unix file system. The facility of the Unix I/O is simple, easy to use, and allows programmers to be independent of data type. It can therefore be applied in a uniform way to all I/O devices. However, it can be a disadvantage in a parallel computing environment because, in that facility, it is difficult to support domain-specific I/O operations and to increase I/O performance.

Some nonconventional file systems, such as Vesta [14, 10, 11, 12, 13] and Galley [72, 73], provide a two- and three- dimensional file structure, respectively. In Galley, however, there is no notion of file-level read/write operations. Each read/write operation is applied to only a single fork of a subfile; each fork of a subfile is similar to a traditional Unix file.

In UPIO the structure of a file is defined by mapping data domain into a logical disk, which is just an n-dimensional abstract block-index domain. The mapping between the data domain and a logical disk is simple: decompose the data domain based on the semantics of the data domain and map each decomposition into one index of the logical disk. We call each index of the logical disk a *logical disk block*; each decomposition of the data domain is associated with a logical disk block.

UPIO allows programmers to define the parameters that specify a file structure (logical disk). That is done by `LD_creat` and `GLD_creat` functions, where `LD_creat` and `GLD_creat` are for the local logical disk model and global logical disk model, respectively. The main parameters of the `LD_creat` and `GLD_creat` are the information for a logical disk and logical disk block that programmers are trying to define.

Logical disk information includes logical disk name,[1] dimension of the logical disk, upper bound of each dimension, type of logical disk block, and indexing scheme of the logical disk blocks. Logical disk block information includes the specification of the data-domain decomposition and the size (type) of the data element (integer, floating point, byte stream, user-defined data type, etc.).

In UPIO the only way to create a logical disk is to use the `LD_creat` or `GLD_creat` function. These functions have I/O nodes assign an *inode*[2] in each I/O node and return a *logical disk descriptor* (LDD), which is the only data structure containing metadata for a logical disk in compute nodes and is used to access file data.

The metadata for an LD is distributed across compute nodes. That is, in an LLDM each compute node has its own metadata for a local LD, and the metadata for a GLD is replicated in each compute node in a GLDM when a GLD is created or open. So there is no single point in UPIO to limit scalability due to metadata.

The functions are different from other Unix-like file systems that can create a file with *open* system call. In UPIO the function of `LD_open` and `GLD_open` is simply to copy the metadata of an LD from the I/O nodes to the compute node. The counterparts of `LD_creat` and `GLD_creat` are `LD_delete` and `GLD_delete`, respectively. The semantics of the functions are straightforward.

A logical disk can grow beyond its initial upper bound toward a certain dimension of the LD by using the `LD_adjust` and `GLD_adjust`, depending on the shape and the indexing scheme of a logical disk. As an example, a two-dimensional logical disk

[1]This is the same as the name of the file in the Unix file system.

[2]The term "inode" is commonly used in literature on the Unix file system. In this dissertation we use the same term for the internal representation of a file, because we have implemented the Unix-like file system layout for the I/O processing in each I/O node.

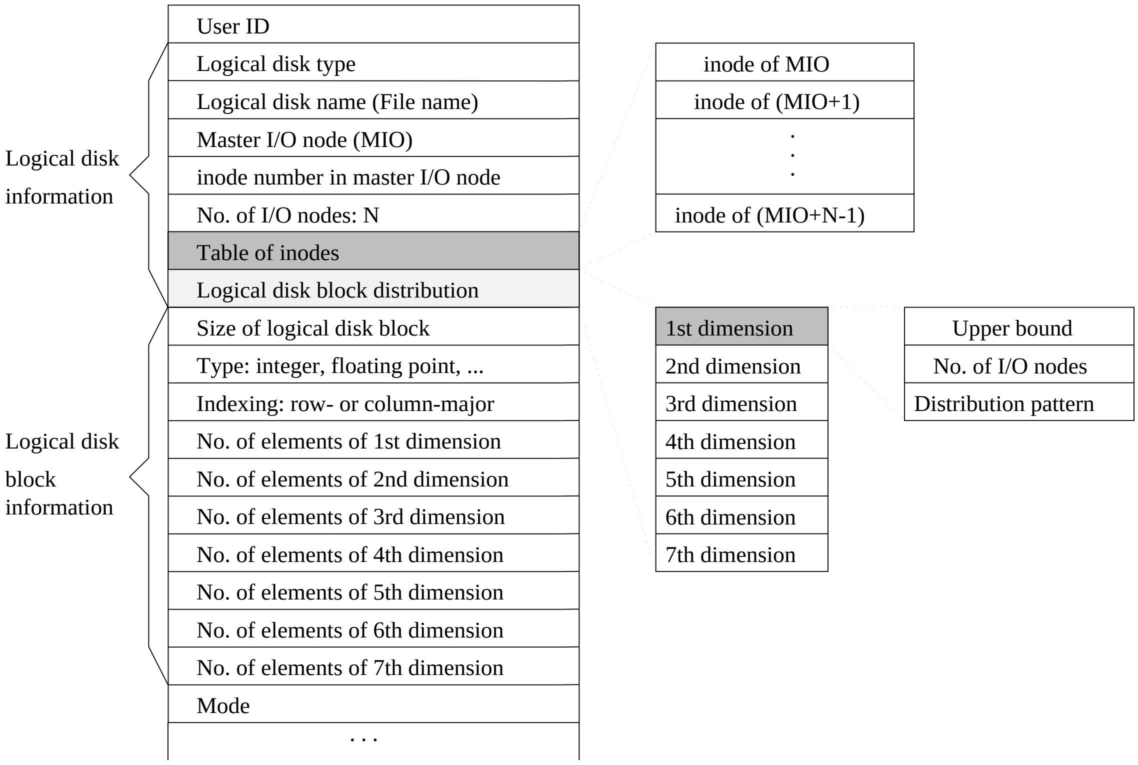


Figure 4.1: Logical disk descriptor (LDD)

ordered in column-major can grow beyond the initial upper bound of the column of the logical disk.

Logical Disk Descriptor

In UPIO the metadata for a logical disk, such as the information of the logical disk and logical disk blocks, is distributed across compute nodes. All this information is stored in a data structure called *logical disk descriptor* and is used in all logical disk access functions. That is, LDD is the only metadata for a logical disk in compute nodes. The main structure of LDD is shown in Figure 4.1, and the following are the main fields of LDD:

- LD owner: Owner of the logical disk. Other authentification information, such as access permissions, is included.
- Logical disk type: Logical disks may be local logical disks or global logical disks.
- Logical disk name: The name of a logical disk
- Master I/O node: The I/O node to place the first block of a logical disk block.
- inode number in master I/O node: Inode index in the master I/O node.
- Number of I/O nodes: The number of I/O nodes in which logical disk blocks are distributed.
- Table of I/O nodes: Inode numbers of each I/O node that stores logical disk blocks.
- Logical block distribution: The upper bound of logical disk blocks in each dimension of a logical disk, the number of I/O nodes (to be) used to distribute the logical disk blocks in each dimension, and the distribution pattern of logical disk blocks in each dimension.
- Size of a logical disk block: We assume that the size of each logical disk block in a logical disk is the same.
- Type: The size of element data contained in a logical disk block.
- Indexing: The indexing scheme of logical disk blocks.
- Number of data elements: The number of data elements in each dimension of physical data consisting of a logical disk block.
- Mode: Represents access mode. For some applications data consistency can be controlled by the applications, but UPIO provides the following access modes

for other applications in which consistency should be enforced. The access mode is defined when a global logical disk is created or open.

- `ANY`: In this mode each compute node accesses global logical disk blocks independently (default access mode).
- `SYNC`: In this mode each compute node accesses global logical disk blocks in a certain order (for example, in the increasing order of the compute node numbers).

Creating a Logical Disk

Figure 4.2 depicts the interaction between compute nodes and I/O nodes for the creation of logical disks. Each compute node assigns a logical disk descriptor for a logical disk as the first step toward creating an LD. That is, the LDD is allocated to replicating metadata for a global logical disk in GLDM and for each local logical disk in LLDM.

After that, in LLDM each compute node sends a message that includes the name of the logical disk, the information for the logical disk and the logical disk block, and the information for authentication to a corresponding I/O node in order to request an inode. The name server in each I/O node inspects the name of the logical disk that is to be newly created. If there is a logical disk with the same name, the I/O nodes send an error message to the compute nodes. Otherwise, the name server registers the name of the logical disk to a directory and calls the disk block handler to assign an inode for the logical disk. Therefore, every logical disk has only one inode.

The disk block handler in each I/O node assigns an inode, stores the information sent by a compute node, and passes the inode number to the compute node

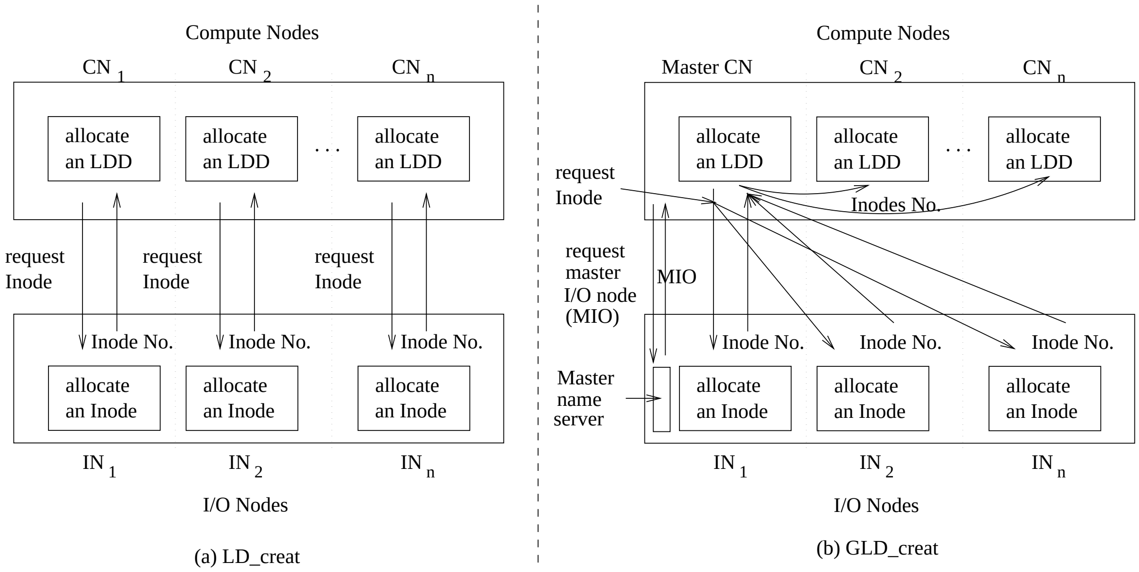


Figure 4.2: Interactions between compute nodes and I/O nodes

as the result of the `LD_creat` operation. Each compute node then stores the inode number in the LDD.

In GLDM the interaction for the creation of a global logical disk is different from that in LLDM, because inodes are not allocated at this time. In this model each compute node simply allocates an LDD and stores the information for the global logical disk in the LDD as in the `LD_creat` function. The global logical disk blocks are distributed across I/O nodes by the `GLD_distribute` function, which is described in the next section; an inode for the global logical disk is assigned to each I/O node at this time.

In this model only one compute node, the *master compute node*, sends a message with the name of a global logical disk to be created to a master name server in an I/O node. The name server checks the existence of the global logical disk. If a logical disk with the same name is found, the name server sends an error message to the master compute node. Otherwise, the name server registers the name of the global logical disk in a directory, determines a master node where the first global logical disk block will be placed, and then sends the information concerning the master I/O node to the master compute node. In UPIO each I/O node becomes a master I/O node in a round-robin fashion. That is, the first global logical disk blocks are allocated in a round-robin fashion on the I/O nodes.

When the master compute node receives the information about the master I/O node, it determines the I/O nodes to which it sends a request based on the parameters specified in `GLD_distribute` function. Each of the I/O nodes then assigns an inode and returns the inode number to the master compute node. The master compute node finally broadcasts the pair of (I/O node number, inode) to the other compute nodes.

4.2 Logical Disk Block Distribution Functions

In the SPMD model, partitioning data among the compute nodes is a very useful feature in scientific and engineering applications, because parallelization is done by data partitioning. Load balance, locality of access, and other optimizations can also be achieved by data partitioning. If the amount of data is too huge to fit into the memory at one time, another distribution should be considered in order to perform I/O efficiently. That is, for higher bandwidth it is also important to distribute data across disks matching the application access patterns.

A number of studies [10, 11, 48, 91] showed that it is important to allow applications to distribute data across disks, matching the application access pattern for exploiting the aggregate bandwidth of a large number of disks. In conventional parallel file systems such as CFS and PFS it is possible to exploit high bandwidth by striping data across the disks. These systems distribute data in a simple round-robin fashion. However, this simple distribution can degrade I/O performance, since the distribution unit size and pattern may not match the application request size and access pattern. In an extreme case, concurrent I/O requests can be directed to a single disk and then to another disk, and so on.

In UPIO, especially for the global logical disk model, we allow programmers to distribute data across a set of I/O nodes by considering the application access patterns. This is possible because programmers map data into a logical disk, which is just a multidimensional, abstract index domain. They are thus able to distribute data across the I/Onodes by decomposing the abstract-index domain, which is similar to data partitioning in some programming languages such as HPF[31], Fortran D[33], or Vienna Fortran[6].

Figure 2.1 shows a simple example of data partitioning[3] that is relatively agreed upon for partitioning multidimensional arrays for internal computations. This is done by a set of programming language directives. There are four representative options: `BLOCK`, `CYCLIC`, `BLOCK_CYCLIC`, and $*$.

In UPIO the distribution of LDBs is done by the `GLD_distribute` function and will affect the ability of programmers to exploit high bandwidth. The `GLD_distribute` function takes an LDD and assigns an *attribute* to each dimension of the decomposition of a global logical disk. All data access functions in UPIO for GLDM apply the distribution to all data blocks mapped to the logical disk blocks. Each attribute represents the mapping of the logical disk blocks in that dimension of the decomposition; attributes in each dimension are independent.

UPIO adopts the four types of attributes (`BLOCK, CYCLIC, BLOCK_CYCLIC,` and $*$) and provides more distribution schemes to support various external algorithms efficiently. The semantics of the adopted options are exactly the same as

[3]In this dissertation we use *partitioning* to mean mapping data to compute nodes, and *distribution* to mean distributing data across I/O nodes.

the `DISTRIBUTE` statement of Fortran D, but the method of BLOCK distribution in UPIO is a little different from that in Fortran D when logical disk blocks are not distributed evenly.

4.2.1 Distribution Functions

All data-access primitives for GLDM in UPIO use the information specified by the `GLD_distribution` function to calculate the mapping of logical disk blocks to I/O nodes. The mapping function, f,

$$f_L(j) = (\imath_L(j), \ell_L(j)) = (d, j_l)$$

is a mapping of the index j of a logical disk block of a global logical disk L to the index j_l of the sub-logical disk blocks assigned to I/O node d. Each mapping function has two component functions, $\imath$ and ℓ. These functions are used to compute the I/O node in which the logical disk block is placed and location information of the logical disk block in the I/O node. For a given global logical disk L, the I/O node function $\imath_L$ maps the index of the global logical disk block to its unique I/O node d, and the local index function ℓ_L maps the index to index j_l of the sub-logical disk blocks assigned to I/O node d.

4.2.2 BLOCK Distribution

`BLOCK` distribution divides a logical disk into equal-sized blocks, that is, an equal number of logical disk blocks, assigning each block to each I/O node successively.

Figure 4.3 shows some examples of `BLOCK` distribution, and following are cor-

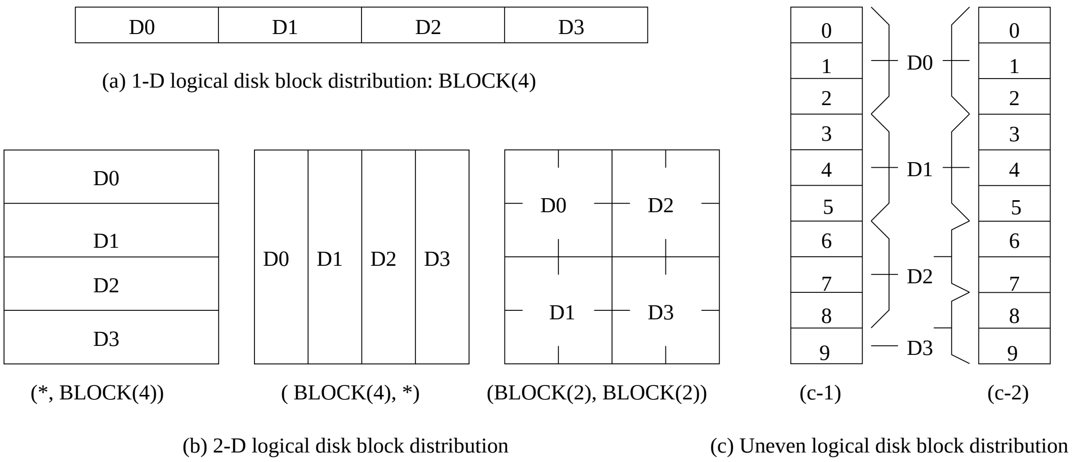


Figure 4.3: Examples of BLOCK distribution

responding usages of the figure.[4]

`GLD_distribute`(*ldd1*, `BLOCK(4)`)

`GLD_distribute`(*ldd2*, $*$, `BLOCK(4)`)

`GLD_distribute`(*ldd2*, `BLOCK(4)`, $*$)

`GLD_distribute`(*ldd2*, `BLOCK(2)`, `BLOCK(2)`)

In the examples, *ldd1* and *ldd2* represent, respectively, the LDDs of the 1-D and 2-D global logical disks shown in Figure 4.3. The global logical disks are distributed across four I/O nodes. The semantics of the distribution function is the same as the `BLOCK` distribution in Fortran D when the logical disk blocks are distributed evenly. However, the distribution is a little different from that in Fortran D when the logical disk blocks are not distributed evenly.

Let's suppose a 1-D array of size N is to be distributed across P processors in Fortran D, then $\lceil \frac{N}{P} \rceil$ elements of the array are assigned to each processor. But, in this distribution, a processor may have $N - \lceil \frac{N}{P} \rceil \times (P-1)$ elements of the array (see

[4]In this dissertation we define dimensions as x and y. The dimension x is defined along columns, and y is defined along rows. The attributes are written from the dimension x to y in the figure.

Figure 4.3(c-1)). If we assume the unit of load of each processor to be the number of elements assigned to each processor, the load imbalance can be

$$\lceil \frac{N}{P} \rceil - (N - \lceil \frac{N}{P} \rceil \times (P-1)) = \lceil \frac{N}{P} \rceil \times P - N.$$

The load imbalance may be a cause of I/O performance degradation, because the utilization of I/O nodes can be decreased by directing I/O requests to some I/O nodes instead of directing them to all I/O nodes evenly. UPIO always maintains a load imbalance of at most one in `BLOCK` distribution by distributing logical disk blocks as follows:

1. Assume that a 1-D logical disk of size N is to be distributed across D I/O nodes.

2. Calculate the minimum number of logical disk blocks, m, that will be distributed to each I/O node and the remaining logical disk blocks, r, where $m = \frac{N}{D}$ and $r = N \bmod D$.

3. Distribute one more logical disk block of the remaining logical disk blocks, r, to each r I/O node if $r > 0$. The logical disk blocks are distributed as follows (see Figure 4.3(c-2)):

   - I/O node 0: logical disk blocks from $0, 1, ..., \frac{N}{D}$
   - I/O node 1: logical disk blocks from $\frac{N}{D}, \frac{N}{D}+1, \frac{N}{D}+2, ..., 2(\frac{N}{D}+1)-1$
   - ...
   - I/O node $r-1$: logical disk blocks from $(r-1)(\frac{N}{D}+1), (r-1)(\frac{N}{D}+1)+1, ..., r(\frac{N}{D}+1)-1$
   - I/O node r: logical disk blocks from $r(\frac{N}{D}+1), r(\frac{N}{D}+1)+1, ..., r(\frac{N}{D}+1)+\frac{N}{D}-1$

- I/O node $r+1$: logical disk blocks from $(r+1)\frac{N}{D}+r, (r+1)(\frac{N}{D}+1), ..., (r+2)\frac{N}{D}+r-1$
- ...
- I/O node $D-1$: logical disk blocks from $(r+2)\frac{N}{D}+r, (r+2)\frac{N}{D}+r+1, ..., N-1$

4.2.3 SKEW_CYCLIC and BLOCK_SKEW_CYCLIC Distributions

`SKEW_CYCLIC` and `BLOCK_SKEW_CYCLIC` are the types of attributes that UPIO most supports. The layout for SKEW_CYCLIC distribution is shown in Figure 4.4. In this figure, 12×12 LDBs are distributed over six I/O nodes that are logically arranged into 1×6 (see Figure 4.4 (a)) and 2×3 (see Figure 4.4 (b)) grids. The number in each square represents the I/O node on which the LDB is stored. This distribution allows efficient access of LDBs in a row or column, because the balanced load in each I/O node yields a high I/O bandwidth.

Programmers can logically arrange I/O nodes into any dimension of grid and distribute LDBs onto the I/O nodes in a skewed-cyclic fashion. Following is the mapping function for distributing 2-D LD in a SKEW_CYCLIC fashion over the I/O nodes, which are logically arranged into an $N_x \times N_y$ grid where N_x and N_y are the number of I/O nodes in each dimension and (x, y) is the index of an LDB:

$$\imath_L(x,y) = (((\tfrac{x}{N_x}+y) \bmod N_y) * N_x + (\tfrac{y}{N_y}+x) \bmod\ ) \bmod (N_x * N_y).$$

BLOCK_SKEW_CYCLIC is similar to SKEW_CYCLIC, but it first divides the LDBs in each dimension into contiguous chunks of certain size and then assigns these chunks in the same fashion.

0	1	2	3	4	5	0	1	2	3	4	5
1	2	3	4	5	0	1	2	3	4	5	0
2	3	4	5	0	1	2	3	4	5	0	1
3	4	5	0	1	2	3	4	5	0	1	2
4	5	0	1	2	3	4	5	0	1	2	3
5	0	1	2	3	4	5	0	1	2	3	4
0	1	2	3	4	5	0	1	2	3	4	5
1	2	3	4	5	0	1	2	3	4	5	0
2	3	4	5	0	1	2	3	4	5	0	1
3	4	5	0	1	2	3	4	5	0	1	2
4	5	0	1	2	3	4	5	0	1	2	3
5	0	1	2	3	4	5	0	1	2	3	4

(a) LDB layout over a 1-by-6 logical grid

0	1	2	3	4	5	0	1	2	3	4	5
3	4	5	0	1	2	3	4	5	0	1	2
1	2	0	4	5	3	1	2	0	4	5	3
4	5	3	1	2	0	4	5	3	1	2	0
2	0	1	5	3	4	2	0	1	5	3	4
5	3	4	2	0	1	5	3	4	2	0	1
0	1	2	3	4	5	0	1	2	3	4	5
3	4	5	0	1	2	3	4	5	0	1	2
1	2	0	4	5	3	1	2	0	4	5	3
4	5	3	1	2	0	4	5	3	1	2	0
2	0	1	5	3	4	2	0	1	5	3	4
5	3	4	2	0	1	5	3	4	2	0	1

(b) LDB layout over a 2-by-3 logical grid

Figure 4.4: Examples of SKEW_CYCLIC distribution

4.3 Accessing Logical Disk Blocks

As it is often necessary to change the distribution of data at runtime in distributed-memory parallel computers for internal computation, a file needs to be used with multiple internal views, either by different programs or by different phases of computation within the same program [17]. In this case, serious mismatches affect the bandwidth and decrease the I/O performance. Some possible alternatives for dealing with the problem are presented by Crockett [17]. Converting the data format according to the internal view of the file can be an alternative, but this could be expensive for large files. Providing a software interface to present multiple internal views when needed becomes another alternative, but with possibly degraded performance.

UPIO provides a variety of data access functions that can present multiple internal views that minimize the degradation of performance. In this chapter we describe basic read and write functions of UPIO. More complex functions for presenting a variety of access patterns will be described in the next chapter.

Many scientific applications require the ability to read and write several gigabytes of data in a random-access and sequential fashion [45]. With the conventional parallel file system interface, it is difficult to present such access patterns efficiently. For example, a number of data chunks that are not contiguous cannot be accessed at one time, because this access entails explicit or implicit *seek* operations to adjust an access point. The interfaces may require cooperation between compute nodes and many I/O requests to access the data chunks, and this may degrade the system's performance.

In UPIO all accesses to the data on physical disks are made through logical disks defined by programmers. Specifying the indices of logical disk blocks is the only thing that should be done by programmers in order to access data on physical disks. Therefore, programmers can easily present access patterns with a minimum number of I/O requests.

Let's suppose a two-dimensional matrix is stored on a physical disk in column-major order and a user is trying to access a data region, say $r \times c$. Then, c I/O operations are required to access the data region in conventional file systems. Of course, c explicit or implicit seek operations are also required to adjust the access point, because the r data elements in c columns are not contiguous. If a logical disk is used, however, one I/O operation is enough to access the data region, because the $r \times c$ region can be defined as a logical disk block (see Figure 4.5). Even if it is not defined as a logical disk block, in UPIO one I/O operation is enough. (This will be explained in the next section.)

It is also possible to access multiple noncontiguous logical disk blocks at one time by specifying the indices of the logical disk blocks if they are random. If the logical disk blocks have a constant distance between LDBs, programmers can access

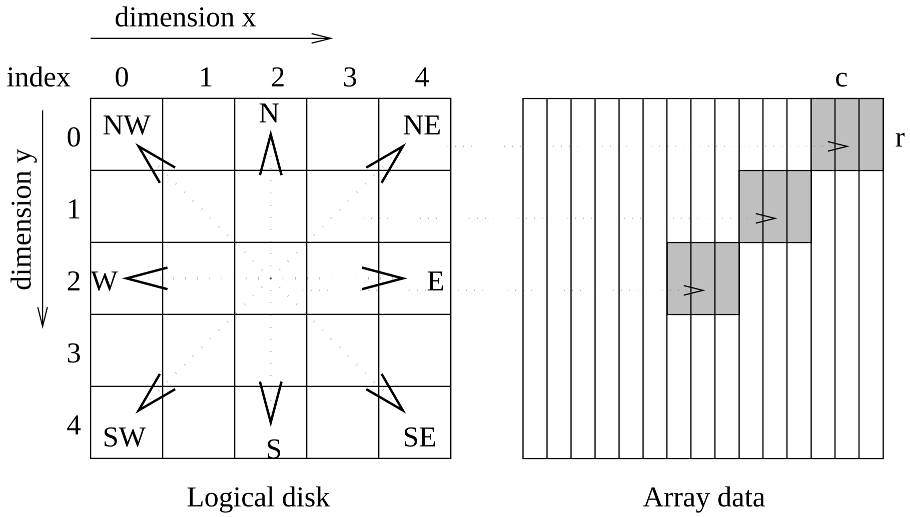


Figure 4.5: Example directions to read/write LDBs in a 2-dimensional LD

them by specifying the index of the first logical disk block, the total number of logical disk blocks to access, and the direction of access. The direction is represented by the distance between LDBs, and following is the data structure for direction information `NEWS` for a two-dimensional logical disk. Figure 4.5 shows some example directions that programmers can specify:

```
typedef struct {
    int x;    /* distance to the next LDB in dimension x */
    int y;    /* distance to the next LDB in dimension y */
} NEWS;
```

Suppose the two-dimensional matrix is mapped into a 5×5 LD as shown in Figure 4.5 and a user wants to access the data chunks represented as shaded rectangles in that figure. Since the data chunks are not contiguous, $3c$ I/O operations are necessary in conventional file systems. In UPIO, however, users can access the data with one I/O operation by identifying the logical disk blocks that are mapped into the data chunks. The data chunks are mapped into the logical disk blocks, (2,2), (1,3), and (0,4) when we assume that rc data elements are mapped into a logical disk block. All of the LDBs can be accessed at one time with the following statement:

`LD_read`(*ldd, mv_cnt, memv, cnt, news, index*)

This function reads three (= *cnt*) logical disk blocks, beginning at the logical disk block *index*, (2, 2), toward the direction *news*, and stores data into the memory as specified by the memory vector *memv*, where *ldd* is the LDD. The direction information is represented by setting the value of x and y as 1 and -1, respectively, (i.e., $news.x = 1$ and $news.y = -1$) in the data structure for direction. The *memv* is a pointer to an array of (*mem, lcnt*); *lcnt* LDBs are stored beginning at *mem*; *mv_cnt* means the number of the array elements.

The I/O latency in UPIO to access the data chunks should be less than that of conventional file systems, because the number of I/O operations is $3c - 1$ times less than that of conventional file systems. With a nested-strided request of Galley [72, 73] it is possible to access the data region, but in that system all data regions should be in a fork, because there are no file-level read and write operations. MPI-IO [8, 9] or Vesta [14, 10] may represent the access pattern with a single I/O request, but it is more complex than UPIO.

The operation explained above is a blocking I/O request. UPIO also provides nonblocking I/O requests.[5] The basic blocking and nonblocking access functions to logical disk blocks are summarized in Figure 4.6. All parameters for nonblocking functions are the same as those for blocking functions, but each nonblocking function returns a request handle, *flag*, which is a pointer to an internal UPIO data structure and is used to check the status of the read or write request (via `LD_done`).

[5]From now on we will describe only blocking I/O operations, but there are corresponding nonblocking I/O operations in UPIO.

After calling `LD_done`, the application will wait until the nonblocking read or write operation has completed.

Local Logical Disk Model
`LD_read(ldd, mv_cnt, memv, cnt, news, index)`
reads `cnt` LDBs beginning at `index` into memory as specified by `memv`
`LD_write(ldd, mv_cnt, memv, cnt, news, index)`
writes data in memory to `cnt` LDBs beginning at `index`
`flag = LD_aread(ldd, mv_cnt, memv, cnt, news, index)`
asynchronous read operation
`flag = LD_awrite(ldd, mv_cnt, memv, cnt, news, index)`
asynchronous write operation
`LD_done(flag)` waits for the completion of asynchronous read/write
Global Logical Disk Model
`GLD_read(ldd, cflag, mv_cnt, memv, cnt, news, index)`
reads `cnt` LDBs beginning at `index` into memory
`GLD_write(ldd, cflag, mv_cnt, memv, cnt, news, index)`
writes data in memory to `cnt` LDBs beginning at `index`
`flag = GLD_aread(ldd, cflag, mv_cnt, memv, cnt, news, index)`
reads `cnt` LDBs beginning at `index` into memory asynchronously
`flag = GLD_awrite(ldd, cflag, mv_cnt, memv, cnt, news, index)`
write data in memory to `cnt` LDBs beginning at `index` asynchronously

Figure 4.6: Basic logical disk access functions

There is no difference in functionality of logical disk access functions for LLDM and GLDM; however, in GLDM the indices of logical disk blocks being accessed are translated into the indices of sub-logical disk blocks assigned to each I/O node, because all logical disk blocks of a global logical disk are distributed across I/O nodes. Each I/O node has sub-logical disk blocks as the portion of the global logical disk allocated to the I/O node, and the I/O requests in GLDM can therefore be directed to several I/O nodes.

The read/write functions for GLDM have one more parameter, *cflag*, than do those for LLDM. It is used to specify whether the I/O request from each compute node should be serviced independently or collectively. The runtime support for user-controllable collective I/O is described in Chapter 8 in detail.

4.4 Accessing Segments of Logical Disk Blocks

In a multicomputer with a parallel file system the I/O performance will be better when processors can make a small number of high granularity I/O requests instead of a large number of low granularity I/O requests. However, Kotz and Nieuwejaar [58] showed with a tracing to file-related activity on a massively parallel computer that while almost all reads and writes were for fewer than 4000 bytes, those reads and writes transferred only 2.0% and 3% of all data read and written, respectively. Ousterhout, Baker, and Miller [69] showed that a common characteristic of file workloads, especially scientific workloads, is that files are accessed sequentially.[6] These entail a number of small requests, because at present the interface provided by conventional parallel file systems does not support accesses with stride.

One of the best ways to generate a small number of high granularity I/O requests is to partition data among processors that conforms to the way it is distributed on disks. If a data array is stored in column-major order on disks, a column-block distribution [31] among processors will make a small number of high granularity I/O requests, since each processor is able to read high granularity of data directly from disks. If data is distributed among processors in a row-block distribution fashion, it will make a large number of low granularity I/O requests.

[6]Kotz [58] defined a *sequential* request to be one that is at a higher file offset than the previous request from the same compute node, and a *consecutive* request to be a sequential request that begins where the previous request ended.

We believe that the abstraction of the conventional one-dimensional file is one of the main reasons for a large number of low granularity I/O requests. However, a logical disk is a good tool for extending the abstraction of files and allowing programmers to manipulate I/O explicitly, because programmers are able to define a logical disk and logical disk block while considering data access patterns and maintaining total control over the disk.

In UPIO, programmers can access a portion of LDBs. We call the data elements in that portion a *segment* of LDBs. A segment can be specified in a single LDB as well as across a number of LDBs with the data structure in Figure 4.7 (e), i.e., the coordinators of the first data element in an LDB and the number of data elements in each dimension. The black rectangles shown in Figure 4.7 are examples of segments of LDBs, which are defined in Figure 4.7 (f).

Figures 4.7 (a), (b), and (c) show the data array stored in column-wise block fashion on physical disks through a 1-D LD, say $A(4)$. Figure 4.7 (d) shows the data array stored in block-wise block fashion through a 2-D LD, say $A(2,2)$.

Let's suppose a compute node accesses the segments in Figure 4.7. In UPIO the segments can be accessed at the same time with the statement:

`LD_reads`(*ldd, mem, cnt, news, segment, order, index*)

Beginning at the LDB *index*, this function reads data specified in *segment* from or across *cnt* LDBs toward the direction *news* and stores the data in memory *mem*. The data segment is specified as shown in Figure 4.7 (f), i.e., the coordinators of the first data element are assumed to be x and y, and the number of data elements in dimensions 0 and 1 is assumed to be w and h, respectively; *order* represents how to read data segments from LDBs and it has the following values and meanings for two- and three-dimensional data array:

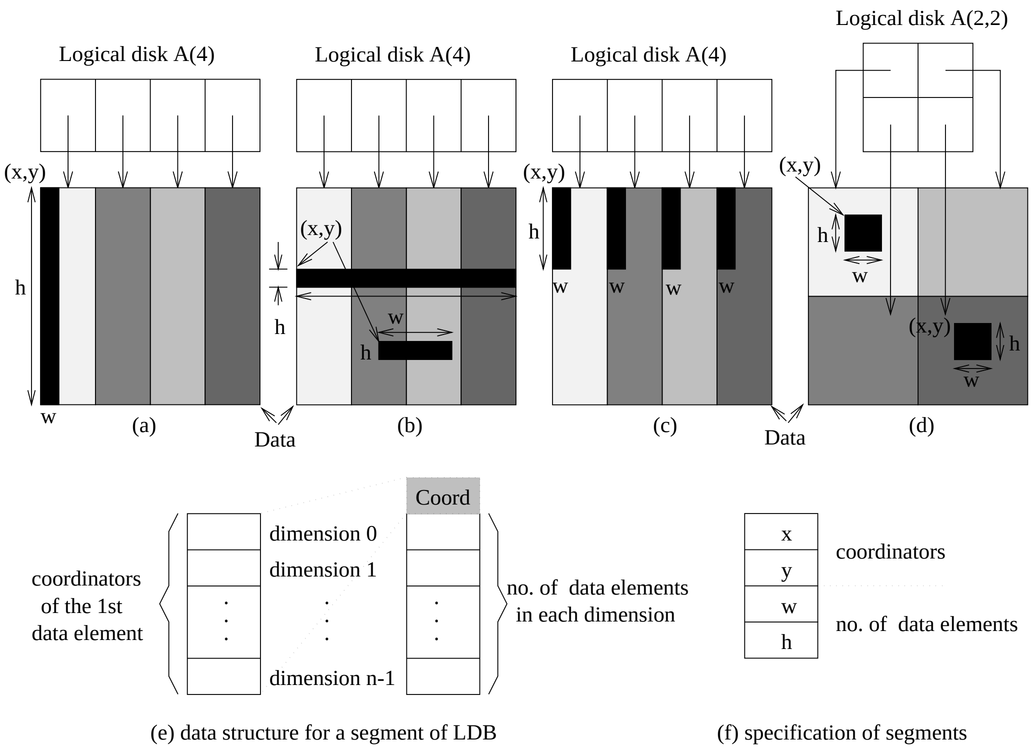


Figure 4.7: Accessing data segments through a logical disk

- X: access data segments from dimension x to y
- Y: access data segments from dimension y to x
- XY: access data segments from dimension x to y and finally to z
- YX: access data segments from dimension y to x and finally to z
- XZ: access data segments from dimension x to z and finally to y
- YZ: access data segments from dimension y to z and finally to x
- ZX: access data segments from dimension z to x and finally to y
- ZY: access data segments from dimension z to y and finally to x

Let's consider Figure 4.7 (a). In this case, conventional parallel file systems can also access the data segment with a single I/O operation. However, in other cases shown in Figure 4.7 it is impossible to access the segment with a single I/O operation; w I/O operations are needed to access the segments shown in Figures 4.7 (b) and (d) if the data is stored in column-major order. Of course, $4w$ and $2w$ I/O operations are necessary to access the segments in Figure 4.7 (c) and (d), respectively.

If a logical disk is used, programmers can access the segments with a single I/O operation. Following are corresponding statements to read the segments shown in Figure 4.7:
`LD_reads`(*ldd, mem,* 1, *news, segment, Y,* $A(0)$): a segment in Figure 4.7 (a),
`LD_reads`(*ldd, mem,* 4, *news, segment, Y,* $A(0)$): a big segment in Figure 4.7 (b),
`LD_reads`(*ldd, mem,* 2, *news, segment, Y,* $A(1)$): a small segment in Figure 4.7 (b),
`LD_reads`(*ldd, mem,* 4, *news, segment, Y,* $A(0)$): all segments in Figure 4.7 (c),
`LD_reads`(*ldd, mem,* 2, *news1, segment, Y,* $A(0, 0)$): all segments in Figure 4.7 (d),
where the segment is defined as shown in Figure 4.7 (f) and the direction information,

Local Logical Disk Model
`LD_reads(ldd, mem, cnt, news, segment, order, index)` beginning at the LDB `index`, reads data segments specified in `segment` from or across `cnt` LDBs toward the direction `news` and stores the data contiguously in `mem`
`LD_writes(ldd, mem, cnt, news, segment, order, index)` beginning at the LDB `index`, writes data segments specified in `segment` from `mem` to disks
`LD_areads(ldd, mem, cnt, news, segment, order, index)` reads data segments asynchronously
`LD_awrites(ldd, mem, cnt, news, segment, order, index)` writes data segments asynchronously
Global Logical Disk Model
`GLD_reads(gldd, cflag, mem, cnt, news, segment, order, index)` reads data segments from a global logical disk
`GLD_writes(gldd, cflag, mem, cnt, news, segment, order, index)` writes data segments to a global logical disk
`GLD_areads(gldd, cflag, mem, cnt, news, segment, order, index)` reads data segments from a global logical disk, asynchronously
`GLD_awrites(gldd, cflag, mem, cnt, news, segment, order, index)` writes data segments to a global logical disk, asynchronously

Figure 4.8: Data segments access functions

news and *news1*, are set as $news.x = 1$, and $news1.x = 1$ and $news1.y = 1$, respectively.

Figure 4.8 shows the representative access functions to segments in logical disks. The structured interface of Galley [72, 73] and MPI-IO [8, 9] can access the segments with a single I/O operation, but in Galley the matrix must be stored in a single fork and in MPI-I/O the access pattern can be presented in a more complex form by using offsets.

Chapter 5

Performance of Basic Application Interface of UPIO

To explore the effects of UPIO we first limit our experiments to simple benchmarks with the basic application interface of UPIO for the local logical disk model. In this model each compute node has its own local file and access to the file; in our experiments we look at the behavior of the individual compute node. We assume that a 1024×1024 matrix is stored in each local file in column-major format using the file pointers in a traditional file system and a 1-D logical disk, respectively, to compare the performance of both cases under a similar environment.

We ran the program with both of the traditional file systems using file pointers and UPIO on top of our simulator.[1] We configured our system with one compute node and an I/O node for our experiments, because this was enough to observe the behavior of the individual compute node in LLDM. Our system has the following system specification. Each node has a 25 MHz CPU and a direct-memory access (DMA) capability. I/O is modeled as message passing with I/O requests and replies sent and received between a compute node and an I/O node by using DMA. The

[1] We implemented the same I/O operations using file pointers.

I/O node has a thirty-two 4-KB-buffer cache that uses an LRU-replacement and write-back strategy and an HP97560[85] disk that is attached to an SCSI bus whose peak bandwidth is 10 MB/s. The disk capacity and peak transfer rate are 1.3 GB and 2.34 MB/s, respectively. The file system block size is 4-KB.

5.1 Access Patterns

Figure 5.1 shows the access patterns that we used as file-access patterns in our experiments. We made these synthetic access patterns similar to the array-distribution possibilities in High-Performance Fortran [31] since, in parallel computation, the order may be mapped into compute node numbers. Patterns are named by the data access formats and access methods (row-wise, column-wise, stride, and cyclic). Names are started using a shorthand beginning with **S** (square region) for data access format. In these examples an 8×8 matrix is stored in column-major order and data is accessed square region by square region. Each square region of the matrix contains only one data element and is labeled with the access order.

In our experiments we defined three different data access formats: 16×16, which is smaller than the file system block size; 32×32, which is the same with the file system block size; and 32×64, which is larger than the file system block size. In all access patterns, stride is eight square regions.

We defined a 2-D logical disk for the matrix, considering the data access formats, and mapped each square region into a logical disk block. In this case we can store the data block-by-block, but not in row/column major order. This will reduce the number of disk accesses, because a region is mapped into one or more contiguous physical disk blocks. We assumed that the logical disk blocks are ordered in column-major manner in our experiments.

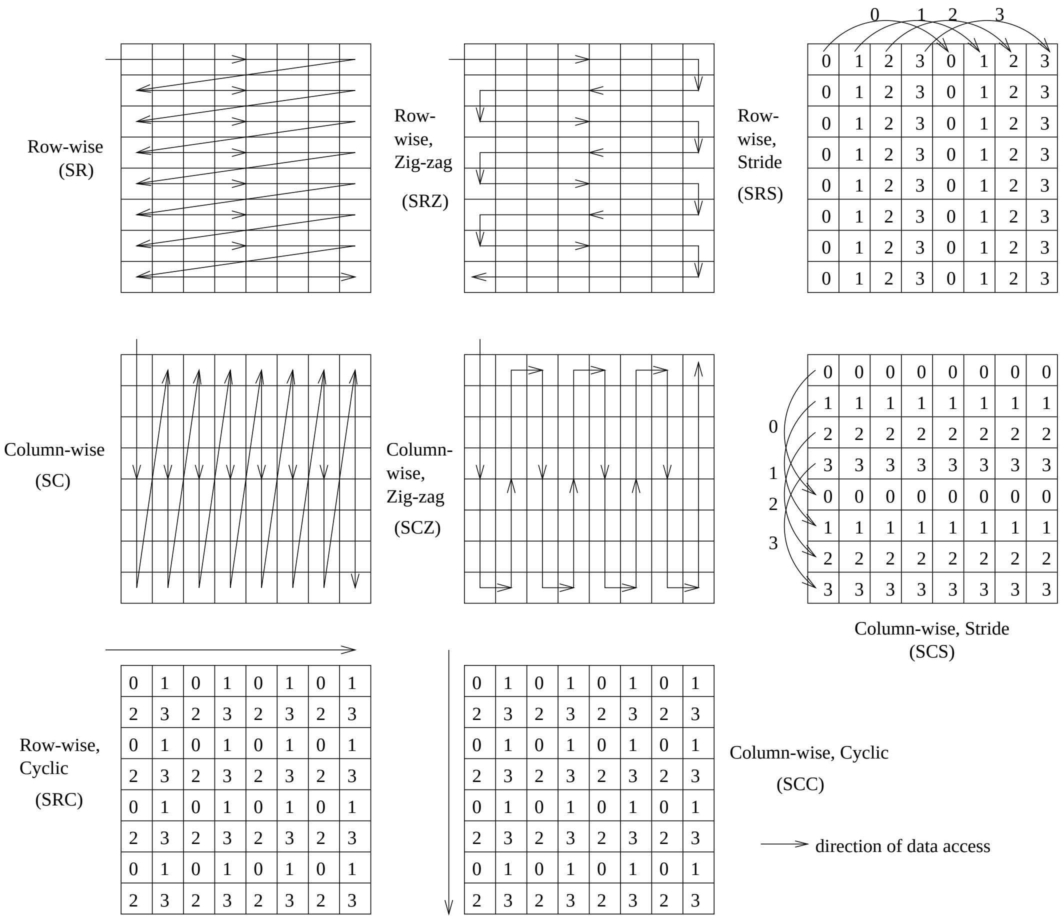


Figure 5.1: Examples of access patterns

5.2 Experimental Results

I/O time is closely related to the number of I/O operations, because I/O latency is high. Let's assume that an I/O overhead is the time taken preparing a disk access: system calls, communications between the compute node and the I/O node, checking the cache in I/O nodes, and so on. Then, as the number of I/O operations increases, the I/O overhead will increase.

When the size of a square region is 16×16 and one data chunk is accessed at a time, the number of I/O operations to access the matrix in any access pattern is 65,536 ($= 16 \times 64 \times 64$) for the traditional parallel file system using FP (File Pointer), because 16 I/O operations are required to access a data chunk. But it is 4,096 ($= 64 \times 64$) for both 1-D LD (Logical Disk) and 2-D LD, because a data chunk can be accessed with a single I/O operation in both versions: a data chunk can be represented with a data segment in 1-D LD and it can be mapped into a logical disk block in 2-D LD.

As the size of a data chunk increases, the number of I/O operations for 1-D and 2-D LD decreases, because a single I/O operation is enough to access each data chunk. If a chunk size is 32×32, the number of I/O operations required to access all data chunks for both 1-D and 2-D LD will be 1,024 ($= 32 \times 32$). However, the number of I/O operations for FP does not decrease as does the number for logical disk versions, because it is impossible for the I/O operations of FP to represent a data chunk with a single I/O operation. It is affected only by the amount of contiguous data sizes accessed at a time along the columns of the matrix. Thus 32 I/O requests are necessary to access a data chunk and the total number of I/O operations will be 32,768 ($= 32 \times 32 \times 32$). Even if the size of a chunk increases as 32×64, the number of I/O operations for FP does not decrease, since 64 I/O operations are needed to access a data chunk. For both 1-D and 2-D, it will be 512

$(= 32 \times 16)$.

Figures 5.2 and 5.3 show the performances of our user-controllable I/O approach and the traditional method using file pointers. In these figures each access pattern has three bars: the first bar is for the traditional I/O method using a file pointer, the second is for a 1-D LD, and the third one is for a 2-D LD.

When a chunk size is 16×16, the read and write time for 2-D LD is much less than that for other approaches in all access patterns except access patterns SCZ and SCS (see Figures 5.2, 5.3). In these access patterns, data accesses do not conform to the layout of data on the disk for 2-D LD, because LDBs are stored in column-major order. On the other hand, in 1-D LD and FP, data accesses conform to the layout of data on the disk. However, the performance of 1-D LD is better than that of FP, because in FP only a part of the data chunk is accessed from a disk block in each I/O operation. On the other hand, in 1-D LD, a data chunk is accessed with a single I/O operation from multiple disk blocks and they can be presorted in the I/O node to optimize the head movement of the physical disk.

As the size of a data chunk increases in these access patterns, the read and write time for 2-D LD decreases while that for other approaches increases, because a logical disk block is mapped into one or several adjacent physical disk blocks in 2-D LD. Thus the read and write time for 2-D LD decreases as the size of a data chunk accessed at a time increases in all access patterns; the I/O time is always much less than that for FP and 1-D LD for any data access format and for all disk block allocation strategies.

For many access patterns, the read and write time for FP and 1-D LD decreases as the size of a data chunk increases, because the number of I/O operations needed to access the matrix decreases. However, as the size increases, the I/O time for FP and 1-D LD can be increased in some access patterns because of the limitation of

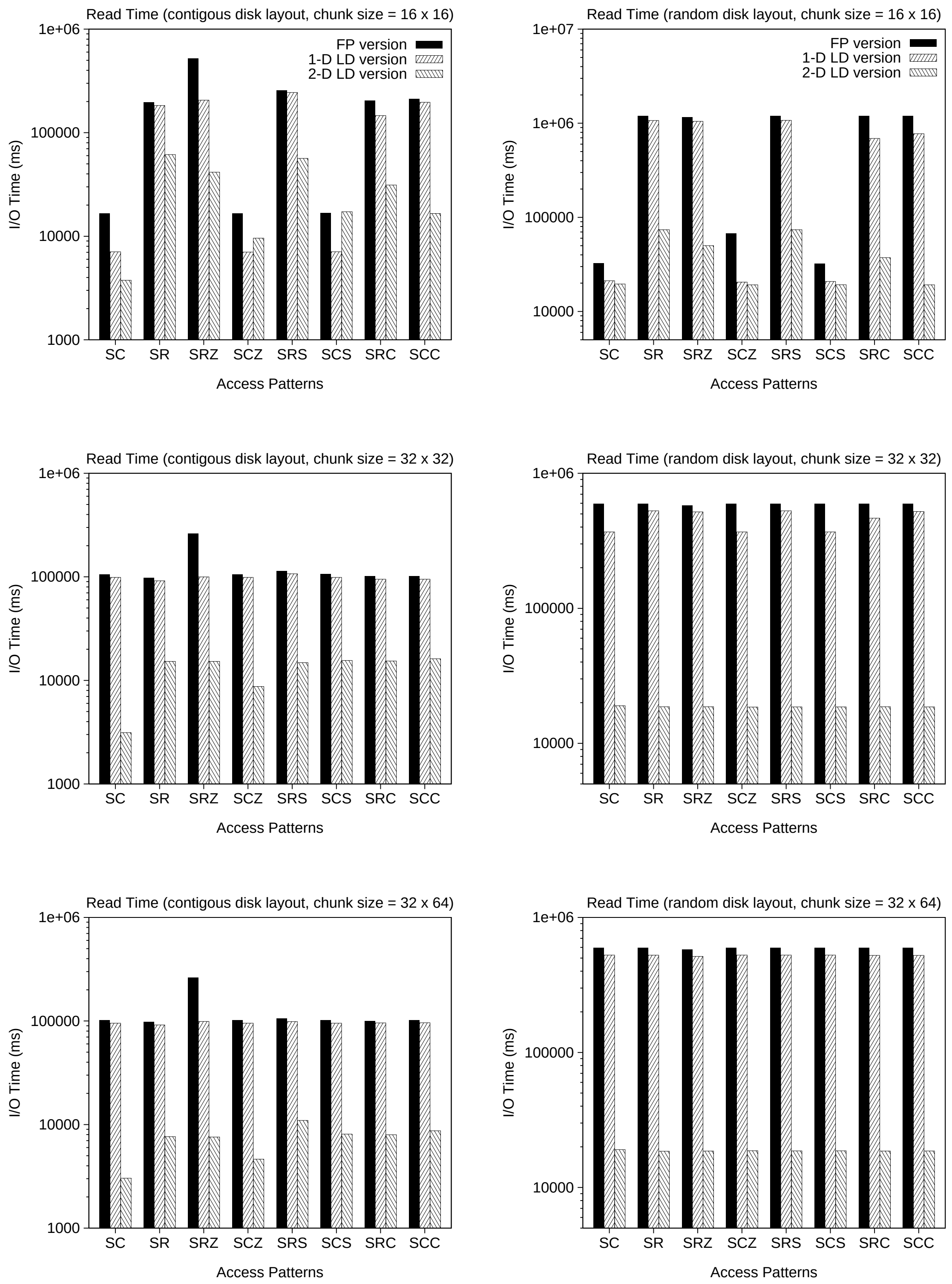


Figure 5.2: Read time

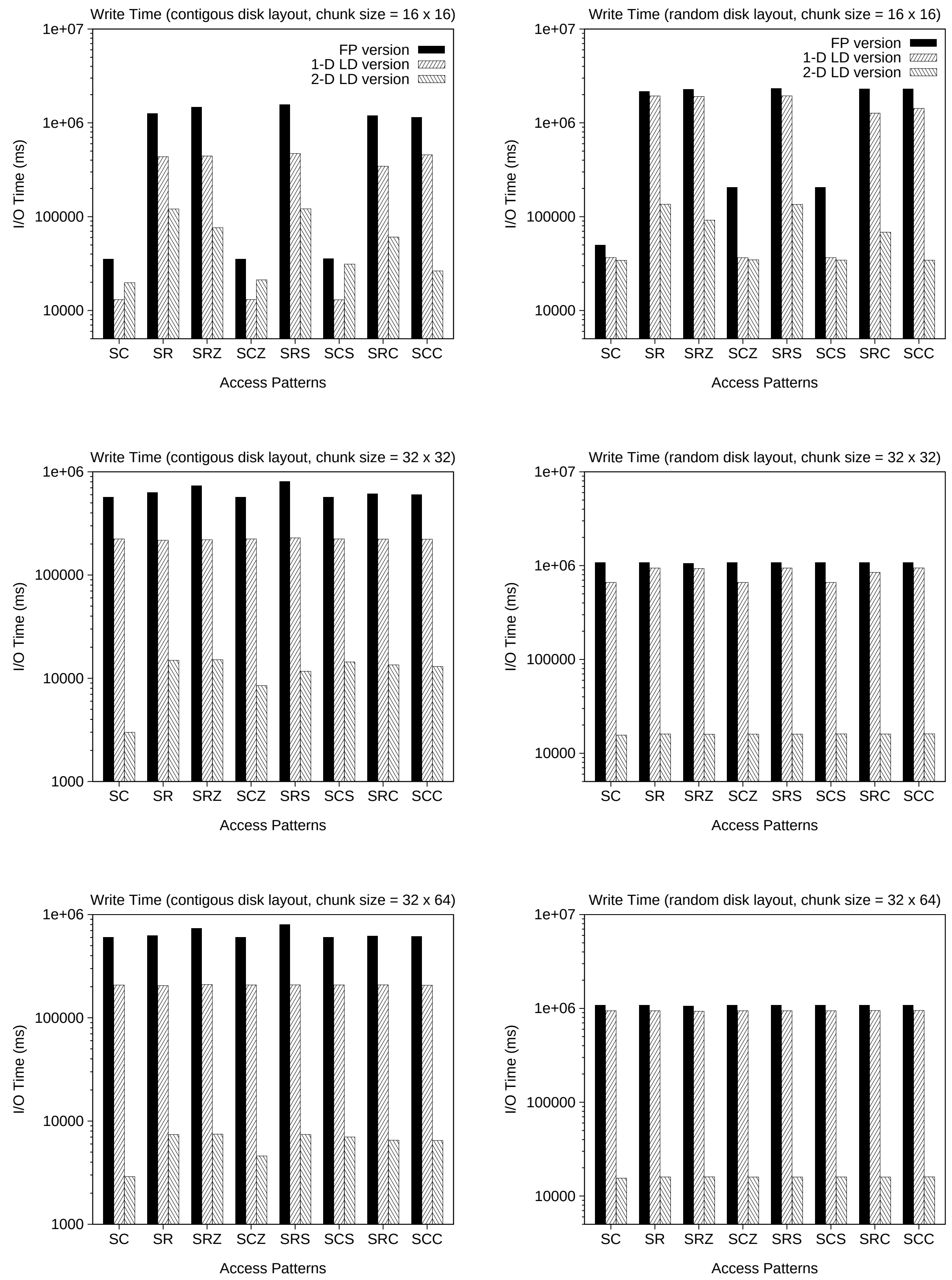


Figure 5.3: Write time

the buffer cache in the I/O node.

Let's suppose in access patterns SC, SCZ, and SCS that the size of a data chunk accessed at a time is $x \times y$. Then the I/O time will be the least if y is within the limitation of the buffer cache in the I/O node, because in the access patterns data accesses conform to the layout of data on the physical disk. In the access patterns, the I/O node can serve $\frac{1024}{x}$ I/O operations for 1-D LD and $\frac{1024 \times y}{x}$ I/O operations for FP without extra disk accesses, once it brings y disk blocks (columns) into the buffer cache from the disk. Therefore, the I/O time for FP and 1-D LD is less than that for the other access patterns when the size of a data chunk is 16×16 in all disk block allocation strategies.

However, as the size of y increases beyond the limit of the buffer cache, I/O time can increase, because the I/O node should read the same y disk blocks whenever a data chunk is accessed, even if the data chunk and the previously accessed data chunk belong to the same disk blocks. Thus, in 1-D LD and FP, the I/O time for access patterns SC, SCZ, and SCS increases as the size of the data chunk increases from 16×16 to 32×32; the I/O time for FP is not nearly as affected even if the size of the data chunk increases from 32×32 to 32×64. The I/O time for 1-D LD decreases a little, because the number of I/O operations decreases as the size of the data chunk increases under the contiguous disk block allocation.

In FP the difference between access patterns SR and SRZ is only the direction of data accesses. That is, in SR data chunks are accessed in the same order in which the matrix is stored, while in SRZ data chunks are accessed in the opposite order. However, the I/O performance of SR is better than that of SRZ, which may be caused by the characteristics of the physical disk. In 1-D LD a data chunk is accessed from multiple disk blocks in every I/O operation and the accessed disk blocks can be presorted in the I/O node to optimize the disk head movement. Thus the I/O time for the access patterns is better than that for FP; the I/O time for SR is a little better than that for SRZ. In 2-D LD the I/O time for access patterns SC and SCZ show a similar result.

In access pattern SRS, data segments are accessed in a strided fashion, and in every access to a data chunk different y disk blocks should be read for FP and 1-D LD. However, the performance of 1-D LD is better than that of FP, because only a single I/O request is enough to access a data chunk in 1-D LD, while y I/O requests are needed in FP. The performance of 2-D LD is always better than those of FP and 1-D LD, because a data chunk is stored in the same disk block. The same analysis can be applied to access patterns SRC and SCC.

In write operations the number of disk accesses increases, because disk blocks should be read for write operations when the data size to write is less than the physical disk block size. Thus the write time is larger than the read time. The performances of 1-D and 2-D LD are always better than that of FP for all access patterns, for the same reason we gave in the read operations.

Under the random block allocation strategy,[2] the I/O time for FP is almost constant in all access patterns when the size of a region accessed at one time is 32×32 or 32×64, because the limitation of the buffer cache in the I/O node and the disk's own cache do not benefit as in contiguous block allocation. However, in SC, SCZ, and SCS, the I/O time is much less than that in any other access pattern when the size of the region is 16×16.

The I/O time for 1-D LD is also almost as constant as FP, but it is less than that for FP because of collective I/O operations. The I/O time for 2-D LD is always much less than that for any other version.

[2]Each graph shown in Figures 5.2 and 5.3 represents an average of five independent experiments.

Our results also show the importance of disk block allocation: the I/O time on contiguous block allocation is always much less than that on random block allocation.

5.2.1 I/O for Best and Worst Cases

The best way to maximize I/O performance is to access data conforming to the layout of data on the disk. In our experiments, a 1024×1024 matrix is stored on the physical disk in column-major order using file pointers. In this case the I/O performance will be best if the data is accessed column by column. On the other hand, if the data is accessed row by row, the I/O performance will be the worst, because 1024 I/O operations and disk accesses are required to access a whole row. Therefore, at least 1,048,576 ($= 1024 \times 1024$) I/O operations and disk accesses are needed to access the whole matrix row by row.

The matrix is also stored on the physical disk using 1-D LD and 2-D LD in column- and block-wise manner, respectively. In all cases, the number of I/O operations needed to access the whole matrix row by row or column by column will be 1024, since a single I/O operation is enough to read a row or column of the matrix.

Table 5.1 shows the read and write time for the extreme cases. The read and write time for FP and 1-D LD is almost the same when the data is accessed column by column. However, when the data is read row by row, the performance of 1-D LD is better than that of FP, because the number of I/O operations for FP is 1,024 times more than that for 1-D LD, even though the same number of disk blocks are accessed in both cases.

To write a row of the matrix, every disk block should be read, because only a single data element is written on each disk block. In 1-D LD, write operations are performed collectively, which optimizes disk head movements as well as reducing the number of I/O operations. On the other hand, only a single data element can be written with an I/O request in FP. Thus the write time for 1-D LD is about three times less than that for FP.

Table 5.1: I/O time (contiguous allocation) – extreme cases (milliseconds)

pattern	operation	FP	1-D LD	2-D LD 16×16	2-D LD 32×32	2-D LD 32×64
column by column	read	3207.46	3130.92	5769.54	98769.45	549977.29
	write	3130.31	3062.03	11709.17	223496.00	675737.37
row by row	read	3018055.77	2924853.66	967494.32	481666.34	251822.93
	write	20165053.19	7056252.90	1961830.96	978616.57	526591.07

In all data access formats, the read and write time for 2-D LD is always larger than that for FP and 1-D LD when the data is read column by column, because a column of the matrix belongs to a column of LDBs in 2-D LD, and thus the LDBs should be read in order to access a column of the matrix.

Because of spatial locality the read time for 2-D LD is a little larger than those for FP and 1-D LD when the data is read column by column and the size of the logical disk block is 16×16. That is, by reading a column of logical disk blocks once, 16 I/O operations can be supported without extra disk accesses. However, as the size of the logical disk block increases, the I/O time increases rapidly because of the lost of spatial locality caused by the limitation of the buffer in the I/O node. (The number of physical disk blocks mapped by one column of logical disk blocks increases as the size of the logical disk block increases.)

When the data is read row by row, the read time for 2-D LD is less than that for FP and 1-D LD. In read operations the read time is about three times less if the size of the logical disk block is 16×16. As the size of the logical disk block increases the read time decreases, because the number of physical disk blocks that are mapped by one row of logical disk blocks has decreased. In write operations the write time is about ten times less than that for FP if the size of the logical disk block is 16×16. As the size of the logical disk block increases the write time also decreases.

Thus, if a two-dimensional matrix is accessed row by row and column by column repeatedly, it may be that the best method to increase I/O performance is to do I/O using 2-D LD. Even if we use 1-D LD, the performance will be much better than that with FP.

5.3 Discussion

The logical disk extends the abstraction of the conventional one-dimensional file to a multidimensional file. We believe that one of the main reasons for a large number of small granularity I/O requests under conventional parallel file systems is the abstraction of the conventional one-dimensional file. Thus, the logical disk plays a very important role in turning a large number of small granularity I/O requests into a small number of large granularity I/O requests.

Our simulations showed that user-controllable I/O operations using a logical disk increased I/O performance by reducing I/O overhead and disk access time. A logical disk allows users to access a number of arbitrary blocks at a time, which reduces the number of I/O operations and makes it possible to presort blocks to minimize the disk-head movements. Since a logical disk block can be defined in any type and size, the user can define a logical disk block by considering the access format, and store the logical disk blocks considering the data access patterns.

In our experiments we defined a 2-D LD and mapped each logical disk block into the data region accessed at one time. The I/O performance using 2-D LD was always the best even when we did not consider access patterns when storing the logical disk blocks. The I/O performance was better than that for the file system using file pointers even in the worst case. The I/O performance using 1-D LD was always the same as or better than that for the file system using file pointers, because the user can make collective I/O requests using the logical disk.

We have presented only the effects of basic user-controllable I/O operations. However, since application programmers have total control over the logical disks defined by them, they are able to control the layout of data blocks on the disk, partition data blocks to the compute nodes, and present complex access patterns with a minimum number of I/O operations. In the next chapter we present advanced user-controllable I/O operations to support the functionality that programmers can use for the purpose of flexibility and higher performance.

Chapter 6

Advanced User-Controllable Collective Parallel I/O Interface

In UPIO, files consist of n-dimensional logical disk blocks, and programmers can access data simply by accessing logical disk blocks. Logical disk blocks are mapped into physical disk blocks automatically by simple calculations. Since a logical disk is just an abstract index domain, programmers can access data by identifying the indices of logical disk blocks to be accessed.

This access method eliminates the overhead of managing the offset of files, making it easy to share data and reducing the number of I/O operations. It also increases the system's performance, since I/O latency is high and closely related to the number of I/O operations.

In UPIO a logical disk block is the unit of access an application may read or write. All or some part of a logical disk block can be accessed, and some data elements can be accessed selectively. The following sections describe the innovative features of data access functions in UPIO that are based on logical disks.

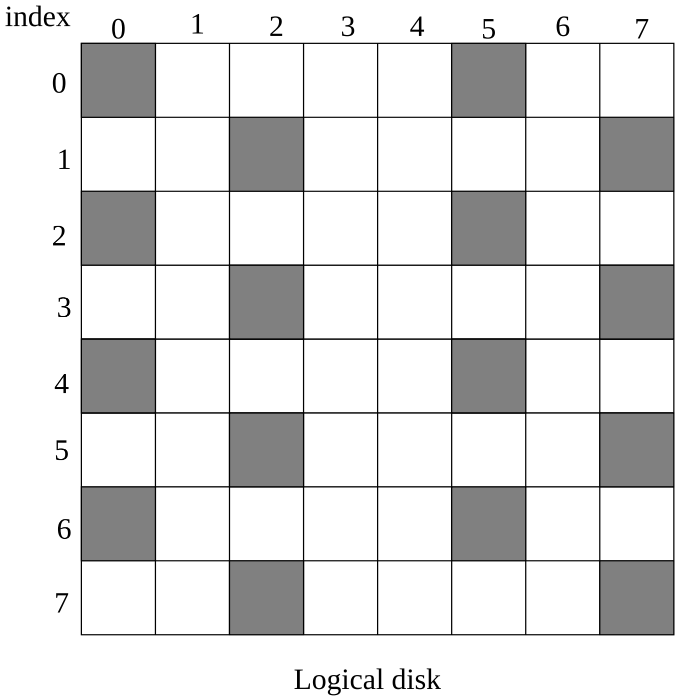


Figure 6.1: Accessing strided logical disk blocks

6.1 Accessing Strided Logical Disk Blocks

The access patterns traced by Purakayastha, Ellis, et al. [82] show that many scientific applications access files sequentially with some intervals. The access patterns may be caused by the limitation of the conventional parallel file systems or by the characteristics of computations.

In UPIO the basic structure of computations can be presented by defining a logical disk block; programmers can access logical disk blocks consecutively or randomly, both of which can support access patterns well. We can also expect logical disk blocks to be accessed in some regular pattern, as are the shaded LDBs shown in Figure 6.1. In this figure we can see that the distance between shaded LDBs is constant in columns 0, 2, 5, and 7 if the LDBs are accessed column-wise; only the index of the first shaded LDB in each column is different. Thus we can access all shaded LDBs with the four basic I/O operations described in Chapter 4, because the shaded LDBs in a column can be accessed with a single I/O operation.

For this kind of access pattern, UPIO provides more complex I/O operations. Following is the data structure involved in the operations, which we call *request vector*:

```
typedef struct {
  int cnt;            /* number of LDBs */
  COORDINATORS coord; /* index of the 1st LDB in a request vector */
  NEWS stride;        /* distance between LDBs */
} Request_Vector;
```

Here *cnt* is the number of LDBs, *coord* represents the index of the first LDB, and *stride* specifies the distance between LDBs in a request vector. The distance is represented by the difference between LDBs in each dimension.

A single instance of the data structure represents a single access pattern, thus the shaded LDBs shown in Figure 6.1 can be represented by four request vectors. The LDBs in columns 0, 2, 5, and 7 are represented by a request vector, as shown in Figure 6.2. Although representing the access pattern by using the request vectors looks cumbersome, it is a proper and necessary procedure with which to request each LDB individually.

Request vectors can also have a regular pattern and can be grouped into an LDB request group where they have the same pattern in each request group. Therefore, by specifying the request vectors in an LDB request group with relationships between groups, all LDBs being accessed can be represented. Note that request vectors 1 and 2, and 3 and 4 can be grouped into an LDB request group, because in each group the request vectors have the same pattern.

			COORDINATORS		NEWS	
Vectors	column index	cnt	coord.x	coord.y	stride.x	stride.y
1	0	4	0	0	0	1
2	2	4	2	1	0	1
3	5	4	5	0	0	1
4	7	4	7	1	0	1

Figure 6.2: Request vectors

The following statement allows applications to access LDBs that have some regular pattern:

`LD_read_stride`(*ldd, mv_cnt, memv, nog, nov, req, stride*)

This function reads the LDBs specified in the request vectors *req* and stores them in memory specified by *memv*; *nov* represents the number of request vectors, which consists of a request group. There can be *nog* request groups that have *stride* distance; the *stride* is specified based on the distance in each dimension between the first LDB of the first request vector in each request group. Figure 6.3 shows a sample code that demonstrates how the function could be used to read the shaded LDBs shown in Figure 6.1.

The counter part of `LD_read_stride` is `LD_write_stride`. There are corresponding functions for GLDM – `GLD_read_stride` and `GLD_write_stride`. With these functions, programmers can access global logical disk blocks with some regular pattern.

```
// Assume that an N * N LD is defined and
// the elementary data type is integer.
int read_logical_disk_blocks (ldd, mem)
  LDD ldd;           /* logical disk descriptor */
  int mem[ ];        /* memory address */
{
   NEWS stride;
   Request_Vector req[2];
   MV  mv;           /* memory vector: array of (mem, lcnt) */
   int nov, nog;       /* no. of request vectors and groups */

   nov = nog = 2;
   /* first request vector */
   req[0].coord.x = 0;  req[0].coord.y = 0; /* index of the 1st LDB */
   req[0].cnt = 4;                          /* no. of LDBs */
   req[0].stride.x = 0; req[0].stride.y = 2;/* distance between LDBs */
   /* second request vector */
   req[1].coord.x = 0;  req[1].coord.y = 1; /* index of the 1st LDB */
   req[1].cnt = 4;                          /* no. of LDBs */
   req[1].stride.x = 0; req[1].stride.y = 2;/* distance between LDBs */
   /* distance between request groups */
   stride.x = 5;    stride.y = 0; /* distance between request groups */
   mv.mem = mem;    mv.lcnt = req[0].cnt*req[1].cnt*nov*nog;

   LD_read_stride(ldd, 1, mv, nog, nov, req, stride);
}
```

Figure 6.3: Reading strided logical disk blocks

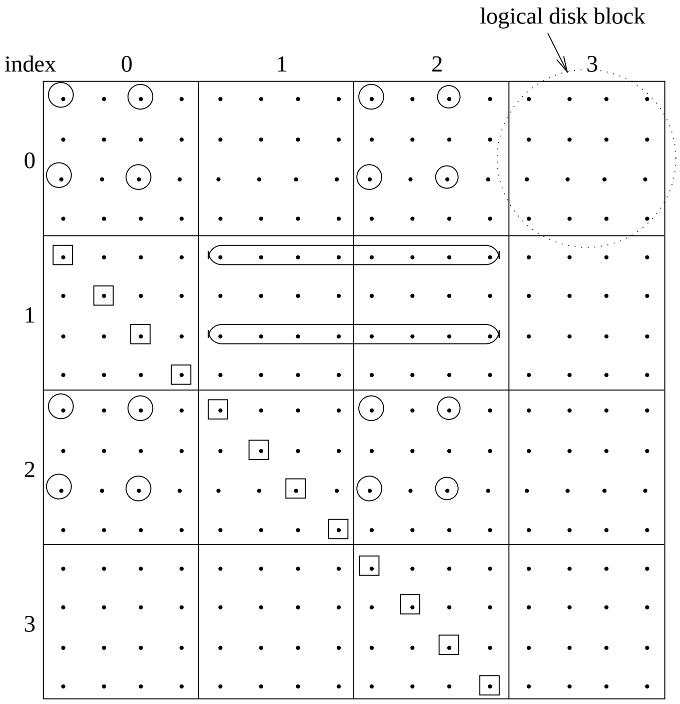


Figure 6.4: Accessing segments with some regular pattern

6.2 Accessing Strided Data Segments

In UPIO a logical disk block is the unit of access an application may read or write. The entire logical disk block or some part of it can be accessed, as well as some data elements that can be accessed selectively in some regular fashions. That is, data segments themselves can have some regular pattern. The pattern can appear in a logical disk block, across logical disk blocks, or in logical disk blocks that have some regular pattern.

Consider a 16×16 array that is mapped into a 4×4 logical disk, as shown in Figure 6.4. Suppose some data elements that are arranged diagonally (the boxed data elements in Figure 6.4) are to be read from some logical disk blocks. Conventional file system interfaces for parallel machines cannot read the data elements with a single I/O request. Programmers should read each data element individually, moving the file pointer to each element in the file systems. Twelve I/O requests are required to read the data elements in the figure using the interfaces of conventional file systems.

In UPIO, however, applications can access data segments with a regular pattern from LDBs with the following statement:

`LD_segs_read`(*ldd, mem, cnt, news, scnt, snews, seg, order, index*)

This function is similar to the function `LD_read`, because it reads data segments from LDBs as specified in `LD_read`. That is, beginning at the LDB *index*, this function reads *scnt* data segments as specified by *seg* (see the data structure shown in Figure 4.7 (e)) toward the direction *snews* (a distance in each dimension between the first data elements in each segment) and stores them in memory *mem*. Other parameters to specify the LDBs are the same as those of `LD_read`. Figure 6.5 shows a sample code for how the function could be used to read the boxed data elements shown in Figure 6.4.

Data segments that have a regular pattern can be specified across a number of logical disk blocks, as are the data elements in the rounded boxes shown in Figure 6.4. Of course, conventional file system interfaces for parallel machines cannot read the data elements with a single I/O request. Sixteen I/O requests are required to read the data elements in the figure, using the interfaces of conventional file systems, if the data is stored in column-major order. In UPIO, however, the data segments can be accessed with a single I/O operation that is specified above. Figure 6.6 shows a sample code for how the function could be used to read the data elements in the rounded boxes shown in Figure 6.4.

Data segments that have a regular pattern can be accessed from strided LDBs with the following statement:

```
// Assume that an N * N LD is defined and
// the elementary data type is integer.
int read_strided_segments_from_logical_disk_blocks (ldd, mem)
    LDD ldd;            /* logical disk descriptor */
    int mem[ ];         /* memory address */
{
    int  cnt, scnt;     /* no. of LDBs and segments */
    NEWS news, snews;
    SEGMENT seg;

    cnt = 3;    scnt = 4;
    /* distance between LDBs and segments */
    news.x = news.y = snews.x = snews.y = 1;
    /* coordinators of the 1st element in the 1st segment */
    seg.coord.x = seg.coord.y = 0;
    seg.size_x = seg.size_y = 1;     /* size of a segment */
    LD_segs_read(ldd, mem, cnt, news, scnt, snews, Y, 0, 1);
}
```

Figure 6.5: Reading strided data segments from logical disk blocks

```
// Assume that an N * N LD is defined and
// the elementary data type is integer.
int read_strided_segments_from_logical_disk_blocks (ldd, mem)
    LDD ldd;           /* logical disk descriptor */
    int mem[ ];        /* memory address */
{
    int  cnt, scnt;
    NEWS news, snews;
    SEGMENT seg;

    cnt = 2;  scnt = 2;
    news.x = 1; news.y = 0;         /* distance between LDBs */
    snews.x = 0; snews.y = 2;   /* distance between segments */
    seg.size_x = 8; seg.size_y = 1;          /* segment size */
    /* coordinators of the 1st element in the 1st segment */
    seg.coord.x = seg.coord.y = 0;
    LD_segs_read(ldd, mem, cnt, news, scnt, snews, X, 1, 1);
}
```

Figure 6.6: Reading strided data segments across logical disk blocks

`LD_segs_read_stride`(*ldd, mem, nog, nov, req, stride, scnt, snews, seg, order*)

This function is the combination of functions `LD_read_stride` and `LD_segs_read`.

Applications may access data segments with several regular patterns in an LDB, as are the data elements in rounded boxes shown in Figure 6.7. (In this figure we assumed that 16×16 data elements are defined as a logical disk block.) In UPIO, data segments with a regular pattern can be grouped together and represented by a *segment request vector* with the following data structure:

```
typedef struct {
     int cnt;        /* number of segments in a vector  */
     int order;      /* accessing order to segments     */
     SEGMENT seg;    /* specification of a data segment */
     NEWS stride;    /* distance between data segment   */
} Segment_Request_Vector;
```

where *cnt* is the number of data segments in a segment request vector, *seg* specifies the first data segment, and *stride* represents the distance between data segments in a segment request vector.

In Figure 6.7, segments 1 and 2, 3 and 4, 5 and 6, and 7 and 8 can be grouped and represented by a segment request vector, respectively, because each pair has a regular pattern. The segment request vectors can be represented as shown in Figure 6.8.

Segment request vectors can also have a regular pattern and can be grouped into a segment request group where all segment request vectors have the same pattern in each request group. Therefore, just by specifying the segment request vectors in a segment request group and relationships between groups, all data segments being accessed can be represented. Note that segment request vectors 1 and 2, and 3 and

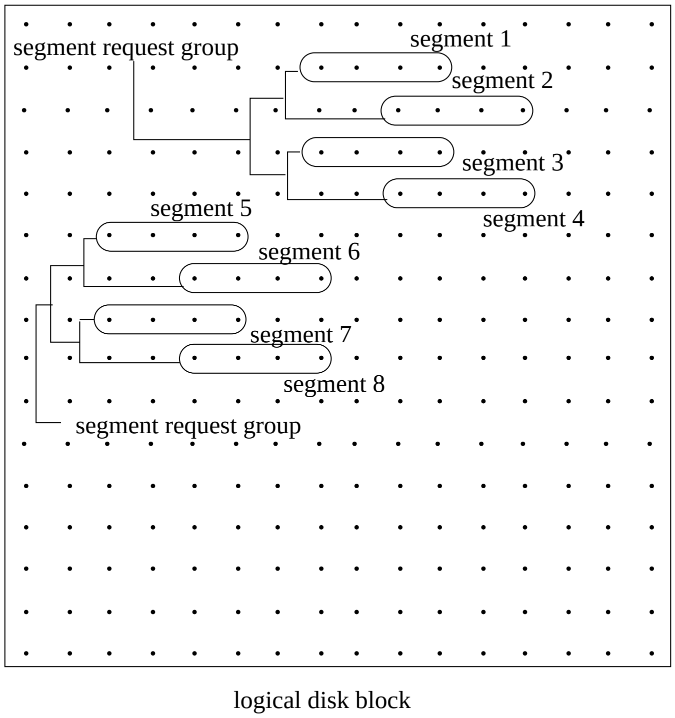


Figure 6.7: Strided segments in a logical disk block

		SEGMENT				NEWS	
Vectors	cnt	seg.x	seg.y	seg.size_x	seg.size_y	stride.x	stride.y
1	2	7	1	4	1	2	1
2	2	7	3	4	1	2	1
3	2	2	5	4	1	2	1
4	2	2	7	4	1	2	1

Figure 6.8: Segment request vectors

4 can be grouped into a segment request group, respectively, because in each group the segment request vectors have the same pattern.

The data segments shown in Figure 6.7 can be accessed with 32 I/O operations if the data is stored in column-major order in conventional file systems. Furthermore, the data segments can be accessed from some logical disk blocks, say L logical disk blocks. Therefore, $32 \times L$ I/O requests are required to access the data elements, using the interface of conventional file systems. However, in UPIO, the data elements can be represented by segment request vectors and accessed with the following statement:

`LD_strided_segs_read`(*ldd, mem, cnt, news, snog, snov, sreq, sstride, order, index*)

This function is similar to the function `LD_read`, because it reads data segments from LDBs as specified in `LD_read`. That is, beginning at the LDB *index*, this function reads data segments as specified in segment request vectors *sreq*; the data segments are read from *cnt* LDBs and the direction information of the LDBs is represented by *news*. In this function *snog* means the number of segment request groups and each request group consists of *snov* segment request vectors. The distance between request groups is specified by *sstride*, which is based on the distance in each dimension between the first data element of the first segment request vector in each request group. So the data segments shown in Figure 6.7 can be specified as follows in this function:

```
snog = snov = 2;
sreq[0].cnt = 2;
sreq[0].seg.coord.x = 7;          sreq[0].seg.coord.y = 1;
sreq[0].seg.size_x = 4;           sreq[0].seg.size_y = 1;
sreq[1].cnt = 2;
sreq[1].seg.coord.x = 7;          sreq[1].seg.coord.y = 3;
sreq[1].seg.size_x = 4;           sreq[1].seg.size_y = 1;
```

```
sstride.x = -5;                    sstride.y = 4;
```

We can also expect the data segments with several regular patterns to be accessed from logical disk blocks with some regular pattern. Suppose the circled data elements shown in Figure 6.4 are to be read; 16 I/O requests are required to read the data elements using the interfaces of conventional file systems.

However, in UPIO, the data elements can be represented by segment request vectors and accessed with the following statement:

`LD_strided_segs_read_stride`(*ldd, mem, nog, nov, req, stride, snog, snov, sreq, sstride, order*)

This function is similar to the function `LD_read_stride`, because the data segments are placed in the LDBs that have a strided pattern. It reads data segments specified in segment request vectors *sreq* from the LDBs and stores them in memory *mem*; *snov* represents the number of segment request vectors that consist of a segment request group. There can be *snog* segment request groups with an *sstride* interval. Figure 6.9 shows a sample code for how the function could be used to read the circled data elements shown in Figure 6.4.

The main functions for LLDM and GLDM used to access strided segments are summarized in Figure 6.10 and Figure 6.11, respectively. All functions shown in Figures 6.10 and 6.11 are for synchronous read/write operations; there are corresponding asynchronous read/write functions to access strided data segments. These functions allow programmers to access data segments that have any patterns with a minimum number of I/O operations.

```
// Assume that an N * N LD is defined and
// the elementary data type is integer.
int read_strided_segments_from_strided_LDBs (ldd, mem)
    LDD ldd;            /* logical disk descriptor */
    int mem[ ];         /* memory address */
{
    int nov, nog, snov, snog;
    NEWS stride, sstride;
    Request_Vector req;
    Segment_Request_Vector sreg;

    nog = snog = 1;     /* no. of LDB and seg. request groups  */
    nov = snov = 1;     /* no. of LDB and seg. request vectors */
    req.cnt = 2;            /* no. of LDBs in a request vector */
    req.coord.x = req.coord.y = 0;/* coordinators of the 1st LDB */
    req.stride.x = 2;  req.stride.y = 0; /* distance betwen LDBs */
    /* distance between LDB request groups */
    stride.x = 0;  stride.y = 2;

    sreg.cnt = 2;   /* no. of segments in an seg. request vector */
    /* coordinators of the 1st element in an seg. request vector */
    sreg.seg.coord.x = sreg.seg.coord.y = 0;
    sreg.seg.size_x = sreg.seg.size_y = 1;  /* size of a segment */
    /* distance between segments */
    sreg.stride.x = 2;  sreg.stride.y = 0;
    /* distance between seg. request groups */
    sstride.x = 0;  sstride.y = 2;
    LD_strided_segs_read_stride(ldd, mem, nog, nov, &req, stride,
        snog, snov, &sreq, sstride, X);
}
```

Figure 6.9: Reading strided segments from strided logical disk blocks

```
LD_segs_read
(ldd, mem, cnt, news, scnt, snews, seg, order, index)
  read scnt data segments specified in seg. The distance between segments is
  specified in snews and other parameters are the same as those of LD_read
LD_segs_write
(ldd, mem, cnt, news, scnt, snews, segment, order, index)
  counter part of LD_segs_read
LD_segs_read_stride
(ldd, mem, nog, nov, req, stride, scnt, snews, seg, order)
  read data segments from strided LDBs
LD_segs_write_stride
(ldd, mem, nog, nov, req, stride, scnt, snews, seg, order)
  counter part of LD_segs_read_stride
LD_strided_segs_read
(ldd, mem, cnt, news, snog, snov, sreq, sstride, order, index)
  reads data segments that have regular patterns.
LD_strided_segs_write
(ldd, mem, cnt, news, snog, snov, sreq, sstride, order, index)
  counter part of LD_strided_segs_read
LD_strided_segs_read_stride (ldd, mem, nog, nov, req,
                      lstride, snog, snov, sreq, sstride, order)
  reads data segments that have regular patterns from strided LDBs
LD_strided_segs_write_stride (ldd, mem, nog, nov, req,
                      lstride, snog, snov, sreq, sstride, order)
  counter part of LD_strided_segs_read_stride
```

Figure 6.10: Strided data segments access functions for LLDM

`GLD_segs_read`
`(ldd, cflag, mem, cnt, news, scnt, snews, seg, order, index)`
reads `scnt` data segments specified in `segment` from a GLD. The distance between segments is specified in `snews` and other parameters are the same as those of `GLD_read`

`GLD_segs_write`
`(ldd, cflag, mem, cnt, news, scnt, snews, seg, order, index)`
counter part of `GLD_segs_read`

`GLD_segs_read_stride`
`(ldd, mem, nog, nov, req, stride, scnt, snews, seg, order)`
read data segments from strided GLDBs

`GLD_segs_write_stride`
`(ldd, mem, nog, nov, req, stride, scnt, snews, seg, order)`
counter part of `GLD_segs_read_stride`

`GLD_strided_segs_read (ldd, cflag, mem, cnt, news, snog, snov, sreq, sstride, order, index)`
reads data segments that have regular patterns from a GLD.

`GLD_strided_segs_write (ldd, cflag, mem, cnt, news, snog, snov, sreq, sstride, order, index)`
counter part of `GLD_strided_segs_read`

`GLD_strided_segs_read_stride (ldd, cflag, mem, nog, nov, req, lstride, snog, snov, sreq, sstride, order)`
reads data segments that have regular patterns from strided GLDBs

`GLD_strided_segs_write_stride (ldd, cflag, mem, nog, nov, req, lstride, snog, snov, sreq, sstride, order)`
counter part of `GLD_strided_segs_read_stride`

Figure 6.11: Strided data segments access functions for GLDM

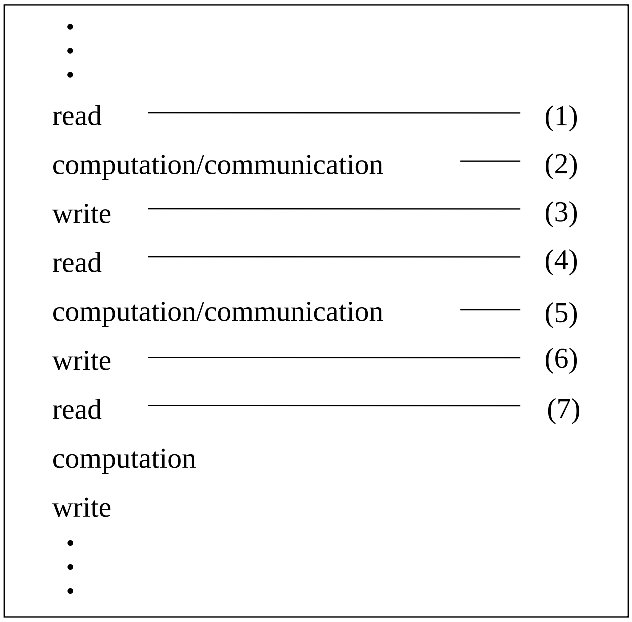


Figure 6.12: Iteration of typical iterative algorithms

6.3 Atomic Read-Write Logical Disk Blocks

The heart of a large class of scientific and engineering applications is an iterative algorithm. Each iteration in the algorithm, in general, consists of computation phases, communication phases, and I/O phases as shown in Figure 6.12. It has been shown that it is possible to improve system performance by overlapping computation and communication (e.g., Active Message) [88], or computation and I/O (e.g., prefetching) [27, 51] in each iteration. That is, phase (2) with phase (3) or (4), as well as phase (5) with phase (6) or (7), can be overlapped in Figure 6.12.

In this figure we can see that temporary values must be read for computation and the result of the computation should be written repeatedly. Thus, if the read and write operations can be done collectively with a single I/O operation, the I/O latency can be reduced and the I/O performance will be increased. In conventional file systems, however, it is impossible to do the operations collectively.

UPIO allows users to do read and write operations collectively with a single I/O request by combining the operations in various patterns as desired. We call the collective read and write operations *atomic read-write* operations.

Atomic read-write operations are similar to the POSIX `lio_listio` [43] in terms of functionality: `lio_listio` is a series of simple reads or writes for a single file. But in UPIO the atomic read-write operations are various combinations of read and write operations. In these operations it is possible to present a series of simple reads or writes for a single file, as well as for multiple files. This allows programmers to express access patterns that have no regular patterns with a single I/O operation, which will reduce the I/O latency by combining multiple I/O operations and increase I/O performance by presorting the blocks being accessed and then optimizing disk head movement.

For some computations it is sometimes necessary to access multiple data files, because the data are logically related. In this case, to obtain a good performance it is desirable to cluster the data files together on the disk. The atomic I/O operations can give information to I/O subsystems by accessing multiple files with a single I/O operation.

UPIO provides blocking and nonblocking I/O operations which are applied to the atomic I/O operations. That is, UPIO allows the combining of blocking and nonblocking operations in a single atomic read-write operation, which will overlap I/O operations and computations as well as reduce I/O latency.

Chapter 7

Evaluation of UPIO with Synthetic Access Patterns

We performed extensive experiments in order to evaluate the performance of the user-controllable collective parallel I/O interface (described in the previous chapter) under various configurations, with various kinds of workloads, and with various parameters.

7.1 Access Patterns

Figure 7.1 shows the synthetic access patterns that we used in our experiments. Most of the patterns are derived from different modes of partitioning two-dimensional matrices among compute nodes, which is relatively agreed upon in partitioning multidimensional arrays and is supported by the available programming languages such as HPF [31], Fortran D [33], and Vienna Fortran [6]. The patterns represent several ways to distribute an 8×8 matrix over four compute nodes. In the figure, each region of the matrix is labeled with the number of the compute node that accesses the region.

Data elements in each dimension of the array can be accessed fully by one compute node (denoted as STAR), accessed by compute nodes in contiguous segments (BLOCK), accessed in a round-robin fashion by compute nodes (CYCLIC), or accessed in contiguous chunks in a round-robin fashion (BLOCK_CYCLIC) by compute nodes. Patterns are named by the first letter of the partitioning pattern for each dimension from dimension x to y.

7.2 Data Format

Since we showed the effects of UPIO for a local logical disk model in Chapter 5, we have focused our experiments on exploring the effects of UPIO for a global logical disk model. For these experiments we stored a 2048×2048 matrix (the size of each element of the matrix is 4 bytes) on the disks in column-major order in single file under a conventional file system environment. The file is striped at block granularity across all the disks in a round-robin fashion, where the file system block size is 4-KB. In all our experiments data is stored contiguously and randomly on the disks.

In UPIO the matrix is stored through one- and two-dimensional global logical disks. A column of the matrix is defined as an LDB when we use a one-dimensional logical disk (1×2048 1-D LD), and LDBs are stored across all the disks in a round-robin fashion. Because this is similar to the way of storing the matrix in the conventional file system environment, we can compare their performances.

UPIO allows users to define an LDB and distribute LDBs across disks by considering a basic computational structure and access pattern. Thus each data segment that is accessed contiguously by a compute node can be defined as an LDB. In an extreme case one data element can be defined for the access patterns, such as CCX and CCY, or a whole data segment for an access pattern, BB, in

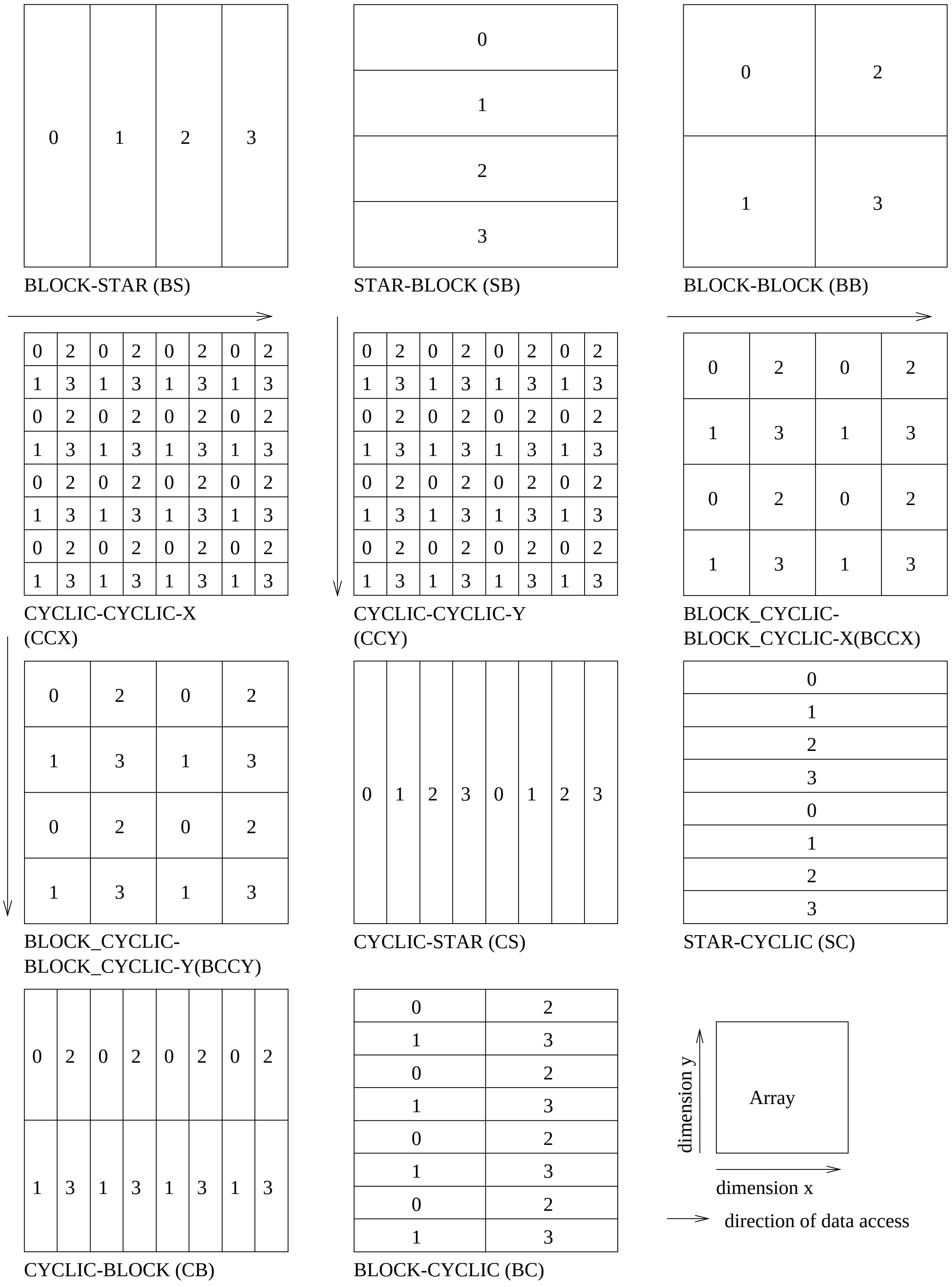


Figure 7.1: Access Patterns

Figure 7.1. We define a 32×32 submatrix as an LDB when we use a two-dimensional logical disk (64×64 2-D LD), which may be intermediate between the two extreme cases. LDBs can be distributed across disks in any mode, as described in Chapter 4, but in our experiments we distributed them across disks in a (SKEW_CYCLIC, SKEW_CYCLIC) mode.

7.3 Performance

In our experiments we configured the system with eight compute nodes (2×4 logical grid) and varied the number of I/O nodes 4, 8, and 16, where I/O nodes are logically arranged into 2×2, 2×4, and 4×4 grids, respectively. We also varied the number of compute nodes 4, 8, and 16 with four I/O nodes in order to study sensitivity.

Each node has a 25 MHz CPU and a direct-memory access (DMA) capability. The I/O node has a thirty-two 4-KB-buffer cache that uses an LRU-replacement, write-back strategy, and an HP97560 [85] disk attached to an SCSI bus whose peak bandwidth is 10 MB/s. The disk capacity and peak transfer rate are 1.3 GB and 2.34 MB/s, respectively.

For the base-configuration experiments, we assume that the system has eight compute nodes (each node has a 32-KB memory) and four I/O nodes. In this chapter we first explore the performance of UPIO and a traditional file system using file pointers (hereafter, we call it FP) on the base configuration. Then we compare the performances of UPIO and FP, varying the system configuration.

7.3.1 Base-Configuration Experiments

Table 7.1 shows the exact shapes used in our base-configuration experiments. Some access patterns (CCX and CCY, BCCX and BCCY) seem to be the same in the table, but they are different in data-access directions. In the CCX and BCCX access patterns, data chunks are accessed from dimension x to y, which is similar to accessing an array in row-major order; in the CCY and BCCY access patterns, data chunks are accessed from dimension y to x.

Table 7.1: Summary of data-access patterns for base configuration. The numbers in parentheses represent the number of chunks that can be accessed by a compute node at a time.

access pattern	partition x	partition y	chunk size (data elements)	no. of I/O requests to access a chunk		
				FP	1-D LD	2-D LD
BS	BLOCK	STAR	2048×4	1	1	1
SB	STAR	BLOCK	4×2048	2048	1	1
BB	BLOCK	BLOCK	1024×8	8	1	1
CCX	CYCLIC	CYCLIC	1	1	1 (8192)	1 (8192)
CCY	CYCLIC	CYCLIC	1	1	1 (8192)	1 (8192)
BCCX	BLOCK_CYCLIC	BLOCK_CYCLIC	32×32	32	1 (8)	1 (8)
BCCY	BLOCK_CYCLIC	BLOCK_CYCLIC	32×32	32	1 (8)	1 (8)
CS	CYCLIC	STAR	2048×1	1	1 (4)	1 (4)
SC	STAR	CYCLIC	1×2048	2048	1 (4)	1 (4)
CB	CYCLIC	BLOCK	1024×1	1	1 (8)	1 (8)
BC	BLOCK	CYCLIC	1×512	512	1 (16)	1 (16)

A data chunk is a set of data elements that are logically contiguous in the data domain, but the size of the chunk that can be accessed at one time by a compute node depends on the memory size. Due to the format of stored data in a file, the number of I/O operations required to access a data chunk can be different, because the data chunk can be stored in disk blocks contiguously or noncontiguously. For example, in the FP version, the SB access pattern requires 2048 I/O operations to access a data chunk, because the data elements in a data chunk are stored in 2048 disk blocks. In 1-D and 2-D LD, however, a single I/O operation is enough to access a data chunk for all access patterns even if the data elements in a chunk are stored noncontiguously. In addition to that, several data chunks can be accessed with a single I/O operation if the data chunks are strided in a certain fashion. (The number of data chunks to be accessed at a time is limited by memory space.) For example, 8192 data chunks can be accessed with an I/O request (`GLD_segs_read` and `GLD_strided_segs_read` for 1-D and 2-D LD, respectively) in the CCX and CCY access patterns.

Figure 7.2 shows the performance of UPIO and FP with a contiguous disk layout. For most access patterns the performance of 2-D LD is better than those of 1-D LD and FP, because the LDBs of 2-D LD are distributed across I/O nodes in SKEW_CYCLIC mode and thus the I/O load for any access pattern is distributed evenly to each I/O node. In addition, the UPIO interface allows programmers to represent the access patterns collectively with a minimum number of I/O operations, which improves performance by reducing overhead and presorting disk blocks to be accessed.

For example, in the BC access pattern the performance of 2-D LD is about 1216 times better than that of FP, because in the FP version each compute node must issue 524,288 ($= 512 \times 1024$) I/O operations to access the data elements and the same I/O node can be accessed by the compute nodes in every I/O request. However, in UPIO, a 1×512 data chunk can be accessed for the access pattern with a single I/O operation, even if the data elements in the chunk are not stored

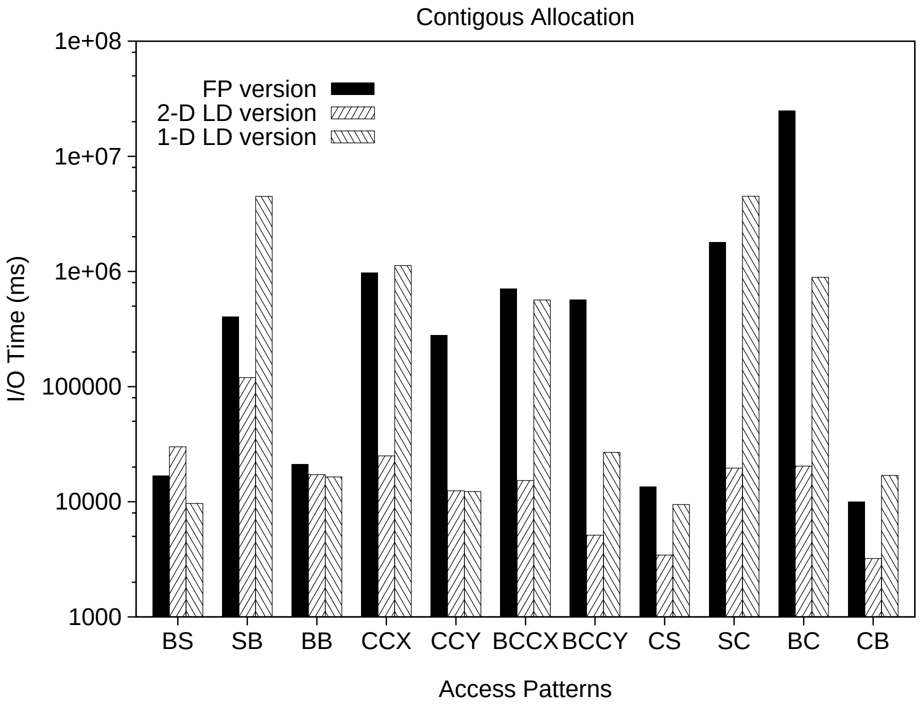


Figure 7.2: Performance for base configuration

contiguously on a disk. In addition, the data chunks to be accessed are strided, which makes it possible for each compute node of the 2-D LD version to access 16 data chunks with a single I/O operation.

The only access pattern in which the performance of FP is better than that of 2-D LD is BS. In this pattern, data is read in column-major order. In 2-D LD, each compute node accesses a different column of LDBs in every I/O operation, but it reads only four columns of data element from each LDB. In 1-D LD and FP, the data accesses conform to the layout of data on the disk, thus the performances of 1-D LD and FP are better than that of 2-D LD.

For most access patterns the performance of 1-D LD is also better than that of FP even if the data format of a file using 1-D LD on disks is similar to that of FP. The LDBs of 1-D LD are distributed in a round-robin fashion as the data blocks of FP. However, the I/O load for any access pattern is distributed evenly to each I/O node, because each compute node can collectively access multiple data chunks or a data chunk that is defined across multiple LDBs with a single I/O request.

For example, in the CCY access pattern, the performance of 1-D is better than that of FP, because in the FP version only one data element is accessed in every I/O operation and thus each compute node must issue 524,288 ($= 512 \times 1024$) I/O operations to access the data elements. In addition to that, the I/O load in each access is distributed to only half of the I/O nodes. On the other hand, in the 1-D LD version, 8192 data chunks can be accessed by a compute node in each I/O operation, and two compute nodes in the same column of a 2×4 logical grid access eight LDBs in the same I/O node; the LDBs accessed by each pair of compute nodes are in different I/O nodes. Therefore, it is possible to use all the disk parallelism, and the performance of 1-D LD is better than that of FP.

There are some cases (SB, CCX, SC, and CB) where the performance of FP is better than that of 1-D LD. In 1-D LD, for the SB access pattern, each compute node can access a 4×2048 data chunk using a single I/O request. However, the data chunk crosses whole LDBs and whole LDBs should be read from the disk in every access because of the limitation of the cache size in I/O nodes, and an I/O node serves one I/O request at a time. In the SC access pattern each compute node can access four 1×2048 data chunks at a time and each data chunk also crosses whole LDBs. Thus whole LDBs should be read from the disk in every access as in SB. On the other hand, in FP, for the SB access pattern each compute node accesses four data elements in the same column at a time. Therefore, the possibility that each compute node accesses data elements in the same disk block is very high and the performance of FP is better that of 1-D LD, even though the number of I/O operations needed to access the data elements is 2048 times more than in 1-D LD. For a similar reason, the performance of 1-D LD for the SC and CCX access patterns is worse than that of FP.

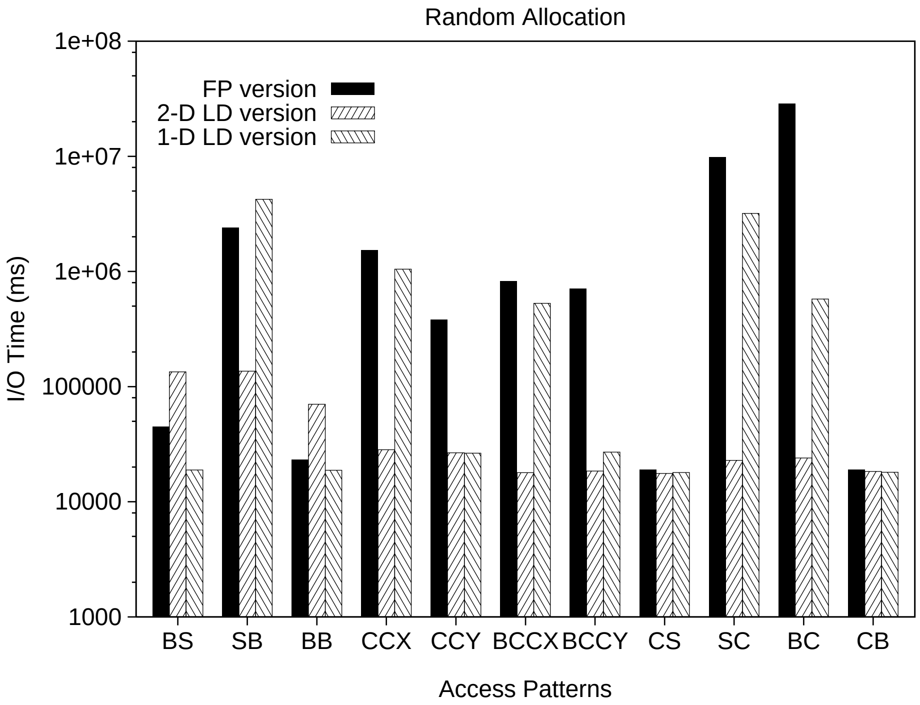


Figure 7.3: Performance for base configuration

For the CB access pattern, in 1-D LD each compute node can access eight 1024×1 data chunks at the same time. An LDB of 1-D LD is mapped into two disk blocks that are contiguous, but each data chunk being accessed by a compute node is just half of a different LDB. Each I/O node serves only one I/O request at a time and accesses eight noncontiguous disk blocks at a time. In FP, however, two contiguous disk blocks in each I/O node are accessed and thus the performance of FP is better than that of 1-D LD.

Figure 7.3 shows the performance of UPIO and FP with a random disk layout.[1] For most access patterns, the performance of 2-D LD is better than that of 1-D LD and FP, as in the contiguous disk layout. Of course, the performance of 1-D LD is better than that of FP for major access patterns, but, for BS access patterns, the performance of 2-D LD is still worse than in the other cases, as we explained in the contiguous disk layout.

[1]Each graph represents the average of five independent experiments.

For the CB access pattern, the performance of FP is better than that of 1-D LD in the contiguous disk layout, because in FP the data accesses conform to the layout of data on the disk. However, in a random disk layout the performances are almost the same, because in 1-D LD multiple disk blocks are all accessed at one time, whereas in FP only a single disk block is accessed. The performance of FP for the SC and CCX access patterns is also better than that of 1-D LD in the contiguous disk layout even though the number of I/O operations needed to access the data elements is 2,048 times more than that in 1-D LD, because in 1-D LD whole LDBs should be read in every I/O operation. However, in a random disk layout the performance of 1-D LD for the access patterns is better than that of FP, because in 1-D LD multiple disk blocks are accessed and extra disk-head movement can be reduced by presorting the disk blocks. On the other hand, in FP only a single data element can be accessed in each I/O request, and thus the I/O performance will depend on the layout of disk blocks for a file.

In this section we primarily explore the performance of UPIO under various workload parameters with a fixed set of system configuration. The only system parameter we vary is the disk layout. In the following section we explore how the changes of system parameters affect UPIO and FP performances.

The main system parameters that we change are the number of compute nodes in the system and the number of I/O nodes. Of course, the cache size of I/O nodes will affect the performance, but we do not change the parameters related to the cache system since we are not concerned with the effects of cache systems. Memory size and the number of disks in each I/O node could affect performance, but we believe that we can get similar effects by changing the number of compute nodes and the number of I/O nodes, respectively.

We use only a contiguous disk layout to explore the effects of the changes of

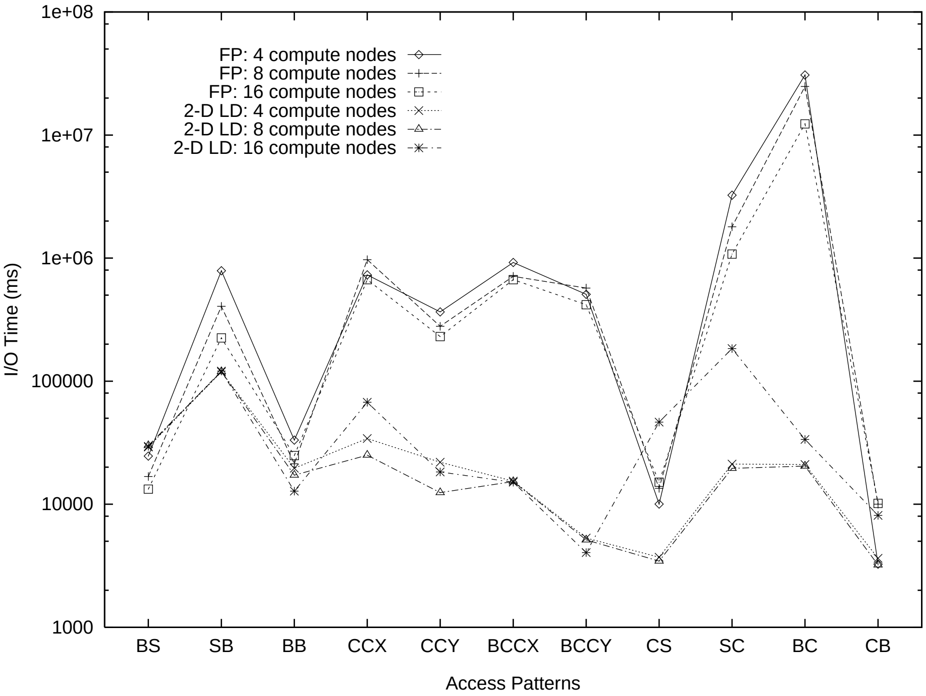


Figure 7.4: Performance comparison of 2-D LD and FP

the system parameters, because the required number of experiments is very high and with the base configuration we could see the performances of both cases.

7.3.2 Effects of Number of Compute Nodes

To explore the effects of the number of compute nodes, we varied the number of compute nodes in our system (4, 8, and 16), holding the number of fixed I/O nodes to four. In this system, the compute nodes are logically arranged into $2 \times 2, 2 \times 4$, or 4×4 grids.

Figures 7.4 and 7.5 show the results of our experiments for 1-D, 2-D LD, and FP. For most access patterns the performance of 1-D and 2-D LD is better than that of FP, as in the base-configuration. However, the performance of UPIO for some access patterns is still worse than that of FP (even though in UPIO multiple data

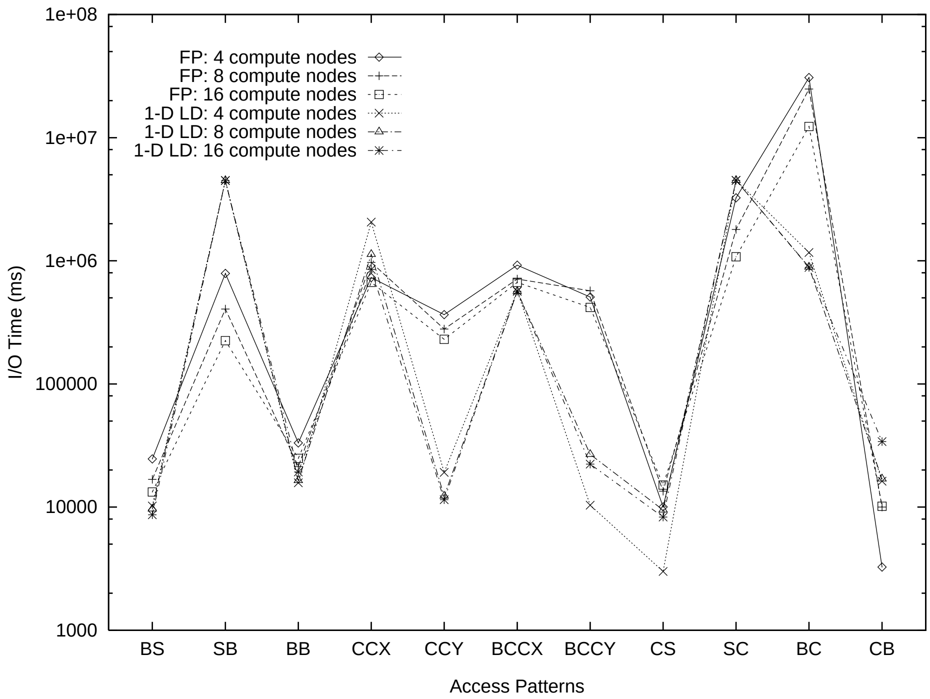


Figure 7.5: Performance comparison of 1-D LD and FP

chunks can be accessed collectively with a single I/O request), because I/O time can depend on the order of service and the size of the buffer cache in I/O nodes.

For example, the performance of 2-D LD for the BS access pattern is still worse than that of FP even if the number of compute nodes increases. In 2-D LD, for this access pattern each compute node accesses a different column of LDBs in every I/O operation, but only four columns of data elements from each LDB are read by the compute node. There is no interprocess spatial locality and the same column of LDBs should be read eight times to serve eight I/O requests[2] due to the limitations of the buffer cache. In FP each compute node accesses four columns in every I/O operation, but the columns accessed by all compute nodes are not stored contiguously. Thus the I/O time depends on the order of the I/O service in the I/O

[2]A 32×32 submatrix is defined as an LDB in 2-D LD. Each compute node fetches only four columns of data elements from each LDB in each I/O operation, and thus eight I/O requests are required to fetch the data elements in an LDB.

nodes. If the order leads to fewer disk-head movements and to better utilization of the disk's internal cache, the I/O time can decrease. The performance of 1-D LD also depends on the order of service. In our experiments the I/O time decreases as the number of compute nodes increases. For this access pattern the performance of FP is still better than that of 2-D LD and worse than that of 1-D LD, as the results show in the base-configuration experiments, for the same reason given in the previous section.

For many access patterns the I/O time of the three versions decreases as the number of compute nodes increases, because the number of I/O operations required to read the array is decreased. However, in some access patterns the I/O time does not decrease even if the number of compute nodes increases, because I/O time can depend on the order of service, the size of the buffer cache in I/O nodes, and the distribution of I/O loads. We analyze the experimental results for those access patterns in the following.

For the SB access pattern, the performance of FP becomes better as the number of compute nodes increases, because in this access pattern each compute node accesses four data elements in the same column at a time and thus the number of data elements in the same column accessed by each compute node increases as the number of compute nodes increases. In 1-D LD each compute node collectively accesses a 4×2048 data chunk that crosses whole LDBs with an I/O request. However, whole LDBs should be read from disks in every I/O request from each compute node because of the limitation of the cache size in I/O nodes, and an I/O node serves one I/O request at a time. Therefore, even if the number of compute nodes increases, the number of disk accesses required to read whole LDBs is not changed, and the performance is not affected by the number of compute nodes. In 2-D LD each compute node also accesses a 4×2048 data chunk with an I/O request as in 1-D LD; the data chunk crosses a row of LDBs. Thus each compute node should access a different row of LDBs in every I/O operation, but only four rows of data elements from each LDB are read by the compute node. There is no interprocess spatial locality and, due to the limitations of the buffer cache the same row of LDBs should be read eight times, as in the BS access pattern, to serve eight I/O requests. Therefore, the performance can be affected by the order of I/O service rather than by the number of compute nodes: the I/O time of 2-D LD is almost the same, regardless of an increased number of compute nodes.

For the access patterns BCCX and BCCY, the performance of 2-D LD is much better than that of FP, because in 2-D LD an LDB is the basic unit of I/O operation. In 2-D LD the I/O time decreases as the number of compute nodes increases, because the number of LDBs that should be accessed by each compute node decreases as the number of compute node increases. In these access patterns all I/O nodes are accessed in every I/O operation, because the I/O request from each compute node is sent to all I/O nodes. Of course, compute nodes can access the same I/O nodes; I/O performance can depend on the order of service in I/O nodes. In FP the compute nodes in the same column of the logical grid access the data elements in the same column of array. However, the performance is worse than that of 1-D LD because only two I/O nodes, at most, are accessed in each operation even if the number of compute nodes increases. FP's performance is best when we use 16 compute nodes (4×4), since four compute nodes access the data elements in the same column, and thus the number of disk accesses to the same disk block is reduced to half that in the other cases. However, I/O time can increase when the number of compute nodes increases from four (2×2 grid) to eight (2×4 grid) (see the graphs for the BCCY access pattern), because in both cases two compute nodes in the same column of the logical grid access the data element in the same column of array and all columns accessed by each compute node are strided in the same disk. Therefore, as the number of compute nodes increases along the x dimension, the performance can depend on the order of service.

In 1-D LD, for the BCCX access pattern, each compute node accesses eight 32×32 data chunks along the x dimension in every I/O operation, but each data chunk is stored in different LDBs and thus 256 ($= 8 \times 32$) LDBs should be read in each I/O operation even though the data chunks accessed by the compute nodes along the same column of a logical grid of compute nodes are in the same LDBs. Therefore, the I/O time for the BCCX pattern is almost the same even if the number of compute nodes increases. For the BCCY access pattern, each compute node accesses eight 32×32 data chunks along the y dimension in every I/O operation. However, the I/O time increases as the number of compute nodes increases. In this access pattern, 32 LDBs are accessed at a time by each compute node and if I/O nodes do not serve the I/O requests from the compute nodes in the same column of a logical grid in a series, the same LDBs should be read again and again whenever a compute node accesses the LDBs. Thus, as the number of compute nodes increases, the possibility that I/O nodes serving the I/O requests from the compute nodes in the same column in a series can be reduced and the I/O time can be increased. In the BCCY pattern the performance is worst when we use eight compute nodes (2×4), because only half of each LDB is accessed in every I/O operation. This is similar to accessing disk blocks in strided fashion, since each LDB consists of two contiguous disk blocks. However, when we use 16 compute nodes (4×4), the two disk blocks are accessed in every I/O operation, and thus the performance is better than that when we use eight compute nodes.

In the BC access pattern the performance of FP becomes better as the number of compute nodes increases. However, it is still much worse than that of 1-D and 2-D LD, because each compute node still accesses only one data element at a time. In 2-D LD, for BC, the performance is the best when we use eight compute nodes, because each I/O node can serve the requests from all compute nodes by reading 16 LDBs in a row of LDBs; but as the number of compute nodes increases to 16, 32 LDBs in two LDB rows should be read in each I/O operation because of the limitation of the cache in I/O nodes. Therefore, the performance becomes worse than that for eight compute nodes. When we use four compute nodes, 32 LDBs in a single LDB row should be read in each I/O operation, and thus the performance is better than that for 16 compute nodes. In 1-D LD, the I/O time decreases as the number of compute nodes increases, because in each I/O operation the number of data elements being accessed in the same column increases.

The SC access pattern is almost the same as the BC access pattern, but in SC all data elements in a row (2,048 data elements) are accessed by a single compute node, while in BC the data elements are accessed by four compute nodes. The performance of FP for the access pattern SC is better than that for BC even if each compute node needs the same number of I/O operations as BC, because for SC the possibility that each compute node accesses a data element in the same column in every I/O operation is higher than that for BC. On the other hand, the performance of 1-D LD for the access pattern SC is worse than that for BC even if each compute node can access the data elements with the same number of I/O operations, since in each operation all LDBs must be accessed for SC.[3]

In the SC access pattern, the I/O time of FP decreases as the number of compute nodes increases, because the number of data elements being accessed in the same column increases as the number of compute nodes increases. In 2-D LD the performance is best when we use eight compute nodes as in the BC access pattern, because by reading the LDBs in an LDB row the requests from all compute nodes can be served. As the number of compute nodes increases to 16, the LDBs in two LDB rows should be read from disks in each operation (16 times) because of the limitation of the cache in I/O nodes. Therefore, the performance becomes worse than that for eight, and even four compute nodes. When we use four compute

[3]Every I/O node serves only a single I/O request at a time, and each data element in a row is mapped into a different LDB.

nodes, the LDBs in a single LDB row should be read in each operation and the data elements in the LDBs fetched into the memory with two requests. The performance of 1-D LD is not affected by the number of compute nodes for the same reason we gave in the SB access pattern. Therefore, the performance of 1-D LD is always worse than that of FP.

The CS and CB access patterns are similar, except that in CS one column of array is accessed by each compute node, while in CB half of a column (which corresponds to a disk block) is accessed by each compute node. For the CB access pattern, the performance of FP is best when we use four compute nodes, because each compute node accesses one disk block in one different I/O node at a time, and the next disk blocks accessed in the following operations are contiguous on the disk. As the number of compute nodes increases, the I/O time in both access patterns increases, because the same I/O nodes are accessed by multiple compute nodes and the possibility of accessing disk blocks noncontiguously can increase.

The performance of 2-D LD for the CS and CB access patterns is best when we use eight compute nodes, but the performance is worst when we use 16 compute nodes. Because in every I/O operation each compute node accesses the data elements in the same LDBs (two columns of LDBs) and the limitation of the buffer cache in I/O nodes, the same disk blocks should be read again to serve the request of each compute node. In 1-D LD the stride between LDBs accessed by each compute node is the number of compute nodes. Since the LDBs are distributed across four I/O nodes in round-robin fashion, the strides in the nodes become one, two, and four when we use four, eight, and sixteen compute nodes, respectively. Thus the performance for four compute nodes is the best.

For the CCX access pattern in 2-D LD the results show similar patterns to those of the access pattern SC, because each compute node accesses LDBs as in SC.

In FP the I/O time decreases as the number of compute nodes increases (from four compute nodes to 16 compute nodes), because the number of disk blocks accessed at a time and the number of data elements being accessed in the same disk block increase. But when the number of compute nodes is eight, the performance can be worse, because the number of data elements being accessed in the same disk block is the same when the number of compute nodes is four, while the number of disk blocks accessed in the same I/O node increases. In 1-D LD the time also decreases as the number of compute nodes increases, as with FP, but the performance of the CCX access pattern is worse than that of FP.

For the CCY access pattern, in 2-D LD, each compute node accesses LDBs in the same pattern as in CCX; the number of disk accesses will be the largest when we use 16 compute nodes. But when we use four or eight compute nodes, the overhead for sending the requested data elements to compute nodes can affect the performance, because in this access pattern the accesses to the LDBs conform to the layout of data on the disks and thus disk access time can be less than the overhead. Thus the graphs can have the patterns as shown in Figure 7.4. In FP, the I/O time decreases as the number of compute nodes increases, but the performance is worse than that of 2-D and 1-D LD, because 524,288 operations are required to access the data elements, as in the base configuration, and the load in each access is distributed to only half of the I/O nodes.

7.3.3 Effects of Number of I/O Nodes

We varied the number of I/O nodes in systems 4, 8, and 16, where I/O nodes are logically arranged into $2 \times 2, 2 \times 4$, and 4×4 grids, respectively. In all such experiments the number of compute nodes is constant at 8 (2×4 logical grid). This enabled us to explore the effects due to changes in the number of I/O nodes: (case

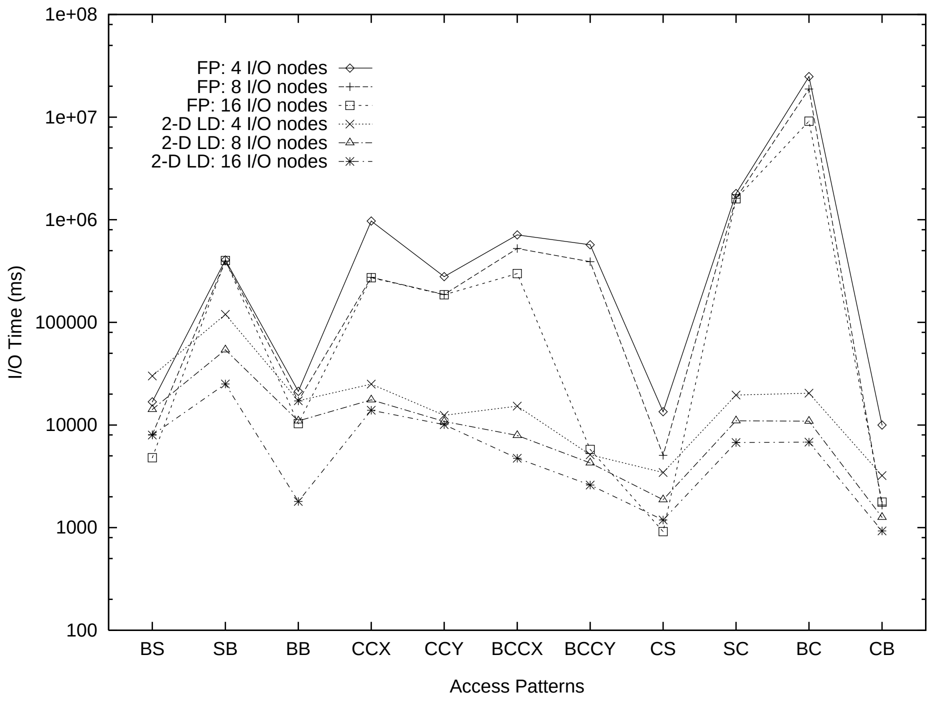


Figure 7.6: Performance comparison of 2-D LD and FP in various number of I/O nodes

1) the number of I/O nodes is larger than the number of compute nodes, (case 2) the number of I/O nodes is less than the number of compute nodes, and (case 3) the number of I/O nodes and the number of compute nodes are the same.

Figures 7.6 and 7.7 show the results of our experiments for 1-D, 2-D LD, and FP. For all access patterns, the performance of 1-D and 2-D LD becomes better as the number of I/O nodes increases, because in 1-D and 2-D LD each compute node can access multiple (logically or physically) noncontiguous data chunks at a time and I/O loads are distributed evenly to all I/O nodes. For many access patterns, the performance of FP becomes better as the number of I/O nodes increases; however, for some access patterns the number of I/O nodes does not affect the performance, because requests are not directed to all I/O nodes in each operation.

For the BCCX and BCCY access patterns, the I/O time of FP decreases as the

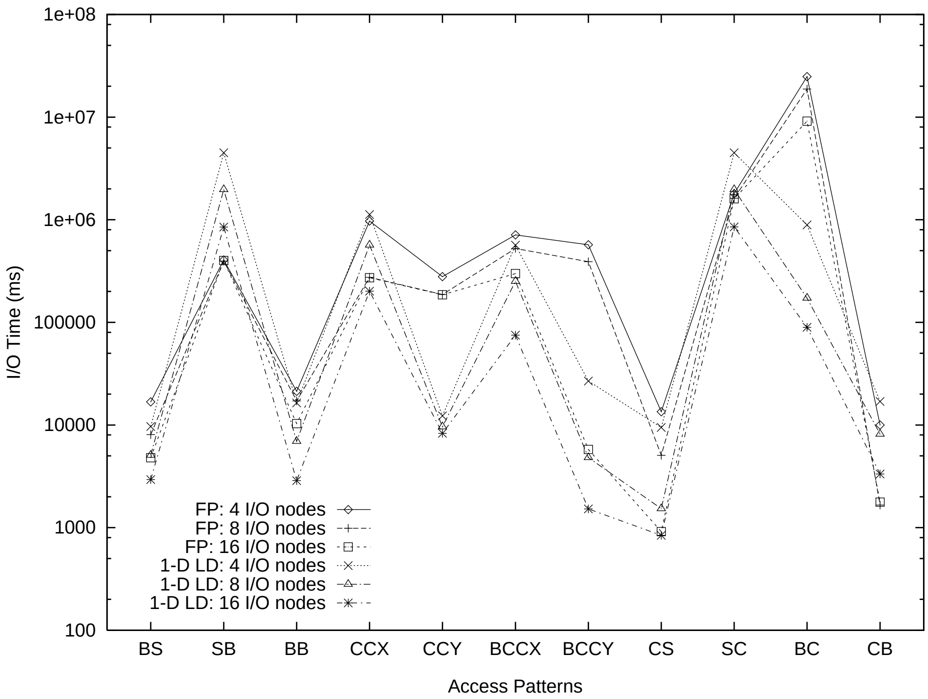


Figure 7.7: Performance comparison of 1-D LD and FP in various number of I/O nodes

number of I/O nodes increases, because more I/O nodes can serve requests at the same time. In BCCY, the I/O time of FP decreases rapidly when we use 16 I/O nodes, because 32 operations can be served by reading a disk block. However, only half of the I/O nodes serve requests in the access patterns, so that the performance of FP is worse than that of 1-D and 2-D LD. The SC and BC access patterns can be explained in the same way, because in the access patterns only half of the I/O nodes serve requests. The performance of 2-D LD for the access patterns is always better than that of FP; the performance of 1-D LD for the SC access pattern becomes better than that of FP as the number of I/O nodes increases, because every request from each compute node is served evenly by all I/O nodes.

For the CCX and CCY access patterns, the I/O time of FP decreases as the number of I/O nodes increases from four to eight, because the requests are directed to more I/O nodes (from two to four nodes). However, the I/O time does not decrease any more even if the number of I/O nodes continues to increase because in each operation all requests from each compute node can be directed to at most four I/O nodes. The performance of 2-D LD is always better than that of FP, as in the base configuration. The performance of 1-D LD for the CCY access pattern is always better than that of FP. For the CCX access pattern the performance becomes better as the number of I/O nodes increases; however, it is still worse than than that of FP, as in the base configuration, except when we use 16 I/O nodes. The graphs for the CB access pattern can be explained in the same way, but in this pattern each compute node accesses a different I/O node in each operation.

The BB access pattern in FP is similar to CB, but in this pattern compute nodes can access the same I/O nodes and the I/O time decreases as the number of I/O nodes increases. When we use the same number of I/O nodes the performance of 1-D and 2-D LD is always better than that of FP, because in 1-D and 2-D LD each compute node can access multiple data chunks at a time through all I/O nodes, while in FP it can access only a data chunk and the requests from all compute nodes are directed to at most eight I/O nodes.

In 1-D LD for the access pattern SB the I/O time decreases as the number of I/O nodes increase. However, the performance is still worse than that of FP, because in 1-D LD all LDBs should be read again in every operation. In FP all disk blocks in each I/O node are accessed contiguously. However, the performance is not affected by the number of I/O nodes, because some compute nodes can access the same I/O node and at most eight I/O nodes can be accessed by all compute nodes at a time.

For the CS access pattern the performance of FP, 1-D, and 2-D LD becomes better as the number of I/O nodes increases, because in this access pattern requests from all compute nodes are directed to all I/O nodes in every operation. The performance of 1-D is always better than that of FP when the same number of I/O nodes is used, because it is possible for 1-D LD to access multiple data chunks with a single operation. For the same reason, the performance of 2-D LD is better than that of FP when we use four or eight I/O nodes. But when we use 16 I/O nodes the performance of FP is better than that of 2-D LD. One column of the array is accessed from multiple LDBs (64 disk blocks) in 2-D LD, while it is accessed from only two disk blocks in FP, even though data accesses of both cases totally conform to the layout of data on the disk. The graphs of FP and 1-D LD for the BS access pattern can be explained in the same way. In this pattern the performance of 2-D LD becomes better as the number of I/O nodes increases, because all requests are served by all I/O nodes in every operation. However, the performance is a little worse than that of FP or 1-D LD when we use the same number of I/O nodes as in the base-configuration experiments.

7.4 Discussion

Our simulations show that user-controllable I/O operations using a global logical disk increase I/O performance. In most access patterns the performance of 1-D and 2-D LD is better than that of FP, because it is possible in UPIO to represent various access patterns collectively with a minimum number of I/O requests.[4] The operations allow compute nodes to access multiple noncontiguous data chunks with a single I/O operation, and the request is distributed to all I/O nodes evenly.

In UPIO, users can distribute data across I/O nodes by considering access patterns. In 2-D LD we distribute LDBs in a (SKEW_CYCLIC, SKEW_CYCLIC) mode. In this distribution I/O requests are always served by all I/O nodes. In FP,

[4] We analyzed only the results of read operations, but we can obtain similar results in write operations.

data is distributed across disks in a simple round-robin fashion. This forces compute nodes to access only a certain number of I/O nodes in some access patterns, because those access patterns do not match with the data distribution pattern. In the access patterns the I/O performance is not affected even if the number of compute or I/O nodes increases. In 1-D LD we distribute data in a way similar to that used in FP, where a column of array is mapped into an LDB. However, since it is possible in UPIO to represent various access patterns collectively with a minimum number of I/O requests, I/O requests are always distributed to all I/O nodes evenly, and I/O time decreases as the number of I/O nodes increases.

Our results show that we can reduce I/O time by increasing the number of compute nodes, because the number of LDBs or disk blocks that can be accessed at a time increases. But it is not always true, because multiple compute nodes can access the data segments in the same LDBs or disk blocks and the number of LDBs or disk blocks accessed at a time can be beyond the size of buffer cache. In this case, the same LDBs should be read in every I/O operation even though the compute nodes access the data segments in the same LDBs. This shows that increasing I/O performance may not be enough to provide a collective I/O interface. The I/O operations that access the data segments in the same LDBs should be collected, analyzed, and changed into another collective I/O request at runtime.

Our results emphasize the importance of data distribution, collective I/O interface, and system support for collective I/O. We have explained the effects of data distribution and collective I/O interface of UPIO up to this point. In the next chapter we describe how collective I/Os are supported in UPIO at runtime.

Chapter 8

User-Controllable Collective I/O

Since most traditional file systems for parallel computers [26, 2, 20, 36, 78] have been developed based on a linear file model and decluster file data across disks, I/O requests from each compute node may be broken into many small I/O requests to different I/O nodes.

To make a single large request, some systems provide an explicit collective-I/O interface [54, 4, 10, 39, 37] that allows programmers to present noncontiguous data in a single I/O request. Current techniques for implementing collective I/O at a system level can be largely classified into two categories: compute-node-oriented I/O (CN-oriented I/O) and I/O-node-oriented I/O (ION-oriented I/O).

In the CN-oriented I/O, compute nodes direct I/O. That is, compute nodes cooperate to make a single I/O request to each I/O node and permute the data among compute node memories before writing or after reading. I/O nodes act like a floating-point coprocessor according to the commands from compute nodes. The two-phase I/O proposed by del Rosario, Bordawekar, and Choudhary [4] is a representative method in this category. In this method a permutation is chosen that will cause I/O requests to conform to the layout of the file. This method improves

I/O performance for some access patterns, but it requires extra memory and time for permuting at compute nodes.

In the ION-oriented I/O, the compute nodes make a single I/O request to each I/O node and one or some of the compute nodes send the I/O request to I/O nodes as in the CN-oriented I/O. However, in this method, data transfers between compute nodes and I/O nodes are directed by I/O nodes. The disk-directed I/O (DDIO) proposed by Kotz [54] is a representative technique for ION-oriented I/O. This technique improves I/O performance without costing the extra time and memory for permutations that are required by two-phase I/O. However, DDIO can have low message-passing bandwidth utilization because of a large number of small-sized data chunk transfers.

UPIO allows programmers to define a file structure by considering the characteristics of the computations or of the application access pattern, distribute file data across disks in order to exploit the aggregate bandwidth of a large number of disks, and access noncontiguous data chunks with a minimum number of I/O operations. This functionality will overcome the limitation of conventional Unix-like file systems and prevent the loss of semantic information from a large, contiguous, parallel file transfer.

However, in GLDM we can expect that some data elements in some logical disk blocks may be accessed by multiple compute nodes in some phases of computations or in some applications, as seen in the access patterns described in Chapter 7. In this case, concurrent I/O requests for the same logical disk blocks can be made by the compute nodes even if each compute node makes a single, large request. Each I/O request can be served independently, but if the request from each compute node accesses the data in the same logical disk blocks, I/O performance would be optimized by serving all requests collectively. UPIO allows programmers to specify how the system serves I/O requests, which is done by setting a parameter, *cflag*, selectively according to access patterns in all data access functions for GLDM.

8.1 Combining I/O Requests

If the parameter *cflag* is set in any data access function for GLDM, each compute node translates the requested logical disk blocks into the indices of physical disk blocks and calculates which data segments are requested for each physical disk block.

Each compute node maintains a hash table that contains all indices of the physical disk blocks and the information concerning the data segments requested for each disk block, as the I/O request handler in I/O nodes does. The hash table also contains the identity of a thread that requests I/O to wake it after completion of I/O. The table is used later to extract or write data segments out of or to the disk-block data that is sent by I/O nodes.

The indices of the requested physical disk blocks are grouped according to destination I/O nodes. In every group a bitmap, of which a bit is for each compute node running a program, is appended to each index. Then all compute nodes running a program cooperate to combine the series of the index of the physical disk block and the bitmap pair, as shown in Figure 8.1. That is, the final combined request contains a series of the index of the physical disk block and the bitmap pair instead of the information for the logical disk block, as in other I/O operations. Finally, one of the compute nodes running a program multicasts the combined I/O request to the corresponding I/O nodes.

```
procedure COMBINING_IO_REQUESTS
// P: number of compute nodes running a program
// Pc: number of candidate compute nodes for communication
// Pa: number of active compute nodes in communication
// mynode: node number (0, 1, 2, ..., P − 1)
begin
  r = ⌈log P⌉;     /* number of iterations */
  p = mynode;
  Pc = p;
  for  (i = 0; i < r; i++)
      rem = Pc % 2;
      Pa = Pc - rem;
      if  (p % 2)
          send a combined request to node 2^i × (p − 1);
          wait for the completion of I/O
      else  if  (p % 2 == 0  and  p < Pa)
          receive a combined request from node 2^i × (p + 1);
          combine the request with its own request;
      p = p/2;
      Pc = Pc/2+ rem;
  multicast a combined I/O request to I/O nodes;
  wait for the completion of I/O
end
```

Figure 8.1: Combining I/O requests

8.2 Data Transfer Between I/O and Compute Nodes

In collective I/O operations of UPIO a disk block is the unit of data transfer between an I/O node and a compute node. In read operations, I/O nodes send a disk block to a compute node with a bitmap, rather than sending data segments one after another to each compute node as in DDIO. The receiving compute node extracts data segments from the disk block according to the data-segment information in the hash table that is constructed in each compute node. It searches the bitmap to find a compute node that requests the disk block and forwards the disk block with the bitmap, after modifying it, to the next compute node if there is one.

In write operations, I/O nodes also send a disk block with a bitmap to a compute node, as in read operations, but each compute node that requested the disk block overwrites the disk block with the data segments in the memory and forwards it to the next compute node, as in read operations. Finally, the disk block is sent back to the original I/O node if there is no compute node left that fills the disk block. If a disk block requested by the compute nodes has never requested, the disk block is not included in the message to be sent by an I/O node to a compute node. The receiving compute node allocates memory space for the disk block and fills data segments.

8.2.1 Read Operations

For each requested disk block, the I/O request handler in each I/O node searches the hash table called the *table of mapped blocks*. If the block is found in the table, it creates a thread called a *permuter* to serve the request for the disk block. Otherwise, it reserves a table entry and passes the index of the block to the cache manager.

The cache manager searches the buffer cache in the I/O node. If the block is found, it passes the address of that buffer for the block to the I/O request handler. Otherwise, it reserves a slot of the buffer cache and passes that buffer's pointer to the I/O request handler. The I/O request handler adds the address passed by the cache manager to the table of mapped blocks and creates a permuter. If the disk block is not in the buffer cache, the I/O request handler passes the entry of the table of mapped blocks to the disk block manager instead of creating a permuter. The disk block manager then places the information for the disk block to a sorted list of disk blocks to be read. The disk block manager creates a permuter after reading a disk block.

The permuter created by the I/O request handler or disk block manager appends the bitmap for a disk block and sends it to a compute node. Once a compute node receives a disk block and bitmap sent by an I/O node, it searches the hash table of requested disk blocks and extracts data from the disk block according to the information concerning the data segments of the requested disk block. It searches the bitmap and decides upon the next compute node to forward the disk block and bitmap. The next node then does the same thing after receiving the disk block including the bitmap, and so on. If there is no compute node left to forward the disk block and bitmap, the read operation for the disk block is complete.

While compute nodes are extracting data segments and forwarding disk blocks, I/O nodes can read other disk blocks and send them to the compute nodes one after another. Therefore, extracting data segments, forwarding disk blocks among compute nodes, and reading disk blocks in I/O nodes can be done in parallel.

Figure 8.2 shows how a disk block is transfered from an I/O node and among compute nodes that need the disk block. In the figure we assume that four compute nodes read the data segments in the same disk block and that the darkened areas in

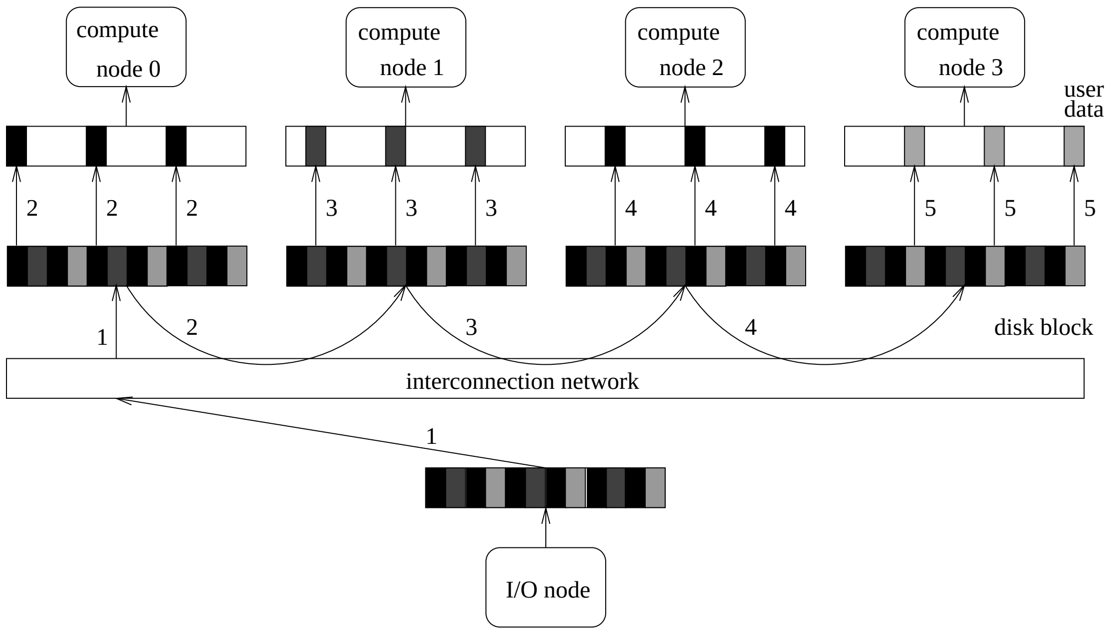


Figure 8.2: Procedure of sending a disk block for read operations in UPIO

the memories represent the data segments to be read by each compute node. When compute node 0 gets the disk block from the I/O node, it forwards the block to compute node 1 while extracting the three data segments. Compute nodes 1 and 2 do the same thing as compute node 0 does when it receives a disk block. Compute node 3 extracts only the data segments from the disk block when it receives it without forwarding the block, because no more compute nodes need the block.

Each compute node accumulates the number of bytes that is read or written, and if it is the same as the requested data size, I/O is complete. There are no interactions between a compute node and an I/O node or among the compute nodes to notify an I/O completion. Therefore, only five communications are needed to read the data segments in a disk block.

If the DDIO technique is applied to read the data segments, the I/O node must send each data segment separately to each compute node, because they are not stored contiguously on the disk. In other words, even DDIO generates a large

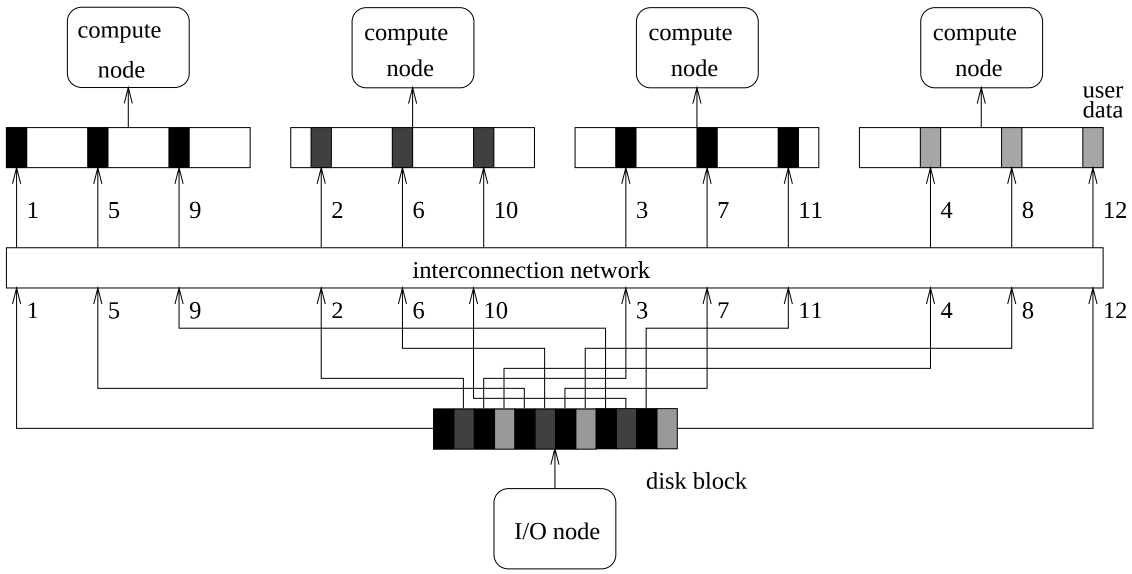


Figure 8.3: Procedure of sending data segments for read operations in DDIO

granularity I/O request to optimize I/O performance, but in some cases it cannot send data segments collectively; thus DDIO can have low message passing bandwidth utilization because of a large number of small-sized data chunk transfers.

Figure 8.3 shows how data segments are transfered from an I/O node to compute nodes. In this figure, note that 12 communications are needed to put the data segments in a disk block to each compute node. DDIO does not require extra space for a disk block in compute nodes and time to extract data segments as UPIO does. However, compared to data transfer time, the communication or DMA setup time is high, even with hardware support.

8.2.2 Write Operations

In write operations the I/O request handler, permuters, and the disk block manager in the I/O nodes do the same things as in read operations. But if a disk block requested by compute nodes has never requested, I/O nodes sends only a bitmap to a compute node.

Once a compute node receives a disk block and bitmap from an I/O node, it searches the hash table of requested disk blocks, extracts data from the memory according to the information for the data segments of the requested disk block, and then overwrites them on the disk block. It searches the bitmap, decides upon the next compute node to ensure that writes can be performed in a certain order, and forwards the disk block and bitmap to the compute node. If a compute node receives only a write request for a bitmap, it allocates a buffer for a disk block, writes the data segments on the buffer, and then forwards the buffer to the next compute node.

The next node then extracts data from the memory according to the information for the data segments of the requested disk block, and then overwrites them on the disk block after receiving the disk block with the bitmap, and so on. If there is no compute node left to write data segments on the disk block, the compute node finally sends back the disk block to the original I/O node and the block is written on the buffer reserved by the I/O node, as in normal write operations. Thus no permutation is required in I/O nodes.

Figure 8.4 shows how a disk block is transferred from an I/O node to a compute node and among compute nodes that need the disk block to write data segments. In this figure, we assume that four compute nodes write the data segments on the same disk block in rank order and that the darkened areas in the memories represents the data segments to be written by each compute node. When compute node 0 gets the disk block from the I/O node, it writes the three data segments on the block and then forwards it to the next compute node. Compute nodes 1 and 2 do the same thing as compute node 0 when it receives the block. Compute node 3 writes data segments on the block when the block arrives and then sends it back to the I/O node, because compute node 3 is the last node to write data segments on the block.

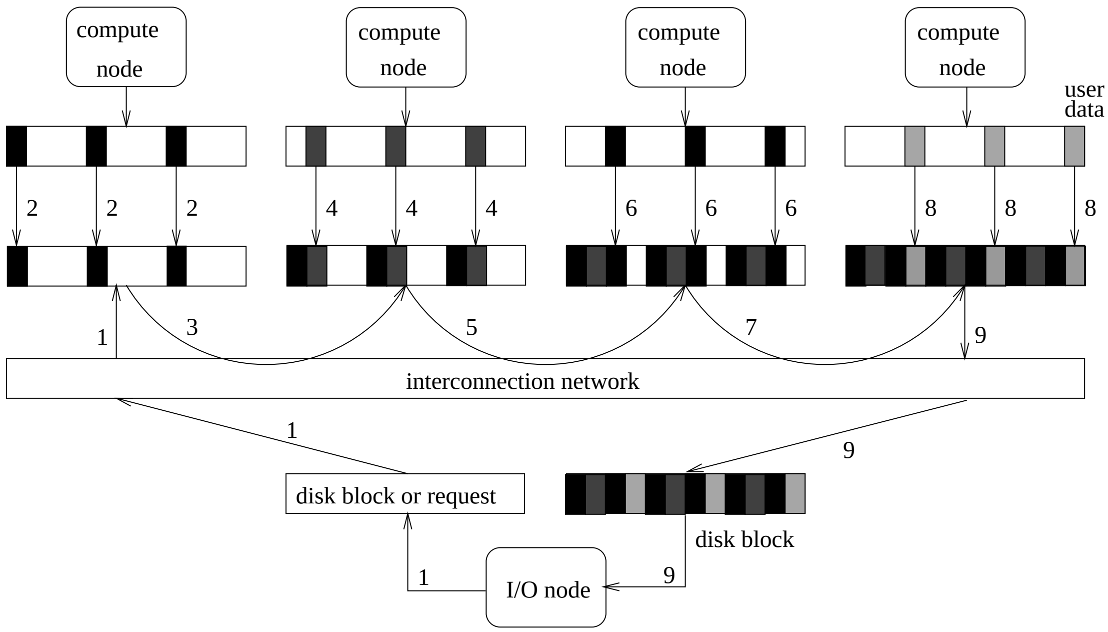


Figure 8.4: Procedure of filling data segments for write operations in UPIO

In write operations, the writing data segments and the forwarding disk block are not overlapped. However, only six communications are needed to write the data segments on a disk block. If the DDIO technique is used to write the data segments, the I/O node must pull each data segment separately from each compute node, because they are not stored contiguously on the disk. Figure 8.5 shows how the data segments are written on a disk block from each compute node. In this figure, note that 12 communications are needed to pull the data segments in a disk block from each compute node.[1]

[1]Actually, the number of communications is 24, because an I/O node sends a message to a compute node in order to request and receive a data segment from the compute node.

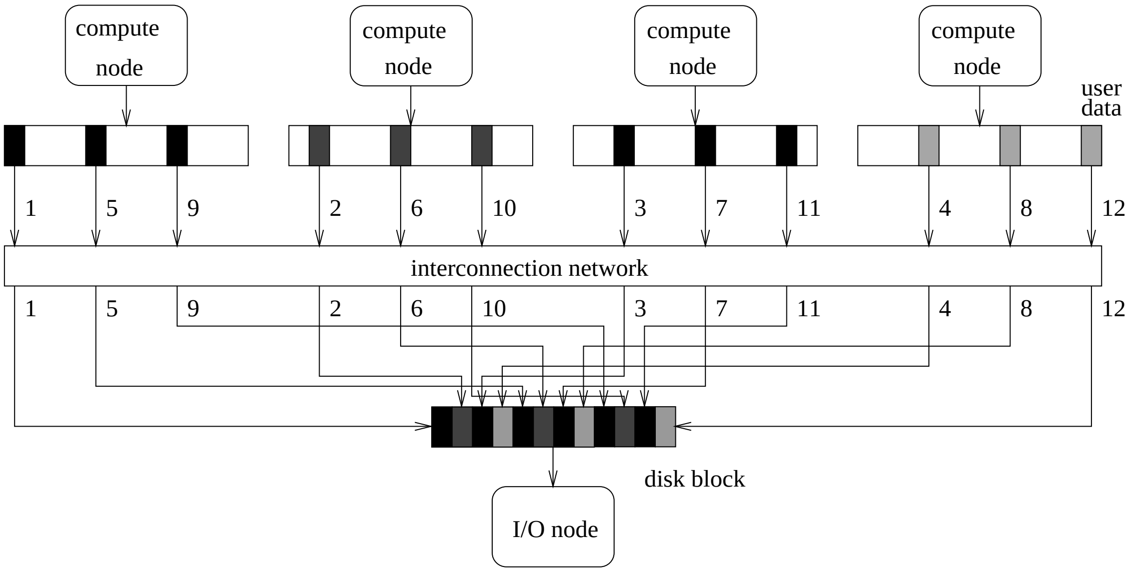


Figure 8.5: Procedure of pulling data segments for write operations in DDIO

8.3 Performance

From the experimental results shown in the previous chapter, we noticed that we can reduce I/O time by increasing the number of compute nodes, because the number of LDBs or disk blocks that can be accessed at a time increases. However, it was not always true, because multiple compute nodes can access the data segments in the same LDBs or disk blocks, and the number of LDBs or disk blocks accessed at a time can be beyond the size of the buffer cache. In this case, the same LDBs should be read in every I/O operation, even though the compute nodes access the data segments in the same LDBs. Among the access patterns described in the previous chapter, SB, CCX, CS, SC, BC, and CB are the representative examples.

We applied the collective I/O technique of UPIO to the access patterns in order to explore the effects of the technique. For our experiments we configured the system with 16 compute nodes (4×4 grid) and four I/O nodes under the same experimental environment described in the previous chapter. In our experiments

I/O requests from each compute node for the access patterns are served separately and collectively, and Figure 8.6 shows the results of our experiments.

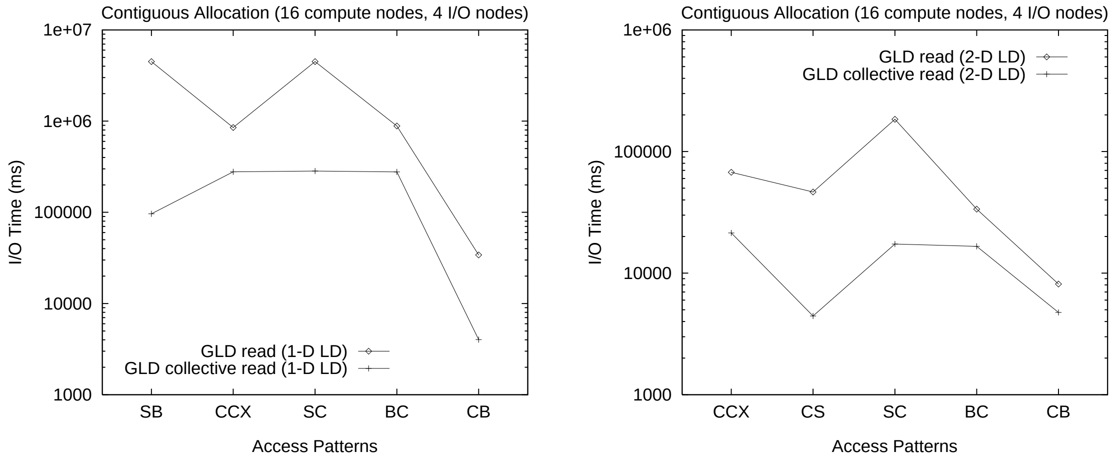


Figure 8.6: Performance of user-controllable collective I/O

In all access patterns the performance of collective I/O is much better than that of separate I/O, because in collective I/O the requested disk blocks are accessed only once even when the total number of disk blocks that should be read is beyond the size of the buffer cache. In 2-D LD, two columns of LDBs should be read from all I/O nodes for the access patterns. The number of LDBs is beyond the size of the buffer cache, but by reading a disk block once, all I/O requests to read the data segments in the disk block can be severed if the requests are performed collectively. Thus the cache is never thrashed while performing I/O to read data segments in the same disk blocks.

In 1-D LD, the performances for the CCX, SC, and BC access patterns are almost the same, because the number of disk accesses and the patterns of disk block accesses (including the permutation pattern in each compute node) are the same. The number of disk accesses for the SB access pattern is the same as for the CCX, SC, and BC access patterns. In this access pattern, each compute node accesses 4×2048 data regions at a time. Thus, in every access, whole LDBs should be read from the disk which is the same as accessing disk blocks contiguously in the same order they are stored. In other access patterns, whole LDBs should be accessed; however, only half of each LDB is read, which is the same as accessing disk blocks in stride, because one LDB is mapped into two contiguous disk blocks.

8.4 Analysis of Data Transfer

In collective I/O operations a compute node can access multiple disk blocks, and the possibility that other compute nodes will access the data segments of the same disk blocks is high. In the previous sections we analyzed data transfers of a single disk block in terms of the number of communications between I/O and compute nodes. In this section we analyze the data transfers of multiple disk blocks and compare UPIO's collective I/O operations with DDIO in terms of the number of communications.

Let's suppose that a system is configured with four compute nodes and four I/O nodes, as shown in Figure 8.7, and each compute node accesses the data segments of the same four disk blocks in every I/O node. If each I/O node sends data segments or disk blocks to compute nodes in the same order, there can be access conflicts to the same compute node and thus data transfers can be serialized. In UPIO and DDIO, data transfers and disk accesses are performed in parallel. But to analyze both cases in terms of data transfers we assumed that data transfers are performed after accessing all data from the disk in each I/O node.[2]

In collective I/O operations of UPIO, a disk block is the unit of a data transfer between I/O and compute node. When a compute node receives a disk block from

[2]Even if data transfer and disk accesses are performed in parallel, data transfers can be analyzed in the same way.

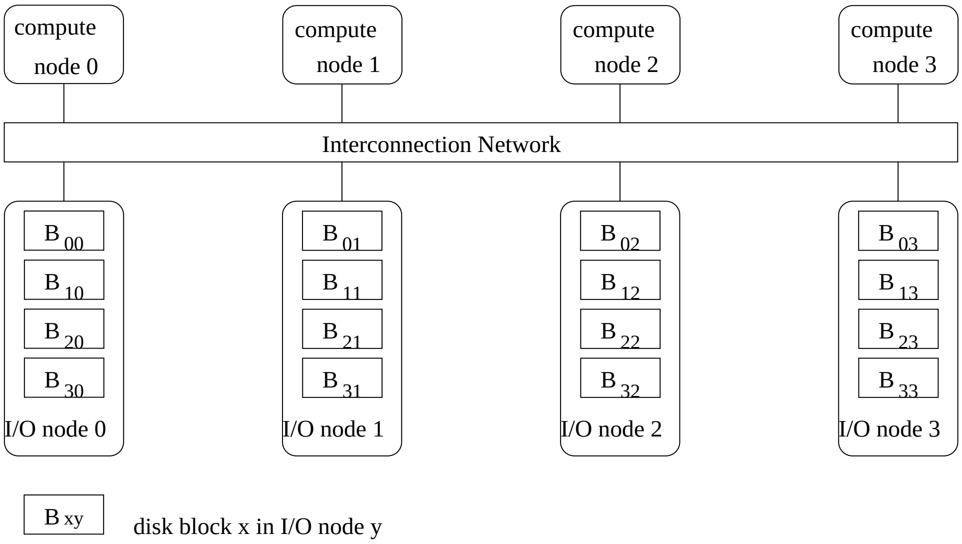


Figure 8.7: An example system configuration

an I/O node, it extracts or fills data segments from or to the disk block and sends the block to the next compute node. After sending the disk block, the compute node receives another one from an I/O node and does the same thing. Thus, the action is like a computer pipeline. Different compute nodes are accessing different data segments of different disk blocks in parallel. Each compute node corresponds to a pipeline stage and is connected through an interconnection network to form a pipe — disk blocks enter at one end, progress through the compute nodes (stages), and exit at the other end.[3]

We use a two-dimensional chart to illustrate the overlapped communications in pipelined compute nodes through an interconnection network. The table is similar to a *space-time* diagram or a *reservation table* [42] depicting the overlapped operations in a linear pipeline processor. We call the chart a *communication table*.

The communication table shown in Figure 8.8 illustrates the communication steps for data transfers of the disk blocks shown in Figure 8.7, where we assume that

[3]In write operations the disk blocks are sent back to I/O nodes.

Compute nodes

B_{xy} : disk block x in I/O node y

P_3	X	X	X	B_{00}	B_{01}	B_{02}	B_{03}	B_{10}	B_{11}	B_{12}	B_{13}	B_{20}	B_{21}	B_{22}	B_{23}	B_{30}	B_{31}	B_{32}	B_{33}
P_2	X	X	B_{00}	B_{01}	B_{02}	B_{03}	B_{10}	B_{11}	B_{12}	B_{13}	B_{20}	B_{21}	B_{22}	B_{23}	B_{30}	B_{31}	B_{32}	B_{33}	X
P_1	X	B_{00}	B_{01}	B_{02}	B_{03}	B_{10}	B_{11}	B_{12}	B_{13}	B_{20}	B_{21}	B_{22}	B_{23}	B_{30}	B_{31}	B_{32}	B_{33}	X	X
P_0	B_{00}	B_{01}	B_{02}	B_{03}	B_{10}	B_{11}	B_{12}	B_{13}	B_{20}	B_{21}	B_{22}	B_{23}	B_{30}	B_{31}	B_{32}	B_{33}	X	X	X
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Time (number of communications)

Figure 8.8: A communication table for data transfers in UPIO

it takes the same time to extract or fill data segments from or to the disk block in each compute node. The rows represent the number of compute nodes that access the disk blocks, and they correspond to a pipeline stage of a four-compute-node pipeline. The total number of time units in the table represents the total number of communications for data transfers.

A marked entry with an X in the (i, j)th square of the table indicates that compute node P_i does not receive any data segment from an I/O or compute node. In Figure 8.7 we assumed that each I/O node sends data segments or disk blocks to the compute nodes in the same order. Thus, there can be access conflicts to the same compute node, so that at the beginning of the data transfers the pipeline of compute nodes is not filled up.

Once the pipe is filled up, it will complete data transfers of one disk block per a certain period of time (communication step), independent of the number of compute nodes in the pipe and the number of I/O nodes. Ideally, B disk blocks can be transferred to p compute nodes in $C_p = p + (B - 1)$ communication steps, where p steps are used to fill up the pipeline or to complete data transfers of the first disk block, and $B - 1$ communication steps are needed to complete the data transfers of the remaining $B - 1$ disk blocks. The number of disk blocks transferred from an

I/O node i, say B_i, can be different and thus the total number of communications for data transfers will be

$$C_p = p + (\sum_{i=0}^{p-1} B_i - 1). \tag{8.1}$$

In DDIO, however, the number of communications depends on the number of compute and I/O nodes, the number of noncontiguous data segments in a disk block that are accessed by a compute node, and the number of disk blocks being accessed, because all data segments in a disk block are sent separately by an I/O node to each compute node and there is no communication between compute nodes.

To analyze the communications for data transfers in DDIO, let's assume that each compute node accesses a single data segment of a disk block in Figure 8.7. Then the communication steps can be illustrated, as shown in Figure 8.9, even if the compute nodes do not form a pipeline. (In this figure we assume that the number of disk blocks transferred by each I/O node is the same and that the number of compute nodes and the number of I/O nodes are the same.) We can notice that the number of communications is the same as that in UPIO. However, if each compute node accesses multiple noncontiguous data segments of a disk block, say S data segments of each disk block (from now on we assume that same number of data segments are accessed from each disk block), the same communication steps should be iterated S times. Therefore, the total number of communications for data transfers between the compute nodes and the I/O nodes will be

$$C_p = p + (S \sum_{i=0}^{p-1} B_i - 1), \tag{8.2}$$

where we assumed that $B_0 = B_1 = ... = B_{p-1}$ and $S \geq 1$.

If we assume that the number of disk blocks accessed by each compute node is different in each I/O node (say $B_0 < B_1 < ... < B_{p-1}$), the number of communication steps can be varied according to the order in which the I/O nodes send data segments

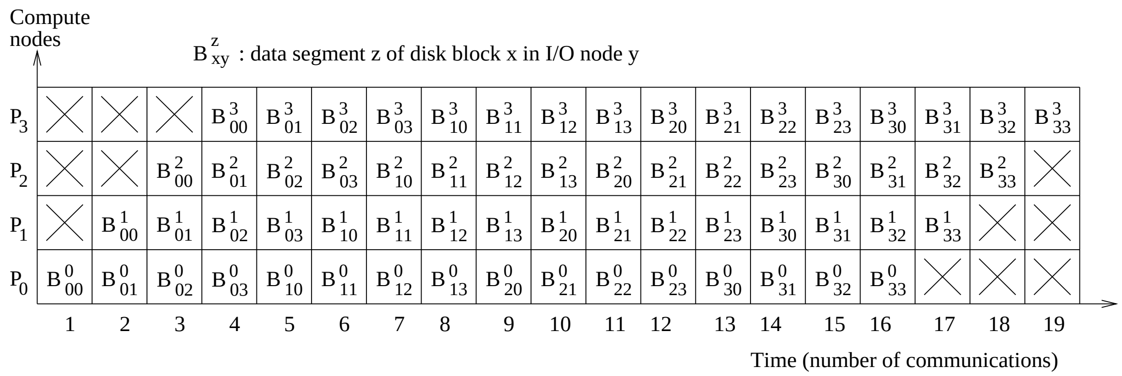


Figure 8.9: A communication table for data transfers in DDIO

to the compute nodes. Let's assume that I/O node $p-1$ sends a data segment first, I/O node $p-2$ sends a data segment next, and so on. Then it can be the best case, because after the pipe of p stage is filled by sending a data segment by I/O node 0 at the pth communication step, data transfers for a disk block are completed in every communication step, at least for $p-1$ steps.

However, if I/O nodes send data segments in an ascending order (that is, I/O node 0 sends a data segment first, I/O node 1 sends a data segment next, and so on), it will be the worst case, because just after the pipe of p stage is filled by sending a data segment by I/O node $p-1$ at the pth communication step, there can be no more data segments sent by I/O node 0. That is, there can be empty stages in the pipe.

In Figure 8.7 let's assume that $S = 1$ and the number of disk blocks accessed by each compute node is different in each I/O node, say one disk block in I/O node 0, two disk blocks in I/O node 1, three disk blocks in I/O node 2, and four disk blocks in I/O node 3. Then the communication table for the data transfers can be depicted as shown in Figure 8.10 (a) if the I/O nodes send data segments in a descending order. If the I/O nodes send data segments in an ascending order, the communication table can be illustrated by Figure 8.10 (b).

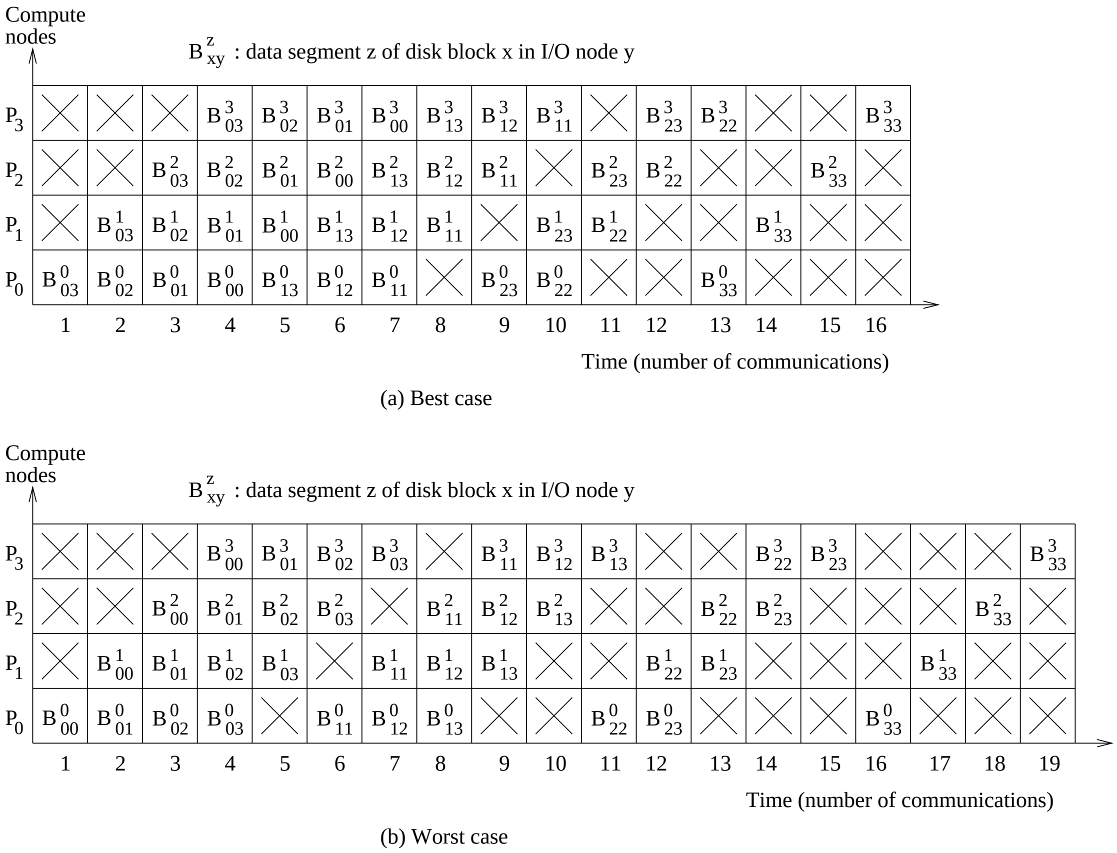


Figure 8.10: Transfers of different numbers of data segments

In Figure 8.10 (a) we can see that only three I/O nodes transfer data segments to three compute nodes at the eighth time unit, even though compute node P_0 can receive a data segment from any I/O node, because each I/O node can send only one data segment to a compute node at one time. From that time unit, only three compute nodes receive data segments for a while. At the eleventh time unit only two compute nodes receive data segments from two I/O nodes (I/O nodes 2 and 3), because the other I/O nodes sent all requested data segments before the time unit.

However, we can assume that the I/O nodes that complete data transfers still send data segments virtually (e.g., I/O node 0 sends data segments virtually at the 8th and 11th time units). Then we can see that each I/O node sends a data segment B_{p-1} times to the same compute node in every pth time unit. Note that when I/O node $p-1$ sends a data segment to a compute node B_{p-1} times, all I/O nodes have completed data transfers, or they complete data transfers while I/O node $p-1$ is sending data segments to the other compute nodes. Therefore, the number of communications depends on the I/O node that transfers the data segments of the largest number of disk blocks, and it will be

$$C_p = S \times p(\max_{0 \le i \le p-1} B_i),^4 \tag{8.3}$$

where we assume that the number of compute nodes and the number of I/O nodes are the same and that each compute node accesses S data segments from each disk block.

In Figure 8.10 (b) note that, in order to complete data transfers, $p-1$ more communication steps are needed, compared to the communication steps in Figure 8.10 (a), because I/O node $p-1$ started to send data segments of disk blocks at the end of the order. That is, in these communication steps, only the I/O node sends data segments to the compute nodes; on the other hand, the other I/O nodes have

[4] If $B_{p-1} = B_{p-2} = ... = B_k$, $k-1$ communication steps should be added, because I/O nodes $p-2, p-3, ..., k$ have to send data segments of the last block to complete data transfers.

completed data transfers. Therefore, the number of communications in the worst case will be

$$C_p = S \times p(\max_{0 \leq i \leq p-1} B_i) + p - 1. \tag{8.4}$$

Until now we have assumed that the number of compute nodes and the number of I/O nodes accessed by the compute nodes are the same. But the number of compute nodes and the number of I/O nodes (say, d I/O nodes) can be different: $p > d$ or $p < d$.

When the number of compute nodes is larger than the number of I/O nodes, i.e., $p > d$, only d compute nodes can, at most, receive data segments from I/O nodes. Therefore, if we depict a communication table for this case, there must be empty slots in the middle of the communication steps. Figure 8.11 illustrates the communication procedure between compute nodes and I/O nodes when the number of compute nodes is larger than the number of I/O nodes ($p = 6$ and $d = 4$).

In Figure 8.11 we assume that each I/O node sends the same number of data segments of the same number of disk blocks (say, three disk blocks) to the compute nodes. We can consider the marked entries with an 'X' in this figure as the data segments that are sent by $p-d$ virtual I/O nodes. Then the number of communications will be

$$C_p = p + (S \times p \times B - 1) - (p - d) = d + S \times p \times B - 1, \tag{8.5}$$

where $B \equiv B_0 = B_1 = ... = B_{d-1}$. If $B_0 < B_1 < ... < B_{d-1}$ and data segments are sent in a descending order as explained above, the number of communications will be

$$C_p = S \times p(\max_{0 \leq i \leq d-1} B_i). \tag{8.6}$$

If $B_0 < B_1 < ... < B_{d-1}$ and data segments are sent in an ascending order, the

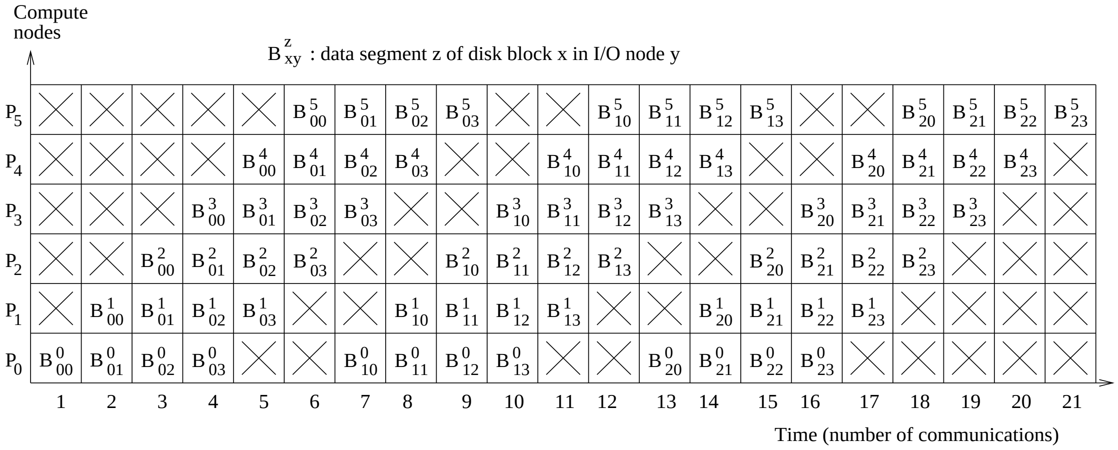


Figure 8.11: Data transfers when the number of compute nodes is larger than the number of I/O nodes

number of communications will be

$$C_p = S \times p(\max_{0 \leq i \leq d-1} B_i) + p - 1. \tag{8.7}$$

However, if the number of I/O nodes is larger than the number of compute nodes (i.e., $p < d$), it can be a little different, because there can always be access conflicts to the same compute node. In this situation we can consider two cases: 1) I/O nodes send data segments in a round-robin fashion to the compute nodes, and 2) I/O nodes send data segments in a random order.

Let's suppose a system is configured with four compute nodes and six I/O nodes and that each I/O node accesses data segments of three disk blocks in each I/O node, and $S = 1$. Then a communication table for the first case can be illustrated as shown in Figure 8.12 (a). A communication table for the second case can be depicted as shown in Figure 8.12 (b), where we assume that I/O nodes 0, 1, 2, and 3 send all data segments to the compute nodes before I/O nodes 4 and 5 send data segments.

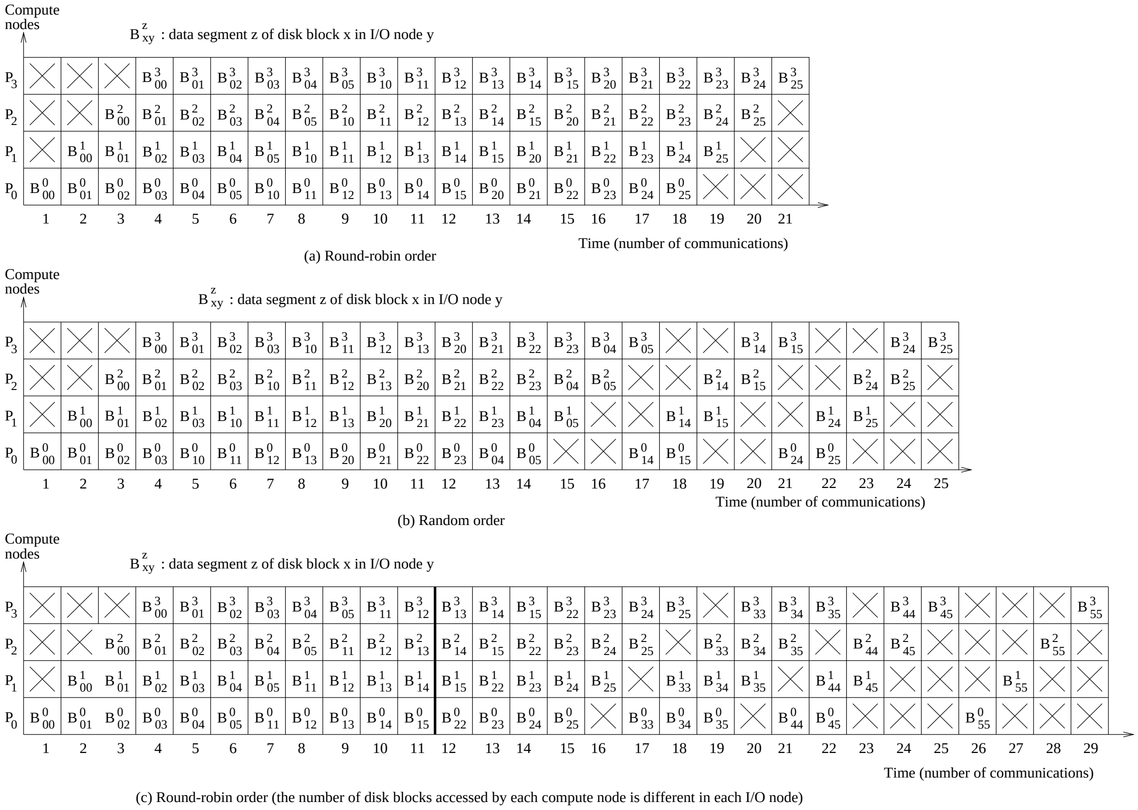


Figure 8.12: Data transfers when the number of I/O nodes is larger than the number of compute nodes

If I/O nodes always send data segments in a round-robin fashion as illustrated in Figure 8.12 (a), data segment transfers will be completed in every communication step after a column of the communication table (a pipeline of compute nodes) is filled. Thus this is the best case. If I/O nodes send data segments in a random order, there can be empty entries in the communication table, as shown in Figure 8.12 (b), depending on the order. Therefore, the number of communications for the best case will be

$$C_p = p + (S \sum_{i=0}^{d-1} B_i - 1), \tag{8.8}$$

where we assume that $B_0 = B_1 = ... = B_{d-1}$.

If we assume that the number of disk blocks accessed by each compute node is different in each I/O node, the number of communication steps can be calculated as follows. Let's assume that the number of disk blocks that will be sent by I/O nodes $0, 1, ..., d-p-1, d-p, d-p+1, ..., d-1$ is $B_0, B_1, ..., B_{d-p-1}, B_{d-p}, B_{d-p+1}, ..., B_{d-1}$, respectively, where $B_0 < B_1 < ... < B_{d-p-1} < B_{d-p} < B_{d-p+1} < ... < B_{d-1}$, and the I/O node sends data segments in rank order. Then we know that a pipeline of p compute nodes will always be filled up after p communication steps until all data segments of B_{d-p-1} blocks are sent, because even though an I/O node completes data transfers, another I/O node can send data segments. After I/O node $d-p-1$ sends the last data segment to a compute node, only p I/O nodes keep sending data segments. From that moment we can consider that I/O nodes $d-p, d-p+1, ..., d-1$ are sending the data segments of the disk blocks $B_{d-p} - B_{d-p-1}, B_{d-p+1} - B_{d-p-1}, ..., B_{d-1} - B_{d-p-1}$, respectively, to the p compute nodes. Therefore, from Equations 8.3 and 8.4, the number of communications of this part will be

$$C_p^a = S \times p(\max_{d-p \leq i \leq d-1} (B_i - B_{d-p-1})), \qquad or$$
$$C_p^a = S \times p(\max_{d-p \leq i \leq d-1} (B_i - B_{d-p-1})) + p - 1.$$

Therefore, if the number of communications required to send all data segments in I/O nodes $0, 1, ..., d-p-1$ to the compute nodes is calculated, we can calculate the total number of communications. From Figure 8.12 (c), note that it takes $B_0 \times d$ communication steps to send the data segments of B_0 disk blocks in each I/O node. (In this figure we assume that $p = 4, d = 6, B_0 = 1, B_1 = 2, B_2 = 3, B_3 = 4, B_4 = 5$, and $B_5 = 6$.) After that, each I/O node will send the data segments of the remaining $B_i - B_0$ $(1 \leq i < d)$ disk blocks. Thus, it will take $(B_1 - B_0) \times (d-1)$ communication steps to send the data segments of $(B_1 - B_0)$ disk blocks in each I/O node. After that it will take $(B_2 - B_1 - B_0) \times (d-2)$ communication steps to send the data segments of $(B_2 - B_1 - B_0)$ disk blocks in each I/O node, and so on.

Therefore, the number of communications to send all data segments in I/O nodes $0, 1, ..., d-p-1$ to the compute nodes will be

$$
\begin{aligned}
& C_p^b = S \times B_0 \times d + \\
& S \times (B_1 - B_0) \times (d-1) + \\
& S \times (B_2 - B_1 - B_0) \times (d-2) + \\
& S \times (B_3 - B_2 - B_1 - B_0) \times (d-3) + \cdots\cdots + \\
& S \times (B_{d-p-1} - B_{d-p-2} - \cdots - B_1 - B_0) \times (d - (d-p-1)) \\
& = S\left(d(\sum_{i=0}^{d-p-1} B_i - \sum_{i=0}^{d-p-2} (d-p-1-i)B_i) - \sum_{i=1}^{d-p-1} iB_i \right) + \\
& \quad S \sum_{i=0}^{d-p-2} B_i \frac{(d-p-1-i)(i+d-p)}{2}.
\end{aligned}
$$

Consequently, if $p < d$, the total number of communications will be

$$
\begin{aligned}
C_p &= C_p^a + C_p^b \\
&= S(p(\max_{d-p \leq i \leq d-1}(B_i - B_{d-p-1})) + p - 1 + d(\sum_{i=0}^{d-p-1} B_i - \sum_{i=0}^{d-p-2} (d-p-1-i)B_i) \\
& \quad - \sum_{i=1}^{d-p-1} iB_i + \sum_{i=0}^{d-p-2} B_i \frac{(d-p-1-i)(i+d-p)}{2}), \qquad or
\end{aligned}
$$

$$
\begin{aligned}
C_p &= S(p(\max_{d-p\leq i\leq d-1}(B_i - B_{d-p-1})) + d(\sum_{i=0}^{d-p-1} B_i - \sum_{i=0}^{d-p-2}(d-p-1-i)B_i) - \\
&\quad \sum_{i=1}^{d-p-1} iB_i + \sum_{i=0}^{d-p-2} B_i \frac{(d-p-1-i)(i+d-p)}{2}). \qquad (8.9)
\end{aligned}
$$

The number of communications for data transfers between I/O and the compute nodes in UPIO depends only on the number of disk blocks accessed by compute nodes in a collective I/O operation and on the number of the compute nodes. In DDIO, however, it depends on the number of I/O and compute nodes, the number of disk blocks accessed in each I/O node, the number of noncontiguous data segments in each disk block, and the order in which I/O nodes send data segments to the compute nodes. Through the analysis described above we know that Equations 8.2, 8.3, 8.5, 8.6, 8.8, and 8.9 represent the number of communications of the best case in each configuration explained above. Therefore, we can see that the number of communications of UPIO is always less than the number of communications of DDIO by comparing the equations with Equation 8.1. In Equation 8.2, if $S = 1$, this is the only case in which the number of communications of DDIO is the same as the number of communications of UPIO.

Chapter 9

Evaluation of UPIO with Real Applications

In this chapter we study the performance of UPIO with some important scientific computations, such as matrix multiplication and the linear equation solver. Designing external algorithms for the computations based on LD is also included in this study.

One of the best ways to design an external algorithm is to base it on an efficient internal algorithm, because that algorithm may already be efficient in terms of computation, communication, or memory usage. Therefore, if we modify the algorithm to be I/O efficient, then the new external algorithm may be optimal in terms of I/O, computation, and communication. In this chapter we show how programmers can produce high-performance external algorithms by using UPIO based on efficient internal algorithms.

9.1 I/O and Memory-Efficient External Matrix Multiplication

There has been a great deal of interest in developing parallel algorithms for multiplying dense matrices, since matrix multiplication is a core component of many scientific applications. Algorithms for multicomputers can be classified as *row-column-oriented* or *block-oriented* algorithms [62]. The performance of block-oriented matrix multiplications is superior to the row-, column-oriented algorithm, since the row-, column-oriented algorithm has a critical deficiency: the communication time increases linearly as the number of processors increases.

Cannon's and Fox's algorithms [62] are well-known, block-oriented, memory-efficient parallel algorithms for matrix multiplication. The performance of Fox's algorithm is inferior to Cannon's if the algorithm does not execute in an asynchronous manner. In this section we show how users can produce high-performance external computation algorithms using UPIO with Cannon's algorithm.

9.1.1 Cannon's Algorithm

Let A, B, and C be $N \times N$ matrices such that $C = A \times B$. A and B are partitioned into submatrices $A_{i,j}$ and $B_{i,j}$ of size $\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}$ each among the $\sqrt{P} \times \sqrt{P}$ logical mesh of compute nodes, where each compute node is labeled $P_{i,j}$ and $0 \leq i, j < \sqrt{P}$. Compute node $P_{i,j}$ stores $A_{i,j}$ and $B_{i,j}$.

In the first phase of the execution of the algorithm, each compute node sends the submatrices $A_{i,j}$ and $B_{i,j}$ to the compute nodes $(i, (j+i) mod \sqrt{P})$ and $((j+i) mod \sqrt{P}, j)$, respectively, to multiply the submatrices together locally. Next, the submatrix of A moves one step left and the submatrix of B moves one step up (with wraparound), and then the submatrices are multiplied and the results added to the partial results in the submatrix C. The multiplication is complete after $\sqrt{P}$ submatrix multiplications and single-step shifts of A and B leftward and upward, respectively.

For external matrix multiplication we assume from now on that $N \times N$ matrices A and B are so large that they do not fit into the memory at the same time, and matrix multiplication is performed on a $\sqrt{P} \times \sqrt{P}$ logical mesh of compute nodes.

9.1.2 Data Format

Defining the data format on the physical disk is important. For example, if a matrix is needed by columns, it does not make sense to store it on the physical disk by rows. Sometimes, rectangular regions of a matrix could be needed, as in Cannon's algorithm. In that case it would degrade I/O performance if the matrix were stored by either column- or row-major order.

UPIO allows users to define the data format on disks to match the computational requirements by defining a logical disk and a logical disk block. Since a submatrix is the basic computational unit for Cannon's algorithm, the submatrix can be defined as an LDB. A square or rectangular submatrix can be an example of an LDB. Mapping a rectangular submatrix into an LDB may be appropriate when N is reasonably large and $\frac{N}{\sqrt{P}}$ elements can fit into the memory at the same time. If N is so large that $\frac{N}{\sqrt{P}}$ elements cannot fit into the memory at the same time, mapping a square matrix into an LDB may be appropriate.

In this study we assumed that the $1 \times \frac{N}{\sqrt{P}}$ submatrix of A and the $\frac{N}{\sqrt{P}} \times 1$ submatrix of B are mapped into an LDB, respectively. That is, matrices A and B are stored on disks through $N \times \sqrt{P}$ and $\sqrt{P} \times N$ LDs, respectively. The resulting matrix

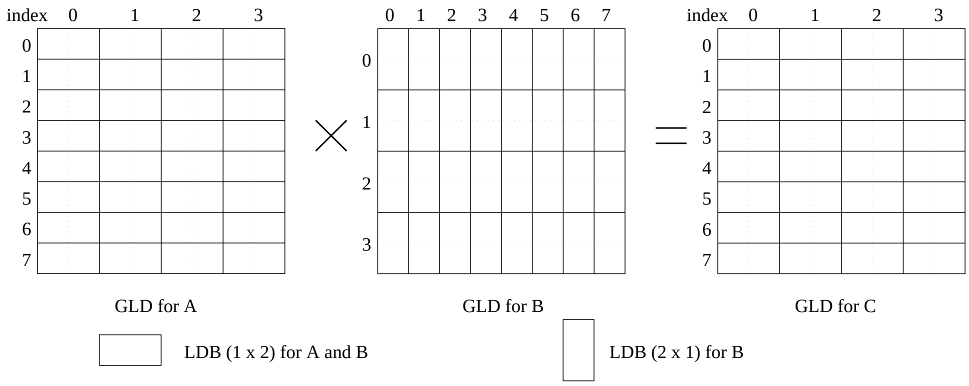


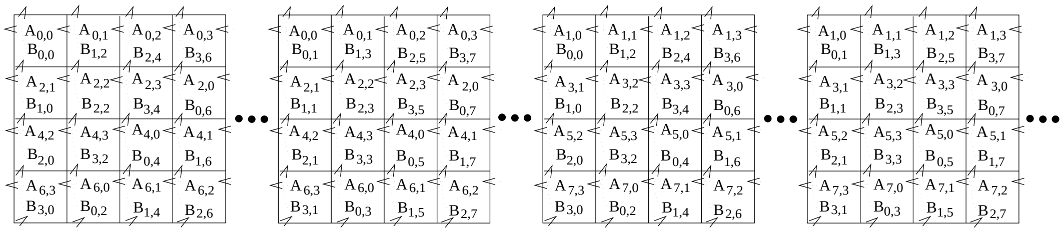
Figure 9.1: External matrix multiplication using UPIO

C is also mapped into a two-dimensional LD as matrix A. The LDBs of all LDs are distributed across I/O nodes in `(BLOCK, BLOCK)` fashion using the `GLD_distribute` operation.

9.1.3 Algorithm Modification

Figure 9.2 shows the scheduling for accessing LDBs and the multiplication of the LDBs based on Cannon's algorithm for 16 (4×4) compute nodes. In this figure we assume that $N = 8$ and matrices A and B are stored on disks through LDs, as shown in Figure 9.1, where the 1×2 submatrices of A and C are mapped into an LDB, the 2×1 submatrix of B is mapped into an LDB, and each LDB for matrices A, B, and C is denoted as $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$, respectively.

Based on Cannon's algorithm, the LDBs are logically assigned to each compute node in `(BLOCK, BLOCK)` fashion; each compute node calculates 2×1 LDBs of C with 2×1 LDBs of A and 1×2 LDBs of B. In Cannon's algorithm, an initial alignment of submatrices A and B is needed to multiply the submatrices together locally, which requires communications. However, for external matrix multiplication, the initial alignment is achieved by reading LDBs as shown in Figure 9.2 (a), since each



(a) Initial reading of A and B (b) Second reading of B and after 4th shift of A (c) Second reading of A and 3rd reading of B (d) Fourth reading of B and after 4th shift of A

Figure 9.2: I/O and communication steps for external matrix multiplication

compute node can directly access any LDB in UPIO.[1]

The following steps are the same as those of the internal algorithm, but a sequence of $\sqrt{P}$ LDB (submatrix) multiplications and communications does not complete the multiplication of matrices A and B as in Cannon's algorithm, because it produces only a portion of an LDB of C. Next, the LDBs of B should be fetched in memory, as shown in Figure 9.2 (b), and the submatrix of A should be shifted one step left to place it at its original compute node for the next computations. The computation sequence should then be continued as in the internal algorithm.

This sequence of computations produces only one LDB of C in each compute node; therefore, each compute node has to fetch the next LDB of A as shown in Figure 9.2 (c). An LDB of B is also fetched again,[2] and the following steps are the same as in the previous computation sequence. The whole procedure for external matrix computation is illustrated in Figure 9.3.

[1] We assumed here that the memory of each compute node can store at least three LDBs.

[2] The LDB of B currently in memory can be reused by algorithm modification to optimize the I/O cost.

procedure EXTERNAL_MATRIX_MULTIPLICATION
// A and B: $N \times N$ matrices
// A and B are stored through $N \times \sqrt{P}$ and $\sqrt{P} \times N$ GLDs, respectively.
// $1 \times \frac{N}{\sqrt{P}}$ and $\frac{N}{\sqrt{P}} \times 1$ submatrix of A and B are mapped into an LDB, respectively.
// The result matrix C is stored through an $N \times \sqrt{P}$ GLD as matrix A.
// M_a, M_b, and M_c: temporary memory for matrices A, B and C, respectively.
// $M_a = M_b = M_c = k\frac{N}{\sqrt{P}}$: the size of each temporary memory, $k \geq 1$.
// LDD_a, LDD_b, and LDD_c are LDDs for matrices A, B, and C, respectively
// mynode.x and mynode.y: coordinators of mynode
begin
    PerNodeLDB = $\frac{N}{\sqrt{P}}$; /* no. of LDBs to be processed by each compute node */
    a_i = mynode.y * PerNodeLDB;
    $a_j = b_i$ =(mynode.x + mynode.y) mod $\sqrt{P}$;
    b_j = mynode.x * PerNodeLDB;
    c_i = mynode.y * PerNodeLDB;
    c_j = mynode.x;
    for ($i = 0$; $i <$ PerNodeLDB: $i = i + k$)
        initialize temporary memory M_c;
        GLD_read(LDD_a, M_a, k, SOUTH, $a_i + i$, a_j)
        for ($j = 0$; j < PerNodeLDB: $j = j + k$)
            GLD_read(LDD_b, M_b, k, EAST, b_i, $b_j + j$)
            internal_processing(M_a, M_b, M_c, j, k, 1, $\frac{N}{\sqrt{P}}$)
            for ($p = 1$; $p < \sqrt{P}$; $p = p + 1$)
                shift LDBs of A and B;
                internal_processing(M_a, M_b, M_c, j, k, 1, $\frac{N}{\sqrt{P}}$)
            shift LDBs of A one step left; /* restore LDBs to its original place */
        GLD_write(LDD_c, M_c, k, SOUTH, $c_i + i$, c_j)
end

Figure 9.3: External matrix multiplication

```
procedure INTERNAL_PROCESSING (M_a, M_b, M_c, j, k, r, c)
// M_a, M_b, and M_c: temporary memory for matrices A, B and C, respectively.
// k: number of LDBs.
// r, c: r × c LDB
// j: jth LDB of B (this procedure calculates k elements of k LDBs of C from jth
// element of each LDB)
begin
    S_ldb = r * c;    /* the number of elements in an LDB */
    for (l = 0; l < k; l++)
        for (m = 0; m < k; m++)
            for (n = 0; n < c; n++)
                M_c[S_ldb * l + j + m] += (M_a[S_ldb * l + n] * M_b[S_ldb * m + n]);
end
```

Figure 9.4: Submatrix multiplication

9.1.4 Complexity

In order to evaluate the performance of the algorithm presented in Figure 9.3, we analyze the code in terms of I/O, communication, and computation cost.

Using the memory size of each compute node, the size of the LDB, and the size of the logical disk, we can calculate the number of I/O operations. Since we assumed that the $1 \times \frac{N}{\sqrt{P}}$ submatrix of A and the $\frac{N}{\sqrt{P}} \times 1$ submatrix of B are mapped into an LDB, respectively, the memory size, M, of each compute node should be a multiple of $\frac{3N}{\sqrt{P}}$.

Note that for each LDB of A, each compute node accesses $\frac{N}{\sqrt{P}}$ LDBs of B. In other words, to calculate an LDB of C and write it back to a disk, the number of I/O operations per compute node is $2 + \frac{N}{\sqrt{P}}$. Therefore, the total number of I/O operations required to calculate all LDBs of C will be

$$(2 + \frac{N}{\sqrt{P}}) \times \frac{N}{\sqrt{P}} = \frac{2N}{\sqrt{P}} + \frac{N^2}{P}.$$

Of course, the number of I/O operations depends on the memory size, M. Let's suppose $M = k\frac{3N}{\sqrt{P}}$ ($k \geq 1$) and the memory is assigned evenly for matrices A, B, and C. Then the total number of I/O operations will be

$$\left(\frac{2N}{\sqrt{P}} + \frac{N^2}{P}\right) \Big/ k. \tag{9.1}$$

To calculate an element of an LDB of C (if $k = 1$), the external matrix multiplication algorithm requires $\sqrt{P}$ two-steps of shift operations (each consists of a single-step shift of LDBs of A and B in all rows and columns of the compute node mesh, respectively) among groups of $\sqrt{P}$ compute nodes. That is, for each LDB of A and B, $2\sqrt{P}$ shifts are required (line 14 of Figure 9.3). Since $\frac{N^2}{kP}$ I/O operations for B are required to calculate all LDBs of C (line 10 of the algorithm), the total required shifts will be $\frac{2N^2}{k\sqrt{P}}$. In each shift the message consists of $k\frac{N}{\sqrt{P}}$ elements; therefore, the total communication cost will be

$$\frac{2N^2}{k\sqrt{P}}\left(\tau + \mu\frac{kN}{\sqrt{P}}\right). \tag{9.2}$$

The procedure for submatrix (LDB) multiplication in memory is presented in Figure 9.4. If we assume that an addition and multiplication pair (line 5) takes unit time,[3] the computation cost of submatrix multiplication will be $k^2\frac{N}{\sqrt{P}}$. This procedure is called in the procedure presented in Figure 9.3 $N^2/k^2\sqrt{P}$ times, and then the total computation cost will be $\frac{N^3}{P}$.

Therefore, the approximate overall parallel runtime[4] of the external matrix multiplication based on Cannon's algorithm will be

$$\frac{N^3}{P} + \frac{2N^2}{k\sqrt{P}}\left(\tau + \mu\frac{kN}{\sqrt{P}}\right) + \frac{1}{k}\left(\frac{2N}{\sqrt{P}} + \frac{N^2}{P}\right), \tag{9.3}$$

[3] We disregard here the cost for calculating the index of memory.

[4] The last term is just the number of I/O operations, but I/O cost is so closely related to it that we added the term here.

where k is the number of LDBs that each temporary memory for the matrices A, B, and C can store at a time.

9.1.5 Experimental Results

In our experiments we configured the simulator with 16 compute nodes and varied the number of I/O nodes 4, 8, and 16, where I/O nodes are logically arranged into 2×2, 2×4, and 4×4 grids, respectively. Each node has a 25 MHz CPU and a direct-memory access (DMA) capability. The I/O node has a thirty-two 4-KB-buffer cache that uses an LRU-replacement, write-behind strategy, and an HP97560 [85] disk attached to an SCSI bus whose peak bandwidth is 10 MB/s. The disk capacity and peak transfer rate are 1.3 GB and 2.34 MB/s, respectively. The file system block size is 4-KB.

To evaluate the performance of the external matrix multiplication based on Cannon's algorithm, two $1K \times 1K$ matrices of integer are multiplied, where the matrices are stored across the I/O nodes contiguously on the physical disk of each I/O node. Table 9.1 shows the performance of the external matrix multiplication, where the number of LDBs each temporary memory stores at a time is varied 4, 8, and 16.

In Equation 9.3 we know that the computation and communication costs (first and second terms) depend on the number of compute nodes or memory size. (We described the computation and communication cost together as computation time in Table 9.1.) The number of I/O operations depends on the number of compute nodes and memory size. However, the I/O cost depends on the number of I/O nodes and disk model. Thus we varied the number of I/O nodes and memory size in our experiments.

Table 9.1: Performance of matrix multiplication for various number of I/O nodes (milliseconds)

	4 I/O nodes		8 I/O nodes		16 I/O nodes	
LDBs	comp	I/O	comp	I/O	comp	I/O
4	233781.75	100230.31	233816.12	25721.51	233783.71	14820.95
8	224011.45	31779.84	224039.07	11104.68	224556.71	7504.61
16	213731.66	12948.10	213742.42	6931.94	213739.82	4026.35

Table 9.1 shows that the computation time is almost the same even if the number of I/O nodes is changed. However, it decreases as the memory size increases, because increasing the memory size decreases the number of I/O operations, and then the number of shifts also decreases. I/O time increases as the number of I/O nodes decrease, because a group of compute nodes are forced to access the same I/O node at the same time. Of course, I/O time decreases as the memory size increases, because more LDBs can be accessed at a time.

In our experiments, since the main focus is to evaluate the I/O performance of UPIO, we also implemented the algorithm with traditional file systems using file pointers on our simulator.[5] In this experiment, data is stored contiguously in skewed round-robin fashion across the I/O nodes; the matrices A and B are stored in row- and column-major order, respectively, and the result matrix C is stored in row-major order.

To run the program, we have to change the `GLD_read` and `GLD_write` operations in lines 9, 11, and 17 of the algorithm presented in Figure 9.3 into k I/O operations, respectively, because the k $1 \times \frac{N}{\sqrt{P}}$ or $\frac{N}{\sqrt{P}} \times 1$ submatrices are not contiguous and cannot be accessed with a single I/O operation in traditional file systems.[6]

[5]We implemented the same I/O operations using file pointers.

[6]In UPIO, even if the submatrix is not defined as an LDB, only single I/O operation is enough

Therefore, despite increasing memory size, the number of I/O operations is always $\left(\frac{2N}{\sqrt{P}} + \frac{N^2}{P}\right)$. In the external matrix multiplication, using a traditional file system, we measure only the I/O time, because the number of I/O operations in the loops of the algorithm does not affect the computation and communication time.

Table 9.2 compares the I/O time of the external matrix multiplication program with UPIO and a traditional file system using file pointers. (Hereafter, we call the programs UPIO and FP, respectively.) In the FP version, the ith row of the matrices A and C and the column of the matrix B are stored in the following I/O node, because the matrices are stored in skewed round-robin fashion across I/O nodes: (i mod $D + i/D$) mod D, where D is the number of I/O nodes.

In the algorithm presented in Figure 9.3, note that compute node $P_{i,j}$ accesses the submatrices in $(\frac{N}{\sqrt{P}} \times i) + k$ rows of matrix A and $(\frac{N}{\sqrt{P}} \times j) + k$ columns of matrix B, where $0 \leq k < \frac{N}{\sqrt{P}}(\frac{1024}{4} = 256)$; $(\frac{N}{\sqrt{P}} \times i) + k$th row of matrix A is stored in I/O node $((\frac{N}{\sqrt{P}} \times i + k) \bmod D + \frac{(\frac{N}{\sqrt{P}} \times i + k)}{D}) \bmod D$, and $(\frac{N}{\sqrt{P}} \times j) + k$ column of matrix B is stored in I/O node $((\frac{N}{\sqrt{P}} \times j + k) \bmod D + \frac{(\frac{N}{\sqrt{P}} \times j + k)}{D}) \bmod D$. However, the I/O node number is not affected by i or j, but by k, because in our experiments we varied D 4, 8, and 16, and $\frac{N}{\sqrt{P}}$ is a multiple of D. Therefore, every compute node accesses data in the same I/O node in every I/O operation.

Table 9.2 shows that the I/O time of the FP version is not much affected by the number of I/O nodes, because every compute node accesses the same I/O node in every I/O operation. On the other hand, UPIO allows users to distribute data across the I/O nodes matching the access patterns, which minimizes the chance of accessing the same I/O node by a group of compute nodes at the same time. In the UPIO version, since the matrices are distributed across I/O nodes in `(BLOCK, BLOCK)` fashion, only $\frac{P}{D}$ compute nodes among P compute nodes access the same

to read the k submatrices [64].

I/O node at the same time. Thus, as the number of I/O nodes increases, the I/O time decreases and the UPIO version performs better than the FP version for any number of I/O nodes.

Of course, as the memory size increases, the I/O time of both versions decreases, because, in the UPIO version, as the memory size increases the number of I/O operations decreases. However, in the FP version the number of I/O operations is not affected by the memory size, because the data that each compute node accesses is not contiguous. The characteristics of the disk model we used affects the I/O time, and the disk buffer and read-ahead function of the disk model reduces I/O times for sequential read requests. As memory size increases, more sequential I/O requests can be issued, which may increase the hit ratio of the disk cache [85] and activate more I/O nodes; thus, the I/O time of the FP version may decrease as memory size increases.

Table 9.2: Comparison of I/O time (milliseconds)

	4 I/O nodes		8 I/O nodes		16 I/O nodes	
LDBs	FP	UPIO	FP	UPIO	FP	UPIO
4	280973.50	100230.31	276124.09	25721.51	257878.28	14820.95
8	133023.48	31779.84	98551.05	11104.68	92997.30	7504.61
16	70389.06	12948.10	48401.88	6931.94	40396.47	4206.35

9.1.6 I/O Optimization

In the external matrix multiplication algorithm presented in Figure 9.3, note that each compute node must access an LDB of A and $\frac{N}{\sqrt{P}}$ LDBs of matrix B to calculate an LDB of C if we assume k is 1. In order to calculate the next LDB of C in the algorithm, the next LDB of A and the same $\frac{N}{\sqrt{P}}$ LDBs of B that are used to calculate

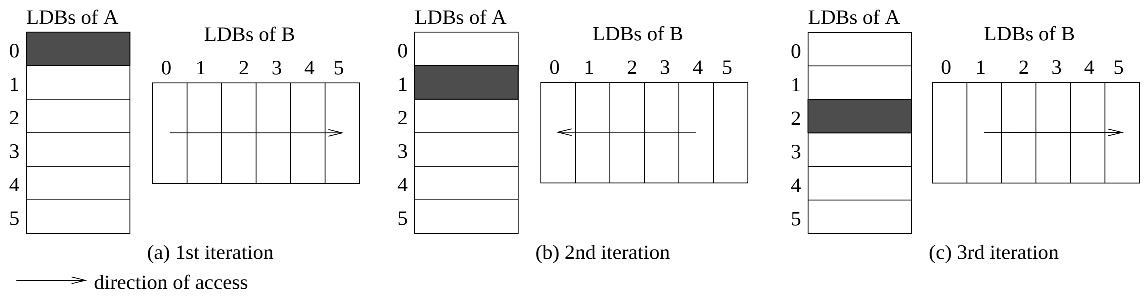


Figure 9.5: Procedure of I/O for I/O efficient matrix multiplication

the previous LDB of C are again fetched. However, if we read the LDBs of B in the opposite direction, it is possible to reuse the LDBs of B that are fetched for the last time to calculate the LDB of C and reduce the I/O time. That is, in each iteration except the first (line 7 in Figure 9.3), we can avoid reading the same LDB of B that is read for the last time in the iteration. The total number of I/O operations will be

$$\frac{1}{k}\left(\frac{2N}{P}+\frac{N^2}{P}-\frac{N}{P}+1\right)=\frac{1}{k}\left(\frac{N^2}{P}+\frac{N}{P}+1\right).$$

Let's suppose six LDBs of A and B are accessed by a compute node to calculate six LDBs of C, as shown in Figure 9.5. At the first iteration (Figure 9.5 (a) and line 7 in Figure 9.3), LDB 0 of A is read and all LDBs of B are read from LDB 0 to LDB 6. In the next iteration (Figure 9.5 (b)), LDB 1 of A is read and LDB 5 of B in memory is reused. The other LDBs of B are read from LDB 4 to LDB 0. In the next iteration, we can reuse the LDB 0 of B. Therefore, in each iteration except the first, we can avoid reading the same LDB of B that is read for the last time in the iteration. In this case, the total number of I/O operations will be $\frac{1}{k}\left(\frac{2N}{P}+\frac{N^2}{P}-\frac{N}{P}+1\right)$.

Table 9.3 compares the I/O times of the UPIO version and the optimized version. In many cases the I/O time of the optimized version is better than that of the UPIO version, because this algorithm reduces the actual amount of data read from disks as well as the number of I/O operations. In some cases the I/O time of the UPIO version is better than that of the optimized version. This is because the I/O time depends on the amount of data read at a time, total amount of data to be read from an I/O node, and the characteristics of the disk model. In fact, in [63], we show that the I/O time to read data that is stored contiguously forward and backward is different.

Table 9.3: I/O time of matrix multiplications (milliseconds)

	4 I/O nodes		8 I/O nodes		16 I/O nodes	
LDBs	Optimal	UPIO	Optimal	UPIO	Optimal	UPIO
4	91254.10	100230.31	26929.90	25721.51	22111.87	14820.95
8	33763.11	31779.84	14615.88	11104.68	7092.06	7504.61
16	11844.49	12948.10	7383.12	6931.94	3874.04	4026.35

9.1.7 Discussion

Our experiments showed that the functionalities of UPIO provide high-performance I/O, minimizing the number of I/O operations and maximizing the aggregate bandwidth of a large number of disks by reducing the possibility of accessing the same I/O node at the same time by a group of compute nodes. Indeed, as the number of I/O nodes increases, the I/O time of the UPIO version decreases rapidly. However, the I/O time of the FP version was not much affected by the number of I/O nodes because, in every I/O operation, all compute nodes access the same I/O node due to the simple round-robin data distribution.

We discussed how the I/O time varies with memory size. The number of I/O operations of the UPIO version decreases as memory size increases, because noncontiguous data blocks can be accessed with a single I/O operation. However, the number of I/O operations of the FP version was not affected by memory size, because non-contiguous data blocks cannot be accessed with a single I/O operation.

Our results emphasize that the abstraction of a one-dimensional file will not suffice to represent the semantics of data domain on disks. To maximize I/O performance and represent the semantics of data domain on disks, an n-dimensional file model and user-controllability in parallel I/O are needed.

9.2 I/O and Communication Efficient Laplace Equation Solver

Finite difference method and *finite element method* [62] are the techniques used to obtain an approximate solution to a partial differential equation governing the behavior of a physical system. These methods impose a regular grid on the physical domain. In order to obtain a solution for a grid point, they impose some operations on the grid point and its neighbors.

In the simplest application of the finite difference method, the problem domain is covered with a uniformly-spaced square mesh of nodal points, so that all points, except those on boundaries, have four equidistant nearest neighbors. For the purpose of discussion of the finite difference method for external computation, we consider a technique for solving a Laplace equation.

A Laplace equation solver by the Jacobi iteration method is one of the simplest iterative techniques. In this method the new values in each iteration are computed, using the values from the previous iteration. We may represent the Jacobi update procedure for any iteration using an array assignment statement as follows:

```
DO i = 2 : n-1
    DO j = 2 : n-1
        A(i, j) = 1/4(B(i, j-1) + B(i, j+1) + B(i+1, j) + B(i-1, j))
```

Figure 9.6: Code segments for the Laplace equation solver

9.2.1 Data Format

Let's assume that an $n \times n$ data array is decomposed on $P_X \times P_Y$ grid compute nodes in (BLOCK, BLOCK) manner, as shown in Figure 9.7, and the subarray (say $H \times N = \frac{n}{P_Y} \times \frac{n}{P_X}$) assigned to each compute node cannot, due to the limits of the main memory, all be in the main memory at the same time and is stored in a local logical disk.

Defining the format of data on the physical disk is important. For example, if a matrix is needed by columns, it does not make sense to store it on the physical disk by columns. Sometimes, rectangular regions of a matrix could be needed. In that case it does not make sense to store the matrix by either column- or row-major order. User-controllable I/O operations allow users to define the data format matching the computational requirements by defining the logical disk block.

We can define the logical disk block as block-wise or row-/column-wise for the Laplace equation solver, because four equidistant nearest neighbors are needed to calculate the new value of a point. A block-wise logical disk block can be defined as the basic unit of I/O, but in this case, in order to calculate the new values of the elements in a logical disk block, at least five logical disk blocks are required. For example, consider the logical disk block 12 shown in Figure 9.8 (a). The new value of each data element in block 12 is calculated using the values of its corresponding

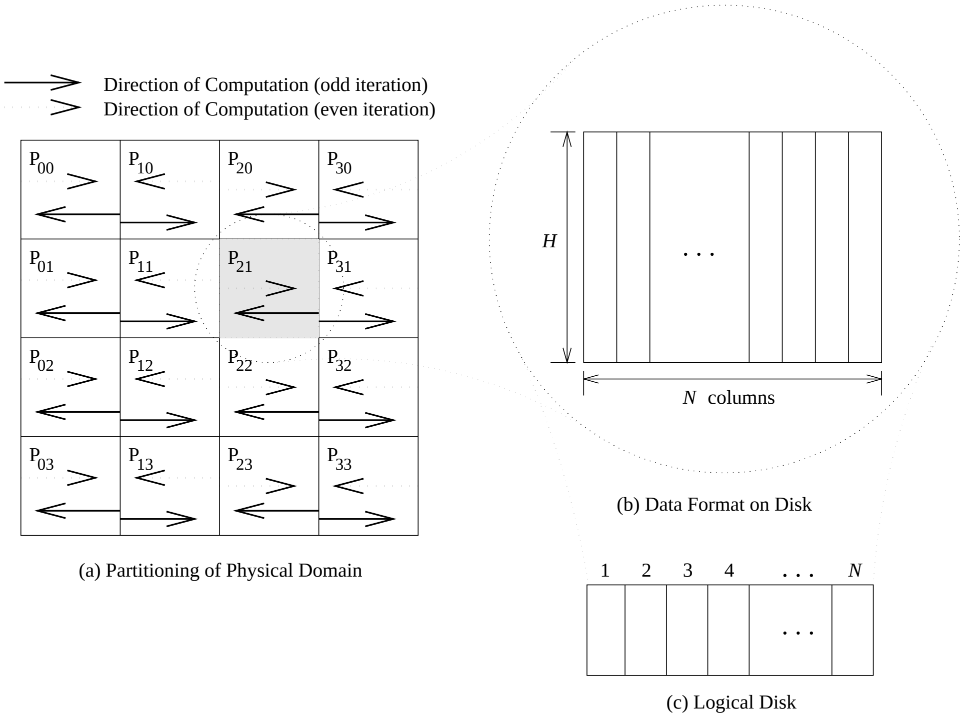


Figure 9.7: Data distribution for the Laplace equation solver

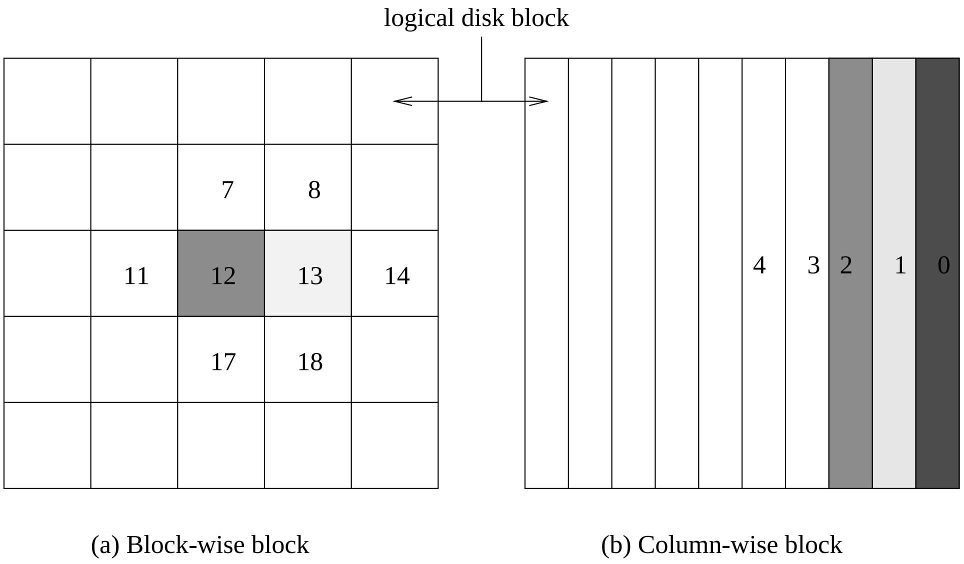


Figure 9.8: Data format on a physical disk

four neighbors. Hence, to calculate the new values at the four boundaries of the block, the values in blocks 7, 11, 13, and 17 are needed. That is, a total of five I/Os are needed to calculate the new value of each data element in block 12. After that, if the new value of each data element in a block of the neighbors of block 12 is calculated, only one block is reused and the other three blocks should be read.

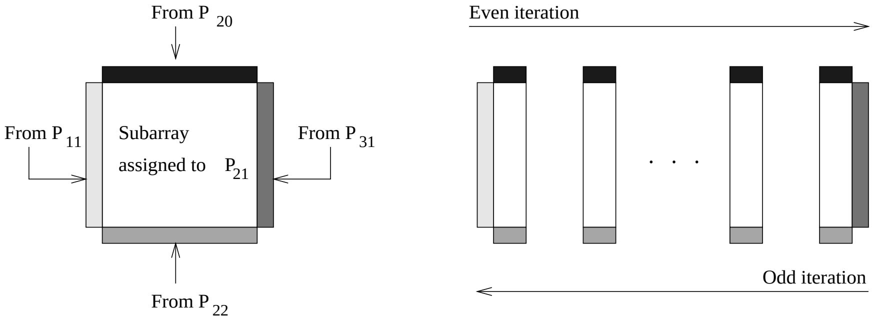
However, if a column is defined as a logical disk block, reading only three logical disk blocks is enough to calculate the new value of the data elements in one logical disk block.[7] That is, the computation for each LDB requires an LDB from the left and an LDB from the right. For example, to calculate the new value of each data element in block 1 (see Figure 9.8 (b)), only blocks 0 and 2 are needed. In order to calculate the new value of each data element in block 2, only one more logical disk block (block 3) should be read and another block can be reused without rereading that block. Therefore, we may get a better performance in terms of I/O if we define a logical disk block as a column-wise block.

9.2.2 Algorithmic Modification

Let's suppose that Figure 9.7 (b) represents the data format assigned to P_{21}. Where N is the number of columns, H is the height of the data array assigned to each compute node. Let's assume that each compute node has its own logical disk to store the data array and each column is defined as a logical disk block (see Figure 9.7 (c)).

Consider the subarray that is assigned to compute node P_{21}, shown in Figure 9.7. According to the Jacobi update procedure, the value of each data element on

[7]We assumed that the main memory of each compute node can contain at least three logical disk blocks.



(a) Overlap area (b) Procedure of I/O and collective communications

Figure 9.9: Procedure of I/O and collective communication with overlapped data

P_{21} is computed using the data elements of its corresponding four neighbors (see the codes in the previous section). Hence, to compute the values in an LDB (say LDB l), LDBs $l-1$ and $l+1$ are required; to compute the values at the four boundaries of the subarray, P_{21} needs the values in compute nodes P_{20}, P_{11}, P_{31}, and P_{22} (see Figure 9.9).

While computing the data elements in LDB l, each compute node has to store the newly computed temporary results in the main memory and then write the results back to the physical disk. Thus it seems that extra memory is required to store the temporary results shown in Figure 9.6. However, if we use the memory space for LDB $l-1$ to store the computation results for LDB l and use the memory as a circular queue, no extra memory is required.

In Figure 9.7, let's suppose that compute node P_{21} is being used to compute the first LDB. Then compute node P_{21} requires two LDBs (LDB 0 and 1) from the local LD, LDB $N-1$ from compute node P_{11}, and the bottom and top elements from compute nodes P_{20} and P_{22}, respectively. Since LDB $N-1$ from compute node P_{11} is just used to compute the first LDB, P_{21} can store the computation results for the first LDB in the memory space for LDB $N-1$. Thus the minimum required memory space, M, for each compute node to compute an LDB should contain $3H+2$ data

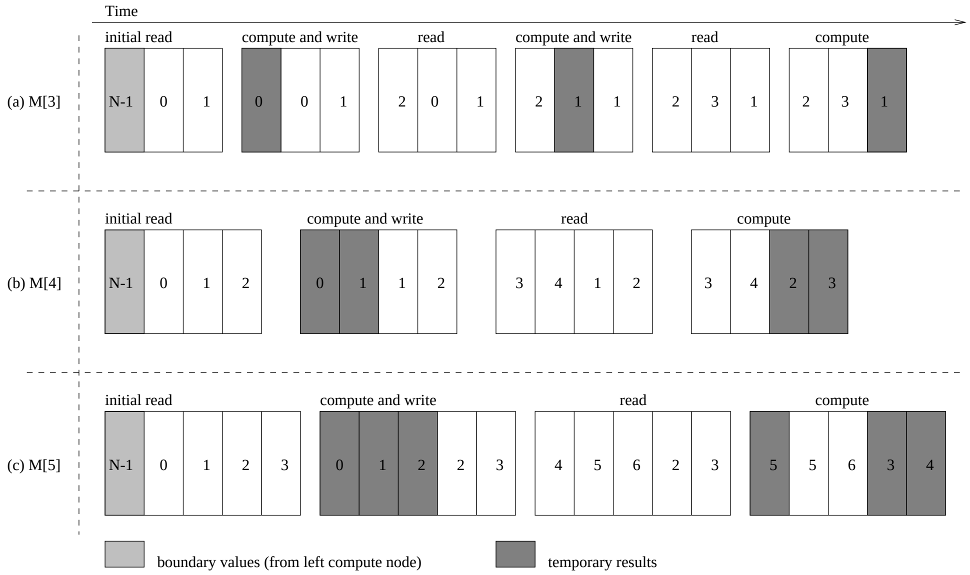


Figure 9.10: Memory usage for an external Laplace equation solver

elements, where $H = \frac{n}{P_Y}$.

Figure 9.10 shows how memory space is used while computing the data elements in LDBs. For simplicity, in this figure the memory space for the data elements from the top and bottom neighbor compute nodes are not represented. Let's suppose one element of M can contain an LDB in the figure. At the first reading, each compute node reads at least two LDBs (LDB 0 and 1) and stores them in $M[1]$ and $M[2]$ to compute LDB 0; LDB $N-1$ from the left neighbor compute node is stored in $M[0]$. Then each compute node can compute LDB 0 and store the results in $M[0]$ (see Figure 9.10 (a)). Newly computed LDB 0 should be stored on the physical disk to read the next LDB, because LDB 2 is required to compute LDB 1. At this time the old LDB 0 is in memory and can be used to compute LDB 1. The computation results of LDB 1 can be stored in $M[1]$ and written back to the physical disk to read LDB 3. When memory space M can contain four LDBs, this pattern of computations is repeated (see Figure 9.10 (b)).

If $M \geq 5H$, I/O operations are a bit different. Let's assume that $M = 5H$ (see

Figure 9.10 (c)). Each compute node will read four LDBs (LDB 0, 1, 2, and 3) at the first reading and store them in memory, $M[1]$ — $M[4]$. Of course, LDB $N-1$ from the left neighbor compute node is stored in $M[0]$, as in other cases. Three LDBs can be computed in each compute node, because all required data elements are in memory. Each compute node stores the computation results for LDB 0 in $M[0]$. The computation results for LDB 1 can be stored in $M[1]$ next, and the newly computed values of LDB 2 are stored in $M[2]$.

The newly computed LDBs ($M[0], M[2]$, and $M[3]$) should be stored on the physical disk to read the next three LDBs (LDB 4, 5, and 6). After reading the LDBs, computations for LDBs 3, 4, and 5 are performed; the computation results for LDBs 3 and 4 are stored in $M[3]$ and $M[4]$, and the computation results for LDB 5 are stored in $M[0]$. To write the results back to the physical disk, two I/O operations are required, because the computation results are stored in noncontiguous memory space. In UPIO, however, the results can be written back to the physical disk with a single I/O operation (see Chapter 4). So the same pattern of computations can be applied even if $M \geq 5H$.

In Figure 9.7, to compute the first LDB, compute node P_{21} requires some data elements from other compute nodes as described above (an LDB $N-1$ from compute node P_{11}, and the bottom and top elements from compute nodes P_{20} and P_{22}, respectively). If we assume that each compute node performs computations from the first LDB, in the SPMD programming model, the other compute nodes are also working on the first LDB. The bottom and top elements are in the memory of the bottom and top neighbor compute nodes, but LDB $N-1$, which will be computed in the last by compute node P_{11}, is not in memory. Thus compute node P_{11} needs to read the LDB and send it to compute node P_{21}. This is an extra I/O, because the LDB should be read again at the end of the iteration. Similarly, compute node P_{xy}, where $1 \leq x \leq P_X - 2$ and $0 \leq y \leq P_Y - 1$, needs to do an extra

I/O to read LDB $N-1$ and send it to the right neighbor compute node. At the end of the iteration, compute node P_{21} requires LDB 0 from compute node P_{31}, and thus P_{31} has to do an extra I/O to read the LDB and send it to P_{21}.

We can eliminate the extra I/Os by simply changing the direction of computation in every iteration. The arrows in Figure 9.7 represent the direction of computation. All compute nodes whose x coordinates are even numbers start computation from right (LDB $N-1$) to left (LDB 0), and other compute nodes whose x coordinates are odd numbers start computation from the opposite direction. In the next iteration, the direction of computation is reversed and will be reversed again after the iteration, and so on. This pattern of computation is possible because there is no dependency between the data in the `DO` execution (see Figure 9.6), and thus the order of LDB computations in each iteration is decided by the order in which the LDBs are read from the disk.

In Figure 9.9, compute node P_{21} begins computing from right to left and P_{31} begins computing from left to right. The compute nodes P_{21} and P_{31} read some logical disk blocks from the last logical disk block and first logical disk block, respectively. They contain boundary values so that the compute nodes can exchange boundary values before starting computation. Compute nodes P_{20} and P_{22} also read the last logical disk block so that they can exchange the boundary values with P_{21}. The compute node P_{11} begins computing from left to right. Therefore, the computations of P_{11} and P_{21} will meet at the boundary and can exchange boundary values. Individual compute nodes do not have to write back the results on the disk at the end of each iteration, because the results are used in the next iteration.

This pattern of computation is possible because the user has control over the user-defined logical disk. This pattern of computation requires no extra I/O operations caused by other compute nodes' requests to get boundary values, which optimizes the I/O operations. Figures 9.11 and 9.12 depict an external Laplace equation solver by the Jacobi iteration method to solve the Laplace equation for the data stored on a logical disk of each compute node.

9.2.3 Complexity

In the algorithms presented in Figures 9.11 and 9.12, we assumed that the $n \times n$ data array is decomposed on $P_X \times P_Y$ grid compute nodes in (BLOCK, BLOCK) manner and that each decomposed data, $H \times N (= \frac{n}{P_X} \times \frac{n}{P_Y})$, is stored using a 1-D LD, where a data column is mapped into an LDB. In this section we analyze the algorithms in terms of I/O and communications, because the complexity of computations is the same as that of an internal algorithm.

In the algorithms each compute node reads $W - 1 (\geq 2)$ LDBs at the beginning (lines 12 and 32), exchanges H elements with the right or left neighbor compute node and $W - 1$ elements with the top and bottom neighbor compute nodes (lines 13 and 33), and computes $W - 2$ logical disk blocks (lines 14 and 34). The computation results ($W - 2$ LDBs) are written back to the physical disk (lines 18 and 38). Thus the minimum required amount of memory for each compute node will be

$$M = WH + W - 1 \geq 3H + 2 = 3\frac{n}{P_Y} + 2, \text{ where } W \geq 3.$$

Reading, communication, computation, and writing LDBs of $W - 2$ will be repeated until the last LDB in each compute node is read (lines 17—26 and 37—46). At the end of each iteration, individual compute nodes do not store the computation results for the last $W - 1$ LDBs and use them in the next iteration. Instead, the results should be stored when the computations are completed, and the total number

```
// P: number of compute nodes (= P_X × P_Y).
// H × N array is assigned to P_xy, 0 ≤ x < X and 0 ≤ y < Y.
// P_xy stores the subarray through a 1-D logical disk L_p[N].
// H: the size of a logical disk block (= n/P_Y).
// M: memory size of P_xy, M ≥ (WH + W − 1) and W ≥ 3.

procedure EXTERNAL_LAPLACE_EQUATION_SOLVER
begin
    w = W − 2;    I = 0;
    while (1)
        if ((x %2 == 0  and  I %2 == 1)  or  (x %2 == 1  and  I %2 == 0))
            j = i = 0;   /* LDB index for reading and writing, respectively */
            if (I == 0) read L_p[j] ... L_p[j + w];
            exchange boundary values collectively if needed;
            compute w logical disk blocks;
            j = j + W − 1 ;
            limit = (N − (W − 1))/w;    more = N − (limit × w + W − 1);
            while  (limit − −)
                store the results to L_p[i] ... L_p[i + w − 1];
                read L_p[j] ... L_p[j + w − 1];
                j = j + w ;    i = i + w ;
                exchange boundary values collectively if needed;
                compute w logical disk blocks;
            store the results to L_p[i] ... L_B[i + more];
            read L_p[j] ... L_p[j + more − 1];
            exchange boundary values collectively if needed;
            compute (more + 1) logical disk blocks;
            if  (condition is satisfied) break;
            I + +;
```

Figure 9.11: External Laplace equation solver (continued)

```
        else if ((x %2 == 0 and I %2 == 0) or (x %2 == 1 and I %2 == 1))
            j = i = N - 1;
            if (I == 0) read L_p[j] ... L_p[j - w];
            exchange boundary values collectively if needed;
            compute w logical disk blocks;
            j = j - W + 1 ;   i = i - w ;
            limit = (N - (W - 1))/w;   more = N - (limit × w + W - 1);
            while (limit - -)
                store the results to L_p[i] ... L_p[i - w + 1];
                read L_p[j] ... L_p[j - w + 1];
                j = j - w ;   i = i - w ;
                exchange boundary values collectively if needed;
                compute w logical disk blocks;
            store the results to L_p[i] ... L_B[i - more];
            read L_p[j] ... L_p[j - more + 1];
            exchange boundary values collectively if needed;
            compute (more + 1) logical disk blocks;
            if (condition is satisfied) break;
            I + +;
    if ((x %2 == 0 and I %2 == 1) or (x %2 == 1 and I %2 == 0))
        i = i + (more + 1);
        store the results to L_p[i] ... L_p[N - 1];
    else
        i = i - (more + 1);
        store the results to L_p[i] ... L_p[0];
end
```

Figure 9.12: External Laplace equation solver (continued from the previous page)

of I/O operations will be

$$
\begin{aligned}
T_{io} &= 2\left(1 + I\left\lceil\frac{N-W+1}{W-2}\right\rceil\right) = 2\left(1 + I\left\lceil\frac{N+1-\frac{M+1}{(H+1)}}{\frac{M+1}{(H+1)}-2}\right\rceil\right)\\
&= 2\left(1 + I\left\lceil\frac{(H+1)(N+1)-(M+1)}{M-1-2H}\right\rceil\right)\\
&= 2\left(1 + I\left\lceil\frac{(n+P_X)(n+P_Y)-(M+1)P}{(M-1)P-2nP_X}\right\rceil\right), \qquad (9.4)
\end{aligned}
$$

where I is the number of iterations and $P = P_X \times P_Y$.

The algorithm is optimal in terms of the number of I/O operations, because each LDB is accessed only once for read and write. As shown in Figure 9.10, when $W \geq 5$, computation results can be stored in memory noncontiguously; more than one I/O operation is required to write the results back to the disk. In UPIO, however, the results can be stored on the disk with a single I/O operation.

Let's consider the communication cost. In every iteration to compute the first and last LDB, each compute node exchanges H elements with the right or left neighbor compute node. While computing whole LDBs (N LDBs) assigned to each compute node, each compute node exchanges N elements with the top and bottom neighbor compute nodes. Exchanging N elements with the top or bottom neighbor compute node requires $1 + \lceil\frac{N-(W-1)}{W-2}\rceil$. Thus the communication cost will be

$$
\begin{aligned}
T_c &= 2I\left((\tau + \mu H) + \left(\tau\left(1 + \left\lceil\frac{N-(W-1)}{W-2}\right\rceil\right) + \mu N\right)\right)\\
&= 2I\left(\tau\left(2 + \left\lceil\frac{N-(W-1)}{W-2}\right\rceil\right) + \mu(H+N)\right)\\
&= 2I\left(\tau\left(2 + \left\lceil\frac{n-P_X(W-1)}{P_X(W-2)}\right\rceil\right) + \mu(\frac{n}{P_Y} + \frac{n}{P_X})\right), \qquad (9.5)
\end{aligned}
$$

where I is the number of iterations and $W \geq 3$.

9.2.4 Experimental Results

In our experiments we configured the simulator with 16 compute nodes (4×4 grid) and 16 I/O nodes. Each node has a 25 MHz CPU and a direct-memory access (DMA) capability. The I/O node has a thirty-two 4-KB-buffer cache that uses an LRU-replacement, write-behind strategy and an HP97560 [85] disk that is attached to an SCSI bus whose peak bandwidth is 10 MB/s. The disk capacity and peak transfer rate are 1.3 GB and 2.34 MB/s, respectively. The file system block size is 4-KB.

In order to evaluate the performance of the external Laplace equation solver based on the algorithm described in the previous section, we implemented the algorithm with a logical disk and a traditional file system using file pointers on our simulator. (Hereafter, we call these programs the LD and FP versions, respectively.) In our experiments a $4K \times 4K$ array is decomposed on 4×4 grid compute nodes in (BLOCK, BLOCK) manner. Each compute node stores a $1K \times 1K$ subarray using a 1-D local logical disk in the LD version; the subarray is stored in a local file in the FP version. In the FP version the subarray is stored in column-major order and a column of the subarray is defined as an LDB in the LD version, where data is stored contiguously and randomly on the disk.

As we explained in Figure 9.10, computation results can be placed in noncontiguous memory space if the memory space can hold more than four columns of the subarray. In UPIO we can write the results back to the physical disk using multiple I/O operations or a single I/O operation with a memory vector (see Chapter 4). In the LD version we implemented both cases in order to explore the effects of the I/O operations. (Hereafter, we refer to the program using a memory vector as the LDmv version.)

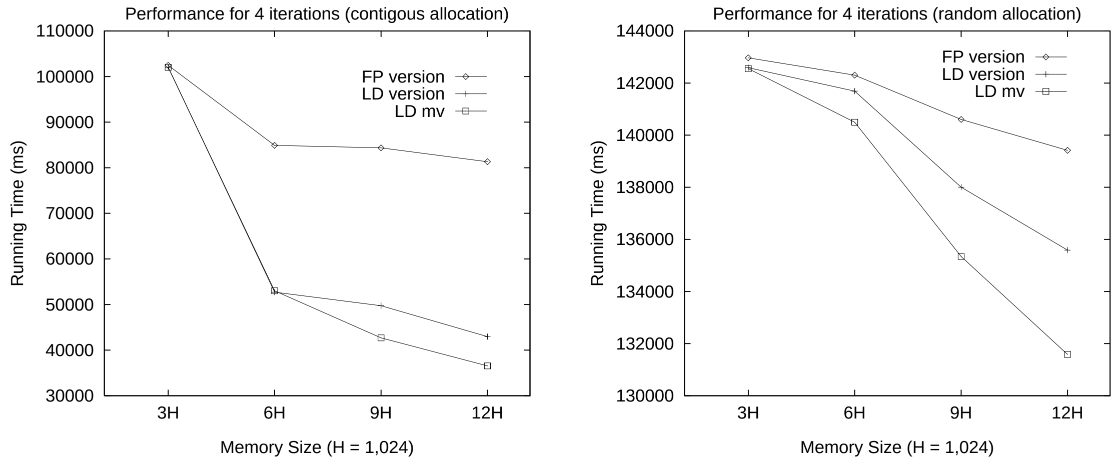


Figure 9.13: Performance of Laplace equation solver

Figure 9.13 compares the performances of the external Laplace equation solver using a file pointer and a logical disk, where the memory contains $W \times 1024 + W - 1$ elements and W is varied 3, 6, 9, and 12. In this figure we can see that the FP version performs the worst in contiguous and random disk-block allocations. This result can be explained as follows.

In the algorithm for the external Laplace equation solver, a submatix is forwardly and backwardly accessed in every other iteration repeatedly. Under a contiguous disk block allocation, disk blocks are accessed contiguously in both data access patterns. The direction of data access is the only difference between the data access patterns. However, from our experiments described in Chapter 5, we observed that the I/O backwardly accessed performance is worse than that for forwardly accessed.

When a subarray is forwardly accessed, multiple columns of the array are accessed at the same time in all program versions of the algorithm. If the subarray is backwardly accessed, only a single column of the array is accessed at one time in the FP version. Thus memory size does not much affect the performance of the FP version. However, in UPIO it is possible to access multiple LDBs form any direction;

requested LDB are translated into physical disk blocks and presorted to optimize disk-head movements in an I/O node. Thus the programs using a logical disk can access multiple columns of the subarray at one time, even data that is accessed backwardly; as the memory size increases, more columns can be accessed and the effect of backwardly accessing data decreases.

If $W = 3$, only one column of a subarray is accessed at a time in the main loops of the algorithm shown in Figure 9.11 and 9.12 (lines 17 and 37). So the performance of all program versions is almost the same. As the memory size increases, the performance of LD versions becomes better, because the effect of backwardly accessing data decreases. The performance of the LDmv version is better than that of the LD version, because more disk blocks can be accessed with a single I/O operation.

9.2.5 I/O Optimization

In the external Laplace equation solver algorithm presented in Figures 9.11 and 9.12, we can observe that each iteration consists of I/O phases, communication phases, and computation phases. In the main loops of the algorithm (lines 17 and 37), temporary result values of computations are written and then different LDBs are read for the next computations. In general, two I/O requests are needed to perform the read and write operations. However, it is not necessary to perform the operations separately. Atomic I/O operations of UPIO allow applications to perform the operations collectively with a single I/O operation. Thus the number of I/O operations could be

$$T_{io} = 1 + I \left\lceil \frac{(n + P_X)(n + P_Y) - (M + 1)P}{(M - 1)P - 2nP_X} \right\rceil, \qquad (9.6)$$

where I is the number of iterations and $P = P_X \times P_Y$ (see Equation 9.4). The reduced number of I/O operations will reduce the I/O latency and increase I/O performance.

We implemented the algorithm using atomic I/O operations of UPIO (hereafter, referred to as the LDatomic version). Table 9.4 compares the performance of the LDmv version and LDatomic version. The performance of the LDatomic version is better than that of the LDmv version in both disk block allocations. The performance gap between the two versions becomes closer as the memory size increases, because the number of I/O operations decreases. However, in our experiments we fixed the number of iterations at four, so as the number of iterations increases, the gap will be larger.

Table 9.4: Comparison of performance for 4 iterations (milliseconds)

Block allocation	Version	W = 3	W = 6	W = 9	W = 12
Contiguous	LDmv	102020.61	53015.83	42689.32	36555.48
	LDatomic	101590.30	52912.33	42631.62	36514.01
Random	LDmv	142555.47	140497.37	135343.06	131586.32
	LDatomic	142122.00	140393.73	135284.78	131545.46

9.2.6 Discussion

In this section we present an I/O and communication efficient external Laplace equation solver algorithm. We reduced the number of I/O operations by reusing data already brought into memory at the end of each iteration and by using a memory vector of UPIO, which is an array of (memory offset, number of LDBs) that makes it possible to read or store data into or from noncontiguous memory space. By explicitly scheduling the direction of computations, it is possible for compute nodes to communicate each other collectively, thereby eliminating extra I/Os caused by communications. This algorithm is also memory efficient, because no extra memory is required to store temporary computation results.

The direction of computation forces compute nodes to repeatedly access data forwardly and backwardly in every other iteration. From our experiments described in Chapter 5, we observed that the I/O performance for backward is worse than that for forward. In UPIO it is possible to access multiple LDBs in any direction; requested LDB are translated into physical disk blocks and presorted to optimize disk-head movements in an I/O node. This reduces the effect of backwardly accessing data and, in fact, our simulations showed that the performance of LD versions became better as the memory size increased. On the other hand, the FP version can only access multiple columns of the subarray when the subarray is forwardly accessed. If the subarray is backwardly accessed, only a single column of the array is accessed at a time in the FP version, thus memory size does not much affect the performance of the FP version.

In our experiments we also showed that we can reduce the number of I/O operations and increase performance by combining write and read operations into an atomic I/O operation. Since, in atomic I/O operations, blocking and nonblocking I/O operations can be combined, it may be possible to improve performance by overlapping I/O and computations.

Chapter 10

Conclusions and Future Work

The primary objective of this dissertation study is to support the necessary I/O requirements to maximize I/O performance for scientific applications on MIMD multicomputers. To meet this objective, we focused our research on two important I/O requirements: a flexible file model and user controllability. For our purposes we designed UPIO, of which the *logical disk* is an important part.

We have showed how a logical disk can extend the abstraction of a linear file model into an n-dimensional file model and give programmers total control over file data. A logical disk corresponds to a file of conventional file systems; however, it is just an abstract block-index domain defined by programmers. Therefore, programmers can define file structures and represent the semantics of data domain by defining the shape of a logical disk and by mapping the basic structure of computation into a logical disk block. A logical disk does not allocate space for itself, neither in memory nor on a physical disk. Therefore, it allows programmers to explicitly control the way of distributing data across I/O subsystems, aggregating disk bandwidth by considering the basic computational structure of applications.

UPIO supports two different models of data storage and access based on logical disks for which there are separate interfaces that can be used together in an application. The results of our experiments showed that a variety of UPIO's interfaces deliver a higher I/O performance by allowing programmers to represent complex data-access patterns collectively with a minimum number of I/O operations, which provides enough information to optimize disk-head movements. UPIO's data-access interfaces do not use a file pointer to access data in a file, which eliminates the overhead of managing the offset of the file and makes it easier to represent multiple small requests as one request. The results also showed the importance of user-controllability over distributing data across disks and in defining file structure by considering access patterns.

Collectively represented I/O requests in each compute node can be served separately or together. To serve the I/O requests collectively, compute nodes cooperate and combine the requests into a larger request. UPIO interfaces allow programmers to specify how the system serves I/O requests according to access patterns, which eliminates combining the overhead for requests that should be served separately. Our experimental results showed that it is possible to improve performance by serving requests collectively in an efficient way that overlaps permutations, data transfers and disk accesses, reduces network traffic, and then increases message-passing bandwidth utilization.

We also evaluated UPIO with two applications, for which we designed new external algorithms. The new algorithms are based on internal algorithms that are efficient in terms of computations, communications, or memory usage. We have shown how UPIO produces high-performance external computation codes by modifying the algorithms to be I/O efficient using UPIO's features. We emphasized in the design of the algorithms that there is no difference between internal and external computations except for data size; we showed that the new external algorithms are efficient in terms of I/O, computation, and communication. Our design and analysis of the algorithms imply that a logical disk can be the base of a computational model for designing and analyzing new external algorithms.

The heart of a large class of scientific and engineering applications is an iterative algorithm. Each iteration in the algorithm generally consists of computation phases, communication phases, and I/O phases. In each iteration, temporary values must be read for computation and the result of the computation should be written repeatedly. UPIO supports atomic I/O operations that can do read and write together. Our experiments with the applications showed that it is possible to reduce I/O time by doing read and write operations collectively with a single I/O operation. Our experimental results with the applications also emphasize the importance of a flexible file model and user controllability in I/O.

While we are satisfied with most of the current work, there are still several interesting work and areas that can be extended. Implementing UPIO on real parallel machines can be most interesting, especially since most of our current work has been done on a simulator.

In this study, we limited the size and shape of a logical disk block should be the same as the size and shape of a logical disk. However, for more flexibility, UPIO should support various types of logical disk blocks and different sizes of logical disk blocks in a logical disk. Exploring the effects of various types and sizes of logical disk blocks is an area that should be extended.

During the course of developing UPIO we focused our research on UPIO's interface and minimized our efforts to design other components of UPIO, because we believe that we can use the software of other I/O subsystems for the other components after mapping LDBs into physical disk blocks. However, other components may affect the system's performance. For example, we noticed that our cache could be thrashed. Therefore, designing other components for UPIO could be interesting work in the future.

With UPIO, programmers can easily create files using logical disks and have total control over the files when accessing and distributing data. However, since the files created using logical disks have a unique nature, we must provide some ways to migrate files between UPIO and an external file system in order to make UPIO useful in a real system.

The von Neumann model is a successful bridge between software and hardware in sequential computation. However, there is no standard model of parallel computation, although the PRAM (Parallel Random Access Machine) [47] model is most widely used. In it all processors are assumed to be connected directly to a global memory and work synchronously. This assumption, however, is not realistic for reducing the gap between parallel software and hardware. Several other models, such as the BSP (Bulk-Synchronous Parallel) model [66] and the LogP model [18], have been suggested in order to reduce the gap.

All models focused on internal computation rather than external computation, and only a few machine models with attached parallel disk systems have appeared for external computation. Vitter and Shriver designed deterministic algorithms for matrix multiplication, FFT, and sorting based on their parallel disk model [89, 90]. In this model it is assumed that a single I/O operation can access all disks simultaneously and that each disk transfers only one block. This assumption, however, is appropriate only in some specific cases, because I/O nodes in the usual parallel processing systems work asynchronously and can access more than one block at a time. In addition, the internal memory is shared by all processors, unlike with real machines, and communication effects between processors are not considered.

We believe that the logical disk could be the base of a computational model for designing new external algorithms or for modifying efficient internal algorithms for external computations, because the basic computation structure can be represented in a logical disk block.

Bibliography

[1] Maurice J. Bach. *The Design of The Unix Operating System.* Prentice Hall, 1986.

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

[3] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 331–338, 1988.

[4] Rajesh Bordawekar, Juan Miguel del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, pages 452–461, 1993.

[5] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. Proteus: A high-performance parallel architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[6] Peter Brezany, Michael Gernt, Piyush Mehotra, and Hans Zima. Concurrent file operations in a High Performance FORTRAN. In *Proceedings of Supercomputing '92*, pages 230–237, 1992.

[7] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[8] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. MPI-IO: a parallel file I/O interface for MPI. Technical Report RC 19841 (87784), IBM T.J. Watson Research Center, November 1994. Version 0.2.

[9] Peter Corbett, Dror Feitelson, Yarson Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson 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.

[10] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993. Also published in *Computer Architecture News* 21(5), December 1993, pages 7–14.

[11] Peter F. Corbett and Dror G. Feitelson. Design and implementation of the Vesta parallel file system. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 63–70, 1994.

[12] Peter F. Corbett and Dror G. Feitelson. Vesta file system programmer's reference. Technical Report Research Report RC 19898 (88058), IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, October 1994. Version 1.01.

[13] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Sandra Johnson Baylor. Parallel access to files in the Vesta file system. In *Proceedings of Supercomputing '93*, pages 472–481, 1993.

[14] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, and Marc Snir. User-friendly and efficient parallel I/Os using the Vesta parallel file system. In

Transputers '94: Advanced Research and Industrial Applications, pages 23–38. IOS Press, September 1994.

[15] Thomas H. Cormen and David Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised as Dartmouth PCS-TR93-188 on 9/20/94.

[16] Parasoft Corp. *Express Version 1.0: A Communication Environment for Parallel Computers*, 1988.

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

[18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proc. of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[19] William J. Dally and Chuck L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[20] Erik DeBenedictis and Juan Miguel del Rosario. nCUBE parallel I/O software. In *Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications*, pages 0117–0124, April 1992.

[21] Juan Miguel del Rosario. A guide to striped files and parallel I/O in ncube system software, release 3.1. Technical Report nCUBE-TR002-920713, nCUBE, July 1992.

[22] Juan Miguel 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.

[23] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok 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.

[24] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improving parallel I/O performance using a two-phase access strategy. Technical Report SCCS–406, NPAC at Syracuse University, 1993.

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

[26] Peter C. Dibble and Michael L. Scott. Beyond striping: The Bridge multiprocessor file system. *Computer Architecture News*, 19(5), September 1989.

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

[28] Rüdiger Esser and Renate Knecht. Intel Paragon XP/S — architecture and software environment. Technical Report KFA-ZAM-IB-9305, Central Institute for Applied Mathematics, Research Center Jülich, Germany, `r.esser@kfa-juelich.de`, April 26 1993.

[29] Dror G. Feitelson, Peter F. Corbett, Yarson Hsu, and Jean-Pierre Prost. Parallel I/O systems and interfaces for parallel computers. In *Multiprocessor Systems — Design and Integration*. World Scientific, 1995. To appear.

[30] R. Floyd and C. Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge Engin.*, 1, June 1989.

[31] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, CRPC, Rice University, January 1993.

[32] Message Passing Interface Forum. Mpi: A message passing interface standard. Technical report, May 1994.

[33] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Uli Kremer, and Chan-Wen Tseng. Fortran D language specification. Technical Report Rice COMP TR90-141, Rice University, December 1990.

[34] Craig S. Freedman, Josef Burger, and David J. Dewitt. SPIFFI — a scalable parallel file system for the Intel Paragon. *IEEE Transactions on Parallel and Distributed Systems*, 1995. To appear.

[35] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

[36] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of the Concurrent File System of the Intel iPSC/2 hypercube. *Journal of Parallel and Distributed Computing*, 17(1–2):115–121, January and February 1993.

[37] N. Galbreath, W. Gropp, and D. Levine. Applications-driven parallel I/O. In *Proceedings of Supercomputing '93*, pages 462–471, 1993.

[38] Hector Garcia-Molina and Kenneth Salem. The impact of disk striping on reliability. *IEEE Database Engineering Bulletin*, 11(1):26–39, March 1988.

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

[40] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David 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.

[41] James 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.

[42] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[43] IBM. *IBM AIX Version 3.2 General Programming Concepts, 12th edition*, October 1994.

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

[45] Concurrent I/O application examples. Intel Corporation Background Information, 1989.

[46] Paragon XP/S product overview. Intel Corporation, 1991.

[47] Josepj JaJa. *An Introduction to Parallel Programming*. Addison Wesley, 1992.

[48] David Wayne Jensen. *Disk I/O In High-Performance Computing Systems*. PhD thesis, Univ. Illinois, Urbana-Champagne, 1993.

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

[50] Michelle Y. Kim and Asser N. Tantawi. Asynchronous disk interleaving: Approximating access delays. *IEEE Transactions on Computers*, 40(7):801–810, July 1991.

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

[52] David Kotz. Multiprocessor file system interfaces. Technical Report PCS-TR92-179, Dept. of Math and Computer Science, Dartmouth College, March 1992. Revised version appeared in PDIS'93.

[53] David Kotz. Disk-directed I/O for MIMD multiprocessors. Technical Report PCS-TR94-226, Dept. of Computer Science, Dartmouth College, July 1994. Revised November 8, 1994.

[54] David Kotz. Disk-directed I/O for MIMD multiprocessors. Submitted to TOCS, January 1995.

[55] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[56] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. In *1991 IEEE Symposium on Parallel and Distributed Processing*, pages 60–67, December 1991.

[57] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

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

[59] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. Technical Report Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.

[60] Orran Krieger. *HFS: A flexible file system for shared-memory multiprocessors*. PhD thesis, University of Toronto, October 1994.

[61] Orran Krieger and Michael Stumm. HFS: a flexible file system for large-scale multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6–14, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[62] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypsis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, 1994.

[63] Jang Sun Lee, Jungmin Kim, P. Bruce Berra, and Sanjay Ranka. Logical disks: User-controllable I/O for scientific applications. In *Proceedings of the 1996 IEEE Symposium on Parallel and Distributed Processing*, pages 340–347, 1996.

[64] Jang Sun Lee, Sang-Gue Oh, P. Bruce Berra, and Sanjay Ranka. User-controllable I/O for parallel computers. In *Proceedings of Parallel and Distributed Processing Techniques and Applications*, pages 442–453, 1996.

[65] C. Leiserson and et al. The network architecture of the connection machine cm-5. In *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[66] L.G.Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[67] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 69–77, May 1987.

[68] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[69] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, 1991.

[70] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, Feb. 1993.

[71] Nils Nieuwejaar and David Kotz. Low-level interfaces for high-level parallel I/O. In *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 47–62, April 1995.

[72] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[73] Nils Nieuwejaar and David Kotz. Performance of the Galley parallel file system. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 83–94, Philadelphia, May 1996. ACM Press.

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

[75] John K. Ousterhout, H. DaCosta, D. Harrison, and et al. A trace driven analysis of the unix 4.2bsd file system. In *Proceedings of 10th Symposium on Operating System Principles*, pages 15–24, 1985.

[76] Barbara K. Pasquale and George C. Polyzos. Dynamic I/O characterization of I/O intensive scientific applications. In *Proceedings of Supercomputing '94*, pages 660–669, November 1994.

[77] David Patterson, Peter Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). In *Proceedings of IEEE Compcon*, pages 112–117, Spring 1989.

[78] Paul 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.

[79] Alan Poston. A high performance file system for UNIX. In *Proceedings of the USENIX Workshop on UNIX and Supercomputers*, pages 215–226, 1988.

[80] Terrence W. Pratt, James C. French, Phillip M. Dickens, and Stanley 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.

[81] Apratim Purakayastha, Carla Schlatter Ellis, and David Kotz. ENWRICH: a compute-processor write caching scheme for parallel file systems. In *Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 55–68, May 1996.

[82] Apratim Purakayastha, Carla Schlatter Ellis, David Kotz, Nils Nieuwejaar, and Michael 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.

[83] James Rothnie. Kendall Square Research: introduction to the KSR1. In *Proceedings of the 1992 DAGS/PC Symposium*, pages 200–210, Hanover, NH, June 23–27 1992. Dartmouth Institute for Advanced Graduate Studies.

[84] Paul J. Roy. Unix file access and caching in a multicomputer environment. In *Proceedings of the Usenix Mach III Symposium*, pages 21–37, 1993.

[85] Chris Ruemmler and John Wilkes. Modelling disks. Technical Report HPL-93-68, revision 1, HP Lab., July 1993.

[86] S. Saini and H. Simon. Enhancing applications performance on intel paragon through dynamic memory allocation. In *Proceedings of the Scalable Parallel Libraries Conference*, Oct 1993.

[87] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *Proceedings of the IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[88] S. C. Goldstein T. Eicken, D. E. Culler and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of ISCA '92*, pages 256–266, 1992.

[89] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.

[90] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, August and September 1994.

[91] David Womble, David Greenberg, Rolf Riesen, and Stephen Wheat. Out of core, out of mind: Practical parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 10–16, Mississippi State University, October 1993.

[92] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer I/O practical. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.