I/O Optimizations for Hierarchical Storage Systems

Gokhan Memik

Technical Report No. CPDC-TR-2000-05-004

May 2000

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

I/O OPTIMIZATIONS FOR HIERARCHICAL STORAGE SYSTEMS

By

Gokhan Memik

A Thesis Submitted to the Graduate

Faculty of Northwestern University

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Approved:

Professor Alok Choudhary
Thesis Adviser

Northwestern University
Evanston, Illinois

May 2000

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Over the last decade, processors have made enormous gains in speed. But increase in the speed of the secondary and tertiary storage devices could not cope with these gains. The result is that the secondary and tertiary storage access times dominate execution time of data intensive computations. Therefore, efficient data access functionality for data stored in secondary and tertiary storage is a must. Hierarchical storage systems (HSS) like HPSS are becoming increasingly popular due to their large storage capabilities and high data transfer rates. In this work, we have devised APRIL, a parallel runtime library, that provides an easy-to-use interface for the data residing on a hierarchical storage systems. The library uses three different optimizations for improving the response times for several I/O patterns: Sub-filing, multiple collective I/O, and selective collective I/O. In *sub-filing*, a large multi-dimensional data residing on the lower levels of the HSS is stored not as a single file but as a number of smaller sub-files. *Multiple Collective I/O* is used to optimize the accesses to several files. *Selective Collective I/O* uses the collective I/O calls selectively, thereby increasing the performance when the synchronization inherent in collective I/O is not needed. In order to evaluate the performance of the library, we have tested it against several access patterns. The experiments show that, these optimizations can reduce the I/O time by an order of magnitude in several cases.

1. Introduction

In this work, we address the problem of managing the movement of very large data sets between different levels of a hierarchical storage system and how to optimize the data accesses. It is now widely acknowledged that the data set sizes manipulated by all applications are getting larger as programmers have access to faster processors and larger main memories. The data sets whose sizes exceed main memories should be stored in secondary and tertiary storages. Although the prices for secondary storage devices are decreasing, tertiary storage devices are still an attractive solution for very large datasets. Hierarchical storage systems, on the other hand, try to combine the advantages of both the storage mechanisms by building a system that consists of both the secondary and tertiary storages. Such systems are becoming increasingly attractive especially for applications that require vast amount of storage capacity which cannot be satisfied by secondary storage devices and for applications which cannot afford the cost or system complexity of a large number of disk drives. Figure 1.1 shows the the massive computing and data storage requirements for the Accelerated Strategic Computing Initiative ("ASCI") project. The figure reveals that the increase in the requirements are higher then the increase in the secondary storages capacities and the transfer rates. One of the attractive solutions to reduce the gap is the use of hierarchical storage systems.

We are introducing a parallel run-time library, which is used to access the data residing on a hierarchical storage system. The library allows users to access the data by an easy-to-use interface. Besides providing a convenient user interface, the library uses three optimizations to reduce the response times of I/O calls: *subfiling*, *multiple collective I/O*, and *selective collective I/O*. These optimizations are discussed in detail in the following sections. In this section, we first give an overview of a popular hierarchical storage system, HPSS. Then, we give information about

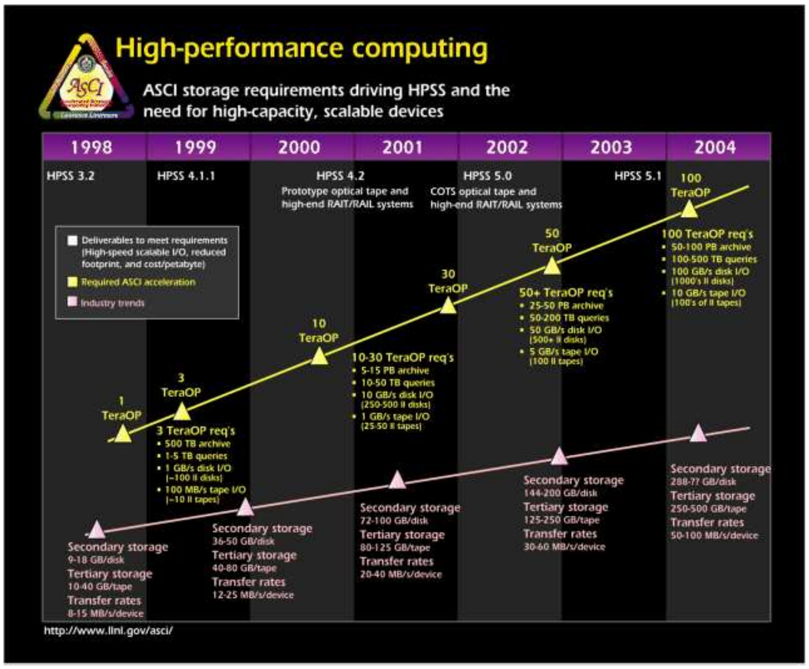

Figure 1.1: Computing and data storage requirements for the Accelerated Strategic Computing Initiative ("ASCI") project along with the secondary storage capacities and transfer rates.

collective I/O. This project is a part of a bigger project called, High-Performance Data Management, Access, and Storage Techniques for Tera-scale Scientific Applications. Subsection 1.3 gives on overview of this project. In the last subsection, we give the outline of the thesis.

1.1 High Performance Storage System

HPSS is a major development project, which began in 1993 as a Cooperative Research and Development Agreement (CRADA) between government and industry, and is now progressing towards its fifth major release. It is a hierarchical storage system software designed to manage and access 100's of terabytes to petabytes of data.

HPSS provides a scalable parallel storage system for highly parallel computers as well as traditional supercomputers and workstation clusters. HPSS requirements

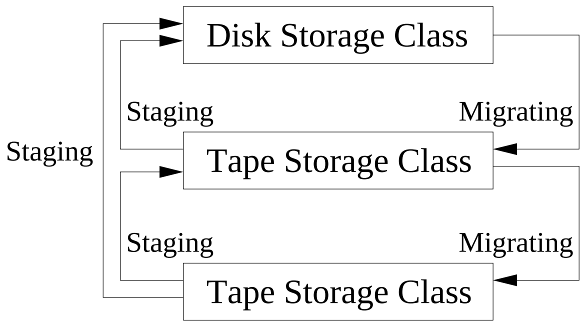

Figure 1.2: An overview of the migrate/stage mechanisms in HPSS.

are driven by high performance computing environments, such as the National Information Infrastructure (NII), in which large amounts of data are generated by massively parallel processors (MPPs) and workstation clusters. Scalability is in several dimensions: data transfer rate, storage size, number of name space objects, size of objects, and geographical distribution. Although developed to scale for order of magnitude improvements, HPSS is a general purpose storage system [45].

It provides the user with a number of transfer interfaces: File Transfer Protocol (ftp), Network File System (NFS) V2, Parallel FTP (PFTP), Client Application Program Interface (Client API), Non-DCE Client Application Program Interface (Non-DCE Client API), MPI-IO Application Programming Interface (MPI-IO API), Distributed File System (DFS).

One of the important aspects of HPSS is the ability to move the data between the levels of hierarchy in the system. This is achieved by the Migration-Purge server. The basic working mechanism is given in Figure 1.2. There are two types of migration, disk migration and tape migration. The main purpose of disk migration is to free up disk storage. The main purpose of tape migration is to free up tape volumes, and not just migrate bitfiles.

1.2 Collective I/O

Collective I/O is a technique, where individual I/O requests are combined to create a single, big I/O request and send to the storage system. Thereby the bandwidth is significantly increased. One of the most important optimizations in MPI-IO [11], an emerging parallel I/O standard, is *collective I/O,* an optimization that allows each processor to do I/O on behalf of other processors if doing so improves the overall performance [9]. This optimization has many variants [28, 26, 38]; the one used in this study is *two-phase I/O* as implemented in ROMIO, a portable implementation of MPI-IO from Argonne National Laboratories [41]. ROMIO has been incorporated into several MPI-libraries, including the MPI implementations of several vendors (e.g., HP, SGI, NEC) and MPICH and LAM, two widely used, freely available, portable MPI implementations. In this implementation, I/O is performed in *two phases*: an I/O phase and a communication phase. In the I/O phase, processors perform I/O in a way that is most beneficial from the storage layout point of view. In the second phase, they engage in a many-to-many type of communication to ensure that each piece of data arrives in its final destination. Previous experiences with collective I/O and its variants show impressive improvements in I/O latencies, and that the extra time spent in performing inter-processor communication is, in most cases, negligible. While collective I/O and its variants are very beneficial if used properly, almost all previous studies considered a *user-oriented* approach in applying collective I/O. For example, Thakur et al suggest programmers to use collective I/O interfaces of MPI-IO instead of easy-to-use Unix-like interfaces [42]. Apart from determining the most suitable collective I/O routine and its corresponding parameters, this also requires, on the user part, analyzing access patterns of her code, detecting parallel I/O opportunities, and finally deciding a parallel I/O strategy.

1.3 Big Picture

The work in this thesis is a part of a bigger project, High-Performance Data Management, Access, and Storage Techniques for Tera-scale Scientific Applications. In this section, we give an overview of this project. The goals in this project determines the goals in this thesis. That is why it is important to get an overview of the project.

In High-Performance Data Management, Access, and Storage Techniques for Tera-scale Scientific Applications project, the objective is to develop a scalable high-performance data management system that will provide support for data management, query capability and high-performance accesses to large datasets stored in hierarchical storage systems (HSS)[44].

The approach tries to combine the advantages of file systems and databases, while avoiding their respective disadvantages. It provides a user-friendly programming environment which allows easy application development, code reuse, and portability; at the same time, it extracts high I/O performance from the underlying parallel I/O architecture by employing advanced I/O optimization techniques like data sieving and collective I/O. It achieves these goals by using an active meta-data management system (MDMS) that interacts with the parallel application in question as well as with the underlying hierarchical storage environment. The proposed programming environment has three key components:

- User applications;
- Meta-data management system; and
- Hierarchical storage system.

These three components can exist in the same site or can be fully distributed across distant sites. For example, as part of our experiments we run a parallel volume rendering application on the SP-2 at Argonne National Lab that interacts with the MDMS located at Northwestern University and accesses its data files (currently) using TCP/IP stored on the HPSS (High Performance Storage System) installed at San Diego Supercomputer Center (SDSC). The experimental configuration is depicted in Figure 1.3.

Normally, application programmers have to deal with the storage systems directly, shown as the two ovals, User Applications and Storage System, in Figure 1.3. Programmers are required to understand the interfaces of storage systems in which the files reside and the associated meta data that describes the internal structures of those files, such as file locations, names of datasets in each file, data structures, types, storage patterns, etc. The information is most likely written in separate text-format files, e.g. README, by the authors who created those data files. Sometimes, programmers even have to inquire additional information from the authors. Traditionally, these work can only be done by the users prior to programming their applications.

In this project, we add the third oval, a meta-data management system, into the figure which acts as an active middle-ware between users' applications and storage systems. This meta-data management system provides users a uniform interface to access resource and information in a heterogeneous computing environment. More information about the meta-data management system and details of the project can be found in [39, 10].

1.4 Outline

The rest of this thesis is organized as follows. In the next chapter, we explain the previous work on the subject. Then, we introduce the APRIL library. We first give an overview, and then explain the interface. In Chapter 4 we explain the sub-filing strategy. In Chapter 5 we describe the multiple collective I/O in detail. After this, we explain the selective collective I/O. Finally, we conclude in the conclusion chapter and discuss the future work.

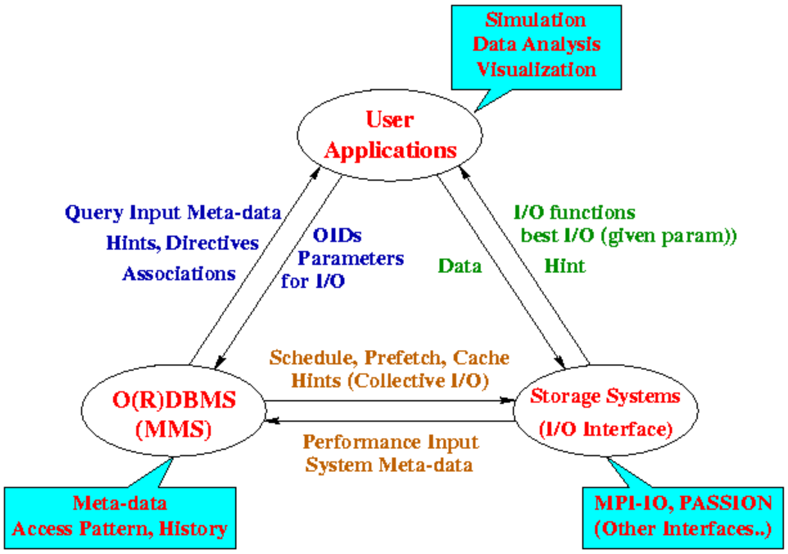

Figure 1.3: An overview of the MDMS system: Three-tiered architecture. Each site can be located distributedly and connect with each other remotely through TCP/IP.

2. Previous Work

It is been widely acknowledged that the increase for the requirements of storage capacities and transfer rates are higher then the increase in the storage capacities and transfer rates. To address this imbalance, hardware designers focus on improving parallel I/O capabilities using multiple disks, I/O processors, and large bandwidth I/O busses [34]. It is generally agreed upon that an optimized I/O software can also play a major role in bridging this performance gap. For example, parallel file systems use several optimization techniques such as data striping, caching, and prefetching to reduce/hide I/O latencies [35, 12, 27]. In order to eliminate the difficulty in using a parallel file system directly, several research groups proposed high-level parallel I/O libraries and runtime systems that allow programmers to express access patterns of their codes using program-level data structures such as rectilinear array regions [9, 38, 11]. While all these software supports provide an invaluable help to boost the I/O performance in parallel architectures, it remains still programmer's responsibility to select appropriate I/O calls to use, to insert these calls in appropriate locations within the code, and to manage the data flow between parallel processors and parallel disks.

Efficient storage organization for multidimensional data has been investigated extensively [36, 8, 23]. Chen et al [8] prove that the problem of efficient organization of multidimensional data on a one-dimensional storage system is NP-complete when arbitrary range queries are allowed. Jagadish et al [23] investigated the problem of efficient organization of a data warehouse on secondary storage.

There has been a considerable amount of work in addressing the flow of data to and from secondary storage devices (e.g., magnetic disks) [6, 9, 11, 34, 16, 29, 31, 32, 5]. There has also been a significant amount of work on the management of large scale data in a storage hierarchy involving tertiary storage devices (e.g., tapes devices) [37, 22, 24, 25, 30]. Striping has been studied to improve the response time of tertiary storage devices [15, 19]. Also, using libraries on top of hierarchical storage systems is not a new concept. Watson et al [46] implemented a MPI-IO version in the HPSS.

3. APRIL

In this section, we give an overview of the APRIL, a parallel run-time library for data manipulation. After giving the overview, we explain the implementation details. Then we discuss the user interface.

3.1 Overview

The library provides routines to efficiently perform I/O required in sequential and parallel applications. It can be used for both in-core and out-of-core applications. It uses a high-level interface which can be used by application programmers and compilers. For example, an application programmer can specify what section of an array she wants to read in terms of lower and upper bounds in each dimension, and the library will fetch it in an efficient manner, first from tape to disk and then from disk to main memory. It provides a portable interface on top of HPSS [14] and MPI-IO [11]. It can also be used by an optimizing compiler that targets programs whose data sets require transfers between secondary storage and tertiary storage. It might even be possible to employ the library within a database management system for multi-dimensional data.

At the heart of the library is the optimization techniques, *sub-filing*, *multiple collective I/O*, and *selective collective I/O*, which is explained in greater detail in the next sections. It also uses collective I/O using a two-phase method, data pre-staging, pre-fetching, and data migration. The main advantage of sub-filing is that it provides low-overhead random access image for the tape-resident data. Sub-filing is invisible to the user and helps to efficiently manage the storage hierarchy which can consist of a tape sub-system, a disk sub-system and a main memory. The main advantage of the collective I/O techniques, on the other hand, is that it results in high-granularity data transfers between processors and disks, and it also makes use of the higher bandwidth of the processor interconnection network.

In general, a processor has to wait while a requested data set is being read from the system. The time taken by the program can be reduced if the computation and tape I/O can be overlapped somehow. The pre-staging achieves this by bringing the required data ahead of the time it will be used. It issues asynchronous read calls to the tape sub-system, which help to overlap the reading of the next data portion with the computation being performed on the current data set. The data pre-fetching is similar except that it overlaps the disk I/O time with the computation time.

3.2 Implementation Details

We have implemented the APRIL library on top of HPSS [14] and MPI-IO [11]. The connections between different components are shown in Figure 3.1(a). In a read call, the sub-files are first read from tape to disk using HPSS and then from disk to main memory using MPI-IO. As mentioned earlier, we employ collective I/O between disk and memory. In a write call the direction of data-flow is reversed. Figure 3.1(b) shows the corresponding storage levels for each action described in the following sections.

To store the information about the file, we are currently using the Postgres95 database [47]. Since we are using a sub-filing strategy, the information about the file file can grow very fast. Therefore, we employed a database system to store the required information. When a new file is created, the user may enter the necessary information about the chunks. The detailed information about the file creation is given in Section 3.3. Then this meta-data is stored in the database for later usage. When a user opens a previously created file, the corresponding meta-data about the file and the chunks are read from the database and cached in the memory. Then the following accesses uses this meta-data. The database is informed about the changes

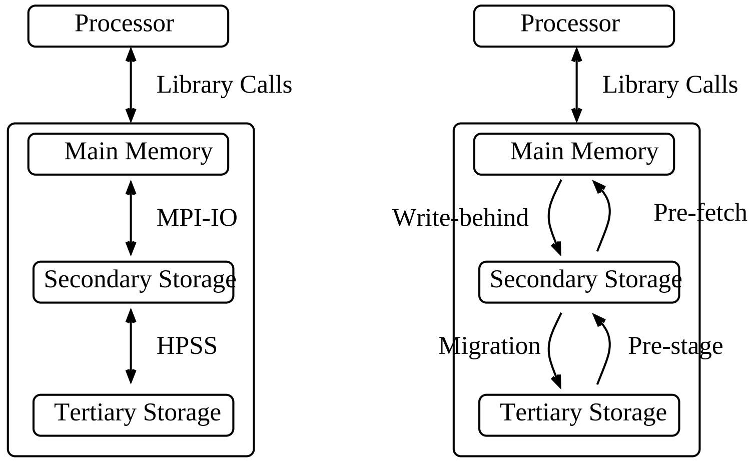

Figure 3.1: (a) Library architecture. (b) Pre-fetching, pre-staging, and migration.

when the file is closed. In other words, the database is accessed only in file open and file close.

3.3 User Interface

The routines in the library can be divided into four major groups based on their functionality: Initialization/Finalization Routines, File Manipulation Routines, Array Access Routines, and Stage/Migration Routines. Table 3.1 lists the basic routines in the library and their functionalities. All the routines listed here are the low level instructions, which should explicitly be called by the user. We are currently adding high level routines to our library, which will call the low level routines implicitly.

Initialization/Finalization Routines: These routines are used to initialize the library buffers and meta-data structures and finalize them when all the work is done. The routine to initialize the system has the format
int T_INITIALIZE ().
This routine initializes the connections to the HPSS and the database. It returns

Table 3.1: The library routines.

Routine	Functionality
Initialization/Finalization Routines	
T_INITIALIZE	Initializes the library structures
T_FINALIZE	Finalizes the library structures
File Manipulation Routines	
Routine	Functionality
T_OPEN	Opens a tape-resident global file for read/write
T_CLOSE	Closes a tape-resident global file
T_REMOVE	Removes both the sub-files of the file and the corresponding info.
Array Access Routines	
Routine	Functionality
T_READ_SECTION	Reads a rectilinear section
T_WRITE_SECTION	Writes a rectilinear section
Stage/Migration Routines	
Routine	Functionality
T_STAGE_SECTION	Stages a rectilinear file section from tape to disk
T_STAGE_WAIT	Waits for a Stage to complete
T_PREFETCH_SECTION	Pre-fetchs a rectilinear file section from tape (or disk) to memory
T_PREFETCH_WAIT	Waits for a Pre-fetch to complete
T_MIGRATE_SECTION	Migrates a rectilinear file section from disk to tape

a positive number upon successful completion. Similarly, int T_FINALIZE () closes the above mentioned connections.

File Manipulation Routines: These routines are used for creating files, opening existing files, closing open files and removing all the chunks and the information related to a global file. T_OPEN is used for creating new files and for opening existing files. It returns a file handle for later referral to the file. The synopsis of T_OPEN is as follows:

T_FILE T_OPEN (char *filename, char *mode, T_INFO *tapeinfo).

'Filename' stands for the name of the file to be opened. 'Mode' indicates whether the file is opened for read, write, or read/write. 'Tapeinfo' is the structure used for entering the necessary information about the file and chunks. It has fields for the elementsize, number of dimensions, the size of each dimension of the chunk and the

```
int main(int argc, char **argv)
{
 T_FILE exfile;
 T_INFO exinfo;
 int start[2];
 int end[2];

 /* Initialize the library */
 T_INITIALIZE();

 /* Open the file for read.  The exinfo will be filled by the library.
 For creating the file (i.e.  if the file is opened for the first time),
 information about the file should be supplied to T_OPEN via exinfo.*/
 exfile = T_OPEN ("file_1","r", &exinfo);

 start[0] = 0;
 start[1] = 0;
 end[0] = 24000;
 end[1] = 80;

 /* Perform the operation */
 T_READ_SECTION (&exfile, &buf, starts, ends);

 /* Close the file */
 T_CLOSE (&exfile);

 T_FINALIZE();
}
```

Figure 3.2: An example code for reading from a two dimensional file.

size of each dimension of the global file.

Array Access Routines: These routines handle the movement of data to and from the tape subsystem. An arbitrary rectilinear portion of a tape-resident array can be read or written using these access routines. Let us focus now on T_READ_SECTION. The signature of this routine is

int T_READ_SECTION (T_FILE *fd, void *buffer, int *start_coordinate, int *end_coordinate)

'fd' is the file descriptor returned by T_OPEN. 'Start_coordinate' and 'end_coordinate' are arrays that hold the boundary coordinates for the section to be read. There are as many elements as the dimensionality of the associated tape-resident global array. This command reads the corresponding elements and stores them in 'buffer'. As discussed earlier, what actually happens here is that the relevant sub-files are read from tape to disk (if they are not on the disk already), and the required sections are read from these sub-files on disk and forwarded to the corresponding positions in the buffer in main memory. An example code for T_READ_SECTION is given in Figure 3.2. In this example, a 24000×80 portion of the file is read to the buffer. The syntax for the T_WRITE_SECTION routine is almost the same except that the direction of the transfer is reversed.

Stage/Migration Routines: These routines are used to stage and migrate the data between the tapes and the disk sub-system. The command

int T_STAGE_SECTION (T_FILE *fd, int *start_coordinate, int *end_coordinate)

immediately returns and starts a data staging operation in the background from tape to disk. It returns an integer to the application which can be interpreted as a descriptor for the associated pre-stage operation. Note that what is actually performed here is to bring the relevant *sub-files* from tape to disk. Note also that there is no 'buffer' parameter in the signature. The routine

int T_STAGE_WAIT (int pre-stage_descriptor)

can be used to wait for a previously initiated pre-stage operation to complete.

int T_PREFETCH_SECTION (T_FILE *fd, void *buffer, int *start_coordinate, int *end_coordinate)

is used to start a pre-fetch operation from disk to memory. The parameters are the same as for T_READ_SECTION. It returns an integer which can be used as a pre-fetch descriptor in a later T_PREFETCH_WAIT call.

int T_MIGRATE_SECTION (T_FILE *fd, int *start_coordinate, int *end_coordinate)

starts to migrate the relevant sub-files (i.e., those corresponding to the section described in the signature) from disk to tape. It should be used with care as these sub-files may contain portions of data that will be requested by a later library call.

4. Sub-filing

In this chapter, we give the details of the sub-filing strategy. We first explain the method. Then we explain our experimental setup. This chapter finishes with the experimental results.

4.1 Sub-filing Strategy

In the sub-filing strategy, each *global array* is divided into *chunks,* each of which is stored in a *separate sub-file*. The chunks are of equal sizes in most cases. Figure 4.1 shows a two-dimensional global array divided into 64 chunks. Each chunk is assigned a unique *chunk coordinate* (x_1, x_2), the first (upper-leftmost) chunk having (0,0) as its coordinate. For the sake of ensuing discussion we assume that the sub-files corresponding to the chunks are stored in row-major as depicted in the figure by horizontal arrows.

A typical access pattern is shown in Figure 4.2. In this access a small two-dimensional portion of the global array is requested. In receiving such a request, the library performs three important tasks:

- Determining the sub-files that collectively contain the requested portion,
- Transferring the sub-files that are *not* already on disk from tape to disk, and
- Extracting the required data items (array elements) from the relevant sub-files from disk and copying the requested portion to a buffer in memory provided by the user call.

In the first step, the set of sub-files that collectively contain the requested portion is called *cover*. In Figure 4.2, the cover contains the sub-files (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,2), (3,3), and (3,4). Assuming for now that all of these sub-files are currently residing on tape, in the second step, the library brings these

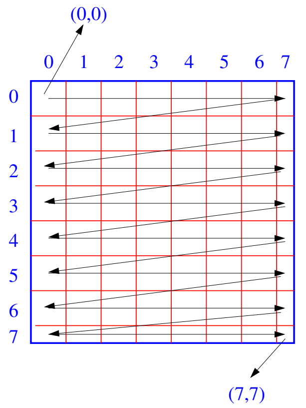

Figure 4.1: A global array divided into 64 chunks.

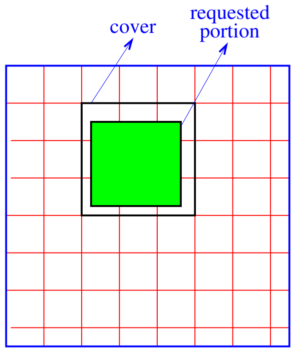

Figure 4.2: An access pattern (shaded portion) and its cover.

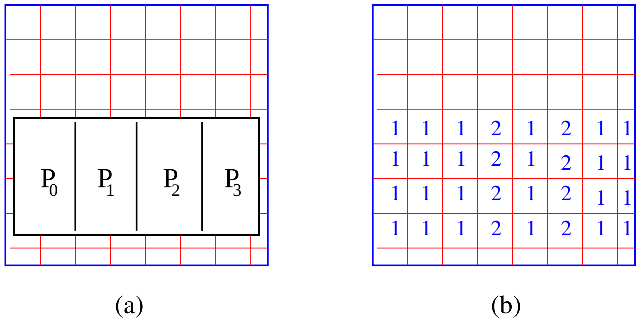

Figure 4.3: (a) An access pattern involving four processors. (b) The global array with each sub-file marked with the number of processors that share it.

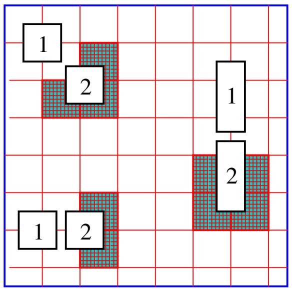

Figure 4.4: Successive array accesses.

sub-files to disk. In the third step, the required portion is extracted from each sub-file and returned to the user buffer. Note that the last step involves some computational overhead incurred for each sub-file. Instead, had we used just one file per global array this computational overhead would be incurred only once. Therefore, the performance gain obtained by dividing the global array into sub-files should be carefully weighed against the extra computational overhead incurred in extracting the requested portions from each sub-file. Our preliminary experiments show that this computational overhead is not too much.

Parallel reads by multiple processors pose additional problems. Although we are going to concantrate on this problem in the next chapter, we are going to discuss some important properties of the problem here. Consider now Figure 4.3(a) where four processors are requesting four different sub-columns of a region. The underlying cover contains 28 sub-files. After bringing these sub-files from tape to disk, we have a problem of reading the required sub-portions (sub-columns) for each processor. We solve this problem by using multiple collective I/O. As stated by del Rosario et al. [33], *collective I/O* is a technique in which processors perform I/O on behalf of each other in order to reduce the time spent in disk-I/O at the expense of some extra communication. Two-phase I/O is a specific implementation of collective I/O, which uses the information available about the access and storage patterns. While it is quite straightforward how to use collective I/O when we have a single file, in our multiple file case it is not clear how to utilize it. One simple approach might be to use collective I/O for each sub-file on disk. In our example, that would mean calling a collective I/O routine 28 times. A better alternative might be to read the data from disk to memory in two steps. In the first step, the processors that have exclusive access to some sub-files perform these independent accesses. In the second step, for each of the remaining sub-files, we can perform collective I/O using only the processors that request some data from the sub-file in question. Consider Figure 4.3(b), this collective I/O scheme corresponds to first reading the sub-files marked '1' and then collectively (using two processors) reading the sub-files marked '2'. In the next chapter, we experiment with several strategies including the above mentioned ones.

During successive reads from the same global file it might happen that the same sub-file can be required by two different reads. Assuming that the sub-file in question still resides on the disk after the first read, it is unnecessary to read it again from tape in the second read. In such a case only the other (additional) sub-files required by the current access are read from tape. The situation is shown in Figure 4.4 for three scenarios. In each case, the first portion read is marked '1' and the second portion read is marked '2'. The shaded parts around the second portions correspond to additional sub-files that are needed for the second read. In other words, the library effectively uses the storage hierarchy.

4.2 Experimental Setup

The experiments are performed using the HPPS at the San Diego Supercomputing Center (SDSC). We have used the low level routines of the SDSC Storage Resource Broker (SRB) to access the HPSS files. SRB is a client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated data sets [4].

We experimented with different *access patterns* in order to evaluate the benefits

Table 4.1: Access patterns used in the experiments.

	Pattern Information		
Access Pattern	Start Coordinate	End Coordinate	Total floating points
A	(0,0)	(1000,1000)	$1 * 10^6$
B	(0,0)	(4000,1000)	$4 * 10^6$
C	(0,0)	(24000,1000)	$24 * 10^6$
D	(5000,5000)	(6000,6000)	$1 * 10^6$
E	(0,0)	(50000,80)	$4 * 10^6$
F	(0,0)	(80,50000)	$4 * 10^6$
G	(0,0)	(1000,4000)	$4 * 10^6$
H	(6000,6000)	(8000,8000)	$4 * 10^6$

of the library. Table 4.1 gives the start and end coordinates (on a two dimensional global array) as well as the number elements read/written for each access pattern (A through H). Note that the coordinate (0,0) corresponds to the upper-left corner of the array. In each case, the accessed array consists of 50000×50000 floating point elements (10 GB total data). We used two different sub-file (chunk) sizes: small (1000×1000 elements) and large (2000×2000 elements).

4.3 Experimental Results

Table 4.2 shows the performance results obtained. For each operation (read or write) we give the response times (in *seconds*) for a naive access strategy and the gains obtained against it using our library which employs sub-filing. The naive strategy reads/writes the required portion from/to the array directly, i.e., it does not use sub-filing and the entire 50000×50000 array is stored as a single large file. For the sub-filing cases we show the *percentage reduction* in response time of the naive scheme. For example, in access pattern A, the sub-filing with small chunk size improved (reduced) the response time for the read operation by 85.2%. Figures 4.5 and 4.6 show the results obtained in graphical form. Note that the y-axes on the figures are *logarithmically scaled.*

Table 4.2: Execution times and percentage gains.

	Write Operations			Read Operations		
Acc. Ptr.	Times w/o chunking	Small Chunk Gain (%)	Large Chunk Gain (%)	Times w/o chunking	Small Chunk Gain (%)	Large Chunk Gain (%)
A	2774.0	96.1	94.5	784.7	85.2	77.1
B	2805.9	83.8	84.9	810.1	43.2	55.6
C	2960.3	8.8	37.9	793.3	-240.5	-172.4
D	3321.2	96.7	95.4	798.4	84.1	79.7
E	151.7	-3525.1	-2437.6	165.2	-3229.3	-2623.9
F	138723.3	96.0	97.2	39214.1	85.9	88.5
G	11096.3	95.9	96.4	3242.9	88.3	88.6
H	5095.2	91.2	96.5	1612.9	76.6	89.9

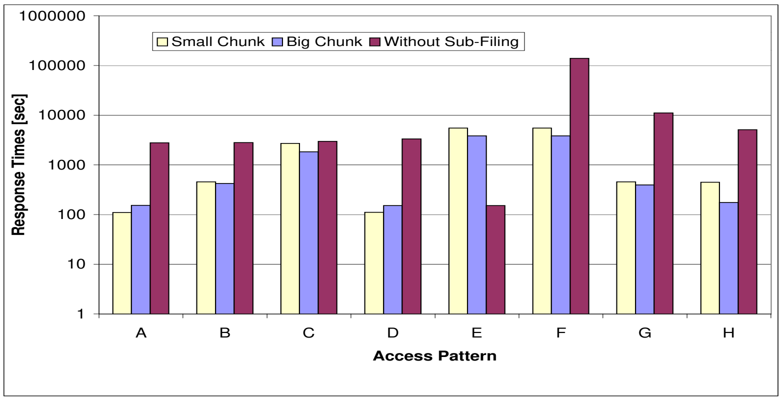

Figure 4.5: Execution times for write operations.

In the patterns A and D, where a 4 MB square chunk is accessed on the left corner and around the middle, respectively, the small chunk size outperforms the large chunk size as the latter accesses extra data elements that do not belong to the required portion. In the pattern H, on the other hand, increasing the chunk size reduces the number of I/O calls which in turn results in the best response time. In B and G, 16 MB of data are accessed in orthogonal directions. In G, since we access a sub-column portion of a row-major array, we need to issue 4000 I/O calls in the naive case. In B, the naive strategy issues only 1000 I/O calls to access the same

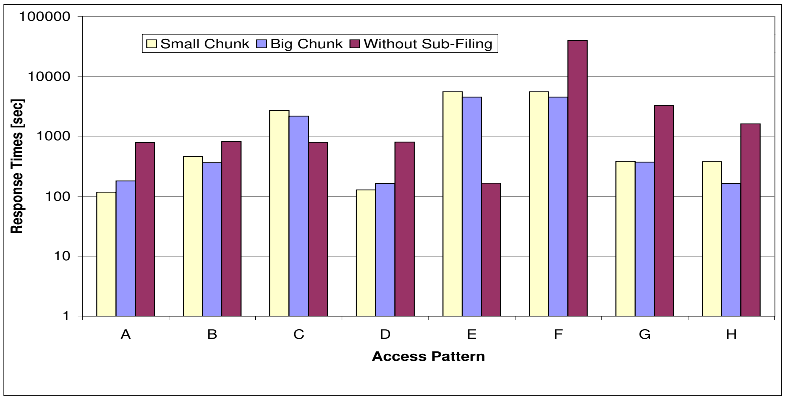

Figure 4.6: Execution times for read operations.

volume of data. Consequently, the impact of sub-filing is more pronounced in G. By comparing the response times of A, B, C, and E, we note that the response times are dominated by the number of I/O calls (in the naive version) and of chunks (in the sub-filed versions–that also corresponds to I/O calls–) rather than by the volume of data accessed. Finally, in the pattern F (whose response time in the naive case was calculated using interpolation from A and G), the sub-filing strategy has the best performance of all and brings a 97.2% improvement in write calls.

In access pattern E, however, the naive strategy outperforms the sub-filing. The sub-filing strategy has two drawbacks for this access pattern. First, the naive strategy completes the whole access with a single I/O call, whereas the sub-filing strategy requires 50 calls to different sub-files to satisfy the access. Secondly, a high percentage of data read by the sub-filing is not used to satisfy the request. As a result of these two drawbacks, the naive strategy performs better than the sub-filing for this access pattern. However, we show in Section 4.3.1 that by chosing an appropriate sub-file size, the sub-filing strategy can perform as good as the naive strategy even for the access pattern E. Note that, HPSS allows the users to access portions of the data residing in tape. The response time of the naive solution for the access pattern E will increase dramatically for the tape architectures, where the granularity of access is a file, because the whole file should be brought to the disk from tape.

An important aspect of our library is its handling of random I/O accesses. When we compare the times for the access patterns A and D, we see that there is a 20% increase in the response time of the naive strategy for the write operation. On the other hand, the times for sub-filed versions remains the same.

Overall, the sub-filing strategy performs very well compared to the naive strategy which performs individual accesses to a large file, except for the cases where the access pattern and the storage pattern of the array match exactly. For large chunk size the average improvement for writing is 93.48%, and for reading it is 73.48%. These data show that, in average our library brings substantial amount of improvement over the naive strategy. The next section shows even in the case where the access and storage pattern match exactly, a suitable chunk shape allows our scheme to match the response time of the naive strategy.

4.3.1 Adaptive Chunk Size

The preliminary results show that our library can bring substantial amount of improvement over the naive case. In the access pattern E, however, the naive strategy performs better. In this section, we experiment with a different chunk size to explore the possibility of matching the performance of the naive scheme in this access pattern.

In the new experiments, the chunk size is set to 50000×80 floating points, which is similar to the access pattern E. The other parameters remain as in Section 4.3. The response time for write operation drops to 148.9 seconds, which is 1.85% better than the naive scheme. For read operation, the response time is 166.2 seconds, which is 0.61% worse than the naive scheme. These results show that our library can perform as good as the naive scheme even in cases, where the access pattern and storage pattern exactly match.

5. Multiple Collective I/O

In this chapter, we give the details of the multiple collective I/O strategy. As explained in the previous sections, with multiple collective I/O we extend the collective I/O for the case where multiple processors read data from multiple files. Plain collective I/O strategy is designed for the case where multiple processors read data from a single file.

This strategy can be used in two ways. First, it can be used within the context of sub-filing. As explained in the previous section, in the sub-filing strategy, a global file is stored as several small subfiles. If a set of this sub-files should be accessed by several processors for an I/O call, then we can use the multiple collective I/O strategy. We can use this strategy even if the sub-filing strategy is not used. As explained in Section 1.3, in the High Performance Data Management project the goal is to collect data about the I/O calls in the applications, thereby creating opportunities for several optimizations. One of the data collected is the set of files an application is accessing. Using this information and the multiple collective I/O strategy explained in this section, we can get significant amount of improvement for the I/O access times.

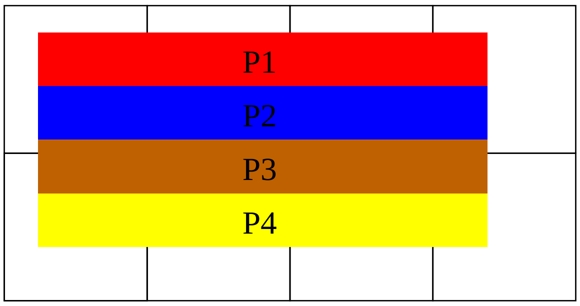

Figure 5.1: An access pattern involving four processors and eight files.

5.1 Multiple Collective I/O Strategy

Consider Figure 5.1 where four processors are requesting different sub-columns of a region. The underlying cover contains 8 files. Now, the problem is to read the corresponding data portions from the corresponding files. There are several methods to achieve this. In a naive method, we would use the collective I/O for all the files. For our example in Figure 5.1, this will result in 8 different I/O calls. In this method, as the number of files increase, the number of I/O calls will increase linearly regardless of the number of processors available. In this section, we are trying to explore methods that will perform better than this naive method. The main idea behind these methods is to assign different files to different processors, thereby increasing the parallelism available in the system. There are several problems to be solved with this method as well. First, how many files should we assign to each processor or vice versa? Then, once we decide the number of processors to be assigned for each file, how do we decide on the specific processor for each file. In the next section, we try to solve the first question. Then, we try to solve the second question. In the following discussions in this chapter, we concentrate on the I/O accesses where the number of files exceed the number of processors. Note that, the problem we are trying to solve is symmetric. If the number of files is larger, we are trying to assign files to processors, if the number of processors exceed the number of files, we will try to assign the processors to files, but both the nature of the problem and the algorithms we will explain in this section will still remain valid regardless of the relation between the number of processors and the number of files.

5.2 Deciding the Number of Processors

In this section, we try to find the number of processors we have to use for each file. For finding this value, we have used the following basic model for the I/O time.

$c_1 + c_2$ * ((Data Size) / (number of processor))

Although this is a very simple model to estimate the I/O times in a parallel environment, it is a good start point to find the number of processors for each file.

The total time for the request will be determined by the processor for which the I/O time is maximum. If we assume the data sizes to be read from each file to be equal and that we assign k processors to each file, the response time will be equal to

$$m * c_1 + m * c_2 * ((\text{Data Size}) / (k)) \quad \text{Eq.1.}$$

, where m is the number of files that the slowest processor is assigned to. If the assignments are done homogeneously among the processors, then

$$k = (m * p) / n$$

, where p is the number of processors and n is the number of files. So, equation 1 becomes

$$m * c_1 + (n / p) * c_2 * \text{Data Size}$$

For a given number of processors and number of files, this expression takes its minimum for the minimum value of m. Since we have to assign at least one processor for each file the minimum value for m is (n / p). That means, we have to assign only one processor to each file to minimize the response time. Note that, in the naive strategy, several processors are assigned to each of the files, which will increase the response time according to our model.

If the number of processors is larger then the number of files, then we have to assign one file to each processor only (the reverse of one processor for each file). The calculations are similar to the calculations given above.

Although we have found that we have to assign one processor to each of the files, we still have to determine which processor to assign to each of the file. Our selection of the processor will have a marginal effect on the I/O times, but it will have a significant effect on the communication times. This is a hard problem to solve. We first build the LP-model of the problem. Since it is slow to solve such an LP model in a run-time system, we have developed two heuristics to solve the problem. The following section section describes the LP-model we have constructed. The next sections discuss the heuristics.

5.3 LP Model of the Assignment Problem

We are given a matrix $a_{i,j}$, which gives the amount of data requested by processor i from file j. And we have to find another matrix $x_{i,j}$, which tells whether the processor i is assigned to file j or not. Although our selection of the processor for each file will not effect the I/O time, it will effect the communication time significantly. In the following discussion, n corresponds to the number of files to be accessed and p corresponds to the number of processors involved. So, we are trying to minimize

$\sum_{i=1}^{p} \sum_{j=1}^{n} a_{i,j} \times (1 - x_{i,j}) \ \forall i, j$

, because this will give us the total amount of data to be communicated. We will have the following constraints on $x_{i,j}$

- $x_{i,j} \in \{0,1\}$

$X_{i,j}$ is a decision variable and should be either “assigned” or “not assigned”.

- $\sum_{i=1}^{p} x_{i,j} = 1 \ \forall j$

In the previous section, we have found that each file will be assigned one processor. If the number of processors exceed the number of files, this constraint becomes

- $\sum_{j=1}^{n} x_{i,j} = 1 \ \forall i$

, making each processor to be assigned to only one file.

- $\sum_{j=1}^{n} x_{i,j} = \frac{n}{p} \ \forall i$

This constraint makes sure that the assignments are homogeneous. It states that the number of files assigned to processor i equals to (n /p). Here, we assumed that regardless of the data size read from the files, we are going to assign same number of files to each processor. Although this assumption makes the calculation of the assignment variables easier, it might decrease the performance if the variance between the data sizes read is large. In such a case we are going to use the constraint

- $\sum_{j=1}^{n}(\sum_{k=1}^{p} a_{k,j}) \times x_{i,j} \leq P \; \forall i$

, which means that the total data to be read by each processor should approximately be the same.

If the number of processors exceed the number of files, then the constraint becomes,

- $\sum_{i=1}^{p} x_{i,j} = F_j \; \forall j$

, where F_j represents the number of processors to be assigned to file j. This number can easily be found using the amount of data read from the file j and all the other files. To calculate F_j, we are going to use the following formula

$$F_k = p \times \frac{\sum_{i=1}^{p} a_{i,k}}{\sum_{i=1}^{p} \sum_{j=1}^{n} a_{i,j}}$$

In other words we distribute the p processors according to the portion of the file j with respect to all the data read.

Note that, in this model the number of variables is equal to (number of processors) $\times$ (number of files). For a reasonable size of input variables, we will have around 100 variables, which will take several minutes to solve even for the fastest LP-solvers available. Therefore, we have developed two heuristics to solve the problem. We discuss these heuristics in the next section.

5.4 Heuristics

In this section, we are going to explain the two heuristics we have developed. The first one uses a sorting algorithm to find the assignment. The second one uses a matching problem solver to make the assignments.

5.4.1 Greedy Heuristic

The first heuristic uses sorting for making the assignments. The main idea behind the heuristic is to assign the specific processors to the file, which are reading

Table 5.1: The amount of data requested by each processor from each file for the access in Figure 5.1. The files are named row-major order starting from the upper left corner.

Processor Number	File Numbers							
	1	2	3	4	5	6	7	8
1	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
2	10 MB	12.8 MB	12.8 MB	4.4 MB	0 MB	0 MB	0 MB	0 MB
3	2.5 MB	3.2 MB	3.2 MB	1.1 MB	7.5 MB	9.6 MB	9.6 MB	3.3 MB
4	0 MB	0 MB	0 MB	0 MB	10 MB	12.8 MB	12.8 MB	4.4 MB

Table 5.2: The list formed for the example access in Figure 5.1.

Access List
$P1 \rightarrow F2, P1 \rightarrow F3, P2 \rightarrow F2, P2 \rightarrow F3, P4 \rightarrow F6, P4 \rightarrow F7, P1 \rightarrow F1, P2 \rightarrow F1, P4 \rightarrow F5, P3 \rightarrow F6, P3 \rightarrow F7,$ $P3 \rightarrow F5, P1 \rightarrow F4, P2 \rightarrow F4, P4 \rightarrow F8, P3 \rightarrow F8, P3 \rightarrow F2, P3 \rightarrow F3, P3 \rightarrow F1, P3 \rightarrow F4$

the largest amount of data from the file. To achieve this, we first create an entry for every case, where processor i is reading a part of file j. Then all these entries are sorted according to the amount of data the processor i is reading from file j. So, the access list for the example in Figure 5.1 is given in Table 5.2, which depends on the data read by each processor from each file given in Table 5.1.

The algorithm tries to pick the first element from the list. If the processor has already been assigned, then the entry is skipped and the next entry is checked. The algorithm continues until we make n assignments. If the entries in the list are finished before we make n assignments, we make the rest of assignments randomly from the remaining processor pool. We do not have to pay attention in this step, because if the entries are finished, it means that all of the remaining processors are not reading any data from the remaining files. So, we can make the assignments arbitrarily because it will not change the amount of data communicated. The pseudo code of the algorithm is given in Figure 5.2. The resultant assignments for the access in Figure 5.1 is given in Figure 5.3. In the result, file 1 and file 4 are assigned to processor 2, file 2 and file 3 are assigned to processor 1, and file 5 and file 8 are assigned to processor 3, file 6 and file 7 are assigned to processor 4.

```
GREEDY_ASSIGN (p, n, a_{i,j})
/*p is the number of processors, n is the number of files,
a_{i,j} represents the amount of data read by processor i from file j.  */
begin
  for i=0 to p do
    for j=0 to n do
      if a_{i,j} ≠ 0 then
        insert_entry (i, j, a_{i,j})
      end if
    end for
  end for
  sort_entries_according_to (last_field)
  while (list not empty) AND (assignments_made < n) do
    entry = list_top
    if entry.processor has been already assigned then
      remove_entry (entry)
    elseif
      assign (entry.processor, entry.file)
      assignments_made ++
    end if
  end while
  assign the remaining files to remaining processors arbitrarily
  return all_assignments
end.
```

Figure 5.2: Greedy algorithm for making the assignments.

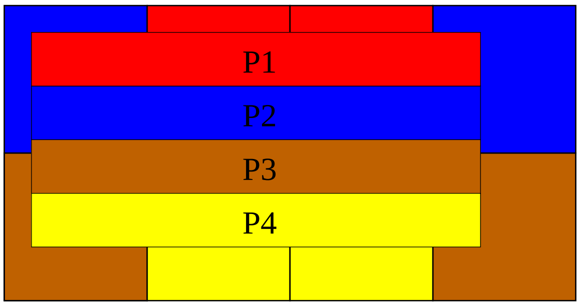

Figure 5.3: The result of the Greedy Heuristic. The colors denote the processor assigned.

The insert_entry function inserts a new entry into the list with the processor number i, file number j, and weight $a_{i,j}$. Similarly, remove_entry function deletes the entry from the list that is equal to its input parameter. At the end, the algorithm returns the list of processor - file pairs that have been assigned.

5.4.2 Matching Heuristic

The second heuristic uses a matching problem solver. We have used the solver from the netflow solver package[13]. The matching problem solver inside netflow implements H. Gabow's N-cubed weighted matching algorithm [17]. This program was written by Ed Rothberg. But to be able to use an existing matching problem solver, we first build a graph representing the $a_{i,j}$'s. Then, we have to modify the graph so that the solver gives the answer we are looking for.

The first step is straightforward. We build a graph G (V,E), in which V contains a vertex for each processor and file. If there are p processors and n files, then the graph is going to have (n+p) vertices. The resultant vertices for the example access in Figure 5.1 is given in Figure 5.4(a). Then we put an edge (u,v) $\in$ E with weight w, when the processor u reads w bytes of data from file v. The resultant graph is shown in Figure 5.4(b).

The existing matching problem solvers do not solve the exact problem we are trying to solve. They basically solve the maximum flow problem given an input graph. The solver tries to match one vertex for each existing vertex. For a bipartite graph G ($V_1 \bigcup V_2$, E) (the graph given to the algorithm is bipartite), it tries to find a different vertex in V_2 for each vertex in V_1. If the number of vertices in V_2 is larger then the number of vertices in V_1, then some of the vertices in V_2 will be left out. Similarly, if the number of vertices in V_1 is larger then the number of vertices in V_2, then some of the vertices in V_1 will be left out. Therefore we have to make sure that the number of vertices for processors equals to the number of processors for files. To be able to make the number of processor vertices equal to the number

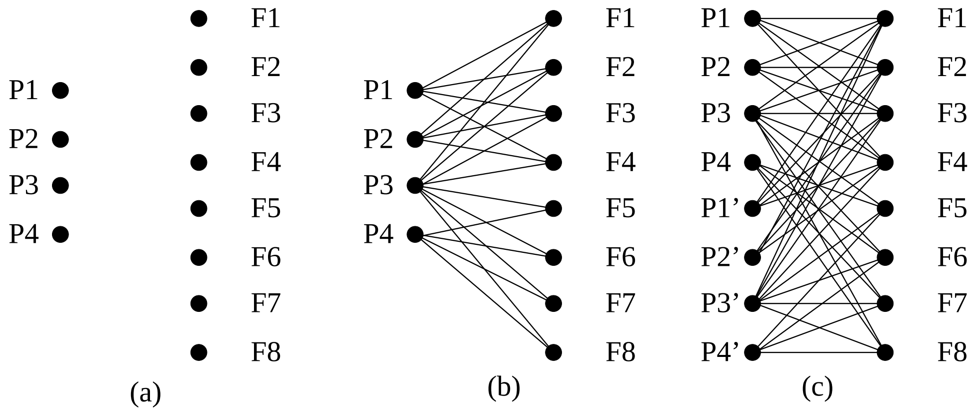

Figure 5.4: The creation of the input graph for the matching problem solver. The weights on the edges are left out for simplicity. (a) The initial graph with a node for each processor and a file. (b) The graph with an edge added for each case where processor i is reading data from file j. (c) The graph after the replication of the processor vertices to make the number of processor vertices and the file vertices to be equal.

of file vertices, we replicate the processor node. For each file node, we replicate F_j times the processor nodes that have an edge to the node[1]. Since the solver is going to assign only one node for each file vertex, the solver will give the result we are looking for. This way, we make sure that the solver makes n assignments. The resultant graph is given in Figure 5.4(c). In this example we assumed that F_j equals to two for every file node.

Once the graph is completed, we give it to the matching problem solver as input. The result of the solver is used as the assignments between the processor and the files. The assignments for the replicated nodes are interpreted like it is an assignment to the original node. Note that, if the number of processors is larger than the number of files, we replicate the file nodes so that in the final result some processors may be assigned to the same file.

[1]The calculation of F_j was explained in Section 5.3.

Table 5.3: Platform used in the experiments.

Number of Processors	128 (120 compute nodes, 8 I/O nodes)
Processor Type	Compute nodes: RS/6000 Model 370, I/O nodes:RS/6000 Model 970
Clock Rate	332 MHz
Out-of-Order Issue	4 instructions
L1 Cache	32 KB split, 2-way associative
L2 Cache	256 KB unified, 2-way associative
Memory Capacity	128 MB per compute node, 256 MB per I/O node
Network	100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI
Disk Space	9 GB per I/O node
Operating System	AIX 4.2.1
Parallel File System	PIOFS

5.5 Experimental Environment

We used the MPI-2 library and an IBM SP-2 in Argonne National Laboratories to evaluate our scheme proposed in this section. The important characteristics of our experimental platform are shown in Table 5.3.

The IBM SP-2 used in the experiments has 128 processors, 8 of which are I/O processors. Each I/O server has a 9 GB of storage space resulting in 72 GB of total disk space. The operating system on each node is AIX 4.2.1. PIOFS provides the parallel access to files. It distributes a file across multiple I/O server nodes.

5.6 Experimental Results

We have first experimented with 4 nodes reading data from 8 files (as shown in Figure 5.1). Each file is a of size 32 MB, representing a two-dimensional matrix of 2048×2048 floating points. So the total amount of data read equals to 256 MB for the base case.

The results for all the experiments are summarized in Table 5.4. It gives the results for the base case and for different number of processors and files. The multiple collective I/O is able to improve the response time by 49.77% in the case where four processors are accessing 8 different files. The results reveal that as the number of processors increase, the improvement also increases. Also when the number of files

Table 5.4: The improvements in [%] with respect to the naive I/O access.

Number of Files	Number of Processors		
	4	8	16
8	49.77	78.33	87.39
16	57.13	80.12	86.67
32	57.57	80.52	87.18

is increased, the improvement increases. Only with 16 processors, the improvement drops when the number of files is increased from 8 to 16. Note that the experiment with 16 processors and 8 files is the only experiment with the number of processors being larger than the number of files. So, when the number of files is increased, there is a transition from number of processors being larger to being equal to the number of files. Therefore, it creates an exception in the trends. This also shows that the algorithm brings significant amount of improvement in the case where the number of processors are less than the number of files.

In conclusion, the results reveal that the multiple collective I/O strategy brings significant amount of improvement over a naive access method. Specifically, we were able to improve the response time by as much as 87.39%.

6. Selective Collective I/O

In this chapter, we give the details of the selective collective I/O strategy. As explained in the previous sections, in selective collective I/O, we use the collective I/O selectively. In other words, we try to eliminate the overhead of the syncheronization and communication by using independent I/O, when possible. An example of collective I/O and independent parallel I/O and collective I/O is given in Figure fig:resim1. We first explain the method. Then we explain our experimental setup. This chapter finishes with the experimental results.

6.1 Selective Collective I/O Strategy

Our approach to collective I/O utilizes a directed graph called *weighted communication graph* (WCG). Each node of a weighted communication graph is a *code block*, which can be defined as a program fragment during which we can keep the datasets in memory; however, between executions of code blocks, the datasets should

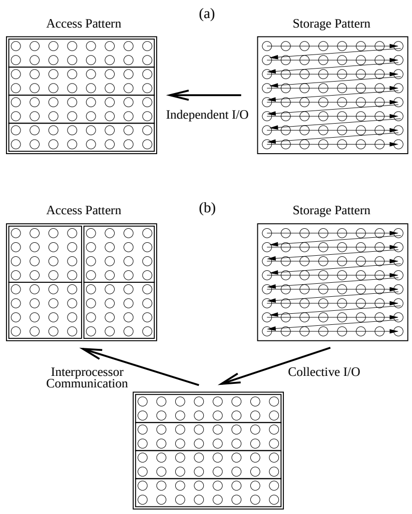

Figure 6.1: (a) Independent parallel I/O and (b) Collective (two-phase) I/O.

be stored on disks. Depending on the applications, the datasets in question, and the available memory, a code block can be as small as a loop nest or can be as large as a full-scale application. An example for the latter is shown in Figure 6.2 that depicts a typical scenario from a scientific working environment. There is a directed edge, $e_{1,2}$, between two nodes, cd_1 and cd_2, of the WCG if and only if there exists at least a dataset that is produced (i.e., created and stored on disk) in cd_1 and used (i.e., read from disk) in cd_2. The *weight* associated with $e_{1,2}$ (written as $w_{1,2}$) corresponds to total number of dynamic control-flow transitions between code blocks cd_1 and cd_2 (e.g., how many times cd_2 is run after cd_1 in a typical setup). Depending on the granularity of code blocks, these weights can be calculated using profiling with typical input sets, can be approximated using weight estimation techniques, or can be entered by a user who observed the scientific working environment for a sufficiently long period of time.

For each dataset of interest, we also mark all the nodes that produce it (called *producers*) and use it (called *consumers*). Note that a given dataset can be produced and consumed *multiple times* and by *multiple nodes* in the WCG. For each producer, we determine all the consumers that use the values coming from that particular producer. *Since, in general, we may have more than one consumer for a given dataset and producer, and each consumer may access the dataset using a different access pattern, selecting an appropriate storage pattern for this dataset in the producer code block is of extreme importance.* For example, suppose that in Figure 6.2 node 0 produces a dataset which is later consumed by nodes 3, 4, and 5. Depending on the access patterns exhibited by these three nodes, we may or may not need to use collective I/O. In the fortunate case where all these three nodes have the same access pattern for the dataset, we can store the dataset in node 0 using this particular pattern (when we create it), resulting in independent parallel I/O in nodes 3, 4, and 5. However, if, say node 5, has a different access pattern

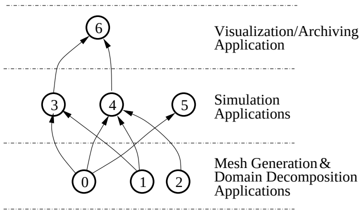

Figure 6.2: Scientific working environment.

than the other two, and we happen to select a storage pattern that is the same as the access pattern of nodes 3 and 4, then we need to perform collective I/O in node 5 in accessing the dataset in question. Nodes 3 and 4 can, on the other hand, still perform independent parallel I/O.

6.1.1 Access Pattern Detection

Access patterns exhibited by each code block can be determined by considering individual loop nests that make up the code block. The crucial step in this process is taking into account the parallelization information [2]. Individual nests can either be parallelized explicitly by programmers using compiler directives [1, 7], or can be parallelized automatically (without user intervention) as a result of intra-procedural and inter-procedural compiler analyses [2, 20, 21]. In either case, after the parallelization step, our approach determines the data regions (for a given dataset) accessed by each processor involved. Three typical access patterns for a given reference to a disk-resident array U are depicted in Figure 6.3 for abstract form of a loop nest taken from a Specfp benchmark, `Swim.256`. Although, in principle, we can have a wide variety of I/O access patterns (e.g., banded, diagonal, block-cyclic, etc.), for simplicity, we confine ourselves to *blocked* access patterns in which each dimension (of the disk-resident dataset in question) is either shared between processors (denoted $block(p)$) or not shared (denoted *), where p is the number of processors sharing the dimension. With this notation, the two access patterns shown in Figures 6.1(a) and (b) can be expressed as $(block(4), *)$ and $(block(2), block(2))$,

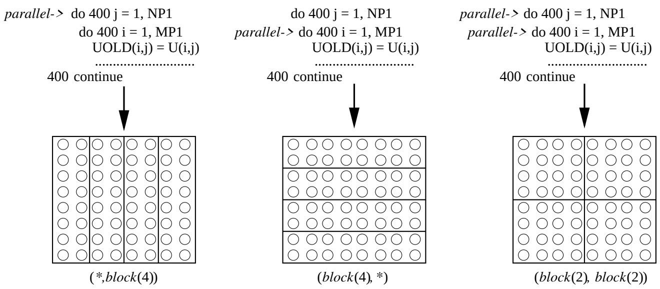

Figure 6.3: Three parallelization strategies and corresponding I/O access patterns.

respectively. In addition to the access pattern information, the compiler also records information regarding inter-processor communication patterns [2].

For each array reference in each loop nest, our compiler determines an access pattern as explained above. Afterwards, it utilizes a *conflict resolution scheme* to resolve intra-nest and inter-nest access pattern conflicts. To achieve a reasonable conflict resolution, we associate a *count* with each reference indicating (or approximating) the number of times that this reference is touched in a typical execution. In addition to that, for each access pattern that exists in the code block, we associate a *counter* that is initialized to zero and incremented by a count amount each time we encounter a reference with that access pattern. In this way, for a given array, we determine the most preferable (or most prevalent) access pattern (also called *representative access pattern*) and mark the code block with that information. Although, at first glance, it seems that, in a typical large-scale application, there will be a lot of conflicting patterns that would make compiler's job of favoring one of them over the others difficult, in reality, most scientific codes have a few preferable access patterns.

6.1.2 Storage Pattern Detection

Having determined an access pattern for each disk-resident dataset in each consumer code block, the next step is to select a suitable storage pattern for each dataset in its producer code block. We have built a prototype tool to achieve this.[2] For a given dataset, the tool takes the representative access patterns detected in the previous step by the compiler for each code block and runs a storage layout detection algorithm. Since this algorithm can be run for each disk-resident dataset separately, without loss of generality, in the following discussion, we focus only on a single dataset. The first step in our approach is to determine *producer-consumer subgraphs* (PCSs) of WCG for the dataset in question. A PCS for a dataset consists of a producer node and a number of consumer nodes that use the data produced by this producer. For instance, suppose that a dataset is produced by node 0 in Figure 6.2 and consumed by nodes 3, 4, and 5. In that case, the PCS, as shown in Figure 6.4(a), contains the nodes 0, 3, 4, and 5 and the edges from 0 to 3, 4, and 5. In the second step, we associate a count with each possible access pattern and initialize it to zero. Then, we traverse all the consumer nodes in turn and for each consumer node add its weight to the count of its access pattern. At the end of this step, for each access pattern, we obtain a count value.

In the third step, we set the storage pattern in the producer node to the access pattern with the highest count. Continuing with our current example, suppose that we have the following weights for the PCS as shown in Figure 6.4(a): $w_{0,3} = 5$, $w_{0,4} = 5$, and $w_{0,5} = 15$. Assume further that the access patterns (found using the technique presented in the previous subsection) for the nodes 3, 4, and 5 are $(*, block(4))$, $(*, block(4))$, and $(block(4), *)$, respectively. Therefore, the final count values for the patterns $(*, block(4))$, and $(block(4), *)$ are 10, and 15, respectively. Considering these values, we select $(block(4), *)$ as the final storage pattern in node 0.

[2] Note that building a separate tool is necessary only if the granularity of code blocks is a full-application. If, on the other hand, the granularity is a single nested-loop or a procedure, the functionality of this tool can be embedded within the compiler framework itself. Our current implementation supports three code block granularities, namely, full-application granularity, single procedure granularity, and loop nest granularity.

Note that, for a given dataset, we need to run the storage pattern detection algorithm multiple times, *one for each producer node* for this dataset. Note also that a given node can be both producer and consumer for a dataset. For instance, assume that, in Figure 6.2, node 0 produces a dataset which is later used by nodes 3 and 4, and the same dataset is overwritten (i.e., updated on the secondary storage) by node 4 and this updated dataset is later consumed by node 6. In this case, for this dataset, we have two PCSs as shown in Figure 6.4(b).

6.1.3 I/O Insertion

The next step is to determine suitable I/O strategies for each consumer node. Let us focus on a specific dataset. If the access pattern (for this dataset) of a consumer node is the same as the storage pattern in the producer node, we perform independent parallel I/O in this consumer node. Otherwise, that is, if the access and storage patterns are different, we perform collective I/O. For our running example in Figure 6.4(a), the final I/O strategies are shown in Figure 6.4(c). We perform this step for each dataset and each PCS. Once the suitable I/O strategies have been determined, the compiler automatically inserts corresponding MPI-IO calls in each code block. Due to space considerations, this last step is not discussed further in this paper.

6.1.4 Discussion

Although, our approach is so far mainly discussed for a setting where individual code blocks correspond to individual applications, it is relatively easy to adapt it to different settings as well. For example, if we consider each code block as a procedure in a given application, then a WCG can be processed using algorithms similar to those utilized in processing *weighted call graphs* [18], where each node (code block) represents a procedure and an edge between two nodes correspond to dynamic control-flow transitions (e.g., procedure calls and returns) between the

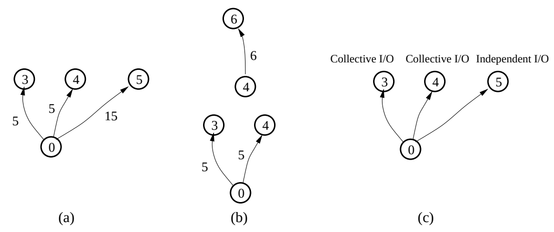

Figure 6.4: (a) A producer-consumer subgraph (PCS). (b) Two PCGs for a dataset. (c) Final I/O strategies for the PCG in (a).

procedures represented by these two nodes. In an out-of-core environment where processor memories are at premium, on the other hand, each node may represent an individual loop nest and edges might represent dynamic control-flow between nests; in this case, the WCG is similar to a control-flow graph. In all cases, though, the weights between edges can be obtained either by using profiling with representative inputs (dynamic approach) [18] or by static weight estimation techniques. For the latter, the control-flow prediction heuristics proposed by Ball and Larus [3] might be useful.

Another important issue that needs to be addressed is what to do (inside a code block) when we come across a reference whose access pattern is not the same as the representative (prevalent) access pattern for this code block. Recall that we assumed that, within a code block, we should be able to keep the datasets in memory. When the access pattern of an individual reference is different from the representative access pattern determined for a code block, we can re-shuffle the data in memory. This may not be a correctness issue for shared-memory machines as each processor has full-access to the global memory space, rather, it may increase the number of nonlocal references, thus, may cause performance degradation. In distributed-memory message-passing architectures, on the other hand, this data re-shuffling in memory is necessary to ensure the correct data accesses in subsequent computations.

Table 6.1: Platform used in the experiments.

Number of Processors	128 (120 compute nodes, 8 I/O nodes)
Processor Type	Compute nodes: RS/6000 Model 370, I/O nodes:RS/6000 Model 970
Clock Rate	332 MHz
Out-of-Order Issue	4 instructions
L1 Cache	32 KB split, 2-way associative
L2 Cache	256 KB unified, 2-way associative
Memory Capacity	128 MB per compute node, 256 MB per I/O node
Network	100 Mbs Ethernet, 155 Mbs ATM and 800 Mbs HiPPI
Disk Space	9 GB per I/O node
Operating System	AIX 4.2.1
Parallel File System	PIOFS

6.2 Experimental Environment

We used the MPI-2 library and an IBM SP-2 in Argonne National Laboratories to evaluate our scheme proposed in this section. The important characteristics of our experimental platform are shown in Table 6.1.

The IBM SP-2 used in the experiments has 128 processors, 8 of which are I/O processors. Each I/O server has a 9 GB of storage space resulting in 72 GB of total disk space. The operating system on each node is AIX 4.2.1. PIOFS provides the parallel access to files. It distributes a file across multiple I/O server nodes.

6.3 Preliminary Experiments

To gain insight on the effect of the storage pattern on I/O access times, we have performed several preliminary experiments. In these experiments, we measured the execution times of several access patterns for different storage patterns. The results are given in Table 6.2. In these experiments, we accessed a 64 MB file using 8 processors of the IBM SP-2. This table gives both the response time of the I/O call for collectively reading the file and the response time of the I/O call for non-collectively reading the file for different access and storage patterns. The storage pattern represent the pattern used for storing the file.

The results reveal that, if we can access the data non-collectively using the

Table 6.2: Average response times for collective I/O (above) and non-collective I/O (below) for different storage and access patterns.

Storage Pattern	Access Pattern		
	Row-Major	Column-Major	Block,Block
Row-Major	23.02	29.31	27.62
Column-Major	27.02	22.58	26.64
Block,Block	25.47	25.32	22.13

Storage Pattern	Access Pattern		
	Row-Major	Column-Major	Block,Block
Row-Major	21.11	175.06	84.59
Column-Major	91.16	20.19	74.19
Block,Block	82.82	151.48	21.03

storage pattern, we can eliminate the synchronization and communication costs of a collective I/O call. This results in the best performance possible.

The preliminary experiments show that, we can get considerable amount of improvement over the collective I/O by using our scheme. Therefore, we designed an environment, where we can experiment with different configurations of several applications. This will allow us to see the advantages and disadvantages of our scheme for a wide variety of applications. The setups explained in the next section are used throughout the experiments discussed in the rest of this chapter.

6.4 Setups and Configurations

To evaluate the possible improvements with our scheme, we have designed 8 different *setups* (*communication graphs*), each built up using 8 different application codes in different ways. The setups built from the applications are given in Figure 6.5.[3] The data size read and/or written by each application in all the setups vary between 64 MB and 128 MB. The data sizes are determined by the number of matrices accessed by the applications (each of which is 32 MB). For each setup given in Figure 6.5, we conducted experiments using two *configurations* (*weighted*

[3]Although Setup 1 and Setup 2 look the same, they differ in the access and storage patterns they employ for different applications. Similarly, Setup 7 and Setup 8 differ in their access and storage patterns. The details are omitted due to lack of space.

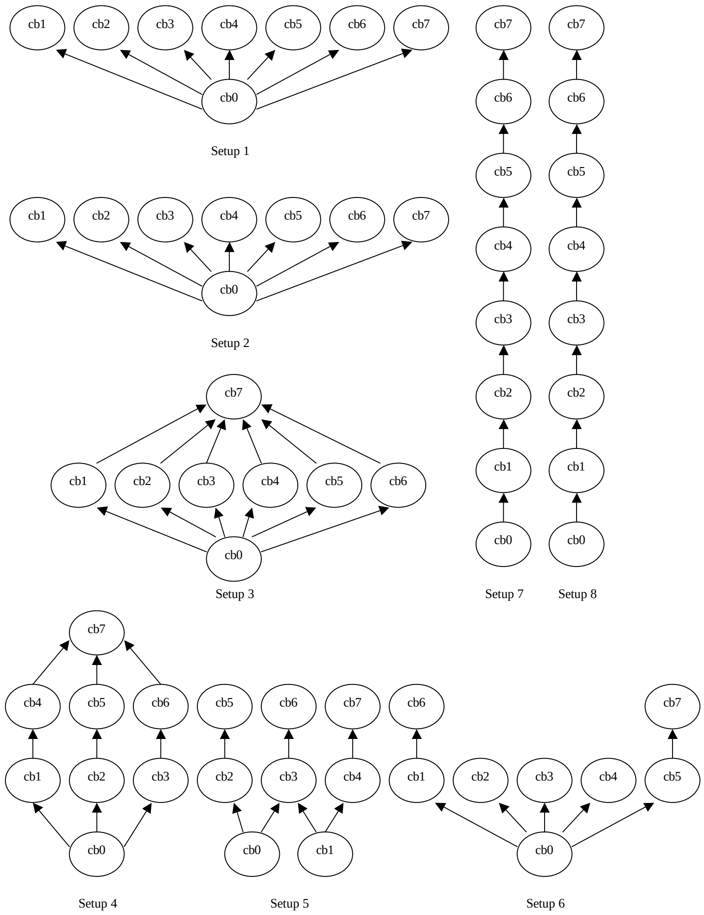

Figure 6.5: Setups (communication graphs) used in the experiments.

communication graphs). As shown in Table 6.3, for a given setup, the configurations differ from each other only in the weights of the edges in the graph. However, these weights are chosen such that the storage patterns detected by the algorithm for different configurations (of the same setup) are different. Each benchmark accesses the data using either (block,*), (*,block) or (block,block) pattern[4].

6.4.1 Different I/O Strategies

In our experiments we are comparing the execution times of three different code versions as explained below.

[4]For a discussion of different access patterns, please refer to Sections 6.1.1 and 6.1.2.

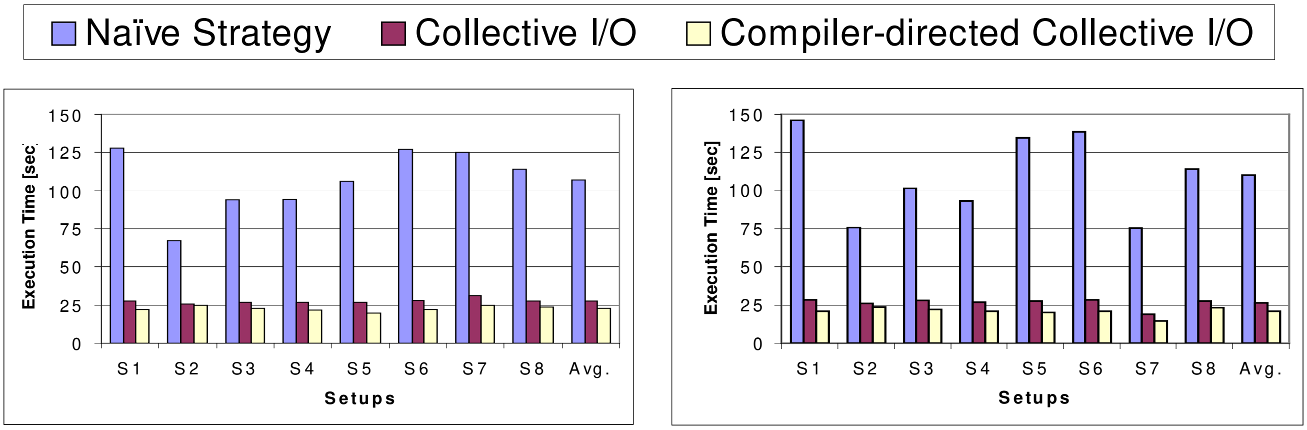

Figure 6: Execution times for the base experiments: Configuration one (left), configuration two (right)

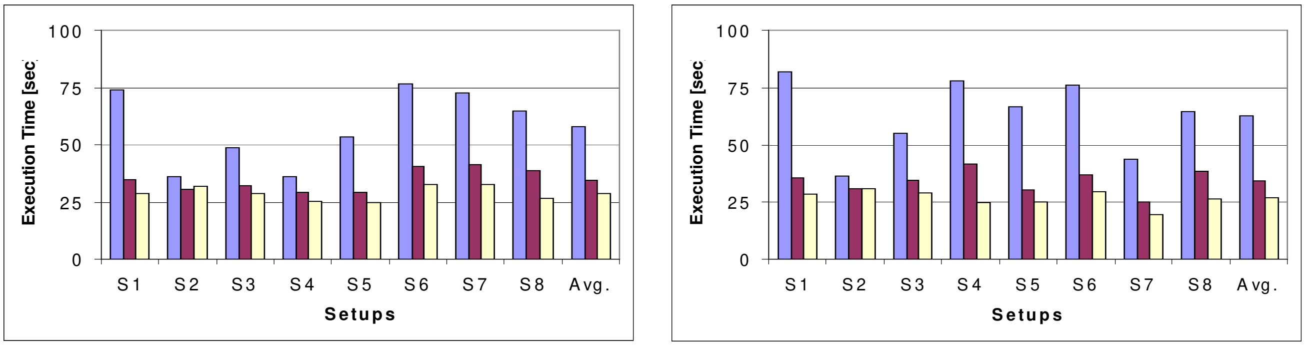

Figure 7: Execution times for 4 processors: Configuration one (left), configuration two (right)

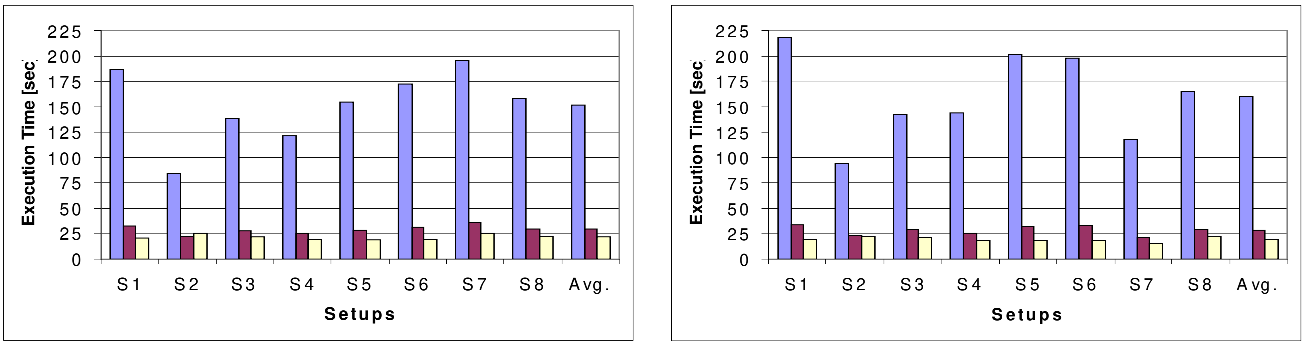

Figure 8: Execution times for 16 processors: Configuration one (left), configuration two (right)

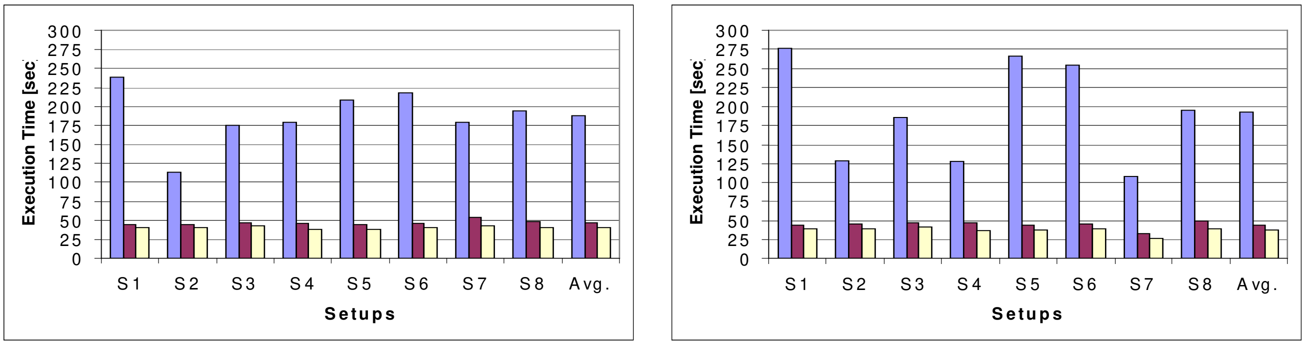

Figure 9: Execution times for larger data size (Data size is doubled): Configuration one (left), configuration two (right)

Three bars for each setup represents the following access strategies, from left to right: naïve strategy (no collective I/O), indiscriminate collective I/O, compiler directed collective I/O

Table 6.3: Configurations used in each setup.

Setup 1:

	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	$w_{0,7}$
Configuration 1.1	5	5	10	12	15	5	6
Configuration 1.2	10	1	10	5	15	5	12

Setup 2:

	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	$w_{0,7}$
Configuration 2.1	5	5	10	12	15	5	6
Configuration 2.2	10	1	10	5	15	5	12

Setup 3:

	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	$w_{1,7}$	$w_{2,7}$	$w_{3,7}$	$w_{4,7}$	$w_{5,7}$	$w_{6,7}$
Configuration 3.1	4	4	4	4	10	4	4	4	10	4	4	4
Configuration 3.2	1	1	10	10	10	10	1	4	4	10	4	4

Setup 4:

	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{1,4}$	$w_{2,5}$	$w_{3,6}$	$w_{4,7}$	$w_{5,7}$	$w_{6,7}$
Configuration 4.1	10	1	1	1	4	4	4	10	4
Configuration 4.2	5	5	5	5	1	1	1	5	1

Setup 5:

	$w_{0,2}$	$w_{0,3}$	$w_{1,3}$	$w_{1,4}$	$w_{2,5}$	$w_{3,6}$	$w_{4,7}$
Configuration 5.1	2	10	10	1	4	4	4
Configuration 5.2	3	3	5	5	4	4	4

Setup 6:

	$w_{0,1}$	$w_{0,2}$	$w_{0,3}$	$w_{0,4}$	$w_{0,5}$	$w_{1,6}$	$w_{5,7}$
Configuration 6.1	8	5	10	10	10	2	2
Configuration 6.2	4	5	5	5	15	2	2

Setup 7:

	$w_{0,1}$	$w_{1,2}$	$w_{2,3}$	$w_{3,4}$	$w_{4,5}$	$w_{5,6}$	$w_{6,7}$
Configuration 7.1	3	5	10	5	5	2	2
Configuration 7.2	5	5	15	12	10	4	2

Setup 8:

	$w_{0,1}$	$w_{1,2}$	$w_{2,3}$	$w_{3,4}$	$w_{4,5}$	$w_{5,6}$	$w_{6,7}$
Configuration 8.1	3	5	10	5	5	2	2
Configuration 8.2	5	5	15	12	10	4	2

Version 1 In this version, each processor performs independent (non-collective) I/O regardless whether the access pattern and the storage pattern are the same or not. This strategy may in general result in an excessive number of I/O calls issued by each processor. We call this version the *naive I/O strategy.*

Version 2 This version performs indiscriminate collective I/O. While we expect this version to bring significant improvements over the naive I/O strategy, in cases where the access pattern and the storage pattern match, the collective I/O is not necessary as the processors can perform independent parallel I/O.

Version 3 This is the strategy using selective collective I/O. The collective I/O is performed selectively, only if the access pattern and the storage pattern do not match. In all other cases, we perform independent parallel I/O.

In Figures 6 through 9, the left-most bar represents the total execution time of version 1, the middle bar represents the total execution time of version 2 and the right-most bar represents the total execution time of version 3.

6.5 Experimental Results

For the base experiments, we have executed the setup and configurations explained in Section 6.4 using 8 processors of the IBM SP-2. The results for the two configurations are given in Figure 6.

For the first configuration, the average improvement over the indiscriminate collective I/O strategy is 18.01%. For setups 5 and 6, we are able to gain more than 21% over the version 2, which performs indiscriminate collective I/O. These two setups give the best results, because a change of the storage pattern effects the most applications. For example, when $cb0$ in Setup 1 changes its storage pattern, the weights of the favoring applications add up to 60% of the sum of all weights, whereas 10% of all the applications disfavor from this change (favoring applications are the ones that were reading the files using the new storage pattern, on the other hand, disfavoring applications are the ones accessing the data using the old storage pattern). So, approximately 50% of the applications favor the change. On the other hand, in Setup 6, the sum of weights of the favoring applications constitutes 60% of all the weights, and there are no disfavoring applications. So, there are 10% more favoring applications in Setup 6. Therefore, the improvement of Setup 6 is more than Setup 1 with our scheme.

For the second configuration, the average improvement over the indiscriminate collective I/O strategy is 21.08%. The relations between the improvements for the individual setups are similar to those of the first configuration.

Note that, both the indiscriminate and selective collective I/O strategies perform well compared to a naive strategy, which does not use collective I/O at all. The improvement of indiscriminate collective I/O is 74.19% over the naive strategy, whereas our scheme brings 78.84% improvement for the first configuration.

6.5.1 Sensitivity Analysis

We have performed a second set of experiments to see how our strategy is affected by the number of processors and the data size. Figures 7 and 8 give the results for 4 and 16 processors, respectively. For 4 processors, our scheme brings 16.42% improvement over the indiscriminate collective I/O version, and 40.23% over the naive strategy for the first configurations. For 16 processors, on the other hand, our scheme brings an 26.27% improvement over the indiscriminate collective I/O version, and 80.74% over the naive strategy for the first configurations. As the number of processors increase, the improvement of our scheme increases, because with larger number of processors, the synchronization and communication costs increase[5]. With 16 processors, our scheme performs 30.95% better than the indiscriminate collective I/O version, and 82.24% over the naive strategy for the second configuration. For 4 processors, on the other hand, in the second configuration, our strategy performs 20.80% and 45.63% better than the indiscriminate collective I/O and naive strategies, respectively.

Figure 9 gives the results for a larger data size. For this experiment, we have doubled the size of the input and/or output data of all the applications. When the data size is increased, the synchronization overhead is reduced. Similarly, communication and I/O can be better overlapped because the I/O calls are longer, so the overall communication cost also reduces. These factors decrease the percentage improvements of our scheme, but, it still performs the best by far. For the first configuration, our scheme brings a 13.57% improvement over the indiscrimi-

[5]The increase in the communication cost is not caused by an increase in the size of the data communicated, but by a decrease in the average size of a message.

nate collective I/O version, and 75.21% improvement over the naive strategy. For the second configuration, on the other hand, the improvement of our scheme over the indiscriminate collective I/O version is 15.57%, and, over the naive strategy, it is 76.99%.

7. Conclusions

In this work, we have analyzed several I/O optimizations for hierarchical storage systems. We first presented a portable interface to the hierarchical storage system. The interface makes it easier for the user to specify the tape I/O required in sequential and parallel applications. The experience gained during its design and development will, hopefully, also help in reaching a set of standard routines for accessing the tape-resident data.

We also presented several optimizations for tape-resident and disk-resident data. We have shown that sub-filing brings significant amount of improvement in the response time for the tape-resident data. We also showed with the multiple collective I/O strategy, we can use the information about the applications to improve the response time of the I/O calls.

We evaluated a compiler-directed collective I/O strategy. Collective I/O plays a major role in parallel I/O systems. Therefore, increasing its performance is very important for many of data-intensive parallel applications. By adopting a selective collective I/O strategy, we are able to bring significant amounts of improvements. In average, our scheme performs 18.01% better than an indiscriminate collective I/O strategy in our base configuration. The scheme performs better as the number of processors increases. Although the improvement decreases with the increased data size, the scheme is still able to perform more than 13% better than an indiscriminate collective I/O strategy.

The interface for parallel I/O systems are usually complex, and they are getting more complex, because the information required by the I/O calls increases. So, it becomes harder for an average user to detect the best possible I/O call for an application. Therefore, detecting the best possible storage and access patterns automatically and providing more user-friendly interfaces are very useful for many programmers and increases the performance of I/O-intensive applications significantly.

References

[1] A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Prog.,* 6(1):3–28, Spring 1997.

[2] J. Anderson. *Automatic Computation and Data Decomposition for Multiprocessors.* Ph.D. dissertation, Computer Systems Lab., Stanford Univ., March 1997.

[3] T. Ball and J. Larus. Branch prediction for free. In *Proc. the SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation,* pp. 300–313, June 1993.

[4] S. Baru. Storage Resource Broker (SRB) Reference Manual. Enabling Technologies Group, San Diego Supercomputer Center, La Jolla, CA, August 1997.

[5] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. *Proceedings of the ACM Symp. on Prin. and Prac. of Paral. Prg.*, pages 1–10, July 1995.

[6] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In *Proc. the 1994 Summer USENIX Technical Conference,* pages 171–182, June 1994.

[7] R. Chandra, D. Chen, R. Cox, D. Maydan, N. Nedeljkovic, and J. Anderson. Data-distribution support on distributed-shared memory multi-processors. In *Proc. Prog. Lang. Design and Implementation*, Las Vegas, NV, 1997.

[8] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. In Information Systems, 20(2):155-183, 1995.

[9] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: parallel and scalable software for input-output. *NPAC Technical Report SCCS-636,* 1994.

[10] A. Choudhary, M. Kandemir, J. No, G. Memik, X. Shen, W. Liao, H. Nagesh, S. More, V. Taylor, R. Thakur, and R. Stevens. Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems. In Cluster Computing: the Journal of Networks, Software Tools and Applications, 2000.

[11] P. Corbett, D. Fietelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface, In *Proc. Third Workshop on I/O in Paral. and Distr. Sys.*, IPPS'95, Santa Barbara, CA, April 1995.

[12] P. F. Corbett, D. G. Feitelson, J-P. Prost, and S. J. Baylor. Parallel access to files in the Vesta file system. In *Proc. Supercomputing'93,* pp. 472–481, Nov 1993.

[13] URL: ftp://dimacs.rutgers.edu/pub/netflow/

[14] R. A. Coyne, H. Hulen, and R. Watson. The high performance storage system. In *Proc. Supercomputing 93*, Portland, OR, November 1993.

[15] A. L. Drapeau and R. H. Katz. Analysis of Striped Tape Systems. In *Proc. of the 12th IEEE Mass Storage Symposium*, Monterey, CA, March 1993.

[16] C. S. Ellis and D. Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proc. of the 1989 Intl. Conf. on Paral. Proc.,* pages I:306–314, St. Charles, IL, August 1989.

[17] H. Gabow. Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs. Ph.D Thesis, Stanford University, 1973.

[18] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proc. Micro-30,* Research Triangle Park, North Carolina, December 1–3, 1997.

[19] L. Golubchik, R. R. Muntz, and R. W. Watson. Analysis of Striping Techniques in Robotic Storage Libraries. In *Proc. of the 14th IEEE Mass Storage Sys. Symp.*, pages 225–238, 1995.

[20] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[21] M. W. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Inter-procedural analysis for parallelization. In *Proc. 8th International Workshop on Lang. and Comp. for Parallel Computers*, pages 61–80, Columbus, Ohio, August 1995.

[22] B. Hillyer and A. Silberschatz. Random I/O scheduling in Online Tertiary Storage Systems. In *Proc. of the 1996 ACM SIGMOD Conference*, pages 195–204, 1996.

[23] H. V. Jagadish L. V. S. Lakshmanan, and D. Srivastava. Snakes and sandwiches: Optimal clustering strategies for a data warehouse. In Proceedings of ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, USA., pages 37-48. ACM Press, 1999.

[24] T. Johnson and E. L. Miller. Performance measurements of Tertiary Storage Devices. In *Proc. of 24rd International Conference on Very Large Data Bases*, pages 50–61, August 1998.

[25] T. Johnson and E. L. Miller. Benchmarking Tape System Performance. In *Proc. of the 15th IEEE Mass Storage Systems Symposium*, 1996.

[26] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation,* pages 61–74, November 1994. Updated as Dartmouth TR PCS-TR94-226 on November 8, 1994.

[27] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proc. the 10th ACM International Conference on Supercomputing,* pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[28] B. J. Nitzberg. *Collective Parallel I/O.* PhD thesis, Department of Computer and Information Science, University of Oregon, December 1995.

[29] J. F. Karpovich, A. S. Grimshaw, and J. C. French. Extensible file systems (ELFS): An object-oriented approach to high performance file I/O. In *Proc. the Ninth Annual Conference on Object-Oriented Prog. Sys., Lang., and Appl.,* pp. 191–204, Oct 1994.

[30] B. Kobler, J. Berbert, P. Caulk and P. C. Hariharan. Architecture and Design of Storage and Data Management for the NASA Earth Observing System Data and Information System (EOSDIS). In *Proc. of the 14th IEEE Mass Storage Systems Symposium*, pages 65–78, 1995.

[31] D. Kotz. Multiprocessor file system interfaces. In *Proc. the Second Intl. Conf. on Paral. and Distr. Info. Sys.,* pages 194–201. IEEE Computer Society Press, 1993.

[32] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *ACM Operating Systems Review*, V 27(2), pp 21–34, April 1993.

[33] J. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proc. the 1993 IPPS Workshop on Input/Output in Paral. Comp. Sys.,* April 1993.

[34] J. del Rosario and A. Choudhary. High performance I/O for parallel computers: problems and prospects. *IEEE Computer,* March 1994.

[35] B. Rullman. Paragon parallel file system. *External Product Specification*, Intel Supercomputer Systems Division.

[36] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In Proceedings of the Tenth International Conference on DAta Engineering, Houston, Texas, USA., pages 328-336. IEEE Computer Society, 1994.

[37] S. Sarawagi: Execution reordering for tertiary memory access. *Data Engineering Bulletin* 20(3): 46–54, 1997.

[38] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing'95,* December 1995.

[39] X. Shen, W. Liao, A. Choudhary, G. Memik, M. Kandemir, S. More, G. Thiruvathukal, and A. Singh. A Novel Application Development Environment for Large-Scale Scientific Computations. In International Conference on Supercomputing, May, 2000.

[40] P. H. Smith and J. van Rosendale. Data and visualization corridors. *Technical Report CACR-164,* CACR, Caltech, Sept 1998.

[41] Rajeev Thakur, Ewing Lusk, and William Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised July 1998.

[42] R. Thakur, W. Gropp, and E. Lusk. A case for using MPI's derived data types to improve I/O performance. In *Proc. of SC'98: High Performance Networking and Computing,* November 1998.

[43] S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations, In *Proc. Workshop on I/O in Paral. and Distr. Sys.,* May 1996.

[44] URL: http://www.ece.nwu.edu/ wkliao/MDMS/index.html

[45] URL: http://www.sdsc.edu/hpss/hpss1.html

[46] R. Watson and R. A. Coyne. The parallel I/O architecture of the high-performance storage system (HPSS). Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, IEEE Computer Society Press, September 1995, pp. 27-44.

[47] A. Yu, and J. Chen. *The POSTGRES95 User Manual.* Dept. of EECS, University of California at Berkeley, July 1995.