Data Management for Large-Scale Scientific Computations in High Performance Distributed Systems

A. Choudhary, M. Kandemir[*], J. No[†], G. Memik, X. Shen, W. Liao, H. Nagesh
S. More, V. Taylor, R. Thakur[†], and R. Stevens[†]
Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208
{choudhar,memik,xhshen,wkliao,harsha,ssmore,taylor}@ece.nwu.edu

October 25, 1999

Abstract

With the increasing number of scientific applications manipulating huge amounts of data, effective high-level data management is an increasingly important problem. Unfortunately, so far the solutions to the high-level data management problem either require deep understanding of specific storage architectures and file layouts (as in high-performance file storage systems) or produce unsatisfactory I/O performance in exchange for ease-of-use and portability (as in relational DBMSs).

In this paper we present a novel application development environment which is built around an active meta-data management system (MDMS) to handle high-level data in an effective manner. The key components of our three-tiered architecture are user application, the MDMS, and a hierarchical storage system (HSS). Our environment overcomes the performance problems of pure database-oriented solutions, while maintaining their advantages in terms of ease-of-use and portability. The high levels of performance are achieved by the MDMS, with the aid of user-specified, performance-oriented directives. Our environment supports a simple, easy-to-use yet powerful user interface, leaving the task of choosing appropriate I/O techniques for the application at hand to the MDMS. We discuss the importance of an active MDMS and show how the three components of our environment, namely application, the MDMS, and the HSS, fit together. We also report performance numbers from our ongoing implementation and illustrate that significant improvements are made possible without undue programming effort.

1 Introduction

Many large-scale scientific applications are data intensive, processing large data sets ranging from megabytes to terabytes. These include applications from the data analysis and mining, image processing, large archive maintenance, real-time processing and so on. As an example, consider a typical computational science analysis cycle, shown in Figure 1. In this cycle there are several steps involved. These include mesh generation, domain decomposition, simulation, visualization, and interpretation of results, archival of data and results

[*]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, kandemir@cse.psu.edu.

[†]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, e-mail: {jano,thakur,stevens}@mcs.anl.gov.

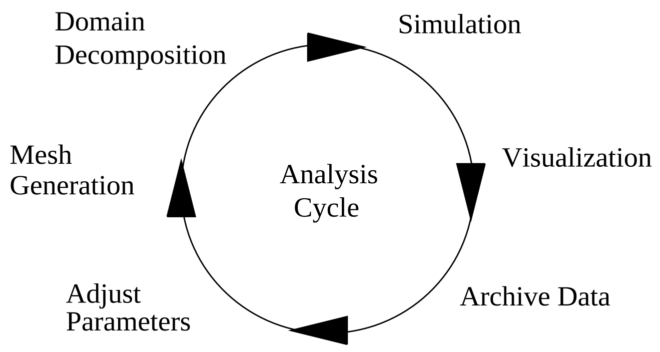

Figure 1: A typical computational science analysis cycle.

for post-processing and check-pointing, and adjustment of parameters. Thus, it is not sufficient to consider simulation alone when determining how to store or access data sets because the data sets in question are used in other phases as well. In addition, these steps may need to be performed in a heterogeneous distributed environment. These considerations require storing visualization and checkpoint data (which can run into 100s of megabytes to terabytes range) in a canonical form so that other tools can use them easily without having to re-organize the data. Furthermore, the re-start of computation with different number of processors requires that data storage be independent of number of processors that produced it. Such requirements present challenging I/O intensive problems and an application programmer may be overwhelmed if required to solve these problems.

Designing efficient I/O schemes for such I/O intensive problems demands expert knowledge and is not suitable for a computational scientist. Consequently, with the increasing number of applications that manipulate huge amounts of data, the effective data management problem is becoming increasingly important. Although this problem has been addressed throughout the years at different levels, there is still little consensus over how to balance the ease-of-use and efficiency.

In one extreme of the spectrum there are high-performance parallel file systems (e.g., Intel's PFS [25] and IBM's Vesta [8]) that have been built to exploit the parallel I/O capabilities provided by modern architectures. They achieve this goal by adopting smart I/O optimization techniques such as prefetching [17], caching [22, 5], and parallel I/O [15, 10]. However, there are serious obstacles preventing the file systems from becoming a real solution to the high-level data management problem. First of all, user interfaces of the file systems are low-level [21]. They force the users to express access patterns of their codes using file pointers, byte offsets, etc., which do not directly match the applications' data structures, which are large multi-dimensional arrays, images and so forth. Second, every file system comes with its own set of I/O calls, which renders ensuring program portability a very difficult task. The third problem with the file systems is that the file system policies and related optimizations are in general hard-coded in it and are tuned to work well for a few commonly occurring cases only. As noted by Karpovich et al. [19] among the others, even if the programmer has full-knowledge of access patterns of her code, it is difficult to convey this information to the file system in a convenient way. Overall, high level I/O performance from parallel file systems is possible only if significant sacrifices are made from portability, code reuse, and ease-of-programming. Notice also that although the parallel I/O libraries (e.g., [6]) built on top of parallel file systems have the potential to provide both ease-of-use and high performance through the use of advanced I/O optimization techniques such as collective I/O [14, 20], and array chunking [26], their extensibility is severely limited by the design principles and the programming language used in their implementation [19].

At the other end of the spectrum are database management systems (DBMS). They provide a layer on top of file systems, which is portable, extensible, easy to use and maintain, and that allows a clear and

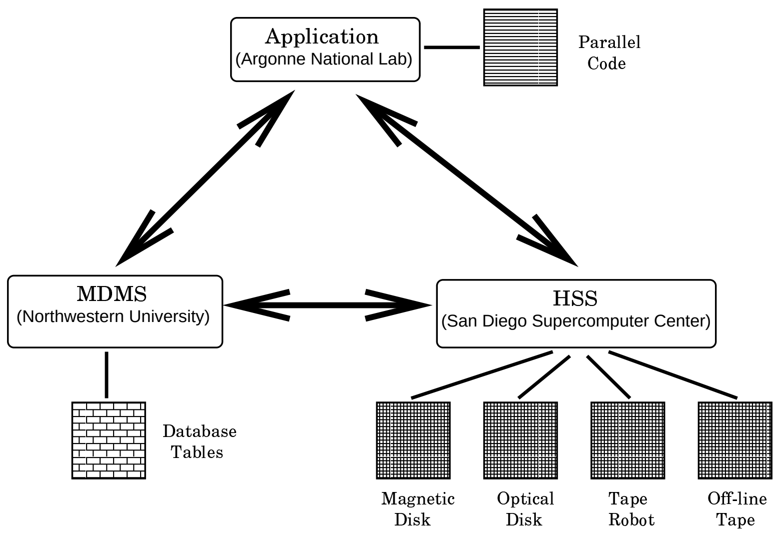

Figure 2: Three-tiered architecture (Three sites in this figure illustrate the current experimental setup to evaluate the architecture).

natural interaction with the applications by abstracting out the file names and file offsets. However, these advantages do not come for free. Since their main target is to be general purpose, they cannot provide high performance on a specific platform. Additionally, the data consistence and integrity semantics provided by almost all DBMS put an added obstacle to high performance. Applications that process large amounts of *read-only* data suffer unnecessarily as a result of these integrity constraints [19]. And finally, most DBMS support only a very limited set of data types and data manipulation models.

In this paper we present a novel approach to the high-level data management problem. Our 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: **(1)** user program; **(2)** meta-data management system (MDMS); and **(3)** hierarchical storage system (HSS). 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) [11] installed at San Diego Supercomputer Center. The experimental configuration is depicted in Figure 2. Functionalities of the thick double-arrows will be explained in subsequent sections.

The remainder of this paper is organized as follows. In section 2 we present the details of the MDMS which is built using Object-Relational DBMS (OR-DBMS) technology [27]. In section 3 we discuss the HSS, focusing in particular on the advanced I/O optimizations and on how they are activated using user directives. In section 4 we discuss basic I/O commands used in applications (i.e., the user interface of our architecture) and make a case for a simpler set of well-implemented I/O functionalities. In section 5 we present performance numbers from our initial implementation using several applications. In section 6 we briefly discuss related work on high-level scientific data management and I/O optimizations and in section 7 we conclude the paper and briefly discuss ongoing work.

2 Meta-data Management System (MDMS)

Our MDMS is an *active* middle-ware currently being built at Northwestern University with the aim of providing a uniform interface to data-intensive applications and hierarchical storage systems. Applications and HSS can communicate with the MDMS to obtain high performance I/O from the underlying architecture. The main functions fulfilled by the MDMS can be summarized as follows:

• It stores information about the abstract storage devices (ASDs) that can be accessed by applications. By querying the MDMS, the applications can learn where in the HSS their data reside (i.e., in what part of the storage hierarchy) without the need of specifying file names. They can also access the performance characteristics (e.g., speed) of the ASDs and select a suitable ASD (e.g., a disk sub-system consisting of eight separate disk arrays) to store their data sets.

• It stores information about the *storage patterns* and *access patterns* of data sets. For example, a specific multi-dimensional array that is striped across four disk devices in round-robin manner will have an entry in the MDMS. The MDMS utilizes this information in a number of ways. The most important usage of this information, however, is to *decide a parallel I/O method* based on *access patterns* (*hints*) provided by the application. By *comparing* the storage pattern and access pattern of a data set, the MDMS can, for example, advise the HSS to perform collective I/O [14] or prefetching [17] for the data set in question.

• It stores information about the *pending* access patterns. It utilizes this information in taking some global decisions, possibly involving data sets from multiple applications (e.g., staging a number of related files from tape sub-system to disk sub-system, or migrating a number of files from disk sub-system to tape sub-system [11]).

• It stores information about the users and applications. A (*user*, *application*) pair is called a *context*. When a user starts to run an application, the MDMS determines the portions of its meta-data that she is allowed to see. As the application runs, the MDMS accumulates access statistics about the user as well as her data sets and utilizes this information in successive runs to enhance the I/O performance further. In other words, the MDMS also keeps meta-data for specifying access history and trail of navigation, though they are not covered in this paper.

Notice that the MDMS is not merely a data repository but also an *active* component in the overall data management process. It *communicates* with applications as well as the HSS and can *influence* the decisions taken by the both.

2.1 Directive Categories

The MDMS is built using the OR-DBMS technology [27] that allows high expressiveness and extensibility. It keeps both *system-level* [4, 7] and *user-level meta-data* [16] about the data sets, data files, ASDs, physical storage devices (PSDs), access patterns, and users. Its communication with the user application is through so called *user directives.* Although these directives come in a variety of flavors, there are two important groups:

• *Layout Directives*: The application can have some control over how its data are laid out in the HSS. These directives are *strong* in the sense that (unless they are inconsistent with each other) the MDMS should take care of them and should advise the HSS to take necessary steps. An example would be a (storage pattern) directive that tells the MDMS that the application wants a specific data set to be stored in a particular fashion. Another example would be a (usage pattern) directive that tells the MDMS to advise the HSS to migrate a specific data set from disk to tape, probably because it will not be used in the remainder part of the application. These directives are important indications about particular uses of data in subsequent computations and in most cases either tell how a data set should be *created* (if the data set does not exist) in the HSS or how a data set should be *re-organized*, e.g., re-striped or re-distributed (if it exists already).

• *Access Pattern Directives*: These directives are used as *hints* which indicate that the user's application is about to start a specified sequence of I/O operations on the HSS. In response to such a directive the MDMS can, for example, advise the HSS to perform a specific I/O optimization in accessing the relevant data. These optimizations include prefetching, caching, collective I/O etc., and are *mild* in the sense that they do not imply a major data re-organization on the HSS part but rather enable a specific I/O optimization to be performed in order to reduce the I/O bottleneck.

Currently both types of directives are being implemented using embedded SQL (E-SQL) functions. It should be mentioned that a directive (whether strong or mild) may be rejected at either of two points. First, the MDMS may decline to take the necessary action due to, for example, the fact that the directives used together are not consistent with each other. Second, even if the MDMS advises an I/O optimization to the HSS, the HSS may reject it due to current state of it (e.g., overloaded). It should be stressed that the HSS is the only component in the proposed architecture that knows the details about its *physical* I/O resources and is free to take any action if doing so helps to improve the overall I/O performance.

One might wonder at this point why instead of using a MDMS (and incurring its overhead) the application code does not directly negotiate with the HSS. This is not a reasonable solution for at least two reasons. First, the application program does not need to know the details of the HSS. Otherwise, it would be very difficult to decide appropriate I/O optimizations. In the proposed architecture the user does not need to know where her data sets reside on the HSS and what their storage patterns are, though she can obtain this information by *querying* the MDMS. The second factor that prevents the user code from communicating directly with the HSS is the fact that a user's code in general cannot have a global information about the other applications concurrently using the HSS. In order to manage the overall I/O activity effectively it requires a *global knowledge* of all users' access patterns and I/O resources; that information is available as *meta-data* in the MDMS. It should be noted, however, the actual data transfer occurs always between the application and the HSS *directly* once the appropriate I/O method has been decided. It should also be mentioned that it is a viable alternative to implement the MDMS as a part of (or on top of) the HSS as long as the semantics of their interaction are preserved.

2.2 Individual Directives

Using directives, an application can convey information about its expected I/O activity to the MDMS. As a minimum, we expect the applications' user to know how her data will be used by parallel processors (henceforth we call this information *access pattern*). However, in general the more information is provided to the MDMS, the better I/O optimizations will be enabled. Table 1 shows the types of information that can be provided by application using directives to the MDMS. These directives can be combined in meaningful ways and can be applied to a number of data sets simultaneously as explained below.

In this environment, an access pattern for a data set is specified by indicating how the data set is to be divided and accessed by parallel processors. For example, an access pattern such as (BLOCK,*) says that the data set in question is divided (logically) into groups of rows and each group of rows will be *mostly* accessed by a single processor. The number of row-groups can also be specified using another directive. Notice that this access pattern information does *not* have to be very accurate, as the processors *may* occasionally access the data elements outside their assigned portions and the exact set of elements accessed will be specified in the actual read/write I/O calls. A few frequently used access patterns are depicted in Figure 3(a) for a four processor case. Each processor's portion are shaded using a different style. A (BLOCK,BLOCK) access pattern indicates that each processor will mostly access a rectangular block and a (*,CYCLIC) pattern involving P processors implies that each processor will mostly access every P^{th} column of the data set. A

Table 1: User directives.

directive explanation	*usage*
storage pattern directive declares a storage pattern for the HSS-resident data set A each <ptrn> can be BLOCK, CYCLIC, BLOCK-CYCLIC(b), or *	`organize` A(<ptrn>,<ptrn>,...)
access pattern directive declares an access pattern for the HSS-resident data set A each <ptrn> can be BLOCK, CYCLIC, BLOCK-CYCLIC(b), or *	`access` A(<ptrn>,<ptrn>,...)
abstract storage directive declares an abstract storage device (ASD) and the number of processors involved e.g. DISK(4,4) indicates a 4×4 processor array will access a disk storage	`storage` D(<np>,<np>,...)
I/O type directive declares the type of I/O that will be performed on data set A <type> can be read-only (RO), write-only (WO), or read-write mix (RW)	`iotype` A(<type>)
sequentiality directive informs about access pattern for each dimension of data set A each <seq,b> can be (sequential,*), (strided,B), or (variable,*), where B is the stride	`sqntl` A(<seq,b>,<seq,b>,...)
repetition directive informs about how many times data set A will be accessed <rep> can be only-once (OT), or multiple-times (MT)	`repeat` A(<rep>)
usage pattern directive informs about what to do with data set A *after* this point in program <usg> can be purge (PG) or migrate (MG)	`usage` A(<usg>)
association (abstract data set space) directive declares that data sets A,B,C,... are associated with T an association implies that the concerned data sets will be treated similarly	`associate`(A,B,C,...) `with` T
data set size directive declares an approximate size for data set A e.g., `size` A(16,777,216) indicates that data set A is approximately 16MB	`dsize` A(<size-in-bytes>)
request size directive declares an approximate size for data set A <rs> can be small request (SR), large request (LR), or variable request (VR)	`rsize` A(<rs>)
meta-data query directive queries a parameter of an entity (data set, ASD, association, etc.) e.g., query(dataset,A,storage-pattern) returns the storage pattern for data set A	`query`(entity,name,parameter)

star ‘*’, on the other hand, indicates that the dimension in question is not partitioned across processors; depending on the parameters of the accompanying `storage` directive, such an access pattern may imply either *replication* (if multiple processors are involved) or *exclusive access* (if a single processor is involved).

In our framework, these patterns is also used as *storage patterns*. For example, a (BLOCK,*) storage pattern corresponds to row-major storage layout (as in C), a (*,BLOCK) storage pattern corresponds to column-major storage layout (as in Fortran), and a (BLOCK,BLOCK) storage pattern, on the other hand, corresponds to blocked storage layout which might be useful for large-scale linear algebra applications whose data sets are amenable to blocking [30].

As an example, consider the following scenario. An I/O-intensive application executes in three steps manipulating five two-dimensional data sets (arrays) P, Q1, Q2, R1 and R2 whose default disk layouts are assumed to be *row-major* (BLOCK,*): **Step (1)**: a single processor reads the data set P and broadcasts its contents to other processors; **Step (2)**: the data set Q1 is created by four processors collectively in row-major order (BLOCK,*) on disk sub-system; also the data set Q2 is created by the same processors in (BLOCK,BLOCK) manner; and finally, **Step (3)**: two data sets, R1 and R2, are read by four processors collectively in row-major order from disk sub-system and then the application does some computation and terminates. In the following we show how these I/O activities can be specified using our user-level directives.
For Step (1), the I/O activity can be captured by the directive:

`access` P(BLOCK,*).

Here it is assumed that the data set P is accessed in row-wise; also since no `storage` directive appears, it is assumed a single processor.
For the I/O activity of Step (2), we can use

`organize` Q1(BLOCK,*) `storage` DISK(4)
`organize` Q2(BLOCK,BLOCK) `storage` DISK(2,2).

The user indicates that four processors will (most probably) write onto four disjoint parts of Q1 (i.e., four row-blocks). The system now has *several* options in response to this directive. It can, for example, use four disks and store each processor’s portion in a separate disk (as shown in Figure 3(b)). This will allow each processor to do I/O independently from others, maximizing the I/O parallelism. Or alternatively, the whole data set Q1 (actually the file contains it) can be striped over four disks (as shown in Figure 3(c), assuming that each processors’ portion contains four stripe units of data). Although this storage style does not necessarily lead to conflict-free disk accesses, in most practical cases it allows sufficient I/O parallelism. Also, more intelligent striping methods such as the one shown in Figure 3(d) can eliminate the potential disk conflicts.

As for the directive that involves Q2, again the system has a number of options. The most interesting one, however, is the one that uses collective I/O [14]. Since the *default* (the final) disk layout is row-major (BLOCK,*) and the processors will write data in (BLOCK,BLOCK) fashion a data re-organization between processors might be necessary for high performance. We will elaborate on this issue later in this paper.
Finally, for Step (3) the following directives can be used:

`associate` (R1,R2) `with` T
`access` T(BLOCK,*) `storage` DISK(4)

First, the `associate` directive indicates that R1 and R2 will be *treated together* (i.e., accessed, staged, migrated in similar fashions) as shown in Figure 3(e). Then the second directive conveys the access pattern. Note that since the storage pattern and the access pattern are the same, no collective I/O is required. An abstract data set space (T, in our example) is a dummy data set variable which helps to specify the desired access pattern of a data set by describing its *relationship* to other data sets. For example, ‘`associate` (A,B,C) `with` T’ implies that whenever the data set A is accessed, the data sets B and C will also *likely* be accessed. This information can then be used to pre-stage the data sets B and C from tape to disk (if they are not already on disk) whenever A is accessed. The `associate` directive also provides convenience in

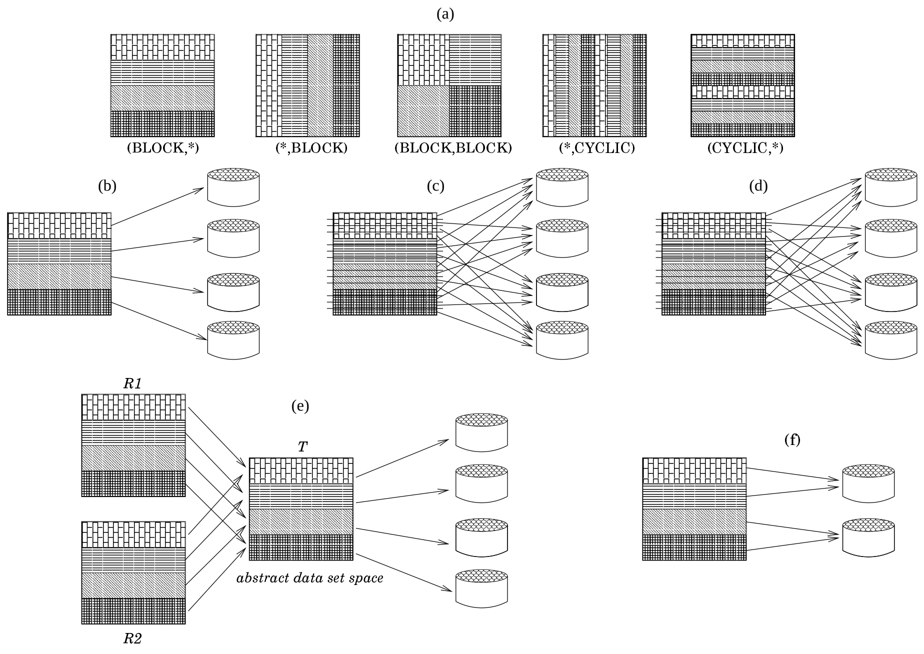

Figure 3: (a) Different access patterns. (b-d) Non-striping vs. striping. (e) Abstract data set space. (f) Contention on disks.

specifying the access pattern of data sets with respect to each other as in Step (3) of the scenario discussed above. It implies that the corresponding portions of R1 and R2 will be used (mostly) by the same processor, and therefore should *preferably* be stored on the same disk.

It should be emphasized that the main idea behind using directives is to help the system *match* the *access pattern* (i.e., how data is accessed) and the *storage pattern* (i.e., how data is stored). When, for example, a DISK(4) directive is sent to the MDMS, what the application program indicates is that four processors will access the data set in question in parallel from disk(s). While the best I/O parallelism can be obtained by allocating each processors' portion on a separate disk, the system does not necessarily has to do so. It should be noticed, though, in case of less than four disks (or I/O nodes) are used the I/O parallelism will suffer because of the possible *contention* on disks (as shown in Figure 3(f)). Thus, a DISK(4) directive essentially reveals to the system that for maximum I/O parallelism at least four disk devices are needed. It should also be noted that the directives explained above are *high-level* and constructed using the *names* of the data sets used in the application which are intuitive to the user. Contrast this with a classical file system interface that boils down everything to linear streams of bytes and offsets.

2.3 Shorthand Notations

In many cases there are a number of data sets that contain the same type of data differing only in the date that the data have been collected. As an example, a broadcast agency can use similar data sets for the broadcast data collected in different days. It can, for example, use the names like BC001, BC002,..., BC365 as data set names, where BC001 is the data set for the first day of the year and so on. In cases such as this it might be very convenient to have some wild-card notations in order to relieve the user from entering the same directive for each data set. Our current notation closely follows the one proposed by Musick et al. :

'*' The single star notation as in 'BC*' indicates that the operation in question should be applied to all data sets with prefix 'BC'. This is exactly the same way that the star is used in the UNIX file system.

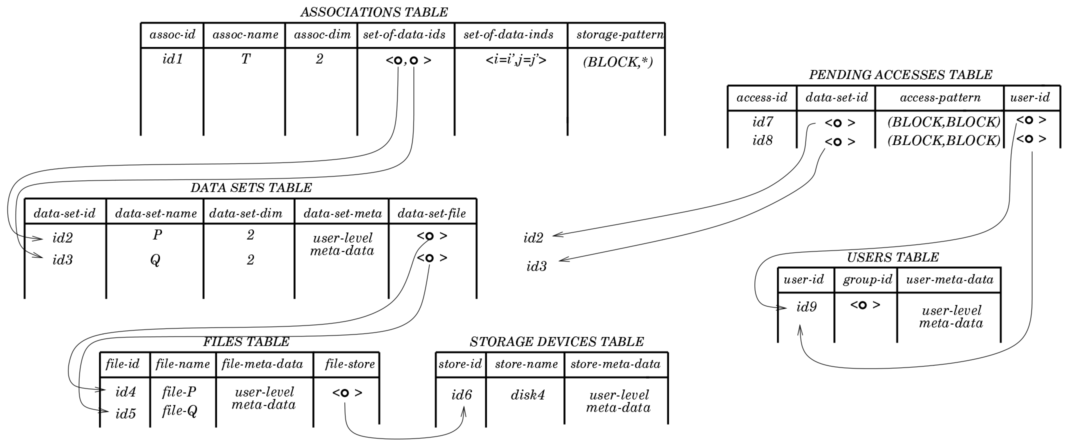

Figure 4: Internal representation in the MDMS.

For example, `associate` R* `with` T will cause all the data sets whose names starting with prefix R to get associated to the abstract data set space T.

'**' The double star notation connects a number of names in the same directive. For example, `associate` (R**,S**) `with` T** means that we want to apply associations to the instances of data sets whose names starting with R and S with *matching* suffixes. If we have the data sets with names R1, S1, R2, and S2, then the effect of the `associate` directive given above will be as if we entered `associate` (R1,S1) `with` T1 followed by `associate` (R2,S2) `with` T2.

2.4 Implementation of User Directives

Internally the MDMS keeps its meta-data in the form of database tables (relations). Figure 4 shows the most important parts for each table in our ongoing implementation. Using an OR-DBMS [27] instead of a pure relational DBMS brings the advantage of using pointers (hence avoiding duplication of meta-data in different tables) as well as extending meta-data as the need arises (using inheritance and/or collection data types [27] for instance). Notice that almost in every table there is a field (attribute) called *user-level meta-data.* Actually, such a field contains a pointer to a separate table where the MDMS stores the *user-level* meta-data (i.e., the meta-data that help user to find her data or to obtain information about the storage sub-systems currently available in the HSS). For example, the *user-level meta-data* field for a file entity can contain information (meta-data) on who created the file, when it was created, what its current size, when it was last modified etc.

The example meta-data entries shown in Figure 4 indicate that two data sets, P and Q, are associated with an abstract data set space T and are stored as (BLOCK,*) fashion (i.e., row-major) in files file-P and file-Q, respectively. These files reside on an ASD called disk4 (which, in turn, can be implemented using 4 physical disk devices). Also, there are two pending access patterns of style (BLOCK,BLOCK) on these data sets that have been initiated by a user whose identity is id9. It should be mentioned that the meta-data shown in Figure 4 do not contain all the data required but rather an important subset for illustrative purposes.

2.5 Common Access (Sharing) Patterns

We studied several I/O-intensive parallel programs to identify commonly occurring data access patterns, paying special attention to select a set of programs that clearly reflects the access behaviors when not a

Table 2: Commonly occurring access patterns.

pattern	*explanation*
read-parallel-mostly	this data set is repeatedly read by multiple processors significantly more frequently than it is written `access` A(...) + `storage` (...) + `iotype` A(RO) + `sqntl` A(...) + `repeat` A(MT)
read-serial-mostly	this data set is read by a single processor significantly more frequently than it is written `access` A(...) + `iotype` A(RO) + `sqntl` A(...) + `repeat` A(...) + `usage` A(...)
private (*exclusively accessed*)	this data set is accessed (read and/or written) exclusively by a single processor `access` A(...) + `iotype` A(...) + `sqntl` A(...) + `repeat` A(...) + `usage` A(...)
write-parallel-mostly	this data set is repeatedly written by multiple processors significantly more frequently than it is read `access` A(...) + `storage` (...) + `iotype` A(WO) + `sqntl` A(...) + `repeat` A(MT)
write-serial-mostly	this data set is written by a single processor significantly more frequently than it is read `access` A(...) + `iotype` A(WO) + `sqntl` A(...) + `repeat` A(...) + `usage` A(...)
producer-consumer	this data set is written once by a single processor and then read by multiple processors in producer part: `access` A(...) + `iotype` A(WO) + `sqntl` A(...) + `repeat` A(OT) + `usage` A(...) in consumers part: `access` A(...) + `storage` (...) + `iotype` A(RO) + `sqntl` A(...) + `repeat` A(MT)
read-write (*mix-shared*)	this data set is accessed (read and/or written) multiple times by multiple processors `access` A(...) + `storage` (...) + `iotype` A(RW) + `sqntl` A(...) + `repeat` A(MT)

special attention is paid to improve the high-level data accesses [18].

Although, as can be expected, there are great variances in access patterns of data sets used in I/O-intensive codes, we have identified a limited set of commonly occurring sharing patterns. Table 2 shows these patterns and how they can be specified by the users through our directives. A (...) means that a suitable parameter should be entered. If a directive is missing, that means it should *not* be entered. Although, our applications may not reflect all possible access patterns that may be exhibited by scientific computations, we believe that it would be relatively straightforward to capture any uncommon pattern as well using the appropriate combinations of the directives given in Table 1.

3 HSS and Utilization of Meta-data

Although in our future experiments we intend employ HPSS [11] as our primary HSS, any hierarchical storage system with a suitable API can be used for that purpose. Currently, we also use parallel file systems such as PFS and PIOFS to conduct experiments with the disk-resident data sets. Basically, the HSS in our environment has two main tasks:

• It keeps the storage related meta-data updated in the MDMS. This is important in order to present the users accurate information about the available I/O resources. In a way, the responsibility of updating the meta-data in the MDMS is *divided* between the application and the HSS. Any data re-organization performed by independently the HSS should be reflected on the MDMS.[1]

• It honors the I/O optimization requests from the MDMS and I/O requests from the user application and returns results to the application.

[1]In future, we intend to use the Datalinks software [13] from IBM. This will relieve us from the responsibility of updating independent HSS data re-organizations as all the I/O activity on the data sets registered with the DB2 will be intercepted and checked for security and consistency.

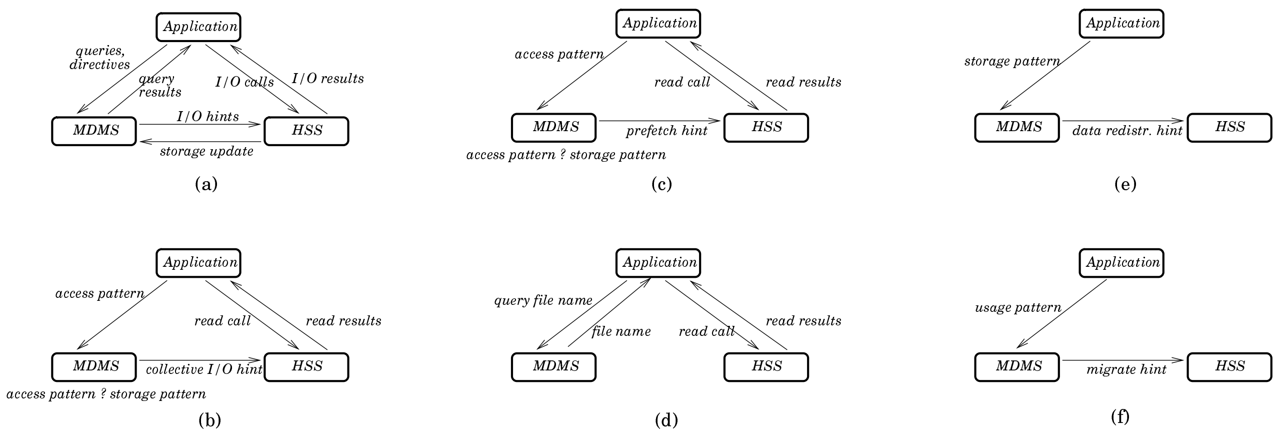

Figure 5: Proposed architecture (a) and different scenarios (b) through (f).

3.1 Example Scenarios

The sketch of the proposed architecture is shown in Figure 5(a). In this three-tiered architecture an application can query meta-data and send I/O directives to the MDMS. The MDMS, in turn, can send the application the query results and can send I/O hints to the HSS (after evaluating the directives it takes from the application). The HSS honors the I/O requests from the application and the I/O hints from the MDMS and send I/O results to the application (when required). It can also update the dynamic storage information kept in the MDMS (when it is necessary to do so). Figures 5(b) through (f) show five example scenarios. In Figure 5(b) the MDMS obtains an access pattern from the application and decides (after comparing it with the storage pattern information) to send a collective I/O hint to the HSS. Then the actual data transfer (which is a read call in this case) occurs directly between the application code and the HSS. Figure 5(c) depicts a similar situation, only this time the MDMS decides to send a prefetch hint (probably because the storage pattern and the access pattern in question are identical). Figure 5(d) shows so called *independent mode of operation* where the application in question negotiates with the MDMS and obtains the file names and their locations in the HSS. Then it accesses the HSS to perform its I/O; no negotiation occurs between the MDMS and the HSS. In Figure 5(e) the application demands a new storage pattern for one of its data sets and the MDMS sends the required data re-distribution hint to the HSS. And finally in Figure 5(f) the application sends a usage pattern which in turn causes the MDMS to instruct the HSS to migrate the data set in question from the disk sub-system to the tape sub-system. Note that all these interactions between the application and MDMS can be specified using the directives in Table 1.

3.2 Advanced I/O Optimizations

In order to exploit the capabilities of modern parallel I/O architectures, it is imperative to use advanced I/O techniques [15]. In principle, these techniques have two main objectives:

• *enhancing I/O parallelism;* that is, to maximize the number of storage units (e.g., disks) that can be kept busy at any given time interval, and

• *improving locality of I/O accesses;* that is, to access as many consecutive data as possible using as few I/O calls as possible (*spatial locality*) *or* to maximize the number of data accesses that can be satisfied from the fast components (i.e., higher levels) of the storage hierarchy (*temporal locality*).

Notice that these objectives can only be realized by careful data placement across storage devices and by careful computation decomposition across processors. Throughout the years several I/O techniques have been designed and implemented [15]. Table 3 briefly summarizes the optimization techniques currently

Table 3: I/O optimization techniques and the corresponding trigger rules.

optimization	*brief explanation*	*suggested if*
parallel I/O	performing I/O using a number of processors in parallel to improve the bandwidth	*access ptrn* $\neq$ (*,*,...,*) $\vee$ $P \neq 1$
collective I/O	distributing the I/O requests of different processors among them so that each processor accesses as many consecutive data as possible it involves some extra communication between processors	*access ptrn* $\neq$ *storage ptrn* $\wedge$ $P \neq 1$
sequential prefetching	bringing consecutive data into higher levels of storage hierarchy before it is needed. it helps to overlap the I/O time and computation time, thus hiding the I/O latency	SQ $\wedge$ RO $\wedge$ SR
strided prefetching	same as sequential prefetching except that data is brought in fixed strides (some elements are skipped)	ST $\wedge$ RO $\wedge$ SR
caching & replacement policy	keeping the data to be used in near future in the higher levels of storage hierarchy (currently two replacement policy is used (LRU-least recently used and MRU-most recently used)	SQ $\wedge$ RO $\wedge$ SR $\wedge$ MT $\rightarrow$ LRU SQ $\wedge$ WO $\wedge$ SR $\wedge$ MT $\rightarrow$ MRU
setting striping unit size	to select a striping unit such that as many storage devices (e.g., disks) as possible will be utilized	`dsize` used
data migration	migrating data from higher levels of storage hierarchy (e.g., disks) to lower levels of storage hierarchy (e.g., tapes)	MG $\wedge$ $\overline{\text{MT}}$ $\wedge$ $\overline{\text{OT}}$
data purging	removing data from the storage hierarchy useful for temporal files whose lifetime is over	PG $\wedge$ $\overline{\text{MT}}$ $\wedge$ $\overline{\text{OT}}$
pre-staging	fetching data from tape sub-system to disk sub-system before it is required	`associate` used
disabling cache & prefetch	used when the benefit of caching or prefetching is not clear	LR

employed by our proposed framework. More detailed descriptions can be found in respective references cited at the end of this paper; here we only discuss collective I/O.

In many parallel applications, the storage pattern of a data set is in general different from its access pattern. The problem here is that if each processor attempts to read its portion of data (specified in the access pattern), it may need to issue a large number of I/O calls. Suppose that four processors want to access (read) a two-dimensional array in (BLOCK,BLOCK) fashion. Assuming that the arrays' storage pattern is (BLOCK,*), each processor will have to issue many I/O calls (to be specific $N/2$ read calls each for $M/2$ consecutive data items, if we assume that the array is $N \times M$). What collective I/O does, instead, is to read the data in (BLOCK,*) fashion (i.e., using minimum number of I/O calls) and then *re-distribute* the data across processors' memories to obtain the desired (BLOCK,BLOCK) pattern. That is, taking the advantage of knowing the access and storage pattern of the array, we can realize the desired access pattern in two phases. In the first phase, the processors access the data in a *layout conformant* way (i.e., (BLOCK,*) in our example), and in the second phase they *re-distribute* the data in memory among themselves such that the desired access pattern is obtained. Considering the fact that I/O in large-scale computations is much more costly than communication, huge performance improvements can be achieved through collective I/O.

The last column in Table 3 gives the conditions under which the respective optimizations will be suggested by the MDMS to the HSS. For example, collective I/O is considered only if *access pattern* $\neq$ *storage pattern* and multiple processors are involved (P denotes the number of processors). The symbols $\vee$ and $\wedge$ are used for *logical or* and *logical and* operations, respectively. SQ denotes 'sequential' and 'ST' means 'strided'; other abbreviations are from Table 1.

3.3 Tape-Specific I/O Optimizations

Another important constituent of our I/O optimization framework is a run-time library that can be used to facilitate the explicit control of data flow for tape-resident data. Our library is activated automatically when the user invokes an I/O call that involves tape-resident data sets (see the next section). Alternatively, this run-time library can be directly used by application programmers and optimizing compilers that manipulate large-scale tape-resident data. The objective is to allow programmers to access data located on tape via a convenient interface expressed in terms of arrays and array portions (regions) rather than files and offsets. The library implements a data storage model on tapes that enables our architecture to access portions of multi-dimensional data in a fast and simple way. In order to eliminate most of the latency in accessing tape-resident data, we employ a *sub-filing strategy* [23] in which a large multi-dimensional tape-resident *global* array is stored not as a single file but as a number of smaller *sub-files*, whose existence is transparent to the programmer. The main advantage of doing this is that the data requests for relatively small portions of the global array can be satisfied without transferring the entire global array from tape-subsystem to disk-subsystem in the HSS as is customary in many hierarchical storage management systems. In addition to read/write access routines, the library also supports pre-staging and migration capabilities which can prove very useful in environments where the data access patterns are predictable and the amount of disk space is limited.

Within the library, each *global tape-resident* array is divided into *chunks*, each of which is stored in a *separate sub-file* on tape. The chunks are of equal sizes in most cases. A typical access pattern might access a small portion of a very large tape-resident data set. 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* [23]. Assuming for now that all of the sub-files that make up the cover are currently residing on tape, in the second step, the library brings these 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 experiments show that this computational overhead is not too much.

4 User Interface

One of the main problems with current parallel file systems and parallel I/O libraries is the excessive number of functions (calls) presented to the user. Then it becomes the task of user to choose the suitable I/O functions that express her access patterns as closely as possible. For example, in the latest MPI-IO standard [9], there are over 30 read/write calls alone which renders the job of selecting the right ones a daunting task. We believe that a majority of these calls can be eliminated if the user is allowed to express access patterns using *directives* at a higher level.

4.1 Supported I/O Functions

In our initial implementation (which targets only scientific codes that use large multi-dimensional arrays) we support the functions shown in Table 4. Notice that these commands are *different from user directives* and are the only commands that can be sent to the HSS directly from the application code. Queries about data sets and storage devices are performed by negotiating with the MDMS through directives. Notice however that the use of directives is optional. Contrast this with the current file system and I/O library interfaces which demand that each and every parameter in the parameter-list of the command should be supplied by the user. In Table 4 *name* can be a data set name or name of an abstract data set space, in which case all the associated data sets are opened. The data in memory is specified by *buffer* that can be either a pointer or a multi-dimensional memory region (e.g., an array). It is assumed that each involved processor will have enough space in their respective *buffer* areas in order to hold its portion of data accessed. The *portion* parameter, on the other hand, denotes the region of data set to be accessed in the global name space; the '*' symbol is used to denote the 'whole data set'.

The *opt* parameter is the *optimization pointer* that is set mainly by the MDMS depending on the directives collected so far from the application. It points to a structure that contains sufficient information to carry out an I/O optimization. Currently we are in the process of implementing these high-level functions on top of MPI-IO [9] and SRB (Storage Resource Broker) [1] from San Diego Supercomputer Center (SDSC). The Storage Resource Broker (SRB) is a middle-ware that provides distributed clients with uniform access to diverse storage resources in a distributed heterogeneous computing environment. We are experimenting with the MPI-IO to evaluate the optimizations involving mainly disk-resident data in parallel file systems and with the SRB to evaluate the optimizations involving tape-resident data.

4.2 Examples for Use of Directives and I/O Calls

Consider the following example.

```
OpenDataSet(P,opt)
access P(*,BLOCK) storage DISK(8)
ReadDataSet(P,bf,*,opt)
```

In this example, the application first opens the data set (here the array P). It also gives an optimization pointer *opt* later to be used. Then it sends an `access` directive to the MDMS which declares that 8 processor will access the data set in a column-major fashion. The MDMS, in turn, compares the storage pattern (the default is (BLOCK,*)) with this access pattern and decides that *collective I/O* needs to be performed. It passes this advice to the HSS by filling out the relevant entries of the data structure pointed by *opt*. When, later, the application issues the `ReadDataSet` command, a collective read operation might be performed (considering the contents of the structure pointed by *opt* and exact parameters in the `ReadDataSet` command). Now suppose that the directive was instead `access` P(BLOCK,*) `storage` DISK(8). In that case since the access pattern and the storage pattern are the same, the MDMS may advise *prefetching* to the HSS by setting the appropriate entries of the structure pointed by *opt*. Notice that in either case the syntax of the actual read call does not change. The only difference is the contents of the data structure pointed by *opt*. Considering the fact that a typical large-scale applications will have only a few directives, the function performed by the application in question can be changed by modifying only a few program lines. This helps readability, reusability, and maintainability.

Let us now consider the following example fragment.

```
associate (P,Q) with T
access T(BLOCK,*) storage TAPE(16)
OpenDataSet(P,opt)
ReadDataSet(P,bf1,*,opt)
```

Table 4: Supported I/O functions.

`OpenDataSet`(*name*,*opt*)
`CloseDataSet`(*name*,*opt*)
`ReadDataSet`(*name*,*buffer*,*portion*,*opt*)
`WriteDataSet`(*name*,*buffer*,*portion*,*opt*)

```
ReadDataSet(Q,bf2,*,opt)
```

In this case, the application first associates two tape-resident arrays with an abstract data set space T. Then the `access` directive indicates that 16 processors are going to access the respective portions of P and Q in row-wise. The application afterwards opens P, an activity which most probably forces the HSS to stage the data set P from tape sub-system to disk sub-system. This also triggers the MDMS to advise the HSS to *pre-stage* the data set Q as well from tape to disk as this array is associated with P and most probably the two arrays will be used together. Assuming that the default layout is row-major, the MDMS also sets the necessary parameters in the structure pointed by *opt* to enable prefetching; no collective I/O is required. Consequently, the following two calls to `ReadDataSet` take advantage of prefetching. Notice that our directives allow two levels of prefetching in this small example: first (through the use of `associate`) from tape to disk (this is called pre-staging), and then (through the use of `access`) from disk to memory (this is called prefetching). Also, note that (when necessary) the `ReadDataSet` call uses sub-filing to stage only the relevant parts of data from tape to disk.

Finally, consider the following example in which the access pattern of a data set P changes during the course of execution. In this example, there are 32 processors involved.

```
OpenDataSet(P,opt)
access P(BLOCK,*) storage DISK(32)
ReadDataSet(P,bf1,*,opt)
organize P(*,BLOCK) storage DISK(32)
access P(*,BLOCK)
ReadDataSet(P,bf2,*,opt)
WriteDataSet(P,bf3,*,opt)
WriteDataSet(P,bf4,*,opt)
```

The application first opens the data set P, and then discloses that its access pattern is row-major. Assuming again the row-major storage layout as default, no collective I/O is required. But then, the application issues an `organize` directive which strongly advises a storage layout change (on disks) from the default row-major layout to column-major layout. The reason for this becomes clear when we take a look at the next `access` directive in the sequence which says that the remaining `ReadDataSet` and `WriteDataSet` commands will access the data in column-major fashion. If the layout of data is not changed from row-major to column-major, each of the remaining three I/O calls will have to use collective I/O; that involves interprocessor communication which (depending on the type of the network and distributed media) can be costly in this case as 32 processors are involved. Therefore, the programmer thinks that it might be better to store the data in the way that it will be accessed later. That is, storing the data as column-major, the last three calls can take the advantage of prefetching as the access pattern and the storage pattern are identical now. Notice, however that the use of the `organize` directive for a data set that has already been created (as in this example) implies a *data re-distribution on disks* and can be quite costly, so it should be used with discretion. Similar to the tables used to keep track of user directives (see Figure 4), we use data base tables to hold necessary information about the read/write calls.

Table 5: Total I/O times (in seconds) for Astro-2D application (Data set size is 256 MB).

	Paragon		SP-2	
	64 procs	128 procs	32 procs	64 procs
Original	64.95	87.02	23.46	39.67
Optimized	27.44	49.37	14.05	11.23

Table 6: Total I/O times (in seconds) for Astro-3D application (Data set size is 8 MB).

	Paragon		SP-2	
	64 procs	128 procs	32 procs	64 procs
Original	51.04	81.45	109.93	211.47
Optimized	36.41	68.02	3.33	3.51

5 Experiments

In this section we present some performance numbers from an ongoing implementation of our MDMS. The experiments were run on an IBM SP-2 and Intel Paragon. Each node of SP-2 is RS/6000 Model 390 with 256 megabytes memory and has an I/O sub-system containing four 9 gigabytes SSA disks attached to it. The nodes on Paragon (Intel i860 XP), on the other hand, are divided into three groups: compute nodes, HIPPI nodes and service nodes. The total memory capacity of the compute partition is around 14.4 gigabytes. The platform where the experiments were conducted has 3 service nodes each with a RAID SCSI disk array attached to it.

We used four different applications: three of them are used to measure the benefits of collective I/O for disk-resident data sets in parallel file systems; the last one is used to see how pre-staging performs in HPSS [11] for tape-resident data. We also performed experiments with sub-filing using synthetic access patterns and present here the first results from these experiments.

Table 5 shows the total I/O times for a 2D astrophysics template (Astro-2D) on the Intel Paragon and IBM SP-2. Here, ‘Original’ refers to the code without collective I/O, and ‘Optimized’ denotes the code with collective I/O. In all cases, the MDMS is run at Northwestern University. The important point here is that in both the ‘Original’ and the ‘Optimized’ versions the user code is essentially the same; the only difference is that in the ‘Optimized’ case, the user code sends *directives* to the MDMS. The MDMS then automatically determines that collective I/O should be performed; this hint is then sent by the MDMS to the HSS. As a result, impressive reductions in I/O times are observed. Since the number of I/O nodes are fixed on both the Paragon and the SP-2, increasing the number of processors causes in general an increase in the I/O times. Tables 6 and 7 report similar results for a 3D astrophysics code (Astro-3D) and for an unstructured (irregular

Table 7: Total I/O times (in seconds) for unstructured code (Data set size is 64 MB).

	Paragon		SP-2	
	64 procs	128 procs	32 procs	64 procs
Original	76.30	142.73	547.61	488.13
Optimized	1.68	0.94	1.25	2.13

Table 8: Total I/O times (in seconds) for volume rendering code on 4 processors (Data set size is 64 MB).

File No →	1	2	3	4
Original	31.18	19.20	61.86	40.22
Optimized	11.90	11.74	20.10	18.38

Table 9: Total I/O times (in seconds) for volume rendering code on 8 processors (Data set size is 64 MB).

File No →	1	2	3	4
Original	18.79	37.69	21.02	14.70
Optimized	10.74	6.23	4.49	6.42

data access pattern) code, respectively. Since these codes have not been modified yet to work through the MDMS, the results reported here are obtained by hand. Nevertheless, they indicate the order of potential savings once collective I/O is used.

Our next example is a parallel volume rendering application. As in previous experiments, the MDMS is run at Northwestern University. The application itself, on the other hand, is executed at Argonne National Lab's SP-2 and the HPSS at San Diego Supercomputer Center (SDSC) is used as the HSS. In the 'Original' code four data files are opened and parallel volume rendering is performed. In the 'Optimized' code ('Original' code + user directives) the four data sets (corresponding to four data files) are associated with each other, and pre-staging (from tape to disk) is applied for these data sets. Tables 8 and 9 give the total read times for each of the four files for the 'Original' and 'Optimized' codes for 4 and 8 processor case, respectively. The results reveal that, for both 4 and 8 processor cases, pre-staging reduces the I/O times significantly. We need to mention that in every application we experimented with the time spent by the application in negotiating with the MDMS is less than 1 second. When considering the fact that for large-scale applications I/O times are likely to be huge (even when optimized), an overhead in this range is acceptable.

In order to measure the usefulness of sub-filing, we performed experiments using the HPPS at 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 middle-ware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated data sets [1].

We experimented with different *access patterns* in order to evaluate the benefits of the library that implements sub-filing. Table 10 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).

The columns 5 through 10 of Table 10 show 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 6 and 7 show the results obtained in graphical form. Note that the y-axes on the figures are *logarithmically scaled.*

Table 10: Access patterns, execution times, and percentage gains.

	Pattern Information			Write Operations			Read Operations		
Acc. Ptr.	Start Coor.	End Coor.	To tal float. points	Times w/o chunking	Small Chunk Gain (%)	Large Chunk Gain (%)	Times w/o chunking	Small Chunk Gain (%)	Large Chunk Gain (%)
A	$(0,0)$	$(1000,1000)$	$1*10^6$	2774.0	96.1	94.5	784.7	85.2	77.1
B	$(0,0)$	$(4000,1000)$	$4*10^6$	2805.9	83.8	84.9	810.1	43.2	55.6
C	$(0,0)$	$(24000,1000)$	$24*10^6$	2960.3	8.8	37.9	793.3	-240.5	-172.4
D	$(5000,5000)$	$(6000,6000)$	$1*10^6$	3321.2	96.7	95.4	798.4	84.1	79.7
E	$(0,0)$	$(50000,80)$	$4*10^6$	151.7	-3525.1	-2437.6	165.2	-3229.3	-2623.9
F	$(0,0)$	$(80,50000)$	$4*10^6$	138723.3	96.0	97.2	39214.1	85.9	88.5
G	$(0,0)$	$(1000,4000)$	$4*10^6$	11096.3	95.9	96.4	3242.9	88.3	88.6
H	$(6000,6000)$	$(8000,8000)$	$4*10^6$	5095.2	91.2	96.5	1612.9	76.6	89.9

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 volume of data. Consequently, the impact of sub-filing is more pronounced in G. In E, the naive strategy outperforms the sub-filing as the entire portion can be accessed in a single I/O call. Note that the 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. 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.

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. Even in that case a suitable chunk shape can be chosen to match the response time of the naive strategy. However, automatic selection of optimal chunk sizes (considering the future access patterns) is beyond the scope of this work.

6 Related Work

There are many proposed techniques for optimizing I/O accesses. These techniques can be divided into three main groups: the parallel file system and run-time system optimizations [21, 6, 9, 17, 19, 14], compiler optimizations [3, 18, 15], and application analysis and optimization [18, 5, 24, 15].

The closest work to ours is the one done by Brown et al. [4]. They propose a similar architecture to ours; however, they do not handle the advanced I/O optimizations proposed in this paper. They build their meta-data system on top of HPSS using DB2 from IBM. In our case, the HSS and the MDMS are loosely coupled allowing us to experiment with different hierarchical storage systems.

Baru et al. [1] investigate use of high-level unified interfaces to data stored on file systems and DBMS. Their system maintains meta-data for data sets, resources, users, and methods (access functions) and provides the ability to create, update, store, and query this meta-data. While the type of meta-data maintained

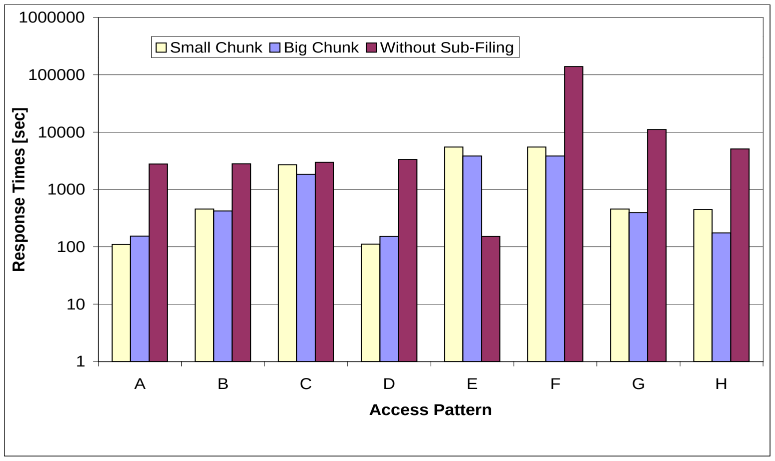

Figure 6: Execution times for **write** operations.

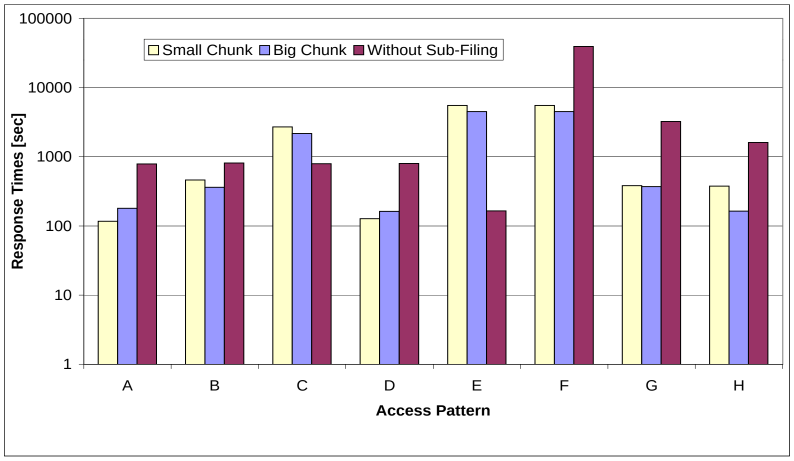

Figure 7: Execution times for **read** operations.

by them is an extension of meta-data maintained by a typical operating system, our meta-data involves *performance-related meta-data* as well which enables automatic high-level I/O optimizations as explained in this paper.

There are several works on I/O characterization of large applications. Three I/O-intensive applications from the Scalable I/O Initiative Application Suite are studied in [12]. An I/O-intensive three-dimensional parallel application code is used to evaluate the I/O performances of the IBM SP and Intel Paragon in [28]. They found IBM SP to be faster with read operations and Paragon for writes. del Rosario and Choudhary [15] discuss several grand-challenge problems and the demands that they place on the I/O performance of parallel machines. The characterization information can be very useful in our framework in selecting suitable user-level directives to implement in order to capture access patterns in a better manner. Finally, some amount of work has been done in the area of out-of-core compilation [3], [2].

7 Conclusions

In this paper we present a program development environment based on maintaining performance-related system-level meta-data. This environment consists of a user code, a meta-data management system (MDMS), and a hierarchical storage system (HSS) and provides a seamless data management and manipulation facility for use by large-scale scientific applications. It combines the advantages of file systems and DBMSs without incurring their respective disadvantages and provides location transparency (through the use of data set names rather than file names or URLs), resource transparency (through the use of ASDs), and access function transparency (through the automatic invocation of high-level I/O optimizations like collective I/O and data sieving).

Also, by storing meta-data and providing means to manipulate it, our framework is able to manage distributed resources in a heterogeneous environment. Preliminary results obtained using several applications are encouraging and motivate us to complete our implementation and make extensive experiments using large-scale data intensive applications. Currently, we have only a single API (which can be used from within the application code); we are also working on GUI and UNIX-style command-line interfaces.

Acknowledgments

This research was supported by Department of Energy under the Accelerated Strategic Computing Initiative (ASCI) Academic Strategic Alliance Program (ASAP) Level 2, under subcontract No W-7405-ENG-48 from Lawrence Livermore National Laboratories. We would like to thank Reagan Moore for discussions and help with SDSC resources. We thank Mike Wan and Mike Gleicher for helping us with the implementation of the volume rendering code and in understanding the SRB and the HPSS. We thank Larry Schoof and Wilbur Johnson for providing the unstructured code used in this paper. We thank Celeste Matarazzo, John Ambrosiano, and Steve Louis for discussions and their input. Finally, we gratefully acknowledge NPACI for allowing us to use their machines at the SDSC.

References

[1] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proc. CASCON'98 Conference,* Dec 1998, Toronto, Canada.

[2] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and implementation of primitives for Parallel I/O, In *Proceedings of Supercomputing'93,* November 1993.

[3] 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 Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995.

[4] P. Brown, R. Troy, D. Fisher, S. Louis, J. R. McGraw, and R. Musick. Meta-data sharing for balanced performance. In *Proc. the First IEEE Meta-data Conference*, Silver Spring, Maryland, 1996.

[5] 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.

[6] 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*, Sept 1994.

[7] A. Choudhary and M. Kandemir. System-level meta-data for high performance data management. To appear in *Proc. the Third IEEE Meta-Data Conference*, Bethesda, Maryland, April 6–7, 1999.

[8] 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.

[9] 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 Parallel and Distributed Systems*, IPPS'95, Santa Barbara, CA, April 1995.

[10] T. H. Cormen and D. M. Nicol. Out-of-core FFTs with parallel disks. *ACM SIGMETRICS Performance Evaluation Review*, 25:3, December 1997, pp. 3–12.

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

[12] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing'95.*

[13] J. R. Davis. Datalinks: Managing external data with DB2 universal database. IBM Corporation White Paper, August, 1997.

[14] 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 Parallel Computer Systems* , April 1993.

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

[16] M. Drewry, H. Conover, S. McCoy, and S. Graves. Meta-data: quality vs. quantity. In *Proc. the Second IEEE Meta-data Conference*, Silver Spring, Maryland, 1997.

[17] C. S. Ellis and D. Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proc. the 1989 International Conference on Parallel Processing*, pages I:306–314, St. Charles, IL, August 1989. Pennsylvania State Univ. Press.

[18] M. Kandaswamy, M. Kandemir, A. Choudhary, and D. Bernholdt. Performance implications of architectural and software techniques on I/O-intensive applications. In *Proc. the International Conference on Parallel Processing (ICPP'98)*, Minneapolis, MN, Aug 1998.

[19] 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 Programming Systems, Languages, and Applications*, pp. 191–204, Oct 1994.

[20] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, Nov 1994.

[21] D. Kotz. Multiprocessor file system interfaces. In *Proc. the Second International Conference on Parallel and Distributed Information Systems*, pages 194–201. IEEE Computer Society Press, 1993.

[22] T. Madhyastha and D. Reed. Intelligent, adaptive file system policy selection. In *Proc. Frontiers of Massively Parallel Computing*, Annapolis, MD, pp. 172–179, Oct 1996.

[23] G. Memik, M. Kandemir, and A. Choudhary. A run-time library for tape resident data. *Technical Report CPDC-TR-9909-014,* Center for Parallel and Distributed Computing, Northwestern University, September 1999.

[24] 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.

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

[26] K. E. Seamons and M. Winslett. Multidimensional array I/O in Panda 1.0. *Journal of Supercomputing*, 10(2):191–211, 1996.

[27] M. Stonebraker. *Object-Relational DBMSs : Tracking the Next Great Wave*. Morgan Kaufman Publishers, ISBN: 1558604529, 1998.

[28] R. Thakur, W. Gropp, and E. Lusk. An experimental evaluation of the parallel I/O systems of the IBM SP and Intel Paragon using a production application. In *Proc. of the 3rd Int'l Conf. of the Austrian Center for Parallel Computation (ACPC) with special emphasis on Parallel Databases and Parallel I/O,* September 1996. Lecture Notes in Computer Science 1127, Springer-Verlag, pp. 24–35.

[29] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. To appear in *Proc. the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.

[30] 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. Fourth Annual Workshop on I/O in Parallel and Distributed Systems*, May 1996.