A Novel Application Development Environment for Large-Scale Scientific Computations

X. Shen, W. Liao, A. Choudhary, G. Memik, M. Kandemir,* S. More, G. Thiruvathukal,† and A. Singh†

Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208 {xhshen,wkliao,choudhar,memik,ssmore}@ece.nwu.edu

Abstract

Effective high-level data management is becoming an important issue with more and more scientific applications manipulating huge amounts of secondary-storage and tertiary-storage data using parallel processors. A major problem facing the current solutions to this data management problem is that these solutions either require a deep understanding of specific data storage architectures and file layouts to obtain the best performance. In this paper, we discuss the design, implementation, and evaluation of a novel application development environment for scientific computations. This environment includes a number of components that make it easy for the programmers to code and run their applications without much programming effort, and at the same time, to harness the available computational and storage power on parallel architectures. Embarking on this ambitious goal, we first present a performance-oriented meta-data management system that governs data flow between storage devices and applications. Another component of our environment is a data analysis and visualization tool which has been integrated with the meta-data management system, storage subsystem, and user applications. We also present an automatic code generator component (ACG) to help users utilize the information in the meta-data management system when they are developing new applications. All these components are tied together using an integrated Java graphical user interface (IJ-GUI) through which the user can launch her applications, can query the meta-data management system to obtain accurate information about the datasets she is interested in and about the current state of the storage devices, and can carry out data analysis and visualization, all in a unified framework. Finally, we present performance numbers from our initial implementation. Our results demonstrate that our novel application development environment provides both ease-of-use and high performance for large-scale, I/O-intensive scientific applications.

1 Introduction

Effective data management is a crucial part of the design of large-scale scientific applications. An important subproblem in this domain is to optimize the data flow between parallel processors and several types of storage devices residing in a storage hierarchy. While a knowledgeable user can manage this data flow by exerting a great effort, this process is time-consuming, error-prone, and

not portable.

To illustrate the complexity of this problem, we consider a typical computational science analysis cycle, shown in Figure 1. As can be seen easily, in this cycle, there are several steps involved. These include mesh generation, domain decomposition, simulation, visualization and interpretation of results, archiving of data and results for post-processing and check-pointing, and adjustment of parameters. Consequently, it may not be sufficient to consider simulation alone when determining how to store or access datasets because these datasets are used in other steps as well. In addition, these steps may need to be performed in a heterogeneous distributed environment and the datasets in question can be persistent on secondary or tertiary storage. Among the important issues in this analysis cycle are detection of I/O access patterns for data files, determination of suitable data storage patterns, and effective data analysis and visualization.

Obviously, designing effective I/O strategies in such an environment is not particularly suitable for a computational scientist. To address this issue, over the years, several solutions have been designed and implemented. While each of these solutions is quite successful for a class of applications, we feel that the growing demand for large-scale data management necessitates novel approaches that combine the best characteristics of current solutions in the market. For example, parallel file systems [10, 29, 8] might be effective for applications whose I/O access patterns fit a few specific forms. They achieve impressive performance for these applications by utilizing smart I/O optimization techniques such as prefetching [18], caching [23, 6], and parallel I/O [16, 11]. However, there are serious obstacles preventing the parallel file systems from becoming a global solution to the data management problem. First of all, user interfaces of the file systems are in general low-level [21], allowing the users to express access patterns of their applications using only low-level structures such as file pointers and byte offsets. Second of all, nearly every file system has its own suite of I/O commands, rendering the process of porting a program from one machine to another a very difficult task. Third, the file system policies and optimization parameters are in general hard-coded within the file system and, consequently, work for only a small set of access patterns. While runtime systems and libraries like MPI-IO [9, 33] and others [35, 3, 7] present users with higher-level, more structured interfaces, the excessive number of calls to select from, each with several parameters, make the user's job very difficult. Also, the usability of these libraries depends largely on how well user's access patterns and library calls' functionality match [20].

An alternative to parallel file systems and runtime libraries is database management systems (DBMS). They present a high-level, easy-to-use interface to the user and are portable across a large number of systems including SMPs and clusters of workstations. In fact, with the advent of object-oriented and object-relational databases [31], they also have the capability of handling large datasets such as multidimensional arrays and image/video files [14]. A major obstacle in front of DBMS (as far as the effective high-level data management is concerned) is the lack of powerful I/O optimizations that can harness parallel I/O capabilities of current multiprocessor architectures. In addition to that, the data consistence
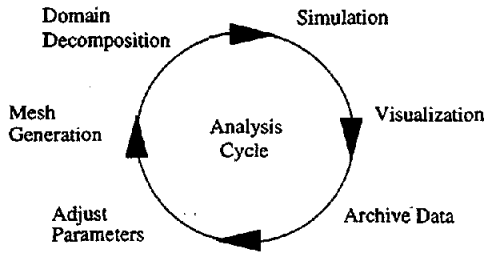
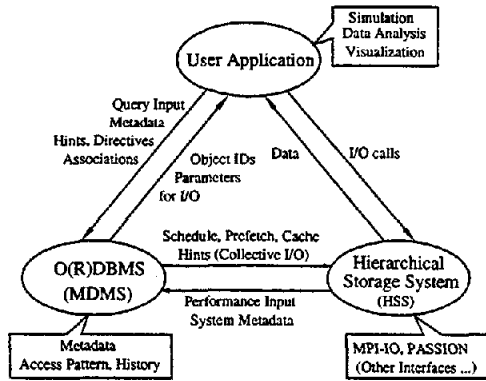Figure 1: A typical computational science analysis cycle.

Figure 2: Three-tiered architecture.

and integrity semantics provided by almost all DBMS put an added obstacle to high performance. Finally, although hierarchical storage management systems (e.g., [36]) are effective in large-scale data transfers between storage devices in different levels of a storage hierarchy, they also, like parallel file systems and DBMS, lack application specific access pattern information, and consequently, their I/O access strategies and optimizations are targeted at only a few well-defined access and storage patterns.

In this paper, we present a novel application development environment for large-scale scientific applications that manipulate secondary storage and tertiary storage resident datasets. Our primary objective is to combine the advantages of parallel file systems and DBMS without suffering from their disadvantages. To accomplish this objective, we designed and implemented a multi-component system that is capable of applying state-of-the-art I/O optimizations without putting an excessive burden on users. Embarking on this ambitious goal, in this paper, we make the following contributions:

- We present a *meta-data management system*, called MDMS, that keeps track of I/O accesses and enables suitable I/O strategies and optimizations depending on the access pattern information. Unlike classical user-level and system-level meta-data systems [17, 27], the main reason for the existence of MDMS is to keep performance-oriented meta-data and utilize these meta-data in deciding suitable I/O strategies.

- We explain how the MDMS interacts with parallel applications and hierarchical storage systems (HSS), relieving the users from the low-level management of data flow across multiple storage devices. In this respect, the MDMS plays the role of an easy-to-use interface between applications and HSS.

- We present a tape device-oriented optimization technique, called *subfiling*, that enables fast access to small portions of tape-resident datasets and show how it fits in the overall application development environment.

- We illustrate how data analysis and visualization tools can be integrated in our environment.

- We propose an automatic code generator component (ACG) to help users utilize the meta-data management system when they are developing new applications.

- We present an integrated Java graphical user interface (IJ-GUI) that makes the entire environment virtually an easy-to-use control platform for managing complex programs and their large datasets.

- We present performance numbers from our initial implementation using four I/O-intensive scientific applications.

The core part of our environment is a three-tiered architecture shown in Figure 2. In this environment, there are three key components: (1) parallel application, (2) meta-data management system (MDMS), and (3) hierarchical storage system (HSS). These three components can co-exist in the same site or can be fully-distributed across distant sites. The MDMS is an active part of the system: it is built around an OR-DBMS [32, 31] and it mediates between the user program and the HSS. The user program can send query requests to MDMS to obtain information about data structures that will be accessed. Then, the user can use this information in accessing the HSS in an optimal manner, taking advantage of powerful I/O optimizations like collective I/O [34, 7, 22], prefetching [18], prestaging [13], and so on. The user program can also send *access pattern hints* to the MDMS and let the MDMS to decide the best I/O strategy considering the storage layout of the data in question. These access pattern hints span a wide spectrum that contains inter-processors I/O access patterns, information about whether the access type is read-only, write-only, or read/write, information about the size (in bytes) of average I/O requests, and so on. We believe that this is one of the first studies evaluating the usefulness of passing large number of user-specified hints to the underlying I/O software layers. In this paper, we focus on the design of MDMS, including the design of database schema and MDMS library (user interface), the optimizations for tape-resident datasets, and an integrated Java graphical user interface (IJ-GUI) to help users efficiently work in our distributed programming environment. Our environment is different from previous platforms (e.g., [24, 2, 1, 5]) in that it provides intelligent data access methods for disk and tape-resident datasets.

The remainder of the paper is organized as follows. In Section 2, we present the design details of meta-data management system including design of database tables and high-level MDMS library (user API). In Section 3, an optimization method to access tape-resident datasets is presented. In Section 4, we present an integrated Java graphical user interface (IJ-GUI) to assist users in distributed environments. In Section 5, our initial performance results are presented. In Section 6, we review the previous work on I/O optimizations. Finally, we conclude the paper and briefly discuss ongoing and future work in Section 7.

2 Design of Meta-data Management System (MDMS)

The meta-data management system is an active middle-ware built at Northwestern University with the aim of providing a uniform interface to data-intensive applications and hierarchical storage systems. Applications can communicate with the MDMS to exploit the high performance I/O capabilities of the underlying parallel 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,[1]

---

[1] These queries are performed using user-friendly constructs. It would be very demanding to expect the user to know SQL or any other query language.

the applications can learn where in the HSS their datasets reside (i.e., in what parts of the storage hierarchy) without the need of specifying file names. They can also access the performance characteristics (e.g., speed, capacity, bandwidth) of the ASDs and select a suitable ASD (e.g., a disk sub-system consisting of eight separate disk arrays or a robotic tape device) to store their datasets. Internal data structures used in the MDMS map ASDs to physical storage devices (PSDs) available in the storage hierarchy.

- It stores information about the *storage patterns* (storage layouts) of data sets. For example, a specific multidimensional array that is striped across four disk devices in round-robin manner will have an entry in the MDMS indicating its storage pattern. 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 dataset, the MDMS can, for example, advise the HSS to perform collective I/O [15] or prefetching [18] for this dataset.

- It stores information about the *pending* access patterns. It utilizes this information in taking some global data movement decisions (e.g., file migration [36, 13] and prestaging [36, 13]), possibly involving datasets from multiple applications.

- It keeps meta-data for specifying access history and trail of navigation. This information can then be utilized in selecting appropriate optimization policies in successive runs.

Overall, the MDMS keeps vital information about the datasets and the storage devices in the HSS. Note 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 both.

The MDMS design consists of the design of database tables and the design of a high-level MDMS API. The database tables keep the meta-data that will be utilized in performance-oriented I/O optimizations. The MDMS API, on the other hand, presents an interface to the clients of the MDMS. They are described in the subsequent subsections.

2.1 MDMS Tables

We have decided that, to achieve effective I/O optimizations automatically, the MDMS should keep five (database) tables for each application. These are *run table, storage pattern table, access pattern table, dataset table,* and *execution table.* Since, in our environment, a single user might have multiple applications running, sharing tables among different applications would not be a good implementation choice because it might slow down the query speed when tables become large. In our implementation, we construct a table name by concatenating the application name and a fixed, table-specific name. Consequently, each application has its own suite of tables. For example, in an astrophysics application (called *astro3d* henceforth), the table names are astro3d-run-table, astro3d-access-pattern-table, and so on, while in a parallel volume rendering application (called *volren* henceforth), they are volren-run-table, volren-access-pattern-table, and so forth. The tables with same fixed table name (e.g., dataset table) have the same attributes for different applications except the run table, which is application specific: the user needs to specify interesting attributes (fields) for a particular application in the run table. For example, in *astrod3d,* the run table may contain the number of dimensions and the dimension sizes of each array, the total number of iterations, the frequency of dumping for data analysis, the frequency of check-point dumping, and so on. The functionality of each table is briefly summarized in Table 1.
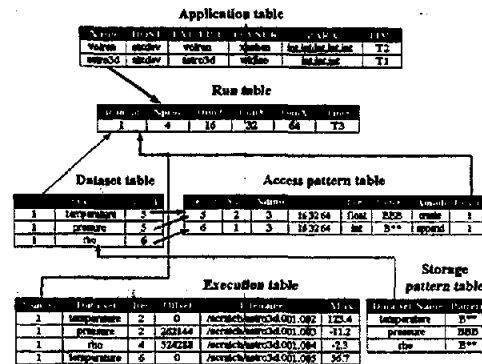

Figure 3: **Internal representation in MDMS.**

Note that, among these tables, the execution table is the most frequently updated one. It is typically updated whenever the application in question dumps data on disk/tape for visualization and data analysis purposes. The run table, on the other hand, is updated once for each run (assigning a new run-id to each run). The dataset table keeps the relevant information about datasets in the application, the access pattern table maintains the access pattern information and the storage pattern table keeps information about storage layouts of the datasets. An advantage of using an OR-DBMS [32] in building the MDMS is being able to use pointers that minimize meta-data replication, thereby keeping the database tables in manageable sizes. The MDMS also has a number of global (inter-application) tables to manage all applications, such as *application table,* which records all the application names, their host machines, and so on in the system, *visualization table,* where location of visualization tools can be found, and *storage devices table,* which maps ASDs to PSDs. An example use of our five database tables is illustrated in Figure 3.

2.2 MDMS API

The MDMS API, which consists of a number of MDMS functions, is in the center of our programming environment. Through this API, the programs can interact with the database tables without getting involved with low-level SQL-like commands. Our MDMS library is built on top of MPI-I/O [9], the emerging parallel I/O standard. MPI-I/O provides many I/O optimization methods such as collective I/O, data sieving, asynchronous I/O, and so forth. But for most computational scientists with little knowledge of I/O optimizations and storage devices, it is very hard to choose the appropriate I/O routines from among numerous complicated MPI-I/O functions. Our MDMS API helps users choose the most suitable I/O functions according to user-specified data access pattern information. In this environment, an access pattern for a dataset is specified by indicating how the dataset is to be shared and accessed by parallel processors. For example, an access pattern such as (Block, *) says that the two-dimensional dataset in question is divided (logically) into groups of rows and each group of rows will be accessed by a single processor. These patterns are also used as storage patterns. As an example, for a two-dimensional disk-resident array, 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 corresponds to blocked storage layout which might be very useful for large-scale linear algebra applications whose datasets are amenable to blocking [35]. Our experience with large-scale, I/O-intensive codes indicates that, usually, the users know how their datasets will be used by parallel processors; that is, they have sufficient information to specify suitable access patterns for the datasets in their applications. Note that con-

Table Name	Functionality	Primary Key
Run table	Records each run of the application with user-specified attributes	Run id
Dataset table	Keeps information about the datasets used each run	run id + association id
Access pattern table	Keeps the access pattern specified by user for each dataset	run id + dataset name
Storage pattern table	Keeps information on how data stored for each dataset	dataset name
Execution table	Records I/O activities of the run, including file path and name, offset, etc.	run id + dataset + iteration number

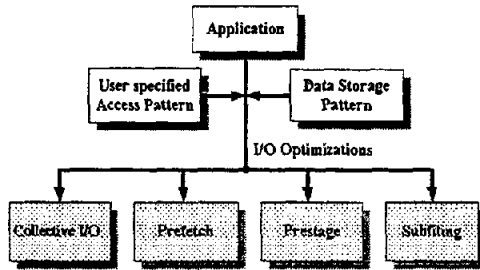Table 1: **Functionality of database tables maintained in the MDMS.**

Figure 4: **Selecting an I/O optimization.**

Figure 5: **A typical MDMS execution flow.**

veying an access pattern to the MDMS can be quite useful, as the MDMS can compare this access pattern with the storage pattern of the dataset (which is kept in the storage pattern table), and can decide an optimal I/O access strategy.

For instance, an example use of this information might occur in the following way. If the user is going to access a dataset in a (Block, Block) fashion while the dataset is stored, say in a file on disk, as (Block, *), the MDMS will automatically choose the MPI-I/O collective I/O function to achieve better performance. Our library also provides other I/O optimization methods that are not found in MPI-I/O but can be built on top of MPI-IO using the access pattern information such as data prefetching (from disk or tape to memory), data prestaging (from tape to disk) and subfiling (for tape-resident data) [25]. For example, when the user is going to access a sequence of datasets and perform some computation on them sequentially, our library can overlap the I/O access and computation by prefetching or prestaging the next dataset while the computation on the current dataset continues. As another example, if the user will access a small chunk of data from a large tape-resident dataset, our tape library, APRIL [25], will be called to achieve low latency in tape accesses. Another feature of the MDMS is that we provide mechanisms to locate the data by dataset names, such as temperature or pressure rather than using file name and offset. The user can also query the MDMS to locate datasets in which she has particular interest and to devise application-specific access strategies for these datasets. Figure 4 depicts a sketch of how an I/O optimization decision is made. In short, comparing the access pattern and storage pattern, and having access to the information about the location of the dataset in the storage hierarchy, the MDMS can decide a suitable I/O optimization.

Note that, in our environment, the users' task is to convey the access pattern information to the MDMS and let the MDMS select a suitable I/O strategy for her. In addition to inter-processor access pattern information (hint), the MDMS also accepts information about, for example, whether the dataset will be accessed sequentially, whether it is read-only for the entire duration of the program, and whether it will be accessed only once or repeatedly. An important problem now is in what part of the program the user should convey this information (hints). While one might think that such user-specified *hints* should be placed at the earliest point in
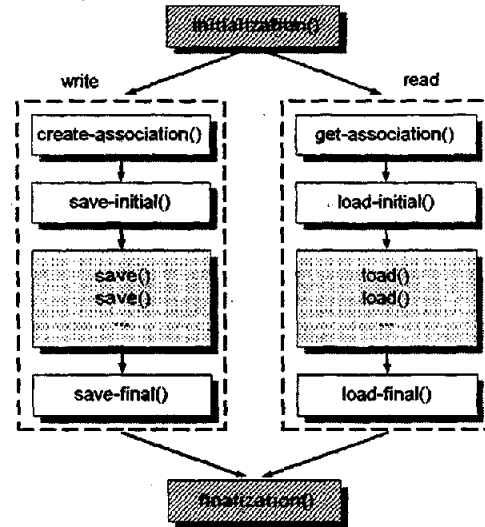
the program to give the MDMS maximum time to develop a corresponding I/O optimization strategy, this may also hurt performance. For example, in receiving a hint, the MDMS can choose to act upon it, an activity that may lead to suboptimal I/O strategies had we considered the next hint. Therefore, sometimes *delaying hints* and issuing them to MDMS collectively might be a better choice. Of course, only the correlated hints must be issued together. While passing (access pattern) hints to file systems and runtime systems was proposed by other researchers [26, 23, 28], we believe that this is the first study that considers a large spectrum (variety) of performance-oriented hints in a unified framework.

The functions used by the MDMS to manipulate the database tables are given in Table 2. Figure 5, on the other hand, shows a typical flow of calls using the MDMS. These routines are meant to substitute the traditional Unix I/O functions or MPI-IO calls that may be used by the programmers when they want to read or dump data. They look very similar to typical Unix I/O functions in appearance, so the users do not have to change their programming practices radically to take advantage of state-of-the-art I/O optimizations. The flow of these functions can be described as follows.

(1) **Initialization** The MDMS flow starts with a call to the initialization() routine.

(2) **Write** The write operations start with create-association() that creates an association for the datasets that can be grouped together for manipulation. The create-association() returns an association-id that can be used later for collectively manipulating all the associated datasets. The subsequent function for the write operations is the save-initial() routine. This can be thought of as 'file open' command in Unix-like I/O.

Name	Functionality	Important Parameters	Tables Involved
initialization()	Initializes the MDMS environment	Application name	Application table
create-association()	Creates an association for the datasets with same behavior	Dataset name, access pattern	Dataset table
get-association()	Obtains the association for the datasets	Dataset name, access pattern	Dataset table
set-run-table()	Adds a row in the run table		Run table
load-initial()	Determines the file name and offset of the dataset; Opens the file; Determines I/O optimization method	Dataset, Association handle	Execution table, Access pattern table, Storage pattern table
load()	Determines whether prefetching should be performed; Performs I/O (read)	Association handle	None
load-final()	Closes the files involved	Association handle	None
save-initial()	Generates file names; Opens files for write; Determines I/O optimization method such as collective I/O, data sieving	Association handle	Execution table, Access pattern table, Storage pattern table
save()	Writes dataset	Dataset, Association handle	None
save-final()	Closes the files involved	Association handle	None

Table 2: **Functions used in the MDMS.**

Then, the user can use the save() function to perform data write operations to the storage hierarchy. Note that in traditional Unix-like I/O, each dataset needs a 'file open', while in the MDMS library, there is only one 'open': the save-initial() routine collectively opens all the associated datasets. The write operations are ended with save-final() that corresponds to a 'file close' operation in Unix-like I/O.

(3) **Read** The read operations start with the get-association() routine that obtains an association handle generated by the create-association() routine during a previous write operation. The next function to continue the read operations is load-initial() which, again, corresponds to 'file-open' in Unix I/O. Then, the user can use the load() routine to perform read operations. The read operations are completed by the load-final() function. Note that the read and write operations can, of course, interleave.

(4) **Finalization** The MDMS flow is ended with the finalization() routine.

As stated earlier, the MDMS library provides transparent access to the database tables, thus users do not need to deal with these tables explicitly. The actions taken by the MDMS for a typical run session are as follows.

(1) A row is added to the run table by set-run-table() to record the user-specified information about this run. Users can search this table by date and time to find information pertaining to a particular run.

(2) For the datasets having similar characteristics such as the same dimension sizes, access pattern and so on, an association is reated by create-association(). Each association with one or several datasets is inserted into the dataset table. The access pattern table and storage pattern table are also accessed by the create-association(): the access pattern and storage pattern of each dataset are inserted into these two tables, respectively. We expect the user to at least specify the access pattern for each dataset. Note that, depending on the program structure, a dataset might have multiple access patterns in different parts of the code. The MDMS also accepts user-specified storage pattern hints. If no storage pattern hint is given, the MDMS selects row-major layout (for C programs) or column-major layout (for Fortran programs).

(3) In load-init(), the file names, offsets, iteration number, etc. of a particular dataset are searched from the execution table.

(4) In save-init(), the execution table may be searched to find out the file name for check-pointing. In save(), a row is inserted into execution table to record the current I/O activity.

(5) Steps 3-4 are repeated until the main loop where the I/O activity occurs is finished.

3 Hierarchical Storage System

The datasets that are generated by large-scale scientific applications might be too large to be held on the secondary storage devices permanently: thus they have to be stored on tertiary storage devices (e.g., robotic tape) depending on their access profile. In many tape-based storage systems, the access granularity is a whole file [36]. Consequently, even if the program tries to access only a section of the tape-resident file, the entire file must be transferred from the tape to the upper level storage media (e.g., magnetic disk). This can result in poor I/O performance for many access patterns. The main optimization schemes in the MDMS we have presented so far, such as collective I/O, prefetching and prestaging, could not help much when the user accesses only a small portion in a huge tape-resident dataset as the tape access times would dominate. In this section, we present an optimization technique called *subfiling* that can significantly reduce the I/O latencies in accessing tape-resident datasets.

3.1 Subfiling

We have developed and integrated into the MDMS a parallel run-time library (called APRIL) for accessing tape-resident datasets efficiently. At the heart of the library lies an optimization scheme called *subfiling*. In subfiling, instead of storing each tape-resident dataset as a single large file, we store it as a collection of small subfiles. In other words, the original large dataset is divided into uniform *chunks*, each of which can be stored independently in the storage hierarchy as a subfile. This storage strategy however, is totally transparent to the user who might assume that the dataset is stored in a single (logical) file. For read or write operations to the tape-resident dataset, the start and end coordinates should be supplied by the user. The MDMS, in turn, determines the set of subfiles that collectively contain the required data segment delimited by the start and end coordinates. These subfiles are brought
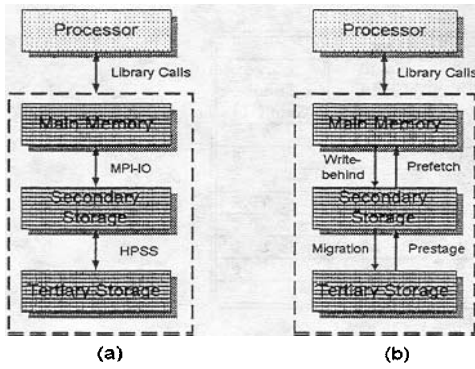
278

Figure 6: (a) **Interaction between the library calls, MPI-IO, and HPSS.** (b) **Prefetching, prestaging, and migration.**

Access Pattern	Start Coordinate	End Coordinate	Num of Floating Point Elements
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)	$4 * 10^6$

Table 3: **Access patterns used in the experiments. Each access pattern is delimited by a start coordinate and an end coordinate and contains all the data points in the rectangular region.**

(using the APRIL API) from the tape to the appropriate storage device and the required data segment is extracted from them and returned to the user buffer supplied in the I/O call. The programmer is not aware of the subfiles used to satisfy the request. This provides a low-overhead (almost) random access for the tape-resident data with an easy-to-use interface.

The interaction between the library calls and the I/O software layers is depicted in Figure 6(a). Our current access to a storage hierarchy that involves tape devices is through HPSS (High Performance Storage System) [13]. The required subfiles are transferred (in a user-transparent manner) using the HPSS calls from the tape device to the disk device and then our usual MDMS calls (built on top of MPI-IO) are used to extract the required subregions from each subfile. Figure 6(b) shows some of the potential I/O optimizations between different layers.

3.2 Experiments with APRIL

We have conducted several experiments using the APRIL library API from within the MDMS. During our experiments, we have used the HPSS at the San Diego Supercomputing Center (SDSC). We have used the low-level routines of the SDSC Storage Resource Broker (SRB) [2] to access the HPSS files. Table 3 shows the access patterns that we have experimented with (A through H). It also gives the start and end coordinates of the access patterns as well as the total number of elements requested by each access. In all these experiments, the global file was a two-dimensional matrix with 50000×50000 floating-point elements. The chunk (subfile) size was set to 2000×2000 (small chunks) and 4000×4000 (large chunks) floating-point elements.

The results from our experiments are summarized in Table 4. The table gives the response times (in seconds) of the naive scheme (i.e., without subfiling) and the percentage gains achieved by our library using two subfile sizes (as given above) over the naive scheme. The results show that the library can, in general, bring about substantial improvements over the naive scheme for both read and write operations. The performance degradations in some patterns are due to the fact that in those cases the original file storage patterns (i.e., without subfiling) were very suitable for the access patterns and the subfiling caused extra file seek operations. We plan to eliminate these problems by developing techniques that help to select optimal subfile shapes given a set of potential access patterns. Our initial observation is that the techniques proposed by Sarawagi [30] might be quite useful for this problem.

4 Design of the Integrated Java Graphical User Interface

As it is distributed in nature, our application development environment involves multiple resources across distant sites. For example,

let us consider our current working environment that consists of different platforms and tools. We do program development using local HP or SUN workstations, the visualization tools used are installed on a Linux machine, our MDMS (database tables built on top of the Postgres DBMS) is located on another machine, and our parallel applications currently run on a 16-node IBM SP-2 distributed-memory message-passing architecture. Although these machines are within our department, they could be distributed across different locations in the Internet.

When the user starts to work on such a distributed environment without the help of our application development system, she normally needs to go through several steps that can be summarized as follows.

(1) Log on to IBM SP2 and submit the parallel application.

(2) When the execution of the application is complete, log on to the database host and use native SQL dialect to find the dataset that would be needed for visualization.

(3) Once the required dataset has been found, transfer the associated file(s) manually, for example using ftp, from SP2 (where data are located) to the visualization host (where visualization tools reside).

(4) Log on to the visualization host (Linux machine) and start the visualization process.

(5) Repeat the steps 2-4 as long as there exist datasets to be visualized.

Obviously, these steps might be very time-consuming and inconvenient for the users. To overcome this problem (which is due to the distributed nature of the environment), an integrated Java graphical user interface (IJ-GUI) is implemented and integrated to our application development environment. The goal of the IJ-GUI is to provide users with an integrated graphical environment that hides all the details of interaction among multiple distributed resources (including storage hierarchies). We use Java because Java is becoming a major language in distributed systems and it is easy to integrate Java in a web-based environment. Java also provides the tools for a complete framework that addresses all aspects of managing the process of application development: processes and threads, database access, networking, and portability. In this environment, the users need to work only with IJ-GUI locally, rather than go to different sites to submit parallel applications or to do file transfers explicitly. Figure 7 shows how IJ-GUI is related to other parts of our system. It actively interacts with three major parts of our system: with parallel machines to launch parallel applications, with the MDMS through JDBC to help users query meta-data from databases, and with visualization tools. The main functions that IJ-GUI provides can be summarized as follows.

279

Access Pattern	Write Operations			Read Operations		
	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

Table 4: Execution times and percentage gains for write and read operations. The second and the fifth columns give the times for the naive I/O (without subfiling) in seconds. The remaining columns (except the first one) show the percentage improvements over the naive I/O method when subfiling is used.
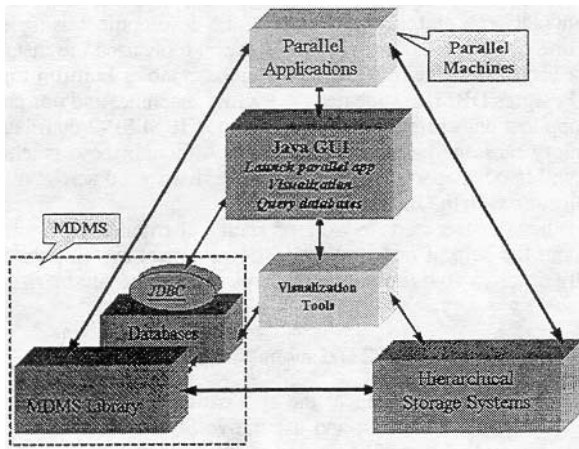

Figure 7: Java GUI and the overall system.

- **Registering new applications** To start a new application, the user needs to create a new suite of tables for the new application. Using the IJ-GUI, the user needs only to specify attributes (fields) of the run table and all other tables (e.g., storage pattern table, execution table, etc.) will be created *automatically* using the information provided in the run table.

- **Running applications remotely** The applications typically run on some form of parallel architecture such as IBM SP2 that can be specified by the user when she registers a new application. Therefore, a remote shell command is used in IJ-GUI to launch the job on remote parallel machines. The user can also specify command line arguments in the small text fields. Defaults are provided and the user can change them as needed. The running results will be returned in the large text area.

- **Data Analysis and Visualization** Users can also carry out data analysis and visualization through the IJ-GUI. In general, data analysis is very application-specific and may come in a variety of flavors. For some applications, data analysis may simply calculate the maximum, minimum, or average value of a given dataset whereas, for some others, it may be plugged into the application and calculate the difference between two datasets and decide whether the datasets should be dumped now or later. The current approach to the data analysis in our environment is to calculate the maximum, minimum, and arithmetic means of each dataset generated. From the IJ-GUIs point of view, this process is no different than

submitting a remote job. Visualization, on the other hand, is an important tool in large-scale scientific simulation, helping the users to inspect the inside nature of datasets. It is in general slightly more complicated than data analysis. First of all, the users' interests in a particular data set may be very arbitrary. Our approach is to list all the candidate datasets by searching the database using the user-specified characteristics such as maximum, minimum, means, iteration numbers, pattern, mode, and so on. Then, the candidates are presented in a radio box for the user so that she can select the dataset she wants. Second, the datasets are created by parallel machines, and they are located on parallel machines or stored in hierarchical storage systems. But our visualization tools are installed in different locations. Therefore, inside IJ-GUI, we transparently copy the data from the remote parallel machine or hierarchical storage systems to the visualization host and then start the visualization process. The user does not need to check the MDMS tables explicitly for interesting datasets or perform data transfers manually. The only thing that she needs to do is to check-mark the radio box for interesting datasets, select a visualization tool (vtk, xv, etc.), and finally, click the visualization button to start the process. The current visualization tools supported in our environment include Visualization Toolkit (vtk), Java 3D, and xv. Figure 8 shows how the user visualizes datasets through vtk and xv.

- **Table browsing and searching** Advanced users may want to search the MDMS tables to find the datasets of particular interest. Therefore, the table browsing and searching functions are provided in the IJ-GUI. The user can just move the mouse and pick a table to browse and search the data without logging on to a database host and typing native SQL script.

- **Automatic Code Generator** Our IJ-GUI relieves users great burden of working in a distributed system with multiple resources. For an application that has already been developed, the user would find it very easy to run her application with any parameters she wants: she can also easily carries out data analysis and visualization, search the database and browse the tables. For a new application to be developed, however, although our high-level MDMS API is easy to learn and use, the user may need to make some efforts to deal with data structure, memory allocations and argument selections for the MDMS functions. Although these tasks may be considered routine, we also want to reduce them to almost zero by designing an Automatic Code Generator (ACG) for MDMS API. The idea is that given a specific MDMS function and other high-level information such as the access pattern of a dataset, ACG will automatically generate a code segment that includes variable declarations, memory allocations, variable assignments and identifications of as many of the argu-
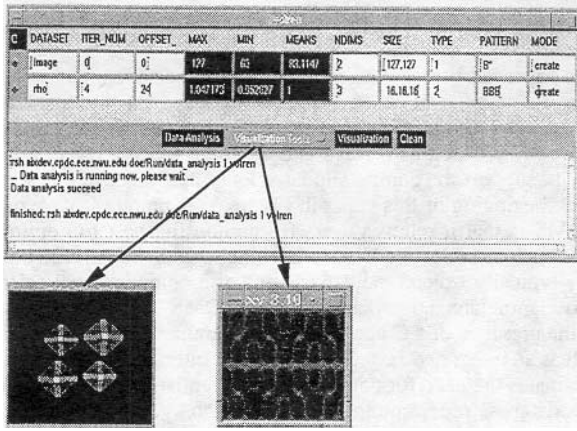
Figure 8: A visualization example. The upper window shows the datasets along with their characteristics such as data sizes, iteration number (in which they are dumped), offset, pattern, and so on. These datasets are chosen by the user for visualization. The lower windows show the visualization results for two different datasets, each using a different visualization tool.

Table 5: Total I/O times (in seconds) for *astro2d* application (Data set size is 256 MB).

	32 procs	64 procs
Original	23.46	39.67
Optimized	14.05	11.23

ments of that API as possible. The most significant feature of ACG is that it does not just works like a MACRO which is substituted for real codes: it may also consult databases for advanced information if necessary. For example, to generate a code segment for set-run-table(), which is to insert one row into the run table to record this run with user-specified attributes, our ACG would first search the database and return these attributes, then, it uses these attributes to fill out a pre-defined data structure as an argument in function set-run-table(). Without consulting the database, the user has to deal with these attributes by hand. Our ACG is integrated within our IJ-GUI as part of its functions. The user can simply copy the code segment generated and paste them in her own program.

Currently, the IJ-GUI is implemented as a stand-alone system, we are in the process of embedding it into the web environment, so the user can work in our integrated environment through a web browser.

5 Experiments

In this section, we present some performance numbers from our current MDMS and IJ-GUI implementations. The experiments were

Table 6: Total I/O times (in seconds) for *astro3d* application (Data set size is 8 MB).

	32 procs	64 procs
Original	109.93	211.47
Optimized	3.33	3.51

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

	32 procs	64 procs
Original	547.61	488.13
Optimized	1.25	2.13

run on an IBM SP-2 at Argonne National Lab. Each node of the SP-2 is RS/6000 Model 390 processor with 256 megabytes memory and has an I/O sub-system containing four 9 gigabytes SSA disks attached to it.

We used four different applications: three of them are used to measure the benefits of collective I/O for disk-resident datasets; the last one is used to see how prestaging (i.e., staging data from tape to disk before they are needed) performs for tape-resident data and how prefetching (i.e., fetching data from disk to memory before they are needed) performs data already on disks. The current implementation of the APRIL library uses HPSS [13] as its main HSS interface to tape devices. HPSS is a scalable, next-generation storage system that provides standard interfaces (including an API) for communication between parallel processors and mass storage devices. Its architecture is based on the IEEE Mass Storage Reference Model Version 5 [12]. Through its parallel storage support by data striping, HPSS can scale upward as additional storage devices are added.

Table 5 shows the total I/O times for a two-dimensional astrophysics template (*astro2d*) on the 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 the Optimized version contains access pattern hints and I/O read/write calls to the MDMS. The MDMS automatically determines that, for the best performance, collective I/O needs to be performed. As a result, impressive reductions in I/O times are observed. Since the number of I/O nodes are fixed on the SP-2, increasing the number of processors may cause (for some codes) an increase in the I/O time.

Tables 6 and 7 report similar results for a three-dimensional astrophysics code (*astro3d*) and for an unstructured (irregular data access pattern) code, respectively. The results indicate two orders of magnitude improvement if collective I/O is used.

Note that an experienced programmer who is familiar with file layouts and storage architectures can obtain the same results by manually optimizing these three applications using collective I/O. This requires, however, significant programming time and effort on the programmers' part. Our work and results show that such improvements can also be possible using a smart meta-data management system and requiring users to indicate only access pattern information.

Our next example is a parallel volume rendering application (*volren*). 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, the four datasets (corresponding to four data files) are associated with each other, and prestaging (from tape to disk) is applied for these datasets. 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, prestaging reduces the I/O times significantly. We need to mention that, in every application we experimented with in our environment, the time spent by the application in negotiating with the MDMS was less than 1 second. When considering the significant

281

Table 8: Total I/O times (in seconds) for *volren* 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 *volren* 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

runtime improvements provided by I/O optimizations, we believe that this overhead in not great.

Finally, we also measure the benefits of prefetching in *volren*. We assume the datasets are stored on local SP-2 disks. In the Original code, four data files are opened and computations are performed sequentially. In the Optimized code, prefetching (from disk to memory) is applied to the next data file when each processor is doing computation on the current data file. Consequently, the I/O time and computation time are overlapped. Table 10 shows the average read times for the four files for the Original and Optimized codes for 4 and 8 processor case, respectively. The results demonstrate that, for both 4 and 8 processor cases, prefetching decreases the I/O time by 15%. Actually, prefetching and prestaging are complementary optimizations. Our environment is able to take advantage of overlapping prestaging, prefetching, and computation, thereby maximizing the I/O performance.

6 Related Work

Numerous techniques for optimizing I/O accesses have been proposed in literature. These techniques can be classified into three categories: the parallel file system and run-time system optimizations [21, 7, 9, 18, 20, 15], compiler optimizations [4, 19, 16], and application analysis and optimization [19, 6, 28, 16]. Brown et al. [5] proposed a meta-data system on top of HPSS using DB2 DBMS. Our work, in contrast, focuses more on utilizing state-of-the-art I/O optimizations with minimal programming effort. Additionally, the design flexibility of our system allows us to easily experiment with other hierarchical storage systems as well. The use of high-level unified interfaces to data stored on file systems and DBMS is investigated by Baru et al. [2]. Their system maintains meta-data for datasets, 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 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.

Table 10: Average I/O times (in seconds) for *volren* (Data set size is 2 MB).

	4 procs	8 procs
Original	2.27	1.34
Optimized	1.91	1.15

7 Conclusions

This paper has presented a novel application development environment for large-scale scientific computations. At the core of our framework is the Metadata Database Management System (MDMS) framework, which uses relational database technology in a novel way to support the computational science analysis cycle described at the beginning of this paper in Figure 1. A unique feature of our MDMS is that it relieves users from choosing best I/O optimizations such as collective I/O, prefetching, prestaging, and so on that may typically exceed the capabilities of a computational scientist who manipulates large datasets. The MDMS itself is made useful by the presence of a C application programming interface (API) as well as an integrated Java Graphical User Interface (IJ-GUI), which eliminates the need for computational scientists to work with complex database programming interfaces such as SQL and its embedded forms, which typically varies from vendor to vendor. The IJ-GUI itself is a key component of the system that allows us to transparently make use of heterogeneously distributed resources without regard to platform. We also presented an optimization for tape-resident datasets, called subfiling, that aims at minimizing the I/O latencies during data transfers between secondary storage and tertiary storage. Our performance results demonstrated that our novel programming environment provided both ease-of-use and high performance.

We are currently investigating other tape-related optimizations and trying to fully-integrate MDMS with hierarchical storage systems such as HPSS. We are also examining other optimizations that can be utilized in our distributed environment when the user carries out visualization. Overall, the work presented in this paper is a first attempt to unify the best characteristics of databases, parallel file systems, hierarchical storage systems, Java, and the web to enable effective high-level data management in scientific computations.

Acknowledgments

References

[1] C. Baru, R. Frost, J. Lopez, R. Marciano, R. Moore, A. Rajasekar, and M. Wan. Meta-data design for a massive data analysis system. In *Proc. CASCON'96 Conference*, 1996.

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

[3] R. Bennett, K. Bryant, A. Sussman, R. Das, and J. S. Jovian. A framework for optimizing parallel i/o. In *Proc. of the 1994 Scalable Parallel Libraries Conference*, 1994.

[4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 1995.

[5] 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.

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

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

[8] P. Corbett, D. Feitelson, J.-P. Prost, G. Almasi, S. J. Baylor, A. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. Morgan, and A. Zlotek. Parallel file systems for the ibm sp computers. *IBM Systems Journal*, 34(2):222–248, Janury 1995.

[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*, 1995.

[10] 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*, pages 472–481, 1993.

[11] T. H. Cormen and D. M. Nicol. Out-of-core ffts with parallel disks. In *ACM SIGMETRICS Performance Evaluation Review*, pages 3–12, 1997.

[12] R. A. Coyne and H. Hulen. An introduction to the mass storage system reference model. In *Proc. 12th IEEE Symposium on Mass STorage Systems, Monterey CA*, 1993.

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

[14] J. R. Davis. Datalinks: Managing external data with db2 universal database. In *IBM Corporation White Paper*, 1997.

[15] 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*, 1993.

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

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

[18] C. S. Ellis and D. Kotz. Prefetching in file systems for mimd multiprocessors. In *Proc. the 1989 International Conference on Parallel Processing*, pages 306–314, 1989.

[19] 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*, 1998.

[20] 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*, pages 191–204, 1994.

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

[22] D. Kotz. Disk-directed i/o for mimd multiprocessors. In *Proc. the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, 1994.

[23] T. Madhyastha and D. R. Intelligent. adaptive file system policy selection. In *Proc. Frontiers of Massively Parallel Computing*, pages 172–179, 1996.

[24] Mcat. In *http://www.npaci.edu/DICE/SRB/mcat.html*.

[25] G. Memik, M. Kandemir, A. Choudhary, and V. E. Taylor. April: A run-time library for tape resident data. In *the 17th IEEE Symposium on Mass Storage Systems*, 2000.

[26] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proc. the Second Symposium on Operating Systems Design and Implementation*, pages 3–17, 1996.

[27] J. Newton. Application of meta-data standards. In *First IEEE Meta-data Conference*, 1996.

[28] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. In *ACM Operating Systems Review*, pages 21–34, 1993.

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

[30] S. Sarawagi. Query processing in tertiary memory databases. In *Proc. the 21st VLDB Conference*, 1995.

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

[32] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proc. the ACM SIGMOD'86 International Conference on Management of Data*, pages 340–355, 1986.

[33] R. Thakur, W. Gropp, and E. Lusk. *On implementing MPI-IO portably and with high performance*. Preprint ANL/MCS-P732-1098, Mathematics and Computer Science Division, Argonne National Laboratory, 1998.

[34] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Proc. the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[35] 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*, 1996.

[36] *UniTree User Guide, Release 2.0*. UniTree Software, Inc., 1998.