Appears in Int. Conf. on High-Performance Computing 2000

Meta-Data Management System for High-Performance Large-Scale Scientific Data Access

Wei-keng Liao, Xaiohui Shen, and Alok Choudhary

Department of Electrical and Computer Engineering
Northwestern University

Abstract. Many scientific applications manipulate large amount of data and, therefore, are parallelized on high-performance computing systems to take advantage of their computational power and memory space. The size of data processed by these large-scale applications can easily overwhelm the disk capacity of most systems. Thus, tertiary storage devices are used to store the data. The parallelization of this type of applications requires understanding of not only the data partition pattern among multiple processors but also the underlying storage architectures and the data storage pattern. In this paper, we present a meta-data management system which uses a database to record the information of datasets and manage these meta data to provide suitable I/O interface. As a result, users specify dataset names instead of data physical location to access data using optimal I/O calls without knowing the underlying storage structure. We use an astrophysics application to demonstrate that the management system can provide convenient programming environment with negligible database access overhead.

1 Introduction

In many scientific domains large volumes of data are often generated or accessed by large-scale simulation programs. Current techniques dealing with such I/O intensive problem use either high-performance parallel file systems or database management systems. Parallel file systems have been built to exploit the parallel I/O capabilities provided by modern architectures and achieve this goal by adopting smart I/O optimization techniques such as prefetching [1], caching [2], and parallel I/O [3]. 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 which forces the users to express details of access attributes for each I/O operation. Secondly, every file system comes with its own set of I/O interface, which renders ensuring program portability a very difficult task. The third problem is that the file system policies and related optimizations are in general hard-coded and are tuned to work well for a few commonly occurring cases only.

At the other end of using database management systems, a database provides a layer on top of file systems, which is portable, extensible, easy to use and maintain, and that allows a clear and natural interaction with the applications by abstracting out the file names and file offsets. However, their main target is to be general purpose and cannot provide high-performance data access. In addition, the data consistence and integrity semantics provided by almost all database management systems put an added obstacle

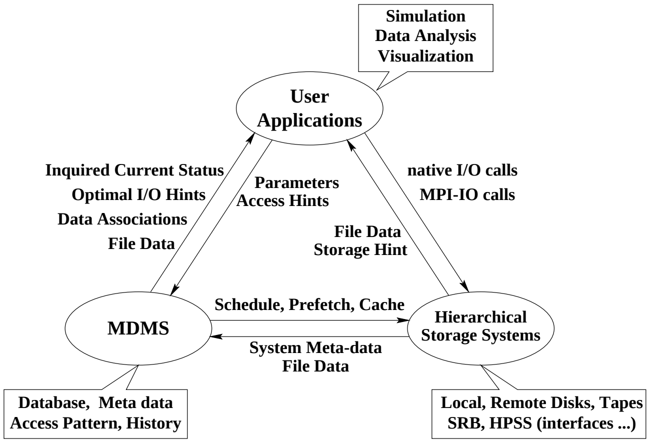


Fig. 1. The meta-data management system environment contains three key components. All three components can exist in the same site or can be located distributedly.

to high performance. Applications that process large amounts of *read-only* data suffer unnecessarily as a result of these integrity constraints [4].

This paper presents preliminary results for our ongoing implementation of a meta-data management system (MDMS) that manages meta data associated to the scientific applications in order to provide optimal I/O performance. Our approach tries to *combine* the advantages of file systems and databases and provides a user-friendly programming environment which allows easy application development, code reuse, and portability; at the same time, it extracts high performance from the underlying I/O architecture. It achieves these goals by using the management system that interacts with the parallel application in question as well as with the underlying hierarchical storage environment.

The remainder of this paper is organized as follows. In Section 2 we present the system architecture. The details of design and implementation is given in Section 3. Section 4 presents preliminary performance numbers using an astrophysics application. Section 5 concludes the paper.

2 System Architecture

Traditionally, the work of parallelization must deals with the problem of data structure used in the applications and the file storage configuration in the storage system. The fact that these two types of information are usually referred from the off-line documents increases the complexity and difficulty of application development. In this paper, we present a meta-data management system (MDMS) which is designed as a active middle-ware to connect users' applications and storage systems. The management system employs a database to store and manage all meta data associated to application's datasets and underlying storage devices. The programming environment of the data management system architecture is depicted in Figure 1. These three components can exist in the same site or can be fully distributed across distant sites.

MDMS 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. Since the meta-data management system stores information describing both application's I/O activity and the storage system, it can provide three easy-programming environments: data transparency through the use of data set names rather than file names; resource transparency through the use of the information about the abstract storage devices; and access function transparency through the automatic invocation of high-level I/O optimization.

3 Design and Implementation

The design of the MDMS is aimed to determine and organize the meta data; to provide a uniform programming interface for accessing data resources; to improve I/O performance by manipulating files within the hierarchical storage devices; to provide a graphic user interface to query the meta data.

3.1 Meta-data Management

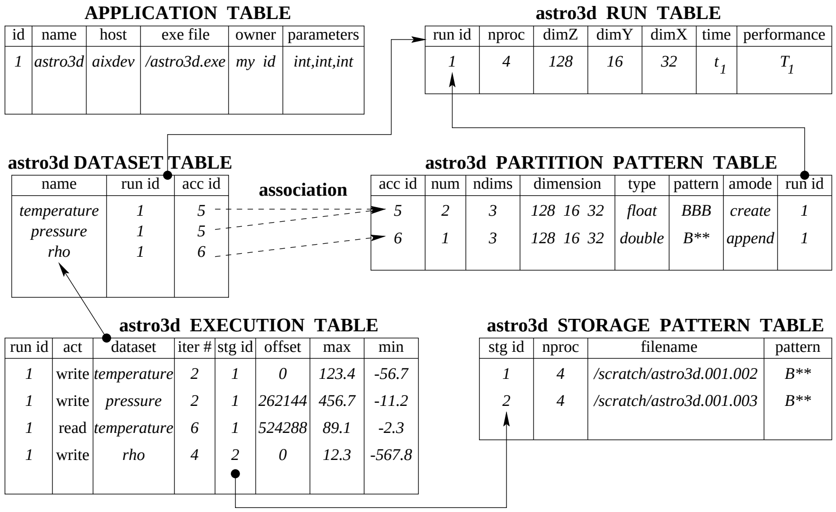
There are four levels of meta data considered in this work that can provide enough information for designing better I/O strategies.

Application Level Two type of meta data exist in this level. The first describes users' applications which contains the algorithms, structure of datasets, compiling, and execution environments. The second type is the historical meta data, for instance, the time stamps, parameters, I/O activities, result summary, and performance numbers. The former is important for understanding the the applications and the management system can use it to provide browsing facility to help program development. The historical meta data can be used to determine the optimal I/O operations for the data access of the future runs.

Program Level This level of meta data mainly describes the attributes of datasets used in the applications. The attributes of datasets includes data type and structure. Since similar datasets may potentially perform the same operations and have the same access pattern, the dataset association provides an opportunity for performance improvement both on computation and I/O. The meta data with respect to I/O activity at this level includes file location, file name, I/O mode, and file structure.

Storage System Level For hierarchical storage system, the storage and file system configuration are considered as valuable meta data. In a distributed environment, since the storage device may not locate at the same site as the machine that runs the application, the meta data describing remote systems must be captured. The meta data at this level mainly deal with the file attributes among different physical devices and can be used to make a suitable I/O decision by moving files within the storage system aggressively.

Performance Level Besides the historical performance results, other valuable meta data includes I/O bandwidth of hierarchical storage system, bandwidth of remote connection, and performance of programming interfaces. The meta data that directly affects



APPLICATION TABLE

id	name	host	exe file	owner	parameters
1	*astro3d*	*aixdev*	*/astro3d.exe*	*my id*	*int,int,int*

astro3d RUN TABLE

run id	nproc	dimZ	dimY	dimX	time	performance
1	*4*	*128*	*16*	*32*	t_1	T_1

astro3d DATASET TABLE

name	run id	acc id
temperature	*1*	*5*
pressure	*1*	*5*
rho	*1*	*6*

astro3d PARTITION PATTERN TABLE

acc id	num	ndims	dimension	type	pattern	amode	run id
5	*2*	*3*	*128 16 32*	*float*	*BBB*	*create*	*1*
6	*1*	*3*	*128 16 32*	*double*	*B***	*append*	*1*

astro3d EXECUTION TABLE

run id	act	dataset	iter #	stg id	offset	max	min
1	write	*temperature*	*2*	*1*	*0*	*123.4*	*-56.7*
1	write	*pressure*	*2*	*1*	*262144*	*456.7*	*-11.2*
1	read	*temperature*	*6*	*1*	*524288*	*89.1*	*-2.3*
1	write	*rho*	*4*	*2*	*0*	*12.3*	*-567.8*

astro3d STORAGE PATTERN TABLE

stg id	nproc	filename	pattern
1	*4*	*/scratch/astro3d.001.002*	*B***
2	*4*	*/scratch/astro3d.001.003*	*B***

Fig. 2. The representation of meta-data in the database. The relationship of tables is depicted by the connected keys. Dataset objects with the same access pattern are associated together.

the I/O performance of parallel applications is the dataset processor partition pattern and data storage pattern within the files. These two patterns can be used to determined the collective or non-collective I/O. In this work, we build the MDMS on top of MPI-IO while the proper meta data is passed to MPI-IO as file hints for further I/O performance improvement. For applications performing a sequence of file accesses, the historical access trail is typically useful for the access prediction.

We use a relational database to store the meta data and organize them into relation tables. Figure 2 shows several tables of our current implementation. For each registered application, five tables are created. Run table records the used run-time parameters for each specific run. The attributes of datasets used in the applications are stored in dataset and access pattern tables. Multiple datasets with the same structure and I/O behavior in terms of size, type, and partition pattern are associated together. In this example, two datasets, *temperature* and *pressure*, are associated together to the same row in the access pattern table. For the same I/O operations performed on the associated datasets, some resources can be re-used, eg. file view, derived data type, or even sharing the same file. The execution table stores all I/O activities for each runs. The storage pattern table contains the file locations and the storage patterns.

3.2 Application Programming Interface

The implementation of the meta-data management system uses a PostgreSQL database [5] and its C programming interface to store and manage the collected meta data. User applications communicate with MDMS through its application programming interface (API) which is built on top of PostgreSQL C programming interface. Since all meta-data queries to the database are carried out by using standard SQL, the overall implementa-

Table 1. Some of the MDMS application programming interfaces.

Function name	Argument list	Description
initialization	(appName, argc, argNames, argValues)	Connect to database, register application, record a new run with a new run id
create_association	(num, datasetNames, dims, sizes pattern, numProcs, eType, handel)	Store dataset attributes into database, create dataset association, return a handle to be used in following I/O functions
get_association	(appName, datasetName, numProcs, handle)	Obtain the dataset metadata from the database, the return handle will be used in the following I/O calls
save_init	(handle, ghosts, status, extraInfo)	Determine file names, open files, decide optimal MPI-IO calls, define file type, set proper file view, calculate file offset
load_init	(handle, pattern, ghosts, status)	Find corresponding file names, open files, decide optimal calls, set proper file view, set file type, find file offsets
save	(handle, datasetName, buffer, count, dataType, iterNum)	Write datasets to files, update execution table in database
load	(handle, datasetName, buffer, count, dataType, iterNum)	Read datasets from files
save_final	(handle)	Close files
load_final	(handle)	Close files
finalization	()	Commit transaction, disconnect from database

tion of the MDMS is portable to all relational databases. Table 1 describes several APIs developed in this work.

MDMS APIs can be categorized into two groups: meta-data query APIs and I/O operation APIs. The first group of APIs, *initialization*, *create_association*, *get_association*, and *finalization*, is used to retrieve, store, and update meta data in the database system. Through this type of APIs, user application can convey information about its expected I/O activity to the MDMS and can request useful meta data from the MDMS including optimal I/O hints, data location, etc. Although user's application can use the inquired information to negotiate with the storage system directly, it may not be reasonable to require users to understand the details of the storage system to perform suitable I/O operations. Since the MDMS is designed to contain necessary information describing the storage system, its I/O operation APIs can act as an I/O broker and with the resourceful meta data inside the system this type of APIs can decide appropriate I/O optimizations.

Figure 3(a) shows a typical I/O application using the MDMS APIs. Through calling *create_association* or *get_association*, user applications can store or retrieve meta data. Functions *save_init* and *load_init* set up proper file view according the access patterns stored in the handle. Then, a sequence of I/O operations can be performed on the same group of associated datasets using *save* and *load*.

3.3 I/O Strategies

The design of I/O strategies focus on two levels of data access: I/O between memory and disk and I/O between disk and tape. The definition of data movement within a hierarchical storage system is given in Figure 3(b).

Data Access Between Memory and Disk For the parallel applications, the I/O costs is mainly determined by the partition pattern among processors and the file storage pattern in the storage system. When the two patterns are matched, the non-collective I/O performs best. Otherwise, non-collective I/O should be used. The MDMS I/O interface

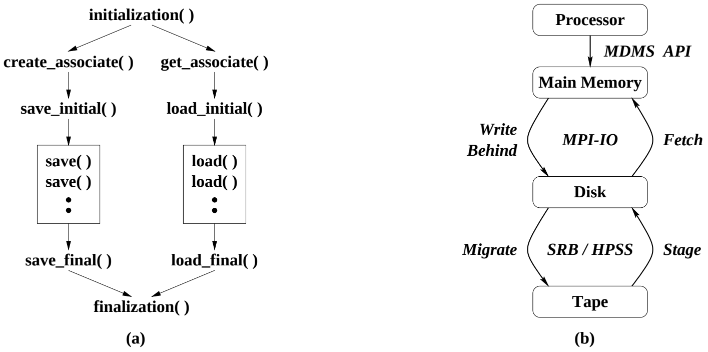


Fig. 3. (a) A typical execution flow of applications using the meta-data management system's APIs to perform I/O operations. (b) Data movement in the hierarchical storage system.

is built on top of MPI-IO [6]. The fact that MPI-IO features provide its I/O calls for different storage platforms leads the implementation of MDMS I/O API to focus on I/O type determination. Other I/O strategies including data caching in the memory and pre-fetching from the disk can also be used to reduce the I/O costs.

Data Access Between Disk and Tape For data access of a single large file, the sub-filing strategy [7] has been designed which divides the file into a number of small chunks, called sub-files. These sub-files are maintained and transparent to the programmers. The main advantage of doing so is that the data requests for relatively small portions of the global array can be satisfied without transferring the entire global array from tape to disk. For accessing a large number of smaller files, we have investigated the techniques of native SRB container and proposed a strategy of super-filing [8].

3.4 Graphic User Interface

In order to provide users a convenient tool for understanding the meta data stored in the MDMS, we have developed a graphic interface for users to interact with the system [9]. The goal of developing this tool is to help users program their applications by examining the current status of underlying dataset configurations.

4 Experimental Results

We use the three-dimensional astrophysics application, *astro3d* [10], developed at University of Chicago as the testing program throughout the preliminary experiments. This application employs six float type datasets for data analysis and seven unsigned character type datasets for data visualization. The *astro3d* is performed in a simulation loop where for every six iterations, the contents of those six float type datasets are written into files for data analysis and checkpoint purposes and two of the seven unsigned character type datasets are dumped into files for every two iterations to represent the current visualization status. Since all datasets are block partitioned in every dimension among processors and have to be stored in files in row major, collective I/O is used. Let X,

Table 2. The amount of data written by *astro3d* with respective to the parameters N(number of iterations), X, Y, and Z(data sizes in three dimensions.)

		$X \times Y \times Z$		
N	No. I/O	$64 \times 64 \times 64$	$128 \times 128 \times 128$	$256 \times 256 \times 256$
6	33	20.97 Mbytes	167.77 Mbytes	1.34 Gbytes
12	56	35.13 Mbytes	281.02 Mbytes	2.25 Gbytes
24	102	63.44 Mbytes	507.51 Mbytes	4.06 Gbytes
36	148	91.75 Mbytes	734.00 Mbytes	5.87 Gbytes
48	194	120.06 Mbytes	960.50 Mbytes	7.68 Gbytes

Y, and Z represent the size of datasets in dimension x, y and z, respectively, and N be the number of iterations. Table 2 gives the amount of I/O performed with different values of parameters specified. The performance results were obtained on the IBM SP at Argonne National Laboratory (ANL) while the PostgreSQL database system is installed on a personal computer running Linux at Northwestern University. The parallel file system, PIOFS [11], on the SP is used to store the data written by *astro3d*. The experiments performed in this work employ 16 compute nodes.

Given different data size and iteration numbers, we compare the performance of original *astro3d* and its implementation using MDMS APIs. The original *astro3d* has already been implemented using optimal MPI I/O calls, that is, collective I/O calls. Therefore, we shall not see any major difference between the two implementations. However, our MDMS will outperform on other applications if they do not optimize their I/O. Figure 4 gives the performance results of overall execution time and the database access time for two data sizes with five iteration numbers. For the case of using $256 \times 256 \times 256$ data size, the total amount of I/O is from 1.34 to 7.68 Gbytes and the overall execution time ranges from 100s to 900s seconds. Since the connection between the IBM SP and the database is through the Internet, the database query times show variance but are all within 3 seconds. Comparing to relatively larger amount of I/O time, the overhead of database query time become negligible. Although using MDMS can result the overhead of negotiation with database, the advantage of dataset association can save the time of setting file views and defining buffer derived data types. For this particular application, *astro3d*, this advantage of using MDMS over the original program can be seen from the slight performance improvement shown in the Figure.

5 Conclusions

In this paper, we present a program development environment based on maintaining performance-related system-level meta data. This environment consists of user's applications, the meta-data management system, and a hierarchical storage system. The MDMS provides a data management and manipulation facility for use by large-scale scientific applications. Preliminary results obtained using an astrophysics application show negligible overhead of database access time comparing to the same application with I/O optimal implementation. The future work will extend the MDMS functionalities for hierarchical storage system including tape and remote file system.

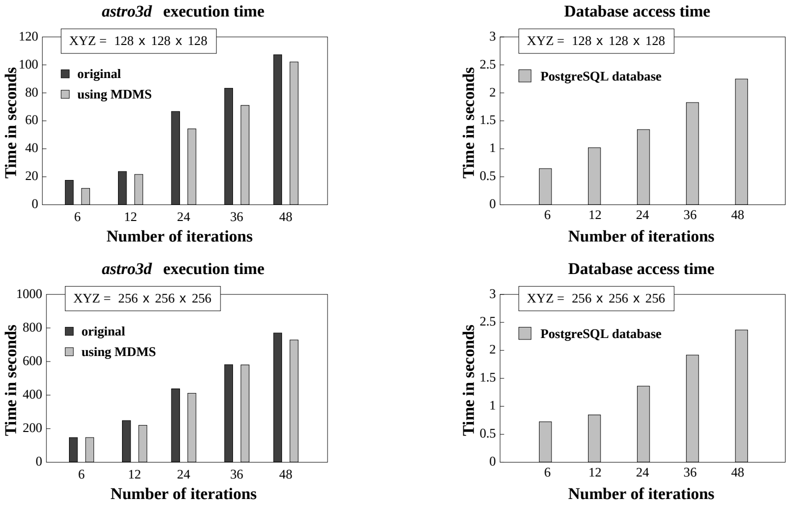


Fig. 4. Execution time of *astro3d* application and the database access time using MDMS. The timing results was obtained by running on 16 processors.

6 Acknowledgments

This work was supported by DOE under the ASCI ASAP Level 2, under subcontract No. W-7405-ENG-48. We acknowledge the use of the IBM SP at ANL.

References

1. C. Ellis and D. Kotz. Prefetching in File Systems for MIMD Multiprocessors. In *International Conference on Parallel Processing*, volume 1, pages 306–314, August 1989.
2. P. Cao, E. Felten, and K. Li. Application-Controlled File Caching Policies. In *the 1994 Summer USENIX Technical Conference*, pages 171–182, June 1994.
3. J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *IEEE Computer*, March 1994.
4. J. Karpovich, A. Grimshaw, and J. French. Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O. In *The Ninth Annual Conference on Object-Oriented Programming Systems*, pages 191–204, October 1994.
5. The PostgreSQL Development Team. *PostgreSQL User's Guide*, 1996.
6. W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* The MIT Press, Cambridge, MA, 1999.
7. G. Memik et al. APRIL: A Run-Time Library for Tape Resident Data. In *NASA Goddard Conference on Mass Storage Systems and Technologies*, March 2000.
8. X. Shen and A. Choudhary. I/O Optimization and Evaluation for Tertiary Storage Systems. In *submitted to International Conference on Parallel Processing*, 2000.
9. X. Shen et al. A Novel Application Development Environment for Large-Scale Scientific Computations. In *International Conference on Supercomputing*, May 2000.
10. A. Malagoli et al. *A Portable and Efficient Parallel Code for Astrophysical Fluid Dynamics.* http://astro.uchicago.edu/Computing/On_Line/cfd95/camelse. html.
11. IBM. *RS/6000 SP Software: Parallel I/O File System*, 1996.