An Integrated Graphical User Interface for High Performance Distributed Computing

Xiaohui Shen, Wei-keng Liao and Alok Choudhary
Center for Parallel and Distributed Computing
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208
{xhshen,wkliao,choudhar}@ece.nwu.edu

Abstract

It is very common that modern large-scale scientific applications employ multiple compute and storage resources in a heterogeneously distributed environment. Working effectively and efficiently in such an environment is one of major concerns for designing meta-data management systems. In this paper, we present an integrated graphical user interface (GUI) that makes the entire environment virtually an easy-to-use control platform for managing complex programs and their large datasets. To hide the I/O latency when the the user carries out interactive visualization, aggressive prefetching and caching techniques are employed in our GUI. The performance numbers show that the design of our Java GUI has achieved the goals of both high performance and ease-of-use.

1 Introduction

Modern simulations not only generate huge amounts of data, they also employ multiple techniques to process data generated: these processes include data analysis, visualization and so on. As data size for these applications is huge, large hierarchical storage system is used as data repository. In addition, databases are also introduced to make data management easily. Therefore, there are multiple resources involved in a modern large-scale scientific environment and these resources are heterogeneous and distributed. Without designing an efficient integrated environment, users have to deal with these resources manually and explicitly. In general, the requirements for the systems to support modern simulations are characterized as follows.

- **High performance simulation** For data intensive applications, state-of-the-art I/O optimizations such as collective I/O, prefetch, prestage and so forth should be employed to alleviate I/O problems of simulations.
- **High performance post-processings** If the user only considers simulation alone, it may result in bad performance when she carries out post-processings such as visualization and data analysis etc because these post-processings may not share the same access pattern with the simulations . The user should be careful to arrange the layouts of her datasets properly on storage systems.
- **Easy-to-use** Databases are employed to fulfill this purpose. The database can maintain meta-data information about the applications, datasets, storage systems and so on and it also provides easy query capabilities.
- **Integrated graphical environment** If the user works on an uniform platform rather than deal with distributed resources manually and explicitly, high efficiency can be achieved. Java proves to be a powerful language for such a task.

- **Latency reducing in the integrated environment** Given the fact that the speed of networks does not meet the user's requirements, aggressive optimizations such as prefetching and caching in a distributed environment are required to hide the network latency and reduce the probability of network failures.

A computational scientist would be overwhelmed in her application development if she has to consider by herself all these issues which are beyond her expertise. A lot of work addressing the above issues has been done separately in literature, few of them have considered them in a completely unified framework. Brown et al. [3] build a meta-data system on top of HPSS using DB2 from IBM. The SRB [2] and MCAT [12] provide a uniform interface for connecting to heterogeneous data resources over a network. Three I/O-intensive applications from the Scalable I/O Initiative Application Suite are studied in [8]. But all these works only address one aspect of the issues we discussed above. Our previous work [6] is a first step toward considering multiple factors in a complete picture. We have designed an active meta-data management system that takes advantage of state-of-the-art I/O optimizations as well as maintaining ease-of-use features by employing relational database techniques. In this paper, we present further considerations about integrated environment and its optimizations in a distributed environment based on our previous work. We make the following contributions:

- Present a high performance data management system in which database performs an active role in making I/O strategies, as well as managing huge amounts of data easily.
- Present the design of an integrated graphical environment in Java for large-scale scientific applications in a distributed environment. In our unified framework, users work only on their local machines and our GUI hides all the details of distributed resources. Users can launch the parallel application, carry out data analysis and visualization, query databases and browse the tables in an uniform interface.
- Present an automatic code generator component (ACG) to help users utilize the meta-data management system when they are developing new applications.
- Present an I/O latency reducing scheme which significantly improves I/O response time when the user carries interactive visualization.

The remainder of the paper is organized as follows. In Section 2 we introduce an astrophysics application and a parallel volume rendering application that we used in our work. A shorthand notation is also introduced for convenience. In Section 3 we give an overview of our design of meta-data systems (MDMS). We present our design of the integrated Java graphical environment for scientific simulations in Section 4. The functions and the inner-mechanisms that GUI provides are presented. In Section 5 we present our I/O latency reducing techniques. Finally we conclude our paper in Section 6.

2 Introduction to Applications

Our first application, called Astro-3D or astro3d [1] henceforth, is a code for scalably parallel architectures to solve the equations of compressible hydrodynamics for a gas in which the thermal conductivity changes as a function of temperature. The code has been developed to study the highly turbulent convective envelopes of stars like the sun, but simple modifications make it suitable for a much wider class of problems in astrophysical fluid dynamics. The algorithm consists of two components: (a) a finite difference higher-order Godunov method for compressible hydrodynamics, and (b) a Crank-Nicholson method based on nonlinear multigrid method to treat the nonlinear thermal diffusion operator. These are combined together using a time splitting formulation to solve the full set of equations.

From computer system's point of view, the application just generates a sequence of data and dumps them on storage. Later, the user may visualize the datasets in which she may be interested. In Astro-3D for

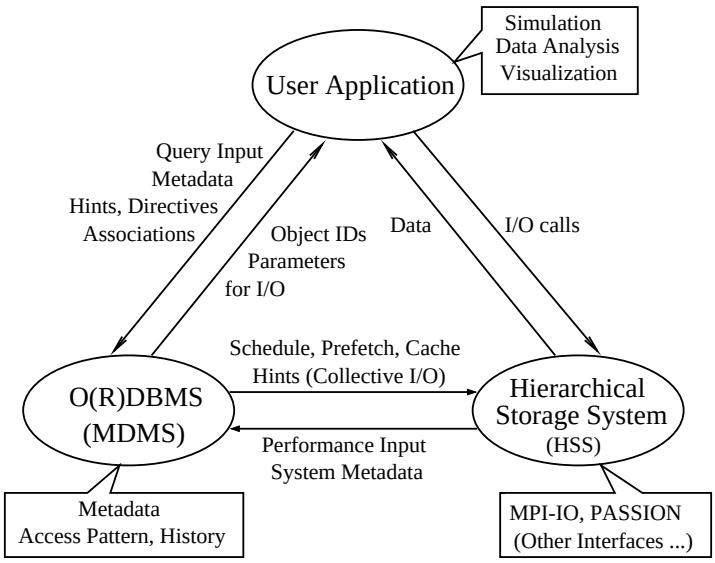

Figure 1: Three-tiered architecture. In this architecture, the user application conveys access pattern information to the MDMS, which, in turn, decides suitable I/O strategies and informs the storage system interface of the HSS (hierarchical storage system). Applications can also query the database tables maintained by the MDMS to obtain information about the locations and current states of their datasets. Once an I/O strategy has been negotiated, the I/O activity itself occurs between the user application and the HSS.

example, it generates six datasets such as temperature, pressure, etc. for each iteration in one run. Each of these datasets is written in a single file. Therefore, a data file is uniquely identified by dataset name, run id and the iteration number. We make the following notations to express a data file by concatenating the dataset name, run id and iteration number: *dataset-runid-iteration*. For example, if the temperature is dumped at the first iteration in the fifth run, it is notated as *temperature-5-1*; if the pressure is dumped at the second iteration in the sixth run, it is expressed as *pressure-6-2* and so on.

Our second application is a parallel volume rendering code (called volren henceforth). It generates a 2D image by projection given a 3D input file. The algorithm is described as follows: The data points of the input file are at the corners of cubic cells. The entire data volume is a rectangular solid made up of these cells. An orthographic projection is done through the volume parallel with one of its three axes. One of the 2 faces of the data volume that are vertical to the projection is used as the image. If there are M x N data points in this face, there are (M-1) x (N-1) pixels in the image. There is one line of projection per pixel. The densities in the stack of cells along this line are added (until their sum is 1.0), and the result becomes the alpha and grey value for that pixel. The command-line arguments include number of iterations per run, sizes of input files and so on.

From computer system's point of view, again, volren just reads a 3D input file and creates an 2D image file for each iteration per run. For example, volren may generate four image files (*image-5-1, image-5-2, image-5-3, image-5-4*) given four input data files at the fifth run.

3 Design of Meta-data Management System (MDMS)

Figure 1 shows a novel architecture we proposed in [6]. The three-tiered architecture contains 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 [16, 15] 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 [17, 5, 11], prefetching [10], prestaging [7], 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. In this section, we give an overview of our MDMS designs.

Our MDMS tries to keep meta-data information summarized as follows.

• It stores information about the abstract storage devices (ASDs) that can be accessed by applications. By querying the MDMS[1] 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 datasets.

• 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 dataset, the MDMS can, for example, advise the HSS to perform collective I/O [9] or prefetching [10] for this dataset.

• It stores information about the *pending* access patterns. It utilizes this information in taking some global decisions (e.g., file migration [19] and staging [19]), possibly involving datasets from multiple applications.

• It keeps meta-data for specifying access history and trail of navigation (not covered in this abstract).

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.

The MDMS design consists of design of database tables and the high-level MDMS user API. The database tables show what meta-data should be maintained and the MDMS user API shows how these meta-data will be used for I/O optimizations. They are described in the subsequent subsections.

3.1 Design of Database Tables

We have identified five tables for each application. They are run table, storage pattern table, access pattern table, dataset table and execution table. One user might have multiple applications running in our system, so we do not think sharing tables among different applications is a good implementation approach because it may slow down the querying speed when tables become large. In our implementation, we complete the table names by concatenating application name and the general table names to avoid sharing of tables. Therefore, each application has its own suite of tables. For example, in an astrophysics application (astro3d), all the table names are called astro3d-run-table, astro3d-access-pattern-table and so on, while in a parallel volume rendering application (volren), they are volren-run-table, volren-access-pattern-table and so forth. The tables with same general table names have same attributes among different applications except the run table, which is application specific: the user needs to specify interesting attributes in that particular application. For example, in astrod3d, the user might be interested in dimension sizes of each array, total number of iterations, frequency of dumping for data analysis, frequency of checkpoint dumping and so on. The functionality of each table is described in table 3.1:

[1] These queries are performed using user-friendly constructs. It would be very demanding to expect the user to know SQL [14] or any other query language.

Table Name	Functionality	Key
Run table	Record each run of the application with user-interested attributes	Run id
Data set table	Data sets used each run	Run id + association id
Access pattern table	Access pattern specified by user for each dataset	Run id + Data set name
Storage pattern table	How data stored for each dataset	Data set name
Execution table	Record I/O activities of the run, including file path and name, offset etc	Run id + Data set + iteration number

Table 1: Functionality of database tables

We also have several global tables to manage all applications, such as application table, which records all the application names and their host machines and so on in the system, and visualization table, where location of visualization tools can be found. The internal representation of the tables is depicted in figure 2.

3.2 Design of MDMS User API

The MDMS user API, which consists of a bunch of MDMS functions, is the kernel of our programming environment. It interacts with database transparently and provides users with data access methods and I/O optimizations. Our MDMS library is built on top of MPI-I/O since MPI-I/O is an emerging standard of MPI. MPI-I/O provides many I/O optimization methods such as collective I/O, data sieving and so forth. But for most computational scientists with little knowledge of I/O optimizations, it is very hard for them to choose appropriate functions among these complicated MPI-I/O functions. Our MDMS API, which is built on top of MPI-I/O, will automatically help users choose best I/O functions according to user specified data access pattern. In this environment, an access pattern for a data set is specified by indicating how the dataset is to be divided and accessed by parallel processors. For example, an access pattern such as (BLOCK,*) says that the 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. 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 corresponds to blocked storage layout which might be very useful for large-scale linear algebra applications whose datasets are amenable to blocking [18]. Usually, the user knows how his data will be used by parallel processors, i.e., user knows access pattern and storage pattern of datasets.

One example of using this information is that if the user is going to access the dataset in a (Block,Block) way while data are stored as (Block,*) , our library will automatically choose MPI-I/O collective I/O function to achieve better performance. Our library also provides other I/O optimization methods which are not found in MPI-I/O such as prefetch (from disk or tape to memory), prestage (from tape to disk) and subfiling [13]. For example, when a 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 doing computation on the current dataset. If the user will access a small chunk of data from a large dataset, our tape library, APRIL [13], will be called. Another feature of our MDMS library is that we provide mechanisms to locate the data by dataset names, such as temperature, pressure rather than using file name and offset which could be very hard to remember. The user can also query the databases to get datasets in which she is interested. Figure 3 shows how I/O optimization decision is made and what kinds of optimizations are used.

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

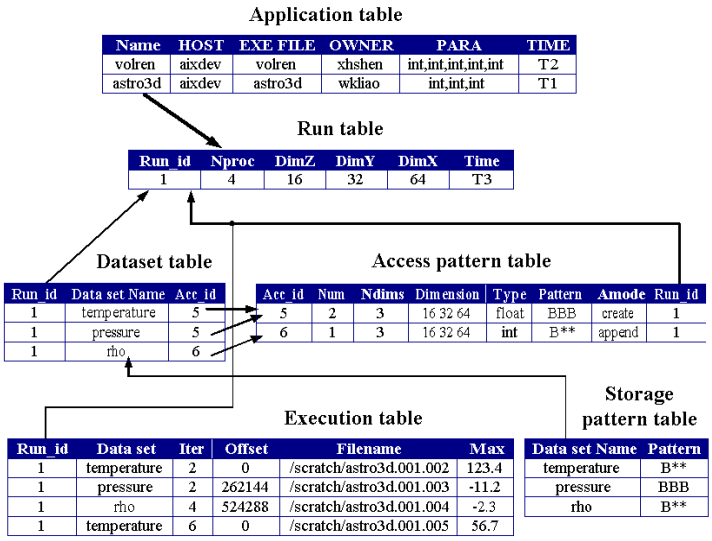

Figure 2: Internal representation in MDMS. At time T3, the user starts to run the *astro3d* application with 4 processors and the size of datasets is $16 \times 32 \times 64$ (see the run table). The information about this run is kept in the run table with a run id of 1. There are three datasets involved in this run, namely, 'temperature', 'pressure', and 'rho' as can be seen from the dataset table. The temperature and pressure datasets are associated together with the same association id (5), they have same dimension sizes, data type, access pattern, and I/O mode (which is 'create'). The rho dataset, on the other hand, has different association id and it will be appended to data file of previous runs (its I/O mode is 'append'). An entry such as 'BBB' in the access and storage pattern tables denotes (Block,Block,Block) access and storage pattern, respectively. The execution table shows the file names and offsets using which these datasets are dumped at each iteration where I/O occurs. Before I/O is performed, the system would first check the storage pattern table for the dataset in question, and then decide (by comparing it with the access pattern), for example, whether collective I/O should be used for optimization.

sequentially, whether it is read-only for the entire duration of the program, and whether it will be accessed only once or repeatedly.

A typical I/O flow using our library in an I/O intensive application is shown in figure 4 and the functionality of MDMS library is depicted in table 3.2.

4 Design of Java Graphical User Interface

4.1 Architecture of Integrated Java GUI

As it is distributed in nature, our programming environment involves multiple resources across distant sites. Consider our current working environment, we are working on local HP or SUN workstations, the visualization tools are installed on a Linux machine, our database (POSTGRESQL) is located on another machine and our parallel applications run on a 16 node IBM SP2. Although these machines are within our department, they could be distributed across any places on Internet.

When a user starts to work in such a distributed environment, she needs to go through the following procedures:

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

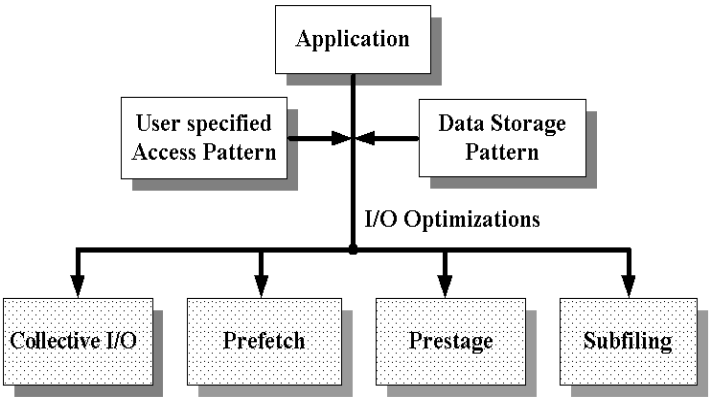

Figure 3: Selecting an I/O optimization. Note that the access pattern in the figure does not only describe the inter-processor data access pattern, but also contains information about whether the dataset is read-only, how many times it will be accessed during the run, etc. All this information will be utilized in deciding a suitable I/O optimization such as collective I/O, prefetching, prestaging, and so on.

(2) When the application is finished, she needs to log on to the database host and use native SQL language to inspect the database to find datasets she would be interested in for visualization.

(3) When the user is interested in a particular dataset, she would transfer the data file explicitly, 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 (DATA) and start the visualization process.

(5) Repeat 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 (GUI) is implemented and integrated to our application development environment. The goal of the 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 itself proves to be a key enabling access language and operating environment with support for all of our platforms of interest, including IBM AIX, Linux, Windows NT, Solaris, and others. Transparency is made possible by the many platform independent abstractions of Java, including process management (a built-in class), multithreading (a language feature), networking and streams (built-in classes), GUI components (the Abstract Windowing Toolkit), and database access (JDBC). Java has proven to be flexible and deliver good performance in all of these dimensions without being in any way on the critical path of performance in the application itself. In this environment, the users need to work only with GUI locally, rather than go to different sites to submit parallel applications or to do file transfers explicitly. Figure 5 shows how GUI is related to other parts of our system. It actively interacts with three major parts of our system:

- Interacts with parallel machines to launch parallel applications.
- Interacts with databases through JDBC to help users query meta-data from databases.
- Interacts with visualization tools to carry out visualization process.

4.2 Main functions of GUI

The main functions that GUI provides are described as follows:

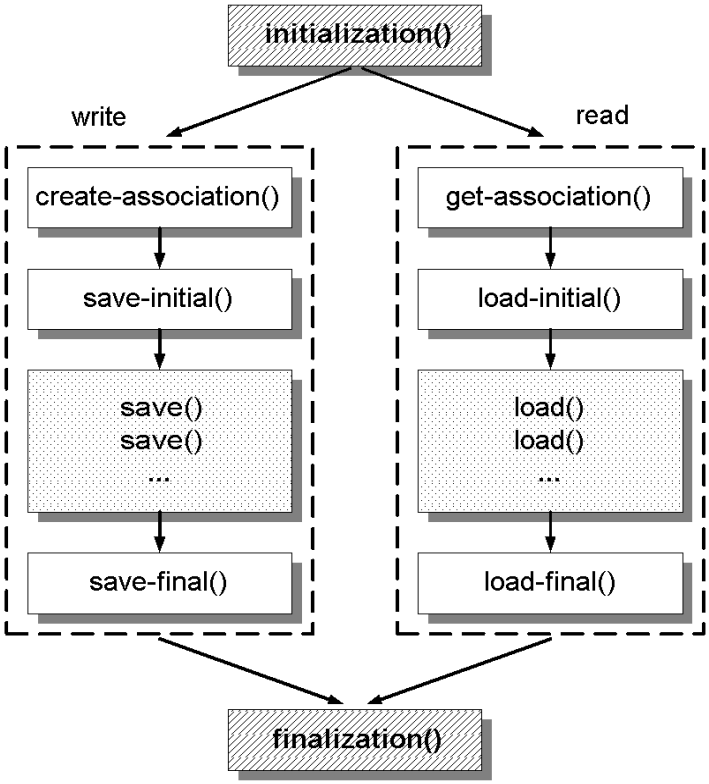

Figure 4: A typical MDMS execution flow: The execution starts with the initialization() routine. The left side of the figure shows how the write operation progresses and right side shows how the read operation progresses. The execution flow is ended by the finalization() routine.

- **Registering new applications** To start a new application, the user needs to create a new suite of tables for the new application. By GUI, the user needs only to specify attributes of run table that she would be interested in, and all the other tables will be created automatically with run table.

- **Running applications remotely** The applications are usually running somewhere on parallel machines such as SP2, which are specified by the user when she registers a new application. Therefore, remote shell command is used in 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. Figure 6 shows an example of an astrophysics application: four processors are used and the sizes of datasets are 16*16*16.

- **Data Analysis and Visualization** Users can also carry out data analysis and visualization through our GUI. Data Analysis may come in a variety of flavors, it is quite application specific. For some applications, data analysis may simply calculate the maximum, minimum or average value of a given dataset, for some others, it may be plugged into the application and calculate the difference between two datasets and decide whether the dataset should be dumped now or later. Our systems current method of data analysis is to calculate the maximum, minimum and means of each dataset generated. From the GUIs point of view, it is no different than just submitting a remote job. Visualization is becoming an important approach in large-scale scientific simulation to inspect the inside nature of datasets. It is often a little more complicated than data analysis: first of all, the users' interest in a particular data set may be very arbitrary. Our approach is to list all the candidate datasets by searching

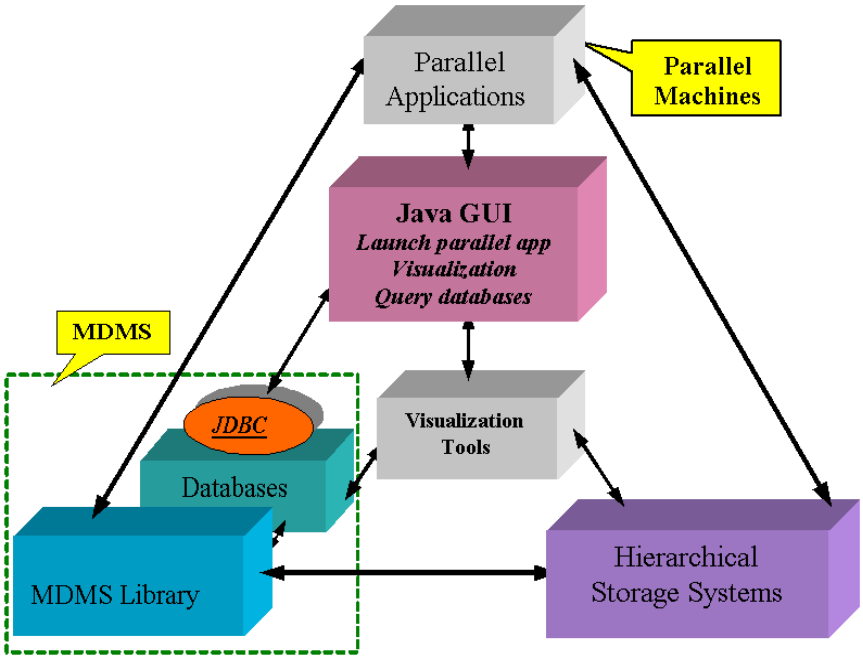

Figure 5: Java GUI in overall system.

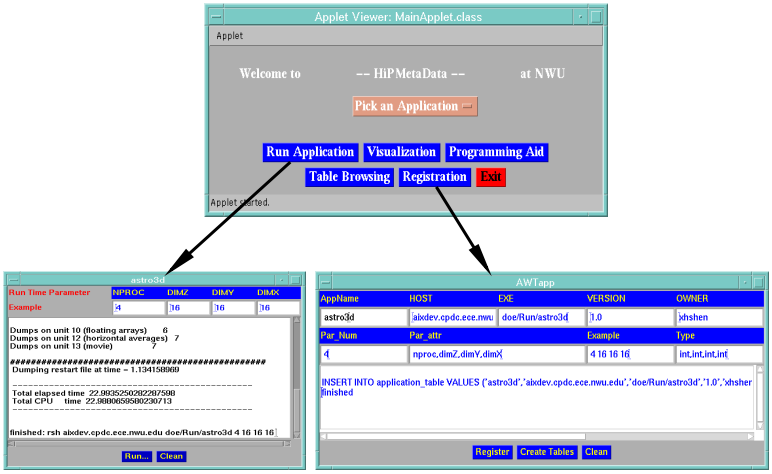

Figure 6: Application registration and running. The upper part shows the main window where the user can pick an application, run it, visualize the datasets generated, and so on. The lower-left window shows that the user is running an application using the 'Run Application' button in main window. The default command line arguments are provided by the system and the user can modify them if needed. The lower-right window shows that the user is registering a new application. The system will ask the user to provide information such as application name, the name of the machine on which the application will run, what command line arguments are, etc., and then it will create a suite of tables for this new application in the MDMS and record this application in the application table.

Name	Functionality	Important Args	Tables Inv'd
initialization	Initialization MDMS environment	Application name	Application table
create-array-asc	Create an association for datasets with same behavior	Data set name, access pattern	Data set table
set-run-table	Add one row in run table		Run table
load-init	Locate file name and offset of the dataset; Open the file; determine I/O optimization method	Data set, association handle	Execution table, access pattern table, storage pattern table
load	Determine whether prefetch should be performed; Perform I/O (read)	association handle	None
load-final	Close files	association handle	None
save-init	Generate file names; open files for write; determine I/O optimization method such as collective I/O	association handle	Execution table, access pattern table, storage pattern table
save	Write dataset	Data set, association handle	None
save-final	Close files	association handle	None

Table 2: MDMS library and its functionality

the database by user-specified characteristics such as maximum, minimum, means, iteration numbers, pattern, mode and so on. Then the candidates are presented in radio box for user to choose easily. Second, the datasets that are created by parallel machines, are located either at parallel machines or stored in hierarchical storage systems. But our visualization tools are installed at other places. Therefore, inside GUI, we transparently transfer the data from the remote parallel machine or hierarchical storage systems to the visualization host's local disks and then start the visualization process. This is implemented by having a server program written in C running on the storage side and the GUI (in Java) serves as the client. The user does not need to check the database for interesting datasets or do data transfer explicitly. The only things the user has to do are to checkmark the radio box for interesting datasets, select a visualization tool (vtk, xv etc.), and then click the Visualization button to start the process of visualization. Our current visualization tools include Visualization Toolkit (VTK), Java 3D, XV etc. Figure 7 shows how the user visualizes the datasets through VTK and XV.

- **Table browsing and searching** Advanced users may want to search the database to find datasets of particular interest. So the table browsing and searching functions are provided in our 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 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 selects 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 arguments of that API as possible. For some functions, the user still needs to complete the code by filling out some blanks

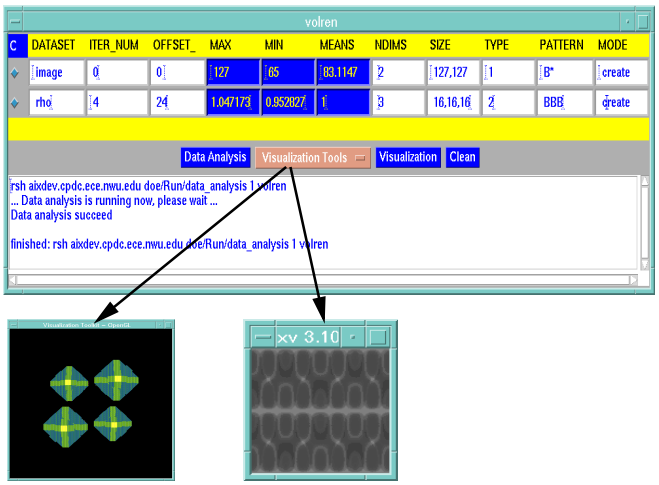

Figure 7: A visualization example. The upper window shows the datasets along with their characteristics such as data sizes, iteration number (in which they are dumped), offset, pattern, and so on. These datasets are chosen by the user for visualization. The lower windows show the visualization results for two different datasets, each using a different visualization tool.

which the ACG has left empty, such as variable names for number of processors, dimension sizes and so on in her application, but it is very easy to complete them because we have provided a code frame and the user only needs to fill out several blanks based on clear directions. A MDMS I/O flow in figure 5 is also provided by the ACG to give user a general concept how our MDMS API works. 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(). Figure 8 shows such a case in a parallel volume rendering application. Without consulting the database, the user has to deal with these attributes by hand. Our ACG is integrated within our GUI as part of its functions. The user can simply copy the code segment generated and paste them in her own program. With our ACG, there is almost no extra effort involved to port an application to our programming environment, and the user does not need to know how exactly our API actually works nor remember the rules to use our API. This is important for computational scientists because it enables them to spend as little time as possible on their programming environment and focus their works on the application itself.

Our current GUI is implemented as a standalone system, we are also embedding it into the web environments, so the user can work in our integrated environment through a web browser.

5 I/O Latency Reducing for Interactive Visualization

Our GUI can help the user work at a single site without consulting a variety of distributed components explicitly. A potential problem in this environment, however, is the I/O response time when the user carries out interactive visualization (VTK) on a sequence of datasets. Note that visualization tools such as VTK etc are usually installed on sequential workstations, while the datasets generated by parallel applications may be stored on the disks of parallel computing systems or on a remote storage system such as tertiary storage systems like HPSS [7]. Therefore, these datasets need to be first transfered to the local site from remote

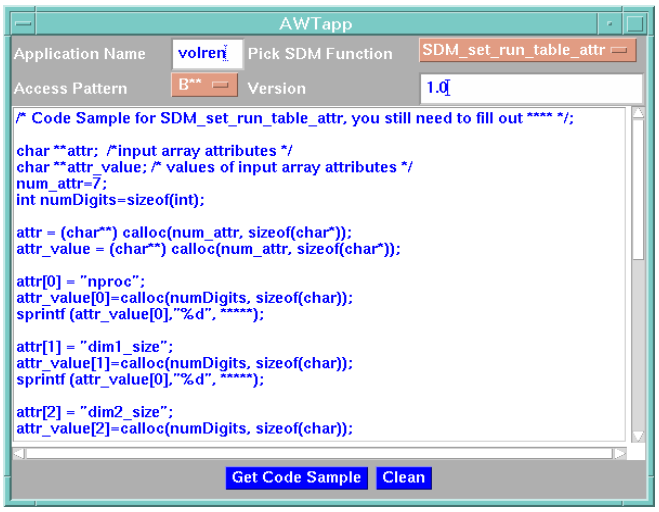

Figure 8: Code segment generated by ACG for set-run-table function.

storage. As the data size could be large and the physical distance could be long, the I/O response time is significant. It is quite annoying that the user has to wait for tens of seconds for the availability of the data after she has launched the visualization process on a dataset. To reduce the I/O latency, a naive approach is that the user explicitly transfers all the data to the local disks first and then starts visualization process. The problems of this approach, however, are three-fold. First, the user has to deal with specific file names and locations manually which are very inconvenient; second, the user may not be sure what datasets she might use in the future and finally, the overall time is still long since the I/O time is not overlapped with the visualization process.

In this section, we present a scheme to address the problems of naive approach. By using prefetching and caching techniques, our approach can effectively hide the I/O latency, improving overall visualization performance in GUI. The basic idea of our approach is that when the user is working on a dataset (visualization), we make a prediction on what the next dataset that might be accessed, then spawn another thread to perform remote I/O and caches it on the local disks. Since visualization process with human interaction is slow, there is ample time to overlap prefetching. In addition, the I/O response time for the next dataset can be significantly reduced when the required dataset can be serviced on local disk cache.

We use database to keep track of data locations and access history. Two tables are created in the database. One table is called *data-access-trace*, whose attributes (fields) include application name, dataset name, iteration number, date and time, run id etc. This table keeps all the datasets visualized by the user. Another table is called *data-cache*, whose attributes (fields) include, application name, dataset name, iteration number, local directory and reference counter. It keeps information about what datasets are currently cached on the local disks and how many times they are accessed by the user.

The key issue for prefetching is to decide which dataset should be fetched. In Section 2 we introduced an astrophysics application which is a representative of many scientific applications. We can see that many scientific applications generate time-serial datasets (each dataset is stored in a separate file) and these datasets will also be accessed by visualization tools one by one in time order. For example, by visualizing datasets ‘temperature’ from time step 0 to 20, the user can know how temperature changes as simulation goes on. This characteristic is good for prediction, because the next time step dataset is the best candidate for prefetching. Figure 9 (column 1,2) shows the results of prefetching. The I/O response time can be reduced significantly.

As some applications may take many time steps and the user may not always visualize all of them. For example, to have a quick view of how dataset ‘temperature’ changes as simulation goes on, the user may only

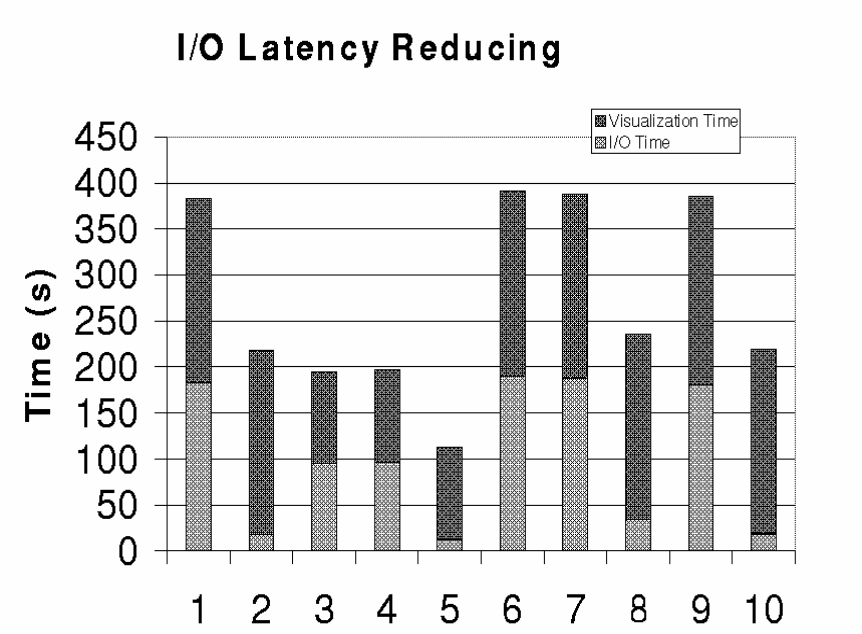

Figure 9: I/O Latency Reducing by Prefetching for Astrophysics data visualization. The data is located at disks of SP2 and visualization tool (VTK) is installed on a Linux machine. The size of each dataset is 8MB. The average visualization time on a dataset is about 10s. (1) No prefetch, Input A; (2) Prefetch, Input A; (3) No prefetch, Input B; (4) Prefetch (fixed stride), Input B; (5) Prefetch (adaptive stride), Input B; (6) No prefetch, Input C; (7) Prefetch (fixed stride), Input C; (8) Prefetch (adaptive stride), Input C; (9) No prefetch, Input D; (10) Prefetch (according to previous runs), Input E. *Input A = (temperature-9-0, temperature-9-1,temperature-9-2, ..., temperature-9-20), B = (temperature-9-0, temperature-9-2, temperature-9-4, ..., temperature-9-20), C = (temperature-9-0,pressure-9-0, temperature-9-2, pressure-9-2, ..., temperature-9-20, pressure-9-20); D = (temperature-9-1, temperature-9-5, temperature-9-4, temperature-9-5, temperature-9-10, ...), E = (temperature-10-1, temperature-10-5, temperature-10-4, temperature-10-5, temperature-10-10, ...).*

pick part of data files in a strided manner such as temperature-9-0, temperature-9-2, temperature-9-4, etc. In this case, the naive prefetching would not help. To address this problem, we proposed an 'adaptive stride' scheme. In this approach, the previous stride is used to predict next dataset. For example, after datasets temperature-9-0 and temperature-9-2 are accessed, we predict the next dataset should be temperature-9-4 since the last stride is 2. Figure 9 (column 3,4,5) shows the results. The adaptive approach dramatically reduces I/O latency. For some cases, the user may be interested in multiple datasets. For example, the changes of temperature and pressure may influence with each other, so the user may inspect temperature and pressure datasets alternatively. The adaptive stride prefetching can still deliver significant performance improvement in Figure 9 (column 6,7,8).

Another scenario could happen is that the datasets to be visualized may not have fixed stride. By checking the data access history in table *data-access-trace* of previous runs, we can still make a good prediction since the user may change run-time parameters for each run and may still interested in the same set of datasets. Figure 9 (column 9, 10) shows the results.

6 Conclusions

In this paper, we presented an integrated Java graphical user interface (GUI) to efficiently help users work on an environment that is characterized by distributed and heterogeneous natures. Our GUI provides users an unified interface to all the resources and platforms presented to large-scale scientific applications. In addition, an I/O response time reducing technique has been integrated into our GUI to hide I/O latency for interactive visualization. All these works take advantage of Java's powerful features such as platform independence, portability, process management, multithreading, networking and streams. The database is also plays an important role which makes the whole framework complete. In the future, we would investigate other optimizations in our environment, such as subfiling [13]. The relationship between prefetch and caching [4] in our context will also be studied.

References

[1] A. Malagoli, A. Dubey, and F. Cattaneo. A Portable and Efficient Parallel Code for Astrophysical Fluid Dynamics. *http://astro.uchicago.edu/Computing/On-Line/cfd95/camelse.html*

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

[3] 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.

[4] 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.

[5] 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.

[6] A. Choudhary, M. Kandemir, H. Nagesh, J. No, X. Shen, V. Taylor, S. More, and R. Thakur. Data management for large-scale scientific computations in high performance distributed systems, In *Proc. the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99),* August 3-6, 1999, Redondo Beach, California.

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

[8] 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.*

[9] 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.

[10] 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.

[11] 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.

[12] MCAT *http://www.npaci.edu/DICE/SRB/mcat.html.*

[13] G. Memik, M. Kandemir, A. Choudhary, Valerie E. Taylor. APRIL: A Run-Time Library for Tape Resident Data. To appear in *the 17th IEEE Symposium on Mass Storage Systems, March 2000.*

[14] R. Ramakrishnan. *Database Management Systems*, The McGraw-Hill Companies, Inc., 1998.

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

[16] M. Stonebraker and L. A. Rowe. The design of Postgres. In *Proc. the ACM SIGMOD'86 International Conference on Management of Data*, Washington, DC, USA, May 1986, pp. 340–355.

[17] 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.

[18] 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.

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