Maitri: A Format-Independent Framework for Managing Large Scale Scientific Data

Rishi R. Sinha, Arash Termehchy, Marianne Winslett, Soumyadeb Mitra, John Norris
University of Illinois
201 North Goodwin Avenue
Urbana, Illinois 61801
{rsinha,termehch,winslett,mitra1,jnorris}@uiuc.edu

ABSTRACT

Even traditional commercial database systems do not scale to the size of today's large scientific data sets, whose growth is outpacing Moore's Law. Instead, scientists are wedded to special-purpose data formats and their associated I/O libraries, even though these libraries provide only basic functionality. Thus there is a need for a scalable data management system that can support these formats and, when needed, provide more sophisticated functionality for indexing, buffering, caching, concurrency control, metadata management, and querying.

This demonstration showcases Maitri, a framework that can be used to address these needs. The Maitri framework consists of a set of standard, very narrow interfaces for format-agnostic, loosely-coupled libraries offering aspects of the functionality listed above. Format independence is provided by Maitri's block manager module, which encapsulates all code that is specific to a particular scientific data format, and calls the appropriate scientific I/O library to read and write data in that format. The demonstration shows the Rocketeer visualization toolkit running with subsets of the Maitri module implementations to visualize rocket simulation output data. The Rocketeer runs show the efficiency that can be gained by layering format-agnostic query, buffering, and/or indexing facilities atop scientific I/O libraries. The demonstration also shows that despite their narrow interfaces, the implementations of Maitri modules work together very effectively.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Scientific Databases; H.3.1 [**Content Analysis and Indexing**]: Indexing Methods

General Terms

Scientific Data Management

3rd *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

Keywords

Indexing, buffer management, block management

1. INTRODUCTION

Today's high-speed computers and automated sensing technology make it very easy to collect and generate data. For example, the 12 TB of data currently exported by the Sloan Digital Sky Survey (SDSS) are used by scientists around the globe to study astronomical phenomena, and a week-long plasma physics simulation at Tech-X Corporation outputs over 1 TB. The upswing in data generation and collection is outpacing Moore's Law, straining the ability of computers and especially disks to keep up. Brute force approaches to data analysis will soon resemble searching for a needle in a haystack. Scientists need innovative data management facilities to store, visualize, and analyze enormous amounts of data efficiently.

Researchers have worked to adapt database technology to the needs of scientists [5, 11]. The most successful of these efforts are probably SQL Server for SDSS and Oracle for biological data. Yet close examination of the SDSS query log shows that scientists use the relational part of SDSS only to find data of potential interest. Once interesting data have been found, the scientists download a less processed form of the data objects in a non-RDBMS format, go through the laborious process of cleaning them at home, and analyze them, all without a relational system. Similarly, there are loud rumblings of discontent with Oracle in the biology community. Most of the reasons for this lack of success have been documented previously, along with the unmet needs and the special characteristics of scientific data and queries [3, 4, 7, 8, 10]. For example, failure has been attributed to the lack of support for arrays, need for considerable performance tuning expertise, lack of portability, expensive or non-existent parallel versions, monolithic architecture, use of proprietary file formats, and scientists' dislike of (learning) SQL. Perhaps surprisingly, performance is also often an issue. For example, a typical commercial data warehouse might hold 2 TB, while the Center for Simulation of Advanced Rockets (CSAR) at UIUC can generate that much data in a week. To quote Mike Folk, CEO of The HDF Group, "It's not unusual for scientists to start with relational databases, which often work well up to a point, but then people find that they don't scale to their problem sizes. This is when people start taking a look at what we're doing with HDF."

In response to the shortcomings of commercial databases, domain and computer scientists developed binary-file-based user-level libraries that store data in formats designed specifically for scientific data. These range from generic formats like HDF and netCDF to domain specific formats like ROOT for high energy physics and FITS for astronomy. These systems allow the scientists to store data with their associated metadata and access them efficiently through the associated libraries. However, the libraries' navigational model for access requires significant programming for each query. Further, the libraries' file-based approach has had the unanticipated side effect that as the data size increases, the number of files usually gets very large (e.g., an 11 GB data set at CSAR has 7152 files). Indexing and multi-file data retrieval are absent from these libraries, and scientific tool programmers expend considerable effort to provide multi-file metadata management, buffering and caching facilities in their tools. Often scientists need data from several sources, each adopting a different format and requiring use of a different API. Thus scientists have to double as data management experts, taking time better spent doing science.

Even if a vastly improved data management facility were made available to scientists, in many cases it would be hard for them to adopt it. A number of scientific communities have already standardized on a data format and schema for their domains, and data that do not adhere to that format will not be sharable or be accessible by the tools developed by the community. Further, specific formats and schemas are often chosen for a large long-term project at its inception, such as while a new satellite is being built. Once a large amount of data has been collected or generated, and tools have been written to access that data in a certain format and schema, it is impractical to convert the data and rewrite the tools.

If neither commercial databases nor current scientific I/O libraries will able to handle the scientific data management challenges of the near future, and it will be very hard for scientists to migrate to new facilities, then how can we improve matters? We must allow scientists to use their choice of format and schema, while introducing new functionality and enhancing scalability. Intuitively, we could satisfy almost everyone by taking the high-level functionality of a DBMS, pasting it on top of a variety of scientific I/O libraries, and changing the look and feel of the query language. In theory, this can be done by enhancing the functionality of today's scientific I/O libraries, by extending ORDBMSs to work better with scientific data, or by introducing an entirely new approach that combines aspects of both worlds. We have adopted the third course of action, for the following reasons.

Assumptions about the data storage format are built into DBMSs, making them incompatible with the vast amounts of legacy scientific data stored in a different format. The removal of these assumptions will impact much of the DBMS (which makes it a great research topic). Further, commercial DBMSs generally use proprietary storage formats, which are unacceptable in most scientific domains.

Without *native* support for arrays in the storage model, access methods, and query language, DBMSs will find it hard to compete with scientific I/O libraries on performance and usability of large data repositories. While commercial DBMSs have made great strides in extensibility, and the interest in column-oriented storage and the decomposition storage model [2] is also a move in the right direction, the array data type remains a second class citizen in commercial DBMSs and it seems unlikely that this will change in the next decade.

If the groups that maintain today's scientific I/O libraries each separately make significant enhancements in library functionality and performance, **scientists will still have to learn a variety of different APIs** to gather all the data they need from different sources. We can avoid this problem (and reduce the cost of the enhancements) by encapsulating the new functionality into modules that can be layered on top of multiple libraries.

To enhance a variety of scientific data storage formats with the high-level functionality of a DBMS, we propose the **Maitri** framework, whose most notable characteristics are as follows:

Loosely coupled, non-monolithic architecture. Maitri's framework consists of very narrow interfaces for a block, buffer, index, metadata, and query manager. With few exceptions, a scientific tool writer can include the managers s/he needs and omit the remainder.

Many different module implementations. The implementations behind Maitri interfaces can take many different forms, making the framework flexible, adaptable, and extensible. This extensibility allows a tool writer to pick the most appropriate implementations for the tool at hand, and upgrade them in the future without disturbing end users.

Format independence. Format- and schema-dependent code resides only in Maitri's block manager module, which must be instantiated by a scientific tool writer before that tool can use Maitri. A block manager implementation will typically use a pre-existing scientific I/O library to navigate through a schema in the native format of that library, and to read and write data in units that are meaningful and appropriate for the tool. In return for the effort of writing a block manager, the tool writer can take advantage of the high-level functionality that is provided by Maitri's other modules and is not available in the scientific I/O library. Further, the high-level functionality can be implemented once and used with a variety of different block managers, each corresponding to a different storage format.

Better scalability. As their users will testify, the most popular scientific I/O libraries are significantly faster than traditional DBMSs for scanning typical large data repositories. The use of scientific I/O libraries as the lowest layer in the Maitri framework means that Maitri users can enjoy this same performance advantage. Where the libraries' performance falls short is in multi-file and selective data access; Maitri's buffer, index, and query manager interfaces and implementations are designed to overcome those limitations.

Better query interfaces. Today's scientific tool query interfaces typically allow users to restrict the values of a handful of variables (e.g., temperature > 0 AND pressure $<$ 100). Though scientists may not like SQL, in the long term a declarative query language is key to providing more powerful query capabilities in tool interfaces. Maitri is designed to allow the graceful adoption of declarative query languages for tools' selective data access.

The remainder of this paper is organized as follows. In Section 2, we present the Maitri architecture and its astonishingly narrow module interfaces. In Section 3, we discuss the functionality of the current implementations of these interfaces and point out numerous open problems specific to scientific data management that can be investigated within

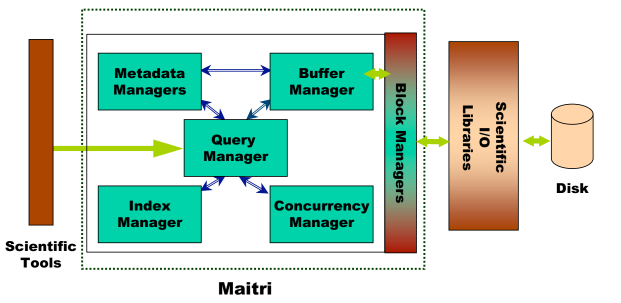


Figure 1. Architecture of the Maitri Framework for Scientific Data Management.

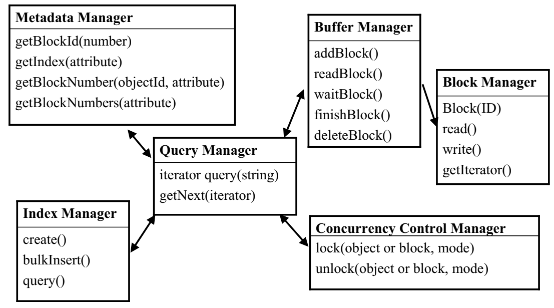


Figure 2. Maitri APIs.

the context of the Maitri framework.

2. MAITRI ARCHITECTURE

As shown in Figure 1, the Maitri framework provides a scientist with the most important features of a database in a modular lightweight package. End-user scientists usually access data through visualization and analysis tools. Maitri is intended to work both in coordination with such tools and as a standalone system. In essence, Maitri is a grey box between the scientific tool writer and the scientific I/O libraries being used to read and write the data.

When all of Maitri's modules are in use, the flow of control for processing a query is as follows. First, a scientific tool or user opens a data repository through a call to the **query manager (QM)**. Subsequently, the tool or user can forward a query to the QM. The QM asks the **metadata manager (MM)** for metadata about the repository and the attributes being queried (such as whether the attributes have indexes). The QM plans how to execute the query; depending on the QM's level of sophistication, this process can be trivial or very complex. During execution of the query, the QM passes control to the **index manager (IM)** to execute selected parts of the query. The QM then uses the MM to find out which data blocks to read to process the rest of the query.

Although scientific data is generally read-only once it is written, there are certain interesting cases where concurrency control is needed. (Unfortunately, space limits prevent us from describing them here.) Before reading or writing data or metadata that may be subject to conflicting access, the QM can invoke the **concurrency manager (CM)** to lock those objects and/or the data blocks containing them, using an appropriate lock mode defined in that particular instantiation of the CM. The QM calls the **buffer manager** to read the blocks into memory buffers efficiently. The buffer manager invokes the **block manager** to efficiently read blocks of data in a particular file format and schema.

Most of the Maitri managers are present in standard DBMSs. The most novel aspect of the Maitri architecture is the use of the block manager to encapsulate format-specific code, as discussed in more detail in the next section. The second novel aspect is in our intent that many scientific codes will not use all of the Maitri managers. If a tool writer chooses to omit the query manager (say), the writer is responsible for implementing the equivalent functionality directly in the tool. The tool can still access Maitri through any other instantiated module interface. For example, simulation codes are write-intensive. Their writers may choose to use just a buffer and block manager, as an easy way to get excellent overlap of writes and computation. The functionality that such codes require in terms of metadata management, a query language, and indexing will often be so trivial that it is simplest to code it directly in the tool. As another example, for archived, read-only data and metadata, there is no need to call a concurrency control manager. On the other hand, visualization tool writers will typically prefer to access data through a query manager. In this case, the writer needs to know the Maitri query API and the query language supported by the particular implementation of the query manager being used. The writer does not need to learn a new query language for each storage format.

Figure 2 shows the interface that each Maitri module currently exports. Due to space limitations, we have omitted most function arguments and return values in the figure. The interfaces are the outcome of over a decade of working directly with scientists and with the HDF group (and thus indirectly with thousands of other scientists); on the basis of this experience we claim that the narrow interfaces for the index, query, buffer, and block managers support 95% of the data management needs of the scientists we have worked with. Our experiments with the indexing, buffering, and block manager modules have left us confident in the effectiveness and completeness of these APIs for today's typical scientific data management needs.

With the current implementations of Maitri interfaces, the flow of control for a typical query is as follows. Consider the query "select velocity where temperature in [100, 500] & pressure in [5, 21]", over a database with an index on temperature. On receiving the query, the QM asks the MM to return links to indexes to temperature and pressure. The MM returns a link to the index to temperature and returns a null for the pressure index. A call to the IM then produces the IDs of the objects that have temperature between 100 and 500. The QM uses the MM to find out which buffers to read to evaluate the constraint on pressure. The QM also uses the MM to determine whether these buffers will also provide values of velocity (in practice, scientists will have stored the velocity and pressure values separately on disk). The QM then asks the buffer manager to read the required blocks in a background thread. The QM tells the buffer manager which blocks it will be reading in the future, so the buffer manager can prefetch them while the QM is working on the pressure constraint for another block. The buffer manager calls the block manager to read the pressure and temperature data in their native formats.

The interfaces in Figure 2 omit 90% of the calls that one might find in the API of a modern scientific I/O library or DBMS; where has all the complexity gone? Some of the complexity lies in the arguments passed to the managers. For example, the QM accepts strings in an arbitrary query language that can be as simple or complex as needed. Some of the complexity is hidden inside the managers. For example, a QM can have a sophisticated optimizer; a CM can have many different lock modes; and a block manager may need to make many calls to the underlying I/O library to instantiate a data block.

The interfaces allow an inept tool designer to really make a mess of things. This decision is deliberate. Without Maitri, tool designers are responsible for all aspects of data management except those encapsulated by the I/O libraries they use. Maitri does not free them from all these responsibilities. Instead, our intent is for Maitri to allow programmers to collaboratively build up layers of enhanced functionality that can be shared across groups and reused in many different contexts, so that tool writers will often be able to manage data at a higher level of abstraction and with greater power than they can without Maitri. Maitri does not free tool writers from all performance tuning responsibilities, either. The responsibility for performance tuning will lie primarily with the writers of block managers.

3. MAITRI MODULES

3.1 Block Manager

In traditional databases, a block refers to a logically contiguous portion of a file on disk. In Maitri, a block refers to a set of *logically* co-located objects that are to be read and written as a unit [6]. Only the block manager can read, write, parse, and understand the internal structure of blocks—and those are its chief responsibilities. The block manager is format dependent, and will usually be schema dependent as well; this sacrifice allows the rest of the managers to be written to be format independent. The block manager interface defines the following methods:

```
INTERFACE BLOCK {
   Block(BlockID id);
   read();
   write();
   Record getInstantiations(Attributes);
   Record getInstantiations();
 }
```

A BlockID object (which encapsulates metadata required to read/write the block) must be initialized before the corresponding block can be read or written using the block manager's read() and write() methods. Once a block has been read into memory, the block manager's getInstantiations() method can be called to parse and extract part or all of the data in the block, and place the data in newly created Maitri Record objects. The developer of a tool that will use Maitri must implement the getInstantiations method. Each Record object contains the values of the attributes for a particular object that resides in the fetched block.

While the block manager looks relatively simple, a good choice of definition of blocks is critical to good runtime performance. For example, if the data set is stored in many individual 2MB files, then a block should be defined as an entire file. On the other hand, if each file is 2GB, it is likely that a different definition of a block is needed for good performance. Without Maitri, scientific programmers are already implicitly saddled with the task of block definition (i.e., deciding how much data to fetch at a time), so in a sense we can hardly make things worse than they already are. However, without good prefetch and write-behind facilities, scientific programmers could never hope to achieve top performance. With the buffer manager's excellent prefetching and write-behind facilities, it is time to raise expectations. At a minimum, we need to help the programmer understand how different implementations of the block manager can change the performance of the tool being developed. Thus an excellent topic for future research is the development of guidelines and automated facilities that will help tool programmers to tune their block managers, i.e., to choose good block definitions.

3.2 Query Manager

Our current query manager implementation supports a very simple query language. All data entry is handled by bulk inserts, and all query conditions take the form of DNF range restrictions on attributes. The query optimizer generates an execution plan that uses all indexes available for restricted attributes, and then performs sequential scans to evaluate the restrictions on the remaining attributes. While this is sufficient for a surprisingly large proportion of scientists' needs, clearly one can do much better. In general, this area consists of uncharted waters packed with interesting research topics.

3.3 Metadata Manager

Scientific data are accompanied by a great deal of metadata, much of it associated with the provenance and storage encoding of the data. Typically, no analysis can be done without the metadata. Researchers who have addressed metadata management for scientific data [1, 8, 9] typically use a database engine to store the metadata, with the actual data residing in scientific-format binary files. This guarantees fast file I/O for data and sophisticated data management capabilities for metadata, but can cause problems for portability and/or performance. Maitri's metadata manager is loosely coupled with the other Maitri modules, so that any existing metadata manager implementation can be wrapped and used in Maitri—DBMS, special-format binary files, ASCII files, mathematical formulas or lookup tables. We have implemented two metadata managers for Maitri and identified several new research challenges for metadata management, which space limits prevent us from describing in this version of the paper.

3.4 Index Manager

Researchers have pointed out that traditional indexes such as B trees, hash tables, and even R trees, KD trees, and oct trees do not work very well with scientific data; bitmap indexes are the leading proposal for a general-purpose alternative [3, 12, 10]. While Maitri's modular architecture supports the use of many different kinds of indexes, we have bet the farm on a hierarchical enhancement of bitmap indexes that we call *multi-resolution bitmap indexes*. Our previous experiments with multi-resolution bitmap indexes showed excellent performance for range queries, regardless of the number of attributes being restricted or the width of the range restrictions [10].

3.5 Buffer Manager

Maitri's buffer manager is a modified and extended version of the Godiva buffer manager [6], which uses a relatively simple API to allow the programmer to specify blocks of data that can be read and stored in memory together. Godiva also allows the application developer to specify prefetching and buffering hints, which Godiva will use in deciding when to fetch/write a block and when to evict a block from its cache. Godiva fetches and writes blocks in a background thread. In Godiva, the block manager is an integral part of the buffer manager; in Maitri, the block manager is an independent entity and the buffer manager is format independent. All Maitri modules must call the appropriate block manager when they need to read or write blocks on disk, and when they need to extract information from blocks in memory.

When the query manager asks the buffer manager to read a block, the buffer manager initializes the block, and then uses the block manager's readBlock() methods (provided by the scientific tool writer) to read the data from disk. The tool writer provides hints to the buffer manager on how to most efficiently read the required data, using the methods addBlock() (which adds the block to the list of active blocks), readBlock() (a blocking call that forces the block to be read), waitBlock() (a blocking call that waits until the block is read in the order in which addBlock() calls were made to the buffer manager), finishBlock() (which removes the block from the list of active blocks but still keeps the block in memory for later use), and deleteBlock() (which removes the block from memory). While the scientific tool processes the data already read, the background I/O thread reads the blocks registered using the addBlock() method. A detailed description of Godiva and a performance study of with visualization tools can be found elsewhere [6].

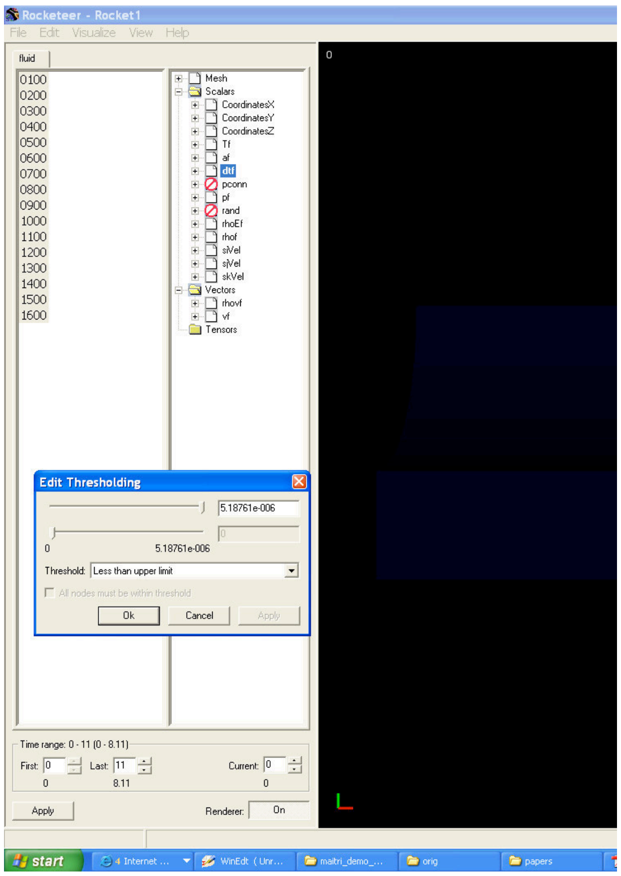
3.6 Concurrency Manager

Once written, scientific data is generally read/append only. Nonetheless, there are at least two interesting research challenges for concurrency control with scientific data and metadata. Both problems have been driven by requests for new HDF functionality from real-world scientists. The first problem occurs when multiple processors want to write compressed simulation snapshot data as the same time. If all processors write to different files, then the huge number of output files causes metadata management headaches. If all processors write to (different parts of) a single file, then it is not clear how much space to allocate for a processor's output, due to the compression. To address this problem, processors can compress relatively small amounts of data and write them out in a log-style approach. Inexpensive concurrency control is needed for the metadata associated with the tail of the log, and also for other metadata that subsequent readers will use to quickly find particular data within the log format.

The second concurrency control problem arises with applications that want to process data from sources that are still producing output. These read-write conflicts can occur even in sequential environments, if data is appended while another process is reading the data. If a scientist is doing online data analysis and visualization as the data arrive, then simultaneous reads and writes to the end of the file can cause inconsistency in the analysis. Inexpensive concurrency control is needed to address this problem.

4. DEMONSTRATION

The demonstration is based on the Rocketeer visualization code (www.csar.uiuc.edu/F_software/rocketeer/). Rocketeer is a powerful tool for visualizing 3D scientific data sets, developed at the Center for Simulation of Advanced Rockets at the University of Illinois. Rocketeer is primarily used to analyze numerical results from rocket simulations, but it can be used for viewing many other types of 3D data. Rocketeer is written in C++ and uses the Visualization Toolkit, which is based on OpenGL for accelerated graphics. Rocketeer differs from many visualization tools in its ability to handle many different types of grids on which the data is defined. The grid may be non-uniform, structured or unstructured, and multiblock. Rocketeer can display data from multiple files and/or multiple data sets from the same file in a single image. Rocketeer can perform a set of graphics operations on a series of data sets, to produce frames for animation. A parallel version of Rocketeer is available, but our demonstration uses the sequential version.



5.18761e-006
0
0 5.18761e-006
Threshold: Less than upper limit
All nodes must be within threshold
Time range: 0 - 11 (0 - 8.11)
First: 0 Last: 11 Current: 0
0 8.11 0
Renderer: On

Figure 3. Rocketeer Visualization Toolkit.

Rocketeer can show a grid on the surface of the computational domain and indicate the value of a scalar variable on that surface using a color scale. It can also display scalar variables as multiple isosurfaces and/or on slices across the x, y, or z axes. The surface, grid, isosurfaces, and slices can be semitransparent and can be cut away to allow a clearer view of the interior of a 3D volume. Rocketeer reads data stored in HDF4 format (www.hdfgroup.org) format or CGNS format (www.cgns.org). Metadata attributes stored with the data help Rocketeer interpret the contents of the file without additional user input.

Figure 3 shows the Rocketeer toolkit. The leftmost text box lists the data sets read (each set is a set of points that were calculated on a single processor). These are the blocks that are being visualized in the image on the right. The text box that is second from the left shows the variables that are being visualized. The scalar values have one value associated with every point, the vector values have 3 values per point, and the tensors have 9 values per point. Each attribute can be visualized on separate isosurfaces, and the "Edit Thresholding" bar can be used to specify constraints for a particular dimension.

We have modified Rocketeer so that it takes its input from a Maitri implementation, rather than from its usual input routines. We have also altered the handling of metadata so that it is stored by the metadata manager, rather than in the data files. This provides much more efficient metadata management, and makes the initial load of Rocketeer much faster for large data sets. We have also modified Rocketeer to support user thresholding on attributes that are *not* being visualized.

The demonstration consists of four main phases. The first phase demonstrates the improvements afforded by the Maitri buffer manager. In this phase, we show how hints from the user can help in overlapping computation with I/O to reduce total run time. In the second phase of the demonstration, we show how multi-resolution bitmap indexing supports efficient subsetting of data.

The third phase of the demonstration shows how the query manager efficiently interleaves computation and I/O. Since bitmap indexes are primarily compute bound, we stream results from the bitmap index manager to the buffer manager in such a way that the background thread of the block manager can start reading the data while the index manager is still determining which data buffers must be read. The query manager also overlaps the reads with the complex calculations of the visualization tool that must be performed before rendering can take place. This efficient overlap of reads and computation is key for high performance. The third phase also shows how consecutive visualizations can reuse results from previous visualizations as beginning blocks for further visualizations, showcasing the ability of the query manager to optimize across multiple queries.

The fourth and final phase shows that the block manager interface allows the visualization tool to switch between data formats easily, by shifting from the HDF block manager to the CGNS block manager.

5. CONCLUSION AND FUTURE WORK

We have proposed the Maitri framework as a way to let scientists have their cake and eat it too. With Maitri, scientists can keep their beloved data formats, along with the good performance of the I/O libraries for those formats, because Maitri uses those libraries to read and write data. Higher-level components of Maitri offer scientists lightweight, mix-and-match, data-format-independent modules that provide many of the features of DBMSs (indexing, buffering, concurrency control, declarative queries), without their drawbacks (proprietary data format, lack of portability, poor support for arrays). Certain aspects of the Maitri architecture make it possible for performance to scale beyond what scientists obtain with DBMSs or scientific I/O libraries. In particular, the superior performance of scientific I/O libraries for reading large repositories carries over to Maitri, through Maitri's use of these libraries for data reads and writes. Maitri helps overcome the libraries' performance problems for many-file data repositories, by supporting background prefetching and buffering across file boundaries. The price of this enhanced functionality is that the scientific tool writer must create a small amount of code to read and write data blocks. Our experimental results show the performance benefits of Maitri's approach with real-world data. We have also pointed out many remaining research challenges in creating and tuning implementations of individual Maitri modules.

6. ACKNOWLEDGMENTS

The research is supported by the Department of Energy under subcontracts B341494, DOE DEFC02- 01ER25508, by NSF grant NSF ACI 02-05611, and by CSE Fellowship.

7. REFERENCES

[1] 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 *High Performance Distributed Computing*, 1999.

[2] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[3] Department of Energy Office of Science Data Management Challenge. http://www.sc.doe.gov/ascr/Final-report-v26.pdf, 2004.

[4] J. Gray, D. T. Liu, M. A. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4), 2005.

[5] R. R. Johnson, M. Goldner, M. Lee, K. McKay, R. Shectman, and J. Woodruff. USD - a database management system for scientific research. In *SIGMOD*, 1992.

[6] X. Ma, M. Winslett, J. Norris, and R. Fiedler. Godiva: Lightweight data management for scientific visualization applications. In *ICDE*, 2004.

[7] R. Music and T. Critchlow. Practical lessons in supporting large-scale computational science. *SIGMOD Record*, 28(4), 1999.

[8] J. No, R. Thakur, and A. Chaudhary. Integrating parallel file I/O and database support for high-performance scientific data management. In *IEEE SC*, 2000.

[9] B. Plale, J. Alameda, B. Wilhelmson, D. Gannon, S. Hampton, A. Rossi, and K. Droegemeier. Active management of scientific data. In *Internet Computing*, 2005.

[10] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: A case study. In *IPDPS*, 2006.

[11] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *VLDB Journal*, 1993.

[12] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(2), 2006.