GODIVA: Lightweight Data Management for Scientific Visualization Applications

Xiaosong Ma
Department of Computer Science
North Carolina State University
ma@cs.ncsu.edu

Marianne Winslett
Department of Computer Science
University of Illinois at Urbana-Champaign
winslett@cs.uiuc.edu

John Norris Xiangmin Jiao Robert Fiedler
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign
{jnorris, jiao, rfiedler}@csar.uiuc.edu

Abstract

Scientific visualization applications are very data-intensive, with high demands for I/O and data management. Developers of many visualization tools hesitate to use traditional DBMSs, due to the lack of support for these DBMSs on parallel platforms and the risk of reducing the portability of their tools and the user data. In this paper, we propose the GODIVA framework, which provides simple database-like interfaces to help visualization tool developers manage their in-memory data, and I/O optimizations such as prefetching and caching to improve input performance at run time. We implemented the GODIVA interfaces in a stand-alone, portable user library, which can be used by all types of visualization codes: interactive and batch-mode, sequential and parallel. Performance results from running a visualization tool using the GODIVA library on multiple platforms show that the GODIVA framework is easy to use, alleviates developers' data management burden, and can bring substantial I/O performance improvement.

1 Introduction

Visualization of scientific and engineering data is becoming increasingly popular due to its ability to represent objects graphically, track their evolution over time using animations, and allow interactive exploration of data. Typically, visualization applications are data-intensive, and visualization software developers face challenges in both data management and I/O performance. In this paper, we address such challenges in visualization codes that visualize time-series data.

As a result of advances in scientific simulation codes and earth/space observational data collection, visualization tools today often need to manage a large number of datasets and do a lot of bookkeeping. In many scientific codes, including both simulation and visualization codes, programmers directly manage their raw datasets and meta data in a straight forward manner as arrays, to represent mesh coordinates, geometric interconnection graphs, unknowns in finite element method applications, DNA sequences, text strings, etc. Of course, keeping track of thousands of flat arrays that might be scattered through a complex code creates a huge management and code maintenance headache.

The fact that traditional database management systems are not suitable for managing scientific data has long been observed by researchers, and many previous studies address this problem [14, 15, 20]. We will discuss such related work in more detail in Section 2. In short, these previously proposed approaches and products provided database solutions to handle scientific data, but most of these solutions might be too *heavyweight* for the majority of parallel scientific applications.

Many scientific data management systems [14, 20] were built on top of full-fledged relational DBMSs such as POSTGRES and MySQL. In practice, most high performance computing users are reluctant to use such systems since it is difficult to build parallel scientific codes as database applications on high-performance platforms. The first reason for this is the lack of inexpensive support for database software on today's popular parallel platforms for scientific computing. Parallel scientific codes typically need to be ported to and run on multiple machines, often with different architectures and operating systems. Also, scientific application users are used to accessing their data stored in plain

arrays rather than through databases [13]. Further, in the previously proposed approaches, all the application data or else all the application meta data are stored in the database, creating data portability and flexibility problems. For example, many scientists need to perform pre-processing or post-processing on different machines than where the simulation or data collection program runs, by migrating their data written in portable files. It is hard to migrate user data when all or part of them are managed by a DBMS and are not externally accessible.

Another drawback is that many DBMS features, such as transaction processing, concurrency control, complex query processing and recovery, are not required or are required in a different form by most scientific applications. For example, scientific applications often periodically save their intermediate computation states, which serve as both snapshots for visualization and checkpoints for flexible restart, making the checkpointing scheme of a DBMS both redundant and unsuitable.

As data management problems have grown, I/O performance has deteriorated, primarily due to the enlarging gap between the performance of the CPU and the secondary storage system. Meanwhile, higher processing power and larger storage capacity enable scientists to produce and save more data, so visualization tools need to process larger volumes of time-series data and their periodic input operations are now more bottleneck-prone than ever.

Visualization applications may also have demands for output. For example, a visualization tool that processes a series of time-step snapshots to make pictures or movies needs to periodically write image files. However, the output workload is usually considerably smaller than the input workload, because the generated image files are normally much smaller in size compared to the input raw data files. Also, scientists often like to write data files using popular, standardized scientific data libraries [12] such as HDF, netCDF, and FITS, which have at visualization time a higher input cost than do plain binary files. In addition, interactive visualization tools often require much more input than output, since the users of these tools may browse a lot of datasets without saving anything, or may save only a small fraction of the data that they have viewed. As a result, input cost is very likely to dominate the total I/O cost in the execution of visualization applications. Therefore, we only examine the input problems of time-series data visualization tools in this paper.

In the past, we showed how to reduce the user-visible periodic output cost in simulation codes [11] by maximizing the overlap between output and other tasks. However, it is more difficult to overlap input operations with data processing in visualization codes. One reason is that a visualization code's data processing module depends on the data read from input files. Another reason is that how the input operations are performed often depends on information retrieved at runtime from the input files, making it hard for a general-purpose I/O library to make required interpretations of input data when performing background I/O. On the other hand, the two key conditions that allowed us to hide output costs in simulations, i.e., idle memory resources and computation phases that hide at least part of the I/O cost, do exist in visualization applications too. This is especially true with parallel visualization codes, where data can be partitioned onto multiple processors for higher aggregate I/O bandwidth, shorter response time and/or shorter total processing time. Exploiting memory space and I/O resources should benefit a visualization application's apparent I/O performance and overall performance. In addition, the idle memory can be used for caching as well, as data may need to be read more than once.

In fact, data access patterns in visualization codes show some advantages for high-level prefetching and caching. Normally, visualization software runs in either *interactive* mode or *batch* mode. In the interactive mode, users are provided with tools to interactively navigate through the data and can manipulate the visual rendering of the data, such as the view angle, color scale, level of detail, etc. In this mode, users' access patterns often bear a certain degree of locality [2]. For example, users may frequently switch back and forth between snapshot images from two different time-steps to observe the changes. Efficient caching can help reduce response time in this case. In the batch processing mode, the user specifies a series of files and lets the visualization program perform similar processing on all of them. The visualization program will go through these files and automatically generate a series of images, often for animation. In this scenario, data are read once and caching is not helpful. However, the set of files is pre-specified and the order of processing is known in advance, making batch-mode visualization applications wonderful candidates for user-level prefetching.

In this paper, we provide a novel solution to the data management and I/O performance problems in visualization applications: *lightweight* database support through the GODIVA (General Object Data Interfaces for Visualization Applications) framework. All data, including raw data arrays and meta data, are still externally stored as disk files in user chosen formats. Meanwhile, visualization tool developers are provided with interfaces to store datasets in an in-memory database (called the GODIVA database hereafter) when they read the file data in. The developers also have interfaces to query a dataset's buffer location during data processing. Further, there are high level interfaces for developers to specify the units for background I/O and to have a certain degree of control over library-level prefetching/caching performed automatically by the database system. Such high-level and general-purpose interfaces are to-

tally independent of the underlying operating system and architecture, and are easily implemented as a portable user library. Because reading the input files and interpreting their contents are done through developer-provided functions, this approach imposes no requirements on file formats whatsoever. If visualization tool developers decide to use GODIVA, they do not have to change how input files are written, and can switch to another input file format just by supplying a different read function.

We implemented the GODIVA interfaces in a standalone user library. Through our experience of using the library in a real-world parallel visualization suite, we found that GODIVA can both alleviate visualization tool programmers' data management burden, and bring significant overall performance improvement. In the following sections, we present our data access interfaces, describe our implementation of these interfaces, and present performance results and analysis.

2 Related work

Three projects closely related to our work are the Tioga system [20], the USD system [7], and the ADR framework [9]. Our GODIVA framework is similar to Tioga and USD in the sense that it was designed as a data management tool for scientific data visualization or analysis. The difference is that the GODIVA framework foregoes most of the standard database functionality, as well as graphical data modeling tools, in exchange for ease-of-use and portability. The ADR framework and GODIVA both facilitate parallel visualization, but unlike ADR, GODIVA does not have any specification regarding the software architecture of visualization applications using the framework or user dataset properties. In addition, as a general-purpose data management facility, GODIVA can be utilized by scientific data management systems that have more specialized data models (e.g., the IBM Visualization Data Explorer [1] and Conquest [19]).

Doshi et al. investigated caching and prefetching strategies to improve interactive visualization performance in exploring huge datasets [2]. These caching and prefetching techniques are built-in features of the authors' software for visualizing multivariate data, XmdvTool [17], and may require manual application-by-application analysis [2]. In contrast, the I/O optimizations in our framework are independent of visualization codes and the data they process.

No and colleagues proposed a scientific data management system [14, 15], which combines database and high-performance file I/O techniques. Their method uses a DBMS for meta data and file I/O for raw array data, creating a portability problem for both the scientific codes and their data. We address the same problem with a different approach: leave the array data and the meta data in user-accessible, portable files, and provide high-level data management interfaces that facilitate both computation and I/O.

Patterson et al. studied informed prefetching and caching [16], in which application codes can disclose future access patterns. This optimization is made at the operating system level and is based on file system buffer cache performance modeling. In contrast, our approach works in a portable user-level library, and allows visualization programmers to control user-level prefetching and caching through high-level interfaces.

Josifovski et al. constructed a lightweight, in-memory object-oriented DBMS named AMOS II [8]. AMOS II is designed for integrating distributed data sources and still provides full database query functionality. In contrast, the GODIVA framework is designed to work directly from user-owned disk files and only provides very limited database support.

3 The GODIVA interfaces

First, we briefly describe the "big picture" of the GODIVA framework. Figure 1 shows how visualization applications use the GODIVA interfaces. In this figure, solid arrows represent data flow and dotted arrows represent control flow.

The GODIVA interfaces can be grouped into three categories: for record operations, for background I/O, and for dataset queries. Typically, the background I/O interfaces and dataset query interfaces are used in the data processing code of a visualization program, while the record operation interfaces are used by developer-supplied read functions in allocating input buffers and reading data from input files into the GODIVA database.

The GODIVA database manages data buffer locations, without interpreting their contents, for a visualization application. The read function supplied by the visualization tool developer creates buffers in the GODIVA database and fills those buffers with data read from input files, while the data processing modules query the GODIVA database to find out data buffer locations and then access the buffers directly during their computation.

In addition, the GODIVA database manages a separate I/O thread for performing I/O in the background. The I/O thread carries out input operations using the developer-supplied read functions. The main thread, which executes the visualization program, can control the background I/O activities through the background I/O interfaces. For example, the main thread may tell the GODIVA library a sequence of files that are going to be processed, along with read functions for reading each file, and the GODIVA database will automatically initiate prefetching using the I/O thread and the specified read functions.

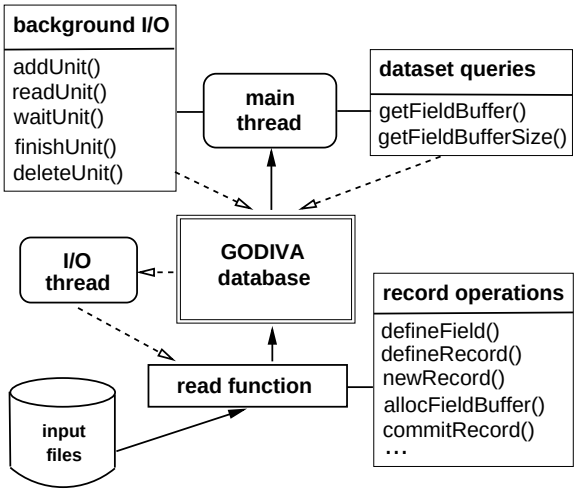


Figure 1. Sample usage of the GODIVA interfaces in a visualization code

In the next two sections, we present the three groups of GODIVA interfaces and their usage in more detail.

3.1 Data management and query interfaces

In this section, we introduce four GODIVA framework concepts: fields, records, field types, and record types.

The GODIVA framework does not distinguish raw array data from meta data. The basic data unit is a named developer-defined *field*, composed of an integer storing the data size and a pointer to a data buffer. Each field data buffer holds a piece of user data, either raw array data or meta data, stored contiguously in memory. Datasets, which may contain both raw array data and meta data items, are organized as *records*. In some sense, a record is very similar to a compound structure in C programming. Each record is a structure and each field of the record is a structure itself, containing a size and a data buffer. Formally, a record is a set of developer-defined fields. GODIVA manages the field data buffer addresses rather than the buffer contents. A program may retrieve a field buffer location from GODIVA, and subsequently access the buffer directly as if the buffer is a user-allocated array. This way, tool developers get the convenience of having a database to manage their datasets without changing the style of their computation code.

Just as database users can add data to a relational database by predefining the schema of a relational table and repeatedly inserting records that adhere to that schema, here tool developers can first define certain *field types* and *record types*, and then repeatedly create records with predefined record types. The interfaces `defineField` and `defineRecord` allow developers to define and name a new field or record type. A record type contains a set of field types, and each newly defined record type has an empty set of field types to begin with. Each field type has three elements: a field name, a field data type, and a pre-defined field data buffer size, all of which are defined through the `defineField` interface. If the data buffer size is not known when the field type is defined, it can be given the value `UNKNOWN`. The `insertField` interface is used to add a predefined field type into the field type set of a record type. When all field types needed have been added to a record type, the `commitRecordType` interface is used to conclude the record definition. Because data accesses in visualization codes using GODIVA are performed on individual fields, the ordering of fields in a record's internal storage is not important.

Table 1 shows a simplified record structure for the datasets recording intermediate results from a fluid dynamics simulation. The record shown stores fluid geometry and physics measurements on a structured 2-D mesh block, used to simulate a part of the fluid propellant in a rocket booster. This record type contains six field types, and each row in Table 1 shows the definition of one field type. The first two fields are meta data items with known sizes. The next four fields store raw array data whose sizes cannot be determined until the input data files are read. Besides the set of field types, a record type contains extra information regarding its key fields, as explained later in this section. In essence, field types and record types provide "templates" for datasets and their logical organization. Listed below is a sample code segment that defines the record type and related field types shown in Table 1.

```
defineField("block id", STRING, 11);
defineField("time-step id", STRING, 9);
defineField("x coordinates", DOUBLE, UNKNOWN);
defineField("x coordinates", DOUBLE, UNKNOWN);
defineField("pressure", DOUBLE, UNKNOWN);
```

field name	data type	buffer size
block ID	STRING	11
time-step ID	STRING	9
x coordinates	DOUBLE	UNKNOWN
y coordinates	DOUBLE	UNKNOWN
gas pressure	DOUBLE	UNKNOWN
gas temperature	DOUBLE	UNKNOWN

Table 1. Sample field types in a record type for a fluid data block

```
defineField("temperature", DOUBLE, UNKNOWN);

defineRecord("fluid", 2); // has 2 key fields

// Insert fields. The last parameter specifies
// whether the field is a key field.
insertField("fluid", "block id", true);
insertField("fluid", "time-step id", true);
insertField("fluid", "x coordinates", false);
insertField("fluid", "y coordinates", false);
insertField("fluid", "pressure", false);
insertField("fluid", "temperature", false);

commitRecordType("fluid");
```

After a record type has been finalized, record instances can be created using this committed record type through the `newRecord` interface. This will create, in the GODIVA database, a new record object with all the fields defined in the specified record type. If a field's size is not `UNKNOWN`, its data buffer will be allocated when the new record is created. Otherwise, the developer needs to explicitly allocate the field buffer using the `allocFieldBuffer` interface, passing the buffer size, field name, and a pointer to the previously created record object. This flexibility is especially useful in the common case where the data array size is not known until the meta data are read.

Figure 2 shows a record created with `newRecord` using the record type shown in Table 1, with all of its field buffers allocated. More specifically, this record stores a 2-D structured mesh block, which contains a 100×100 grid, with 101 coordinates each in the x and y directions. It therefore has $10,000$ rectangular elements, each with two element-based variables: pressure and temperature. All the coordinates and element-based variables are double-precision floating point numbers, as defined in Table 1.

To facilitate queries on data stored in the GODIVA database, we adopt the concept of a *key* from relational database systems. The combination of all the data buffer values associated with the *key fields* in a record uniquely identifies this record among all records with the same record type. Tool developers can specify a field as a key field when inserting this field type into a record type. When the key fields are filled with appropriate values, developers can use the `commitRecord` interface to insert the record into the GODIVA database's index system. The application code can then locate data stored in the database through the `getFieldBuffer` and the `getFieldBufferSize` interfaces. The `getFieldBuffer` interface takes a record type name, a field type name and an array of pointers to buffers holding key field values, and returns a pointer to the data buffer of the specified field in the record identified by the key value combination. The `getFieldBufferSize` works similarly, but returns the buffer size instead of its location. Unlike traditional databases, the GODIVA database does not handle queries over data buffer values, except for the above key lookup queries. For example, it can not return all the records whose field F has a value greater than a. The GODIVA database only organizes data as a collection of buffers, and field value checking is only used in key fields to identify a record. This minimal query interface design allows the GODIVA interfaces to work with all scientific data formats, and reduces the number of modifications for existing codes to use these interfaces.

For example, the record type shown in Table 1 has two key fields: "block ID" and "time-step ID". The GODIVA database can answer queries such as "give me the address of the pressure data buffer of the block with ID block_0003 from the time-step with ID 0.000075".

How does the application code know such key field values? Information such as the range of valid meta data values can be passed from the developer-provided read functions to the data processing modules via shared buffers known to both components, either managed by the GODIVA database or explicitly by the visualization code.

In summary, user data in the GODIVA database are organized in a way that can be viewed as a combination of the conventional database approach and the flat, straightforward "array-and-buffer" approach used in most scientific codes. We use database concepts such as records and fields to help developers keep track of their data, based on shared structures of datasets. At the same time, the database only

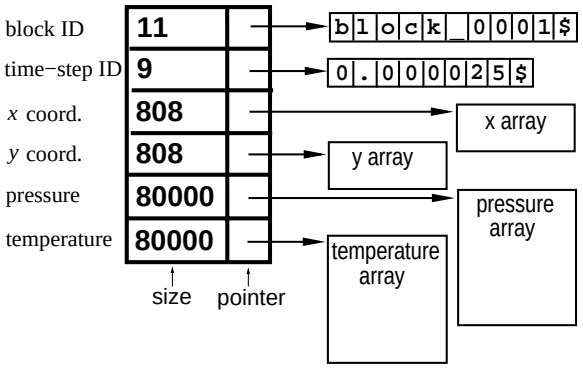


Figure 2. Sample of a record instance

manages data locations rather than data values, and the visualization code can directly access field buffers managed by the GODIVA database.

3.2 Overlapping I/O with data processing

In this section, we present the scheme for automatic yet developer-controllable prefetching to overlap I/O with data processing in visualization programs.

A *processing unit* is a set of records that will be brought in or evicted from the GODIVA database as a whole. Developers can define their own processing units by giving a unit name and a function that reads records belonging to this unit into the GODIVA database. A processing unit is the unit of data flow from the background I/O module to the data processing module. For example, a visualization tool developer may decide to define the set of all records read from the same input file as a processing unit. Or she may group records read from multiple input files that are part of the same time-step snapshot as a processing unit (data from these files are likely to be processed together to generate one or more images), and have a coarser prefetching granularity. Or she may use a subset of records from the same file or even a single record as a processing unit, and have a finer granularity. For brevity, we simply call a processing unit a *unit* in the rest of the paper.

Meanwhile, developers are provided with high-level interfaces to trigger prefetching or cache eviction. There are five such interfaces: `addUnit`, `readUnit`, `waitUnit`, `finishUnit`, and `deleteUnit`. These interfaces allow the application code to give hints to the GODIVA database regarding how prefetching and caching should be performed. When a unit is added with the non-blocking call `addUnit`, it is appended to GODIVA's prefetching list, and gets into line for its records to be prefetched into the database through the developer-provided read function for this unit. This read function will typically create records, allocate field buffers if necessary, and fill the buffers with contents read from input files. `readUnit` is a blocking call that explicitly reads a unit into the database using a developer-supplied read function. A `waitUnit` call will block the caller until the named unit has been read into the database. `finishUnit` tells the database that the processing of the named unit has been completed and the database may feel free to evict all its records. `deleteUnit`, on the other hand, explicitly orders the database to delete records of the named unit.

The GODIVA interfaces enable simple and effective prefetching or caching at the application level. For example, a visualization program running in batch mode can notify the GODIVA database about all the units to be read from all the input files (using the `addUnit` interface) in the order that they are going to be processed, and wait for each unit before the processing of that unit begins. The GODIVA database will automatically prefetch those units with the corresponding developer-provided read functions in the same order. Therefore, I/O to read subsequent units is overlapped as much as possible with the computation on the unit currently being processed. The `addUnit` interface allows developers to express their knowledge of future data access patterns, which can be far more accurate than the guesses of the GODIVA database or the underlying file system. Meanwhile, the batch-mode processing program code may want to issue the `deleteUnit` request immediately after a unit has been processed because it knows that the data will not be needed again. In contrast, an interactive visualization program may not be able to add units in advance since it does not know what the user sitting in front of the monitor will request next, and may simply use the explicit `readUnit` interface to perform foreground blocking I/O. However, an interactive tool perhaps will not delete units voluntarily, hoping that the user revisits some data that are still in the database. It is more likely for such a tool to mark a processed unit "finished" using `finishUnit` instead of deleting it. When GODIVA's memory space is low, it will choose "finished" units to evict from the database.

How much can the database prefetch or cache? This is limited by the amount of idle memory available. The maximum amount of memory that can be used for prefetching and caching is set by the visualization code when the GODIVA database is created, and this limit can be adjusted dynamically at runtime by the GODIVA interface `setMemSpace`. We assume that all the active data currently needed for processing can fit into the memory and whatever memory is left can be used for prefetching and/or caching, minus a small overhead for the record indexing system. The GODIVA database will keep prefetching as long as there are more units to prefetch, and there is memory space to hold more data. When there is no unit to prefetch, the database will try to keep as much data in memory as allowed by the developer-specified maximum database memory space. If the application code needs to read in more units and there is not enough memory space left, a caching replacement policy is used to choose units to evict from the database and make room for the incoming data.

To get benefits from the prefetching or caching mechanism, there must be at least enough idle space to hold one more processing unit than those currently being processed. This memory requirement is similar to that of the traditional double buffering approach. Scientific applications normally have very stable or at least predictable memory consumption, and today's parallel visualization tools can spread their work across more nodes to reduce the memory usage per node. In the GODIVA framework, a visualization code can set the maximum memory space to be used by the GODIVA database, and all the work in managing background I/O is hidden behind the high-level interfaces and taken care of by the database.

3.3 GODIVA interface implementation

In this section, we describe our implementation of the GODIVA interfaces and the GODIVA database as a user library. The library performs background I/O using an I/O thread that calls back developer-supplied functions to perform the file I/O and bring data into the database. Therefore, the GODIVA library does not have to understand how the actual reading is performed, which makes it independent of file formats, data layouts in files, meta data definition, and underlying scientific data library upgrades. Treating the developer-supplied read functions as black boxes does forfeit opportunities for the GODIVA library to perform additional I/O performance optimization. However, it allows the library to work with existing and future visualization tools in a much more flexible way and contributes to the library's portability.

We implemented the GODIVA library in C++, and implemented the GODIVA database as one object called GBO (GODIVA Buffer Object). The GODIVA user APIs are implemented as class member functions of the GBO class. We use the portable *pthread* library and a single background I/O thread is used for all the prefetching work. Each processor has its own database, which manages its local data, and there is no need for any communication between the GBO objects on different processors. Therefore the GODIVA library itself does not use parallel communication libraries such as MPI, making it more lightweight and portable.

Note that the read functions passed by the visualization code to the GODIVA database may perform parallel I/O that requires inter-processor communication between the background threads. In this case, it is the developer's responsibility to ensure that the communication library is thread safe when prefetching is used.

Once the GBO object is created, the background I/O thread is spawned and will start prefetching if there are units added. These units are internally organized in a FIFO queue, where each newly added unit is appended to the end of the queue. The records in the GODIVA database are organized in a C++ STL (Standard Template Library) map [18], indexed with the key field values in a RB-tree. Also, the records are indexed by units, so that when a unit is evicted from the cache, all of its records can be deleted efficiently. The `waitUnit` implementation blocks the main thread until the named unit is ready in the database, which is done through inter-thread communication mechanisms such as signals and semaphores. Reference counts are kept at the unit level. When memory space runs low and there is more reading to do, the database uses the LRU algorithm for cache replacement.

Listed below is a sample main program using the high-level unit interfaces. The whole GODIVA interface is managed by one GBO object, which `godiva` in the sample code below points to, and all the GODIVA interfaces are invoked through this object. In the code below, `godiva` is created with one integer parameter, which specifies the total memory size, in MB, that can be used by this GODIVA database. The program shown uses a file as a processing unit, and `read_file` is a pointer to a developer-defined function that reads datasets from an input file into the GODIVA database as one unit. Typically, this function defines the field and record types, creates and commits new records in the database, and fills field buffers with data read from the specified file. Here the same function is passed in adding both units, "fluid_file1" and "fluid_file2". This is because the unit name is passed back to the read function, and two different names can trigger different operations such as reading different files. The function `process_data`, on the other hand, is likely to contain the bulk of computation, where it uses the query interfaces to access datasets read into the memory. The background I/O thread is terminated when the GBO object is deleted.

```
main()
{
  GBO *godiva;

  godiva = new GBO(400);

  // add all units. "read_file" is the
  // function to read each file
  godiva->addUnit("fluid_file1", read_file);
  godiva->addUnit("fluid_file2", read_file);

  // process array records in fluid_file1
  godiva->waitUnit("fluid_file1");
  process_data("fluid_file1");
  godiva->deleteUnit("fluid_file1");

  // process array records in fluid_file2
  godiva->waitUnit("fluid_file2");
  process_data("fluid_file2");
  godiva->deleteUnit("fluid_file2");

  delete godiva;
}
```

As the GODIVA library is aimed at the needs of scientific applications, we do not provide as much concurrency control and data integrity checking as conventional business-oriented database systems do. For example, a developer may allocate field buffers without reading valid data into those buffers. It is the visualization tool's responsibility to make sensible use of buffer contents. Similarly, since key field values are stored in ordinary field buffers too, there is no way to prevent visualization codes from modifying key field values after a record has been committed to the database, rendering the indexing in the GODIVA database inconsistent with the actual key field values. However, as key fields store meta data items that should not require any modification in a visualization application, we believe this is not a problem. We do provide deadlock detection in our program, for the case when the main thread is waiting for a unit to be ready, and the background thread is blocked for lack of memory space. Given our assumption that the "active" data under processing can all fit into the memory, this should only happen when developers neglect to delete processed units or mark those units "finished".

4 Performance studies

4.1 The Rocketeer visualization tool overview

We evaluated the performance and ease-of-use of the GODIVA interfaces using Rocketeer, the in-house visualization tool at the Center for the Simulation of Advanced Rockets (CSAR) at Illinois, used in visualizing the rocket simulation data produced by CSAR's simulation code GENx. For example, the images and videos at www.csar.uiuc.edu/F_viz are all produced by Rocketeer. The Rocketeer visualization suite contains an interactive serial tool, an interactive tool with parallel processing in a client-server mode called Apollo/Houston, and a parallel batch mode program called Voyager. Programs in the Rocketeer suite are written in C++ and use the Visualization Toolkit [6]. As of this writing, Rocketeer reads data written in the HDF4 format [5] [1]. Currently the tools run on Linux, Solaris, AIX, and Microsoft Windows.

Rocketeer is a powerful visualization tool. It can handle many different types of grids on which the data is defined: non-uniform, structured, unstructured, and multi-block. It can display data from multiple files and/or multiple datasets from the same file in a single image. Unlike many visualization tools, Rocketeer provides users with both interactive and batch operation modes and users often need to use both of them. They can first take a look at the images generated from a few sample time-steps, rotate the camera angle, play with the color scale, etc., until they are satisfied with the visualization results. Then they may initiate a batch mode processing program, which grinds through a collection of files and makes a series of images with the visualization parameter settings chosen from the interactive process. The option of parallel processing enables Rocketeer to spread its work onto more processors, increase the utilization of I/O bandwidth, and potentially, have enough memory to hide I/O costs as proposed in this paper. This is especially important because we have observed relatively low data transfer rates in accessing files written using scientific data libraries such as HDF [11].

In this paper, we present results from Voyager [3], the batch mode parallel visualization tool of Rocketeer. Voyager is a command line tool that takes as arguments a camera position file, a graphics operations file, and a list of HDF files to process. The camera position and graphics operations files are generated during an interactive session of Rocketeer using a representative snapshot. Voyager uses MPI for inter-processor message passing and is scalable to a large number of processors.

4.2 Performance results with Voyager

In our experiments, we processed a subset of the snapshot files generated in a GENx simulation run. These snapshots store intermediate states of the solid propellant in a NASA Titan IV rocket body. The datasets contain the unstructured tetrahedral mesh, the connectivity information, and several node-based or element-based quantities:

[1] Upgrades to HDF5 or other scientific data libraries have been planned for Rocketeer and the rocket simulation codes that generate the data. Based on our previous experience with other file formats including HDF5 [10], we believe that our approach will deliver comparable performance benefit with file formats other than HDF4, in addition to the file-format-independent data management functionality.

a scalar measure of average stress, six components of the stress tensor stored as scalars, the displacement, velocity, and acceleration vectors, and several other quantities required for restarting. The original mesh contains 120481 nodes and 679008 elements in total, partitioned into 120 blocks (with a small amount of duplication of the boundary data). For each time-step snapshot, there are eight HDF4 files. In all of our experiments, we process 32 time-step snapshots. When GODIVA is used, Voyager uses all the files in the same time-step snapshot as a processing unit. Voyager calls GODIVA to add all the units to be processed at the beginning of the run, and to delete a unit after the processing of that snapshot finishes.

To test Voyager's performance with background I/O, we varied the relative amount of I/O by performing three visualization tests, called "simple", "medium", and "complex". The tests process different variables (e.g., velocity and stress) or have different visualization features (such as the requested surfaces, slices, and cutting planes). The "simple" test has the smallest ratio of computation work load to I/O load, while the "complex" test has the largest. For "simple", "medium", and "complex", the total size of input data per snapshot is 19.2MB, 30.1MB, and 16.6MB respectively.

We found that GODIVA brings a two-fold I/O performance benefit to Voyager. As expected, the background prefetching overlaps I/O with computation. In addition, GODIVA's data management facilities can help Voyager *reduce* the amount of I/O. With the original Voyager, reading data and processing data are closely coupled, and certain mesh data may need to be read in repeatedly if there is more than one variable to visualize. With GODIVA, when Voyager knows a particular unit is ready in the GODIVA database, it can retrieve the locations of the data buffers using the query interfaces, and reuse the data in those buffers. Therefore, redundant reads are eliminated. To separate GODIVA's effect in reducing I/O volume from its effect in hiding periodic input costs, we used two versions of the GODIVA library: single-thread and multi-thread. The single-thread version has all of GODIVA's usual record operation and query interfaces, but the background I/O is disabled: there is no I/O thread performing the `readUnit` operations in the background. Instead, a `readUnit` operation is performed inside the corresponding `waitUnit` call. This way, each `waitUnit` operation is equivalent to an explicit, blocking `readUnit` operation. The multi-thread version does the standard background I/O.

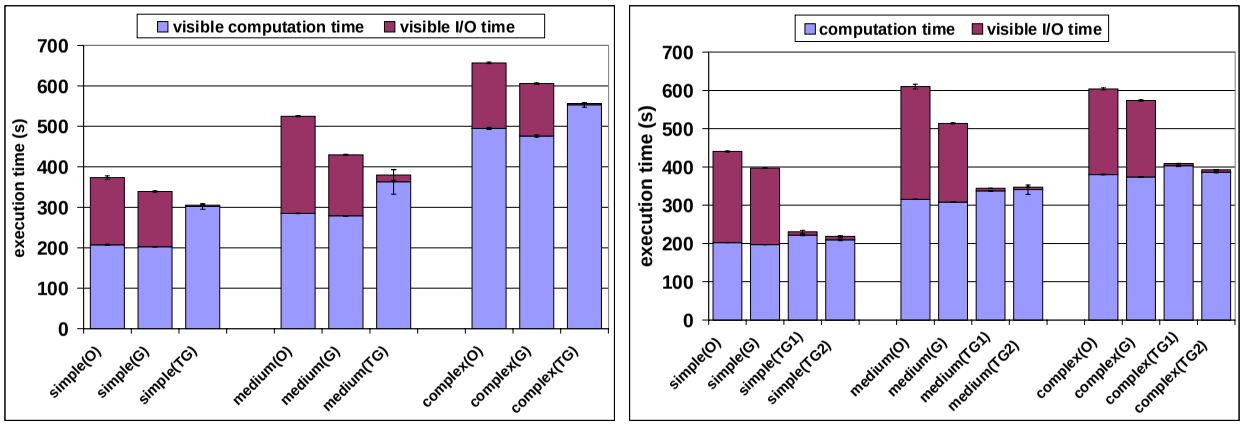
For each of the three visualization tests, we measured the total computation time and visible I/O time using three versions of Voyager: the original implementation without GODIVA (O), with the single-thread GODIVA library (G), and with the multi-thread one (TG). For each visualization test, these three versions of Voyager process the same datasets and the same visualization tasks. The visible I/O time (more accurately, visible input time) is the total time spent on reading the datasets with explicit, blocking read operations or waiting for units to be ready in memory. The computation time is calculated by subtracting the visible I/O time from the total execution time. We ran the tests on two platforms, a single-processor workstation and a dual-processor PC cluster. For each test, we report the average results from five runs. The error bars in the performance charts show 95% confidence intervals.

First, we discuss experiment results from a single-processor workstation called Engle. Engle is a Dell Precision 340 workstation with a 2.0 GHz Pentium 4 processor running Linux 2.4.20. It has 1GB RDRAM memory, and an 80 GB ATA-100 IDE 7200 RPM hard disk. Our experiments used the Linux ext2 file system. GODIVA's total memory space is configured as 384MB.

Results from Engle are shown in Figure 3(a). For all three visualization tests, GODIVA brings improvements to Voyager in both the I/O and the overall performance. By analyzing the Voyager code and the input datasets, we know that by using the GODIVA database, the volume of reads can be reduced by approximately 14%, 24%, and 16%, in the "simple", "medium", and "complex" tests respectively. These numbers tell us the difference in I/O volume between the original Voyager (O) and the one using single-thread GODIVA (G). Since O and G do not overlap I/O with computation, their total visible I/O time is the total actual I/O time. By comparing the I/O time of O and G, we can see that GODIVA reduces the total I/O time by 17.6%, 37.2%, and 20.1% respectively, in the corresponding tests. The extra savings in I/O time are mainly due to the much reduced disk seek time: the original Voyager needs to go back and force in a file to read the mesh data multiple times.

To measure how much I/O time can be hidden by performing prefetching, we compare the performance of the Voyager version using the multi-thread GODIVA library (TG) against that of Voyager using the single-thread one (G). These two versions perform the same amount of I/O, but with TG, all the units are added at the beginning of the run and GODIVA keeps prefetching as long as there is enough memory. In contrast, G does not overlap I/O with computation at all. From Figure 3(a), we see that with TG, the visible I/O time is dramatically reduced, but at the same time, the computation is considerably slowed down.

We measure the fraction of I/O time that is hidden behind computation by calculating what percentage of the total I/O time is saved: $\frac{total\_execution\_time_G - total\_execution\_time_{TG}}{total\_I/O\_time_G}$. The percentage of I/O cost that is hidden is 24.7% in the "simple" test, 33.1% in the "medium" test, and 37.8% in the "complex" test. The "medium" test results show larger variances and larger visible I/O time than those of the other two tests. This is primarily because the "medium" test has



(a) On the Engle workstation

(b) On a Turing cluster node

Figure 3. Voyager running time

the largest total data size, as well as the largest total number of record fields, among the three tests. Compared to the other two tests, in the "medium" test, the main thread is more likely to be blocked when it waits for the first unit, and the I/O thread is more frequently blocked due to a memory shortage. This causes more context switches, creates higher visible I/O time, and makes the computation performance less stable.

We calculate GODIVA's overall performance benefit by repeating the above calculation, but comparing the performance of TG to that of the original implementation (O). Overall, by both reducing the I/O volume and overlapping I/O with computation, GODIVA is able to reduce the input cost significantly from the original implementation on this single-processor workstation: 40.9% in the "simple" test, 60.5% in the "medium" test, and 61.9% in the "complex" test.

Next, we present the performance results from a dual-processor cluster node. We used one node of the Turing cluster, which is owned and managed by CSAR. Turing has 208 compute nodes running Linux 2.4.18, each with dual 1GHz Pentium III processors and 2GB memory. The nodes are connected with Myrinet. The file system used is REISERFS. Again, the GODIVA database is configured to use 384MB of memory.

Because each Turing node has two processors, we perform two sets of experiments with Voyager using the multi-thread GODIVA library, TG1 and TG2, to test the influence of computation on GODIVA's performance. With TG1, we run Voyager and another computation-intensive program to occupy both processors, and with TG2, we run Voyager only, which allows the background I/O thread to occupy the second processor most of the time. With the original Voyager implementation (O) and the one using the single-thread GODIVA library (G), only one processor is used.

Figure 3(b) shows results from a Turing node. Similar to the results from Engle, single-thread GODIVA reduces visible I/O time by 16.0%, 30.0%, and 10.7% in the "simple", "medium", and "complex" tests respectively. The performance of multi-thread GODIVA, however, shows a great difference from that on Engle. With the dual processor architecture, GODIVA is able to hide a much larger fraction of I/O costs: from 81.1% to 90.8% in all the TG1 and TG2 cases. In both the "simple" and the "complex" tests, TG2 shows a smaller computation slowdown factor than TG1, while in the "medium" test, TG2 and TG1 perform almost the same. Overall, I/O and computation overlaps much better on Turing than on Engle, primarily because of two reasons. First, the SMP-aware operating system kernel on Turing has a better scheduling scheme for multiple user threads. Second, by having a second processor, even when an extra computation-intensive process is running, the background I/O's impact on the main thread's computation is reduced, since the processes are scheduled in a round-robin way. Also, Turing has certain graphics software not available on Engle, making the computation time there impressive given its slower CPUs.

Overall, GODIVA is able to reduce the total input cost by up to 93.2% in the "simple" test, by up to 90.3% in the "medium" test, and by up to 94.7% in the "complex" test. These results show the great performance advantage of using GODIVA when multi-processor workstations are available. By having a background I/O thread performing automatic prefetching, GODIVA allows easy and efficient

utilization of idle resources that are normally not used effectively by the original visualization code running on such systems.

It is worth noting that the datasets we used are relatively small, with arrays (for mesh and variable data) ranging from 9600 to 48000 bytes. Therefore, our tests issued a large number of relatively small I/O requests, which is not ideal for overlapping I/O and computation. Applications that have larger data granularity should see bigger performance improvement using GODIVA.

We mentioned that GODIVA can work with both sequential and parallel visualization codes, and Voyager is indeed a parallel visualization tool. Currently however, Voyager can use at most one processor per node in parallel processing. Because Voyager partitions its workload between processors by assigning different processors different snapshots to process, there is little communication involved during the parallel processing except at the beginning of the run. In this case, we expect the speedup brought by GODIVA in parallel mode to be similar to that obtained in our sequential mode tests discussed above. This is confirmed by the results from a series of parallel experiments on Turing using four Voyager processes [10].

5 Conclusions and future work

Recently, computer scientists have realized that new DBMS technology is needed for scientists. For example, Peter Freeman and Lawrence Landweber, both of NSF, noted that such technology "should be able to manage data in a format dictated by the scientists rather than by the DBMS" [4]. Our work presented in this paper can be viewed as an effort to find an affordable and portable database solution for scientific applications.

The main contributions of this paper are as follows:

1. We designed the GODIVA framework, which contains an in-memory database and a small set of interfaces that provide light-weight data management support for visualization tools. The GODIVA database manages data buffer locations only, and helps developers better organize visualization data without modifying the way visualization computation is done. The GODIVA interfaces provide relational-database-like query interfaces for visualization codes to easily access their in-memory data. This is especially helpful for applications that need to process a large number of datasets.

2. We proposed a general-purpose approach to prefetching and caching for visualization tools. Our approach is not tailored to any specific visualization application or problem domain.

3. The GODIVA interfaces are portable and flexible. They have no requirements regarding the underlying machine architecture/operating system, or the visualization tool's software architecture, and can be used in either sequential or parallel visualization programs. Also, they place no restrictions regarding dataset properties or file formats. In addition, GODIVA interfaces allow visualization tool developers to supply their own input modules, and to specify the granularity of prefetching and caching. This way, developers can customize their codes in domain-specific ways, read data in any format, and perform prefetching/caching in the granularity most appropriate for their specific kind of visualization. GODIVA interfaces may also be used as a building block in implementing previously proposed domain-specific prefetching/caching techniques [2].

4. GODIVA's prefetching and caching can benefit both interactive and batch processing tools. Visualization tool developers can choose which GODIVA facilities to use in their codes.

Both the GODIVA framework and the Voyager visualization suite are on-going projects and we have several studies planned as future work:

1. We plan to evaluate the performance of GODIVA in the interactive versions of Voyager, when enough user traces are accumulated.

2. Voyager currently has problems in running multiple processes within an SMP node. When these problems are solved, we plan to experiment with GODIVA in a fully parallel version of Voyager, and investigate the GODIVA framework's performance benefit, as well as its scalability. Also, we plan to evaluate GODIVA on multiple parallel platforms.

3. We plan to experiment with GODIVA and other parallel visualization tools. Because parallel visualization is a new technology, this analysis of GODIVA must wait for such tools to become available.

4. Finally, as a relatively long-term plan, we plan to expand the GODIVA interfaces and implementation to support output through developer-supplied output functions, and enable the GODIVA database to facilitate developer-controllable caching that is not necessarily related to I/O.

6 Acknowledgments

This research was funded by the U.S. Department of Energy, through the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois at Urbana-Champaign, and through the Center for Programming Models for Scalable Parallel Computing.

References

[1] G. Abram and L. Treinish. An extended data flow architecture for data analysis and visualization. In *Proceedings of the IEEE Visualization*, 1995.

[2] P. Doshi, E. Rundensteiner, and M. Ward. Prefetching for visual data exploration. In *Proceedings of the 8th International Conference on Database Systems for Advanced Applications*, 2003.

[3] R. Fiedler and J. Norris. Massively parallel visualization on Linux clusters with Rocketeer Voyager. In *Proceedings of Linux Clusters: the HPC Revolution*, 2001.

[4] P. Freeman and L. Landweber. Cyberinfrastructure: Challenges for computer science and engineering research. Computing Research News, May 2003.

[5] http://hdf.ncsa.uiuc.edu/UG41r3_html/. *HDF 4.1r3 User's Guide*.

[6] http://www.kitware.com/products/vtkguide.html. *The Visualization Toolkit User's Guide*.

[7] R. Johnson, M. Goldner, M. Lee, K. McKay, R. Shectman, and J. Woodruff. USD - a database management system for scientific research. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992.

[8] V. Josifovski and T. Risch. Distributed mediation using a light-weight OODBMS. In *ECOOP Workshop on Object-Oriented Databases*, 1999.

[9] T. Kurc, U. atalyurek, C. Chang, A. Sussman, and J. Salz. Exploration and visualization of very large datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4), 2001.

[10] X. Ma. Hiding periodic I/O cost in parallel applications. PhD thesis, Dept. of Computer Science, University of Illinois, August 2003.

[11] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[12] J. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

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

[14] J. No, R. Thakur, and A. Choudhary. Integrating parallel file I/O and database support for high-performance scientific data management. In *Proceedings of Supercomputing '00*, 2000.

[15] J. No, R. Thakur, D. Kaushik, L. Freitag, and A. Choudhary. A scientific data management system for irregular applications. In *Proceedings of the 8th International Workshop on Solving Irregularly Structured Problems in Parallel*, 2001.

[16] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, 1995.

[17] E. Rundensteiner, M. Ward, J. Yang, and P. Doshi. XmdvTool: visual interactive data exploration and trend discovery of high-dimensional data sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[18] H. Schildt. *C++: The Complete Reference, Third Edition*. McGraw-Hill Companies, 1998.

[19] E. Shek, E. Mesrobian, and R. Muntz. On heterogeneous distributed scientific query processing. In *Proceedings of the 6th International Workshop on Research Issues in Data Engineering - Interoperability of Nontraditional Database Systems*, 1996.

[20] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *Proceedings of the 19th International Conference on Very Large Data Bases*, 1993.