A Distributed Scientific Data Visualization Framework for CFD Applications

Ramkrishna Chakrabarty

The Pennsylvania State University, University Park, PA

Alan M. Shih

University of Alabama at Birmingham, Birmingham, AL

Abstract

High-fidelity scientific simulations generate large amounts of data that need to be analyzed. Scientific visualization using computer graphics has become one of the major approaches to analyzing and interpreting such data. However, due to the ever-growing amount of data generated by the simulations, it is now a major challenge for scientists to visualize and analyze such extreme-scale datasets to explore the crucial information contained within. The efficiency of visualization techniques and approaches is therefore immensely important. There are two major challenges in addressing this issue, namely, a) the management of these huge quantities of data generated from various scientific experiments and simulations, and b) processing these enormous amounts of data in a more efficient manner. In this paper we present a distributed Computational Fluid Dynamics (CFD) data visualization framework that addresses these two challenges by implementing an efficient and scalable database for data storage. We also developed a proof-of-concept distributed visualization technique for visualizing streamlines that uses the aforementioned storage mechanism. Visualization examples of various CFD datasets will be presented to demonstrate the success of this approach.

Keywords: Scientific Visualization, High Performance Computing, CFD, Database

1. Introduction

High-resolution display systems such as the tiled display walls have become a popular way to display scientific data in high resolution. The use of such scalable display systems has enabled the display of high-resolution images with astonishing clarity that a single display system cannot achieve. However, most of the tiled display wall systems today only handle the rendering portion of the entire visualization pipeline, and thus do not efficiently use the computing power of the high-performance rendering cluster that drives the display wall. The entire visualization pipeline is implemented in a serial manner that reads data and is executed on the master node. Since visualization is a computing-intensive and data-heavy process, one can leverage the computing resources on this high-performance cluster for scientific visualization (SciVis) computation as well as the data storage involved in visualization. This research effort is to develop a Distributed Visualization Framework (DVF) that utilizes the high-performance cluster of a tiled display wall to perform complex computations involved in the entire visualization pipeline over and above rendering to improve the overall performance. There are two major challenges to addressing this issue. First is the efficient management of these huge quantities of data generated from various scientific experiments and simulations, and second, are the improvements necessary in computations for visualization involving enormous amount of data. To address these challenges, a distributed visualization framework was developed to support three different kinds of data management infrastructures to study their efficiency in the process, namely, a centralized database, a parallel database, and flat files, to store the data.

Typical scientific visualization applications nowadays are both computation-intensive and data-heavy that a local workstation can no longer hold and process the datasets. It challenges the user in both data preprocessing and management. One of the simplest database

approaches for storage and management is using a single database for storing both the metadata and the raw cell data [1]. In this approach, a multidimensional array is simply written to a BLOB (Binary Large Object), which is then stored in a field of a table in the database. Applications fetch the contents of the BLOB only when it is necessary to operate on the data. This method is made more efficient by storing the grid manipulation logic inside the server as UDRs (User Defined Routine) to minimize frequent BLOB input/output operations to read just the portions of the grid that are required. No *et al.* [2] proposed a Scientific Data Manager that uses parallel I/O and database to access large datasets. In this method the metadata is stored in a database, and the file is not split but is stored in a parallel file system and is accessed using an MPI File I/O library. To store very large datasets, this method has a lot of advantages. For example, there is no need to split the dataset, and the asynchronous reads are helpful in predictive techniques. However, this method is plagued by a major disadvantage due to the fast data retrieval requirement of this project. As the dataset is stored in a parallel file system, each file read call is made over the network, which can hamper the performance that is of paramount importance in this application. Doshi *et al.* [3] investigated caching and pre-fetching strategies to improve interactive visualization performance in exploring huge datasets. These caching and pre-fetching techniques are built-in features for visualizing multivariate data, XmdvTool [4], and may require manual application-by-application analysis. The GODIVA framework proposed by Ma *et al.* [5] provides simple database-like interfaces to help visualization tool developers manage their in-memory data and I/O optimizations, such as pre-fetching and caching, to improve input performance at run time. This interface is a stand-alone, portable user library, which can be used by all fashions of visualization codes: interactive or batch-mode, sequential or parallel. USD (Unstructured Scientific Data) [6] is a database system developed at Lawrence Livermore

National Laboratory that provides database capabilities required when doing scientific researches.

In recent years CFD has evolved into a widely used engineering tool to analyze the flow field associated with complex designs using sophisticated multi-physics models. It can produce very large sets of data. One of the common and important approaches to understand the flow field is to visualize the simulated data in a meaningful manner using various rendering techniques, such as streamlines or iso-surfaces. A streamline [7] is a line that is tangential to the instantaneous velocity direction, wherein velocity is a vector that has a magnitude and a direction. In other words, a streamline is a path traced out by a mass-less particle as it moves with the flow. It is primarily used to visualize vector data of the flow fields. Streamline calculations for large datasets is a computation intensive operation, since it is not easy to parallelize the calculation. Early research projects such as pV3 (parallel Visual3) [8, 9] breaks up the problem domain in space and places each partition on an individual workstation; streamlines are then calculated in a distributed, interactive manner. In particular, pV3 can couple visualization calculations with the simulation. Another popular solution involves using out-of-core processing of the dataset [10]. In this out-of-core approach, octrees are used as the framework for the data partitioning, since both the shape and resolution of unstructured data vary widely throughout the grid. Octree is a data structure that subdivides space recursively into eight cubes, each of which may be subdivided into eight sub-cubes, and so on. Using octrees makes it possible to refine the data partitioning in the regions where the grids are dense, such that subsets are relatively equal in data size. An out-of-core method is the only solution in the absence of large memory space and parallel computers.

2. Distributed Visualization Framework

The architecture of the distributed visualization framework (DVF) in this research effort and its various components are described in this section. This framework has been designed with the growing needs of scientific data management and high-performance visualization in mind. This framework is implemented on a 3x3 tiled display system at the University of Alabama at Birmingham. A 10-node dual-processor Linux cluster drives this nine-tile visualization wall. The bulk of the design considerations for the framework have derived from the following requirements: 1) *Efficient data management*. Efficient data management becomes imperative for this framework in the wake of the extremely large datasets to be visualized today. This efficient data management technique allows this framework to achieve significant data I/O speedup compared to the traditional file system based approach, and enable transparent high-level API for data access; 2) *Independence of individual visualization techniques*. The framework needs to provide a solution for any visualization technique or algorithm. In other words, this framework should be designed with generic and general algorithms in mind; and 3) *Scalability*. Another requirement for this framework is the scalability, as it is a necessary feature that allows the framework to maintain efficiency when the sizes of the data sets increase over time.

2.1 Hardware Architecture

An end-to-end framework has been implemented for distributed scientific data management and visualization. This framework primarily provides a solution for efficient management of scientific data, distributed visualization algorithms for high-resolution displays, and intuitive data management designs tailored for specific visualization algorithms. The distributed nature of the framework is primarily due to the distributed implementation of the visualization algorithms. This design consists of following main components: master

workstation, visualization cluster and data storage. The master workstation is a high-end workstation meant to initiate the algorithm, oversee the running of the framework, and render the geometry obtained at the end of the visualization pipeline. The visualization cluster is a computational cluster used for executing the compute-intensive visualization pipeline. The cluster helps in faster execution of the visualization algorithms facilitated by task parallelism and distribution. The data storage can be applied to any data modality, such as a database, a parallel database, a regular data file, or a data grid, depending on the user's choice. The visualization pipeline in the framework is executed as a five-step process in a distributed manner. In the first step, the dataset to be visualized is uploaded to the data storage and preprocessed. Next, the visualization algorithm is triggered on the visualization cluster. Then the cluster queries the data storage as and, when necessary, to create the basic visualization geometry. In the fourth step, the final geometry is sent back to the master. Finally, in the master workstation, the geometry is rendered on the high-resolution tiled display using the Chromium framework [11]. The dependence of the framework on the various dispersed resources for executing various tasks of the visualization pipeline makes it a distributed visualization framework.

2.2 Software Architecture

This framework has been implemented by designing and developing a collection of software tools that facilitate in the execution of the pipeline. The software that drives the framework comprises of Data Storage (DS), Data Abstraction Layer (DAL), and Distributed Visualization Engine (DVE) modules. DS module utilizes various types of data storage paradigms supported by the framework. This includes data files, databases, and parallel databases. However, this component can be easily extended to other modalities, such as data grids. DAL module acts as a generic wrapper surrounding the DS module, thus enabling the

visualization algorithms to be independent of the storage implementation. DVE module is designed to be component with visualization algorithms for a distributed environment. In this work, a distributed version of streamline visualization technique is developed for the DVE.

Data Storage (DS) Module

As a part of the data management services, this module performs functions such as data preprocessing, efficient data storage, and fast data retrieval. Three different kinds of data storage have been prototyped. These include data files, databases, and parallel databases. These are three different of data storage models, each of which is efficient for a specific range of data size. Data storage comes in as the first step in the DVF pipeline. It is divided into a master database and data file, a central database, or a parallel database containing the mesh/solution data, depending on the data storage model. The metadata is stored as a set of database tables in the master database. The list of metadata extracted from the input data includes dataset name, data type, number of cells, etc. This also contains an octree generated from the mesh data. Octrees [12] are used as the framework for the data partitioning, since both the shape and resolution of unstructured data vary widely throughout the grid. Using an octree allows us to refine the data partitioning in the regions where the grids are dense, such that subsets are relatively equal in data size (i.e., in terms of the number of cells). For constructing an octree, the data partitioning is carried out in a top-down manner. First, the whole data set is considered as one octant. This octant is then decomposed further using three cutting planes perpendicular to the x, y, and z-axes. If the number of cells in a child octant exceeds a predefined limit, the maximum octant size, this child octant is partitioned further. The above procedure is performed recursively until all octants contain fewer cells than the maximum octant size. Each octant stores the bounding box of the octant and the number of cells in the octant. Each leaf octant also *stores* the list of

cells in that node. This metadata is common to all the three models of data storage supported by this framework. Using a flat data file to store the data is the most common and a conventional approach for CFD data storage. This approach is simplest and well understood. But it may not be the best approach since the entire data has to be accessed in a sequential manner. When using the central database approach in DVF to store the data, a single database server is used as the primary data storage. In this case the framework uses the database facilities such as indexing, synchronized concurrent access to data i.e., multiple users of the database can safely query the same data simultaneously, and user-defined functions and multithreading to achieve faster data access. In addition to these metadata tables, the master database also stores two more tables, one each for mesh data and solution data. The mesh data is stored "cell-wise" in a single table, since most of the applications require mesh data in terms of cells. Likewise the solution data is also arranged cell-wise, with scalar(s) and vector(s) for each point.

In a more sophisticated storage scheme, a parallel database is employed. This approach is similar to the central database approach; the only difference being that a parallel database is used instead of a single central database. This approach is best when the dataset runs into terabytes or more that is too large to fit in a single database. In the parallel database approach the data preprocessor creates the metadata and splits the dataset into smaller lists of cells and distributes these groups of cells among the different database nodes in the cluster. Another metadata table is created to keep track of which cell is going to which database node. The parallel nature of the database is made transparent to the user by the database head-node. Just as in the database-driven approach above, here also all the data retrieving is implemented using SQL queries. However, in this case the database head-node parses the SQL query and determines which node to run the query on with the help of the metadata table, and then this query is

forwarded to the target database node that actually executes the query. In this manner, the user or the visualization engine never comes to know the nature of data storage. An independent instance of the database server is run on each of the nodes of the parallel database. The slave nodes store a subpart of the celldata/solution tables, as mentioned previously. Like the central database model, here also the metadata is stored in the master database, however, in this case an extra table is created to store the cell extents stored in each slave database node. PostgreSQL [13] is used as the database for storing metadata and the raw cell data. The POSIX threads [14] library is used for implementing multiple thread support in the framework.

Data Abstraction Layer Module

One of the most important features of DVF is its interoperability across central databases, parallel databases or data file as the data storage implementation. However, it would be a very inefficient design if all applications, including the CFD solver and visualization algorithms have to interface with all of these different types of data storage supported. On the other hand, it will be difficult to implement new data storage types if it is necessary to interface them with all of the visualization techniques implemented in the framework. Therefore, there is a need to provide an abstraction layer between the DVE and data storage. This way, the DAL eliminates the need for different upstream or downstream algorithms to maintain different data interfaces in the code in order to acquire data from a specific data storage implementation. DAL exposes a standard API on the input and output sides, allowing DVE and data storage to interact with each other and thus remain transparent to each other. It will also allow different applications other than visualization to access the data through the same APIs provided in DAL. The DAL hides all of the complexities of all the different data storage implementations such as central and parallel databases. DAL provides a simple, common, and unified API for the client application. When

using the DAL API, the client application requires *no database knowledge*. DAL encapsulates table creation, SQL statements to fetch data, and all other database specific jobs. It is implemented as a C^{++} class library. DAL exposes an interface for client applications for uploading the dataset onto the database. It also provides another interface for data query and data download purpose. *DALWriter()* and *DALReader(),* provide functionalities to upload or download the data to and from the data storage, respectively. A schematic layout of the DAL design is shown in Figure 1.

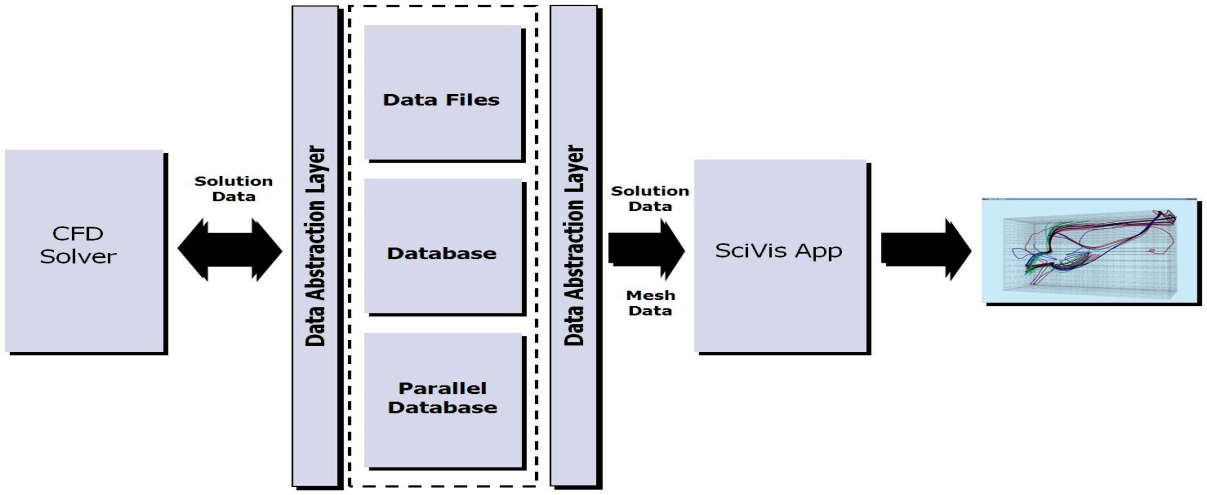


Figure 1. Schematic design of the data abstraction layer

Distributed Streamline Computation Module

A streamline is a path traced out by a mass-less particle as it moves with the flow. The position of each point in the path depends on the location of the previous point, the flow velocity vector on that point, and the scale of the time step to determine the next location for the streamline. Thus, to calculate a streamline, a starting point, a time step, and the maximum propagation time are required. Initially, the velocity vector at the starting point is fetched from the database. Finding the vector at a point is two-step process that is executed inside the database by user-defined functions. First, the cell is searched in the entire dataset, which contains

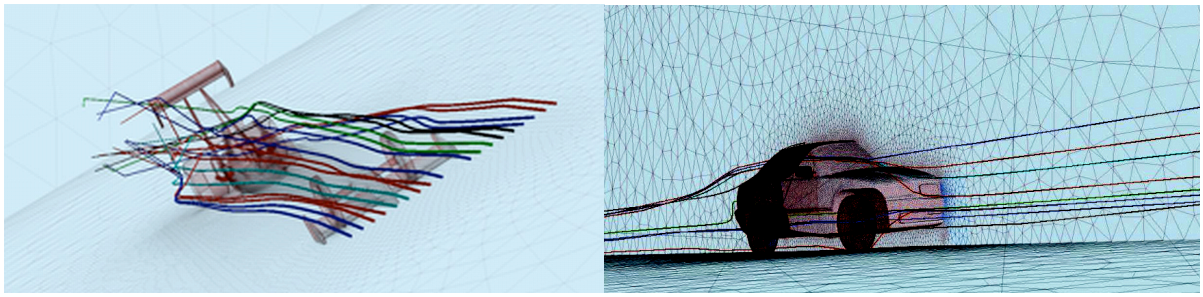
the point. Then, the vectors at the vertices of the cell are interpolated to find the vector. This operation is the most compute-intensive step in the entire streamline computation process. When the vector is obtained from the database, it is multiplied with the time step to get the next point on the streamline. Next, it is determined whether this point lies inside the dataset and if the number-of-streamline points calculated so far is less than the maximum propagation time. Finally, when all the points are calculated, they are sent to the visualization application for rendering.

This component's distributed nature comes not only from the fact that each streamline algorithm is computed using multiple nodes (compute and data), but also from the fact that multiple nodes can compute multiple streamlines simultaneously. The visualization module is a separate MPI application, which is linked to the DAL library. Here the master node sends out the initial parameters of each streamline to be generated to each of the slave compute nodes. The slave nodes compute the streamline using the *DALReader* class to fetch data, and finally return the streamline points to the master. In the final step the master collects all of the streamline points from all of the slaves and renders the streamline and the mesh geometry. This streamline calculation module was developed using LAM-MPI [15] implementation of MPI [16] for message passing in the distributed approach. The final visualization is done using Visualization Toolkit (VTK) [17].

3. Benchmarks

Two CFD datasets were used in this study. The first dataset is for the flow field about a drag racer vehicle, as shown in Figure 2(a). The second dataset is for a BMW 315 sedan, as shown in Figure 2(b). Streamline calculations were performed using the distributed visualization engine. Note the streamlines are of different colors, each generated by a separate thread of the

visualization module. The final rendering of the visualization is executed in the master node along with the rest of the mesh structure.



(a) (b)

Figure 2. Distributed streamlines calculation and visualization for (a) a drag racer vehicle and (b) a BMW315 sedan.

The benchmarking of the distributed visualization framework is done against a serial version with similar functionality that served as the baseline case. This baseline approach uses a flat file as the data storage. In the case of streamline visualization, it has to perform sequential searches through the flat files to find the cells containing the target point. The performance tests between the distributed and the serial frameworks are done using GNU *gprof*. In the case of streamlines, the time taken to compute a set of streamlines using different data storage techniques is compared.

3.1 Comparison between Times Taken to Compute a Set of Streamlines

A test dataset was initially uploaded to all of the three supported data storage implementations. Then a set of 16 streamlines is generated on 1, 4, and 8 processors. In this metric, a fixed set of 16 streamlines is computed on a varying number of processors. This test dataset with 17,515 cells stored in all of the three data storage implementations. Results from this metric show that, as the number of processors is increased, the time taken to compute the set of 16 streamlines decreases. However, an interesting observation can be made from Figure 3, that the time taken to compute the streamlines when the dataset is stored in a data file is more

than in the cases where the dataset is stored in either of the database implementations. The time taken by the central database and the parallel database remains nearly the same. This observation can be attributed to the fact that since multiple processes are trying to access a single data file simultaneously, this implementation becomes slower than any of the database implementations, as they support concurrent synchronous access to multiple users. Another reason for the database implementations being faster is the presence of such sophisticated search mechanisms as indexing, which is a standard feature of most databases.

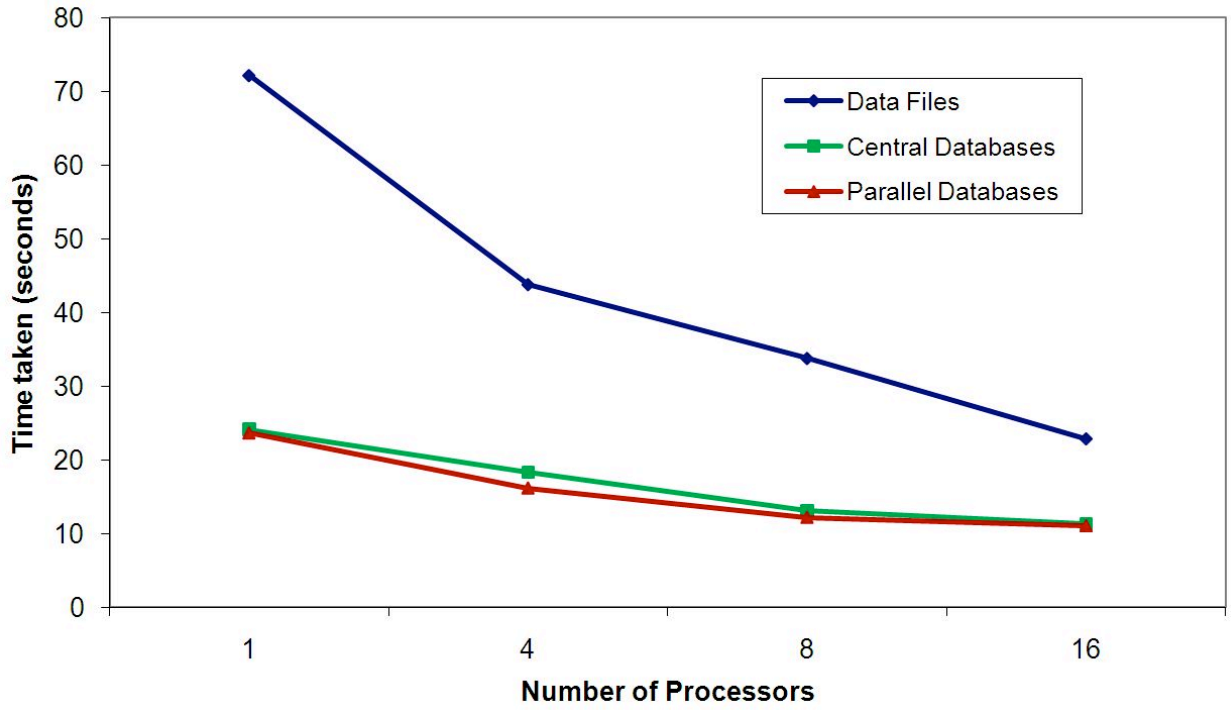


Figure 3. Comparison between times taken to compute a set of 16 streamlines for the test dataset

3.2 Comparison between Times Taken to Compute Varying Number of Streamlines

In this metric, the times taken by different data storage implementations are compared with an increasing number of streamlines computed. This test is conducted to study the effect of the number of streamlines and thereby the number of processors used simultaneously on different data storage implementations. Further this test is done to confirm the conclusions drawn

in the previous test. The result shows that the time taken by all of the three different data storage implementations do go up as the number of streamlines computed is increased. However, as concluded in the last test, the data file implementation becomes slower than the database implementations as the number of simultaneous streamlines computed is increased. As is clear from Figure 4, the data file implementation is faster when just one streamline is computed. However, as the number of streamlines computed is increased, the number of processes accessing the same file also increases, and thus the data file implementation becomes slower than the database implementations.

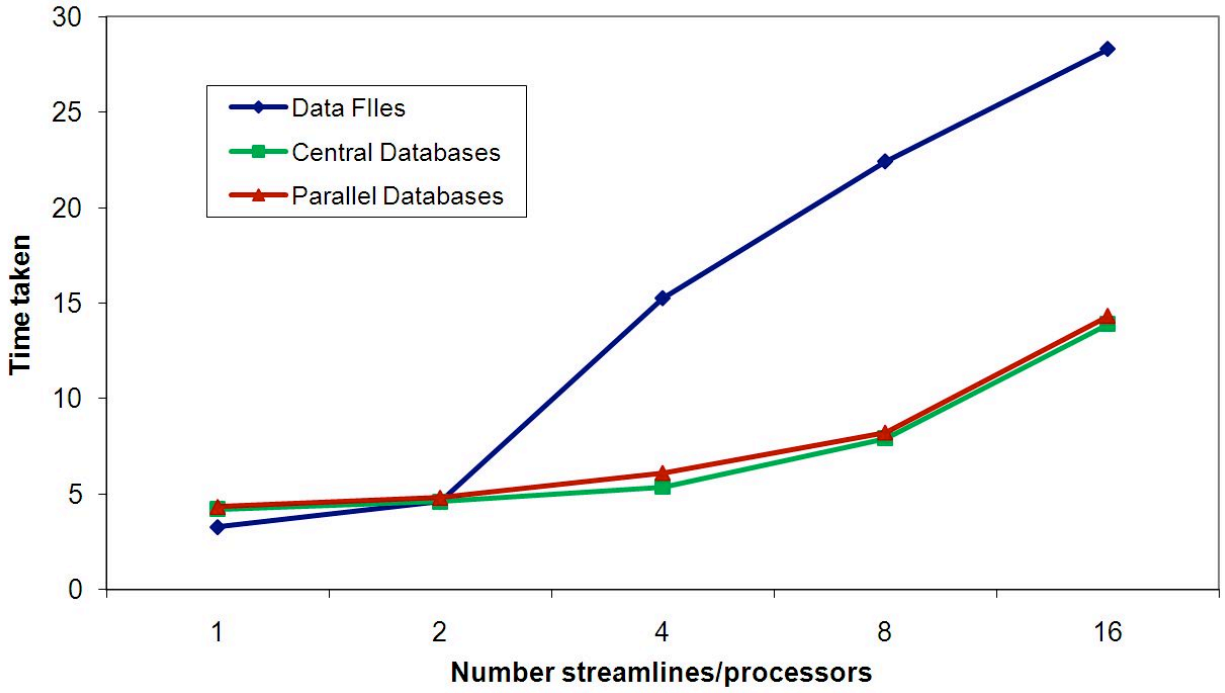


Figure 4. Comparison between times taken to compute varying number of streamlines for the test dataset

4. Conclusion

Distributed visualization is useful in increasing the performance of visualization techniques for large datasets. This work studied the effect of different data storage implementations, such as data files, central databases, and parallel databases, for high-

performance computing data on distributed visualization techniques. The main motivation of this work is to explore the performance differences of data storage implementations using database and traditional flat data files in the distributed visualization framework, especially in the case of large datasets. To achieve this objective, an elaborate distributed visualization framework was developed that supports data files, central databases, and parallel databases. A distributed version of the streamline algorithm was also developed to serve as the test-bed algorithm for comparison. Various fast data retrieval methods were implemented to improve the performance, such as the octree search algorithm and cell caching. Several metrics were developed to benchmark the performance of the distributed visualization framework. The results showed that in the case of computing a set of streamlines, the parallel database is the fastest. This can be attributed to various kinds of sophisticated search mechanisms already built into databases. Also, in databases there is concurrent synchronous access to multiple users, which is absent in normal data files, making them a slower data storage implementation.

Acknowledgement

The authors would like to express their gratitude to Dr. Purushotham Bangalore of Computer and Information Sciences department at UAB for his consultation on parallel computing, and Dr. Roy Koomullil of Mechanical Engineering department at UAB for providing the CFD datasets used in this study.

References

[1] Barrodale Computing Services Ltd. Storing and Manipulating Gridded Data in Databases. http://www.barrodale.com/grid_Demo/index.html, 2002

[2] 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

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

[4] 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

[5] Xiaosong Ma, Marianne Winslett, John Norris, Xiangmin Jiao, and Robert Fiedler. GODIVA: Lightweight Data Management for Scientific Visualization. To appear in Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)

[6] 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.

[7] L. Resenblum. Scientific Visualization: Advances and Challenges. Academic Publishers, 1994

[8] R. Haimes. pV3: A Distributed System for Large-Scale Unsteady CFD Visualization. AIAA Paper 94-0321, In Proceedings of AIAA 32nd Aerospace Science Meeting and Exhibit, Reno, Nev., January 1994.

[9] R. Haimes, and T. Barth. Application of the pV3 Co-Processing Visualization Environment to 3-D Unstructured Mesh Calculations on the IBM SP2 Parallel Computer. In Proceedings of CAS Workshop, NASA Ames Research Center, Mar. 1995.

[10] S. Ueng, K. Sikorski, and K. Ma. Out-of-Core Streamline Visualization on Large Unstructured Meshes. IEEE Transactions on Visualization and Computer Graphics 3(4): December 1997.

[11] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahem, P. Kirchner, and J.T. Klosowski, Chromium: A Stream Processing Framework for Interactive Rendering on Clusters of Workstations, ACM Transactions on Graphics (Proceedings of SIGGRAPH) 21(3), 2002.

[12] J. Wilhems and A. Van Gelder. Octrees for Faster Isosurface Generation. ACM Transactions on Graphics, 11(3): 201-227, July 1992

[13] The PostgreSQL Global Development Group. PostgreSQL 7.3.2 Programmer's Guide. http://www.postgresql.org/docs/, 2002

[14] Programming POSIX Threads. http://www.humanfactor.com/pthreads/posix-threads.html

[15] The LAM/MPI Team. LAM/MPI User Guide version 7.1.1. www.lam-mpi.org, 2004

[16] Message Passing Interface Forum. The MPI message-passing interface standard vers. 2.0. http://www.mpi-forum.org/docs/docs.html, May 1998.

[17] Kitware Inc. The Visualization Toolkit User's Guide. Kitware Inc Publishers, 2004