Exploration and Visualization of Very Large Datasets with the Active Data Repository *

Tahsin Kurc+, Ümit Çatalyürek+, Chialin Chang†, Alan Sussman†, Joel Saltz†+

† Institute for Advanced Computer Studies and Dept. of Computer Science
University of Maryland
College Park, MD 20742

+ Dept. of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

{kurc,umit,chialin,als,saltz}@cs.umd.edu

Abstract

The analysis and exploration of scientific datasets is a challenging and increasingly important part of scientific research. Visualization is a powerful analysis tool that converts numerical values into an image, which can be more easily analyzed by a human. We have developed a framework, called the Active Data Repository, that is designed to provide support for applications that analyze, explore and visualize very large multi-dimensional datasets. ADR targets distributed memory parallel machines with one or more disks attached to each node. In this paper, we present implementations of ray-casting based volume rendering and iso-surface rendering methods using ADR for visualizing out-of-core datasets, and describe experimental performance results.

1 Introduction

The primary goal of gathering data from experimental measurements and scientific simulations is to better understand the problem at hand. Understanding can be achieved by analyzing the data generated by simulations and experiments. With the help of more powerful computers and advanced sensors, long running, large scale simulations [8, 29, 40] and experimental measurements [1, 34, 42] are generating collections of very large datasets. The availability of low-cost systems built from networks of high-performance commodity computers and high-capacity, commodity disks has greatly enhanced a scientist's ability to store large-scale scientific data. For instance, a PC cluster with 800GB of disk storage can be built with 5 Pentium III PCs, each with two 80GB EIDE disks, for about $8K using off-the-shelf components. However, the vast amount of data in scientific datasets makes it an onerous task for a scientist to efficiently store the data, access the data of interest, and manage system resources to process the data.

Consider a scientist developing a characterization of ground waterways and examining various contamination remediation strategies. In some cases, the properties of a ground waterway or a reservoir may not

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408) and #ACI-9982087, the Office of Naval Research under Grant #N6600197C8534, Lawrence Livermore National Laboratory under Grant #B500288 (UC Subcontract #10184497), and the Department of Defense, Advanced Research Projects Agency, USAF, AFMC through Science Applications International Corporation under Grant #F30602-00-C-0009 (SAIC Subcontract #4400025559).

$O \leftarrow Select$(Output Dataset, Range Query)
$I \leftarrow Select$(Input Dataset, Range Query)
(* Initialization *)
foreach o_e **in** O **do**
  read o_e
  $a_e \leftarrow Initialize(o_e)$
endfor
(* Reduction *)
foreach i_e **in** I **do**
  read i_e
  $S_A \leftarrow Map(i_e)$
  foreach a_e **in** S_A **do**
    $a_e \leftarrow Aggregate(i_e, a_e)$
endfor
(* Output *)
foreach a_e **do**
  $o_e \leftarrow Finalize(a_e)$
  write o_e
endfor

Figure 1: The basic loop for processing data in the analysis of scientific datasets.

be well known, due to insufficient field data. Thus, several hydrodynamics simulations of the same region need to be run using different simulation parameters over many time steps. In one analysis scenario, the scientist visualizes the data from one of the datasets generated by one of the simulations over a time period of interest. In another case, the data values from two datasets, corresponding to two different simulations, are compared and the difference between data values from the two simulations is visualized. This analysis scenario would require applying application-specific comparison and difference operators on the data before the result can be visualized. For studying various contamination scenarios, on the other hand, data from hydrodynamics datasets in a region of interest over a subset of time steps is processed to provide input to a chemical transport simulator [25]. In all of these cases, an application developer could benefit from a framework that provides programming and runtime support for applications that make use of large scientific datasets.

Analysis usually involves extracting the data of interest from the dataset, and processing and transforming it into a new data product that can be more efficiently consumed by another program or by a human. In many scientific datasets, data items are associated with points in a multi-dimensional attribute space. The data dimensions can be spatial coordinates, time, or varying experimental conditions such as temperature, velocity or magnetic field values. Oftentimes, the reference to the data of interest can be described by a *range query*, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset(s). Only the data items whose associated coordinates fall within the multi-dimensional box are retrieved and processed. Figure 1 shows the high-level pseudo-code for the basic processing loop commonly executed by applications that analyze scientific datasets. The function $Select(...)$ identifies the set of data items in a dataset that intersect a given range query. An intermediate data structure, referred to as an *accumulator* can be used to hold intermediate results during processing. For example, a z-buffer can be used as an accumulator to hold color and distance values in a polygon rendering application [44]. Accumulator items are allocated and initialized during the initialization phase (steps 3–6). The processing steps consist of retrieving data items that intersect the range query (step 8), mapping the retrieved input items to the corresponding output items (step 9), and aggregating, in some application specific way, all the input items that map to the same output data item (steps 10–11). The mapping function, $Map(i_e)$, may map an input item to a set of output items. The aggregation function, $Aggregate(i_e, a_e)$, aggregates the value(s) of an input item i_e with the intermediate results stored in the accumulator item a_e that corresponds to one of the output items that i_e maps to. Finally, the intermediate results stored in the accumulator are post-processed to produce the final results for the output dataset (steps 13–16). Steps 1 and 4 are needed when the processing of data updates an already existing dataset, and data items are needed to initialize accumulator elements. The output can be stored on disks in the system (step 15) or can be consumed by another program (e.g., displayed by a client program in a visualization application). The output dataset is usually much smaller than the input dataset, hence steps 7–12 are called the *reduction* phase of the processing. Aggregation functions are usually *commutative* and *associative*, i.e., correctness of the output data values does not depend on the order input data items are aggregated.

We have developed a framework, called the Active Data Repository (ADR) [10, 19], that is designed to provide support for applications that analyze very large multi-dimensional datasets on a distributed memory parallel machine with one or more disks attached to each node. ADR targets applications with the processing structure shown in Figure 1. It is designed as a set of modular services implemented in C++, which can be customized for application-specific processing. The ADR runtime system supports common operations such as memory management, data retrieval, and scheduling of processing across a parallel machine. In this paper we describe the implementation of ray-casting based volume rendering and iso-surface rendering using ADR for visualizing large datasets, and present experimental results on a cluster of PCs.

2 Active Data Repository

The Active Data Repository (ADR) [10, 19] is an object-oriented framework designed to efficiently integrate application-specific processing with the storage and retrieval of multi-dimensional datasets on a parallel machine with a disk farm. ADR consists of a set of modular services, implemented as a C++ class library, and a runtime system. Several of the services allow customization for user-defined processing. An application developer has to provide accumulator data structures, and functions that operate on *in-core* data (i.e. the $Initialize$, Map, $Aggregate$, and $Output$ functions) to implement application-specific processing of *out-of-core* data. A unified interface is provided for customizing ADR services via C++ class inheritance and virtual functions. The runtime infrastructure provides support for common operations such as index creation and lookup, management of system memory, and scheduling of data retrieval and processing operations across a parallel machine.

Multiple application-specific customizations of ADR services can co-exist in a single ADR instance, and the runtime system can manage multiple datasets simultaneously. This makes it possible to develop and deploy data servers that enable, besides direct analysis and visualization of a dataset, generation of data products from one or more datasets that can be used by another program.

2.1 System Architecture

An application implemented using ADR consists of one or more *clients*, a *front-end process*, and a customized *back-end* (Figure 2). The front-end interacts with clients, translates client requests into queries and sends one or more queries to the parallel back-end. Since the clients can connect and generate queries in an asynchronous manner, the existence of a front-end relieves the back-end from being interrupted by clients

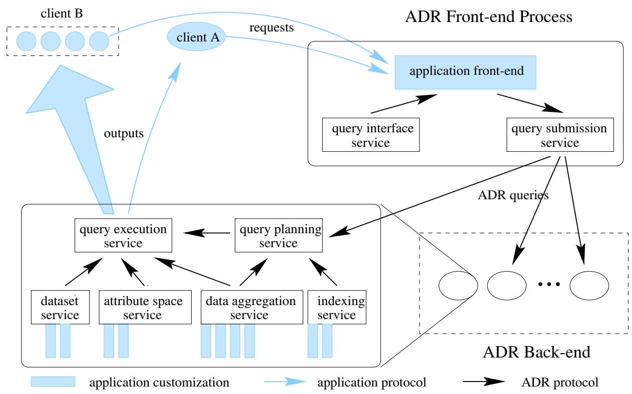


Figure 2: An application suite implemented using ADR. The shaded bars represent functions added to ADR by the user as part of the customization process. Client A is a sequential program while client B is a parallel program.

during processing of queries. The back-end is responsible for storing datasets and carrying out application-specific processing of the data on the parallel machine. The customizable ADR services in the back-end include: (1) an *attribute space service* that manages the registration and use of user-defined mapping functions; (2) a *dataset service* that manages the datasets stored in the ADR back-end and provides utility functions for loading datasets into ADR; (3) an *indexing service* that manages various indices (default and user-provided) for the datasets stored in ADR; and (4) a *data aggregation service* that manages the user-provided functions to be used in aggregation operations, and functions to generate the final outputs. This service also encapsulates the data types of both the intermediate results (accumulator) used by those functions and the final output datasets.

2.2 Datasets in ADR

A dataset in ADR is stored as a set of data chunks, each of which consists of a subset of data items. A chunk is the unit of data retrieval in ADR. That is, a chunk is retrieved as a whole during processing. Retrieving data in chunks instead of as individual data items reduces I/O overheads (e.g., seek time), resulting in higher application level I/O bandwidth. As every data item is associated with a point in a multi-dimensional attribute space, every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates of all the items in the chunk. The dataset is partitioned into data chunks by the application developer, and data chunks in a dataset can have different sizes. Since data is accessed through range queries, it is desirable to have data items that are close to each other in the multi-dimensional space placed in the same data chunk.

Data chunks are distributed across the disks in the system to fully utilize the aggregate storage space and disk bandwidth. In order to take advantage of the data access patterns exhibited by range queries, data chunks that are close to each other in the underlying attribute space should be assigned to different disks. By default, the ADR data loading service employs a Hilbert curve-based declustering algorithm [18] to distribute the chunks across the disks. Hilbert curve algorithms are fast and exhibit good clustering and declustering properties. Other declustering algorithms, such as those based on graph partitioning [33], can also be used by the application developer. Each chunk is assigned to a single disk, and is read and written only by the local processor to which the disk is attached. After data chunks are assigned to disks, a multi-dimensional index is constructed using the MBRs of the chunks. The index on each processor is used to quickly locate the chunks with MBRs that intersect a given range query. An R-tree [21] implementation is provided as the default indexing method in ADR, but user-defined indexing methods can also be provided.

2.3 Processing in ADR

The processing of a query in ADR is accomplished in two steps: a *query plan* is computed in the *query planning* step, and the actual data retrieval and processing is carried out in the *query execution* step according to the query plan.

Query planning is carried out in three phases: *index lookup*, *tiling* and *workload partitioning*. In the index lookup phase, indices associated with the datasets are used to identify all the chunks that intersect with the query. If the output/accumulator data structure is too large to fit entirely in memory, it is partitioned into *tiles* in the tiling phase. The ADR data aggregation service provides C++ base classes, which are customized by an application developer for tiling the accumulator data structure. Each tile contains a subset of the accumulator elements so that the total size of a tile is less than the amount of memory available for the accumulator. A tiling of the accumulator implicitly results in a tiling of the input dataset. Each *input tile* contains the input chunks that map to the corresponding output tile. Since an input element may map to multiple accumulator elements, the corresponding input chunk may appear in more than one input tile if the accumulator elements are assigned to different tiles. During query execution, the input chunks placed in multiple input tiles are retrieved multiple times, once per output tile. In the workload partitioning phase, the workload associated with each tile is partitioned among processors. In the current ADR implementation, the entire accumulator tile is replicated on each back-end processor, and each processor is responsible for processing local input data chunks. This strategy is similar to the sort-last rendering approach [32]. In the query execution step, the processing of an output tile is carried out according to the query plan. A tile is processed in four phases.

1. **Initialization**. Accumulator elements for the current tile are allocated space in memory and initialized in each processor.

2. **Local Reduction**. Each processor retrieves and processes data chunks stored on local disks. Data items in a data chunk are mapped to accumulator elements and aggregated using user-defined functions. Partial results are stored in the local copy of the accumulator tile on a processor.

3. **Global Combine**. Partial results computed in each processor in phase 2 are combined across the processors via inter-processor communication to compute final results for the accumulator.

4. **Output Handling**. The final output for the current tile is computed from the corresponding accumulator values computed in phase 3. The output is either sent back to a client or stored back into ADR.

A query iterates through these phases repeatedly until all tiles have been processed and the entire output has been computed. The output can be returned to the client from the back-end nodes, either through a socket interface or via Meta-Chaos [17]. The socket interface is used for sequential clients, while the Meta-Chaos interface is mainly used for parallel clients. Figure 3 illustrates the processing of an output tile in ADR through the four phases.

Note that ADR assumes the order the input data items are processed does not affect the correctness of the result, i.e., aggregation operations are commutative and associative. Therefore, the runtime system can order the retrieval of input data chunks to minimize I/O overheads. Moreover, disk operations, network operations and processing are overlapped as much as possible during query processing. Overlap is achieved by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switching between queued operations as required. Pending asynchronous I/O and communication operations in the operation queues are polled and, upon their completion, new asynchronous operations are initiated when more work is required and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion. For portability reasons, the current

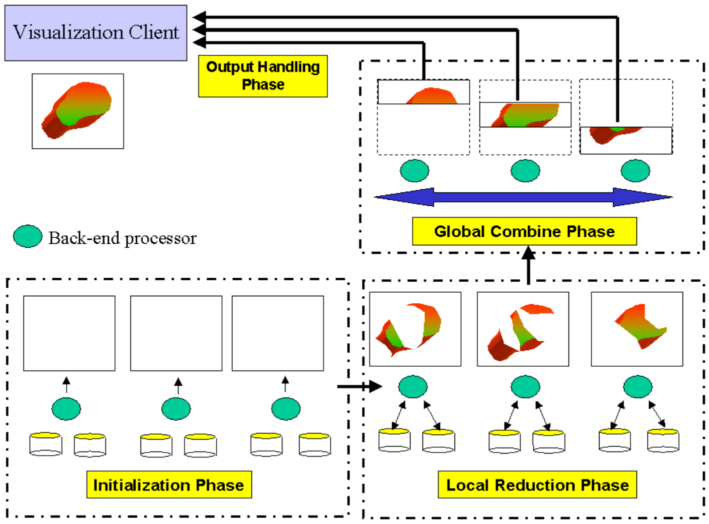


Figure 3: Processing of an output tile in ADR.

ADR implementation uses the POSIX `lio_listio` interface for its non-blocking I/O operations, and MPI [39] as its underlying interprocessor communication layer.

The back-end can execute multiple queries concurrently. Each query is assigned its own workspace (e.g., memory for accumulator data structure). The runtime system switches between queries to issue I/O and communication operations, and handles the computation for a query when the corresponding I/O and communication operations complete.

3 Visualization Applications

In this section we present the implementations of ray-casting based volume rendering and iso-surface rendering using ADR for visualizing out-of-core simulation datasets.

3.1 Out-of-core Iso-surface Rendering of Datasets from Environmental Simulations

The study and characterization of surface and ground waterways (e.g., aquifers, bays and estuaries) and oil reservoirs involve simulation of the transport and reaction of various chemicals over many time steps on a three-dimensional grid that represents the region of interest. In a typical analysis of datasets generated by the simulations, a scientist examines the transport of one or more chemicals in the region being studied over several time steps. Iso-surface rendering is a well-suited method to visualize the density distributions of chemicals in a region. In this section we present an iso-surface renderer implemented using ADR to render large, out-of-core datasets, where data values are associated with points in a rectilinear grid. Figure 4 shows the client interface for the prototype implementation.

Given a three-dimensional grid with scalar values at grid points and a user-defined scalar value, called the iso-surface value, an iso-surface rendering algorithm extracts the surface on which the scalar value is equal to the iso-surface value. The extracted surface (iso-surface) is rendered to generate an image. The ADR implementation uses the marching cubes and polygon rendering functions of the Visualization Toolkit (VTK) [37] for extracting and rendering an iso-surface. The marching cubes algorithm [28] is a widely used algorithm for extracting iso-surfaces from a rectilinear mesh. In this algorithm, volume elements (voxels)

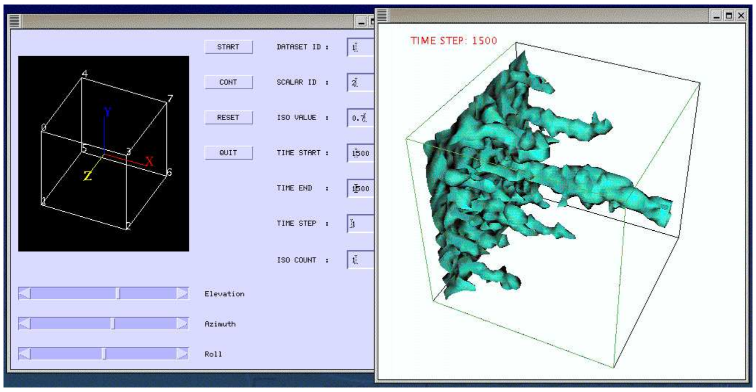


Figure 4: Client interface for the iso-surface renderer implemented with ADR.

are visited one by one. At each voxel, the scalar values at the corners of the voxel are compared to the iso-surface value. If all the scalar values are less than or greater than the iso-surface value, no surface passes through the voxel. Otherwise, using the scalar values at the corners, the algorithm generates a set of polygons that approximate the surface passing through the voxel. The polygons from the current voxel are added into a list of polygons. An image of the surface from a viewing direction can be generated from the list of polygons using a polygon rendering algorithm (e.g., the z-buffer algorithm).

3.1.1 Out-of-core Algorithm

In the out-of-core algorithm, the three-dimensional grid is partitioned into sub-volumes in each dimension. For a grid with $N \times M \times K$ voxels, a sub-volume contains $\frac{N}{n} \times \frac{M}{m} \times \frac{K}{k}$ adjacent voxels, where n, m, and k are the number of divisions in each dimension. Note that two voxels in the grid may share one or more grid points. If two such voxels are assigned to different sub-volumes, the shared grid points are replicated in each sub-volume. A sub-volume corresponds to a data chunk in ADR. Sub-volumes are declustered across the disks in the system, and a sub-volume is assigned to a single disk.

The bounding box of a sub-volume is defined by the coordinates of its eight corner points in three-dimensional space. The metadata for a sub-volume consists of the bounding box, a pointer to the file that contains the sub-volume, an offset in the file and the size of the sub-volume. The minimum and maximum scalar values across all the voxels in a sub-volume are stored along with the bounding box information. The minimum and maximum scalar values for a sub-volume are used to quickly test if an iso-surface passes through the voxels in the sub-volume, to decrease the amount of I/O during processing. If the iso-surface value is less than the minimum or greater than the maximum scalar value for a sub-volume, no surface can pass through the voxels in the sub-volume, and the sub-volume is not retrieved from the disk. The current implementation uses a simple index. An array is used as an index for each time step in the simulation. Each entry in the array corresponds to a sub-volume and stores a pointer to the meta-data for the sub-volume. Another array is used to access the index array for a single time step. Each entry of this array corresponds to a time step and stores a pointer to the index array for the time step. Both the index array for all time steps in the dataset, and the index arrays for individual time steps are stored in a file. The ADR indexing service has been customized to implement this indexing scheme.

The ADR services have been customized using the rendering functions from the VTK library [37]. For iso-surface rendering, the VTK functions that extract an iso-surface from an *in-core* volume and render the iso-surface to produce a partial image, correspond to the user-defined aggregation functions in ADR. A 2-dimensional z-buffer [44] is used as the ADR accumulator structure to store partial images. Each z-buffer entry corresponds to a pixel in the image plane, and stores a color value (an RGB value) and a z-value. The z-value is the distance of the foremost point to the image plane among all the points on the iso-surface that map to the same pixel with respect to a viewing direction. The color value is the color of the foremost point.

During the ADR local reduction phase, each processor reads the sub-volumes that intersect with the query from its disks. When a sub-volume is retrieved and available in memory, VTK functions are called by the ADR aggregation service to apply the marching cubes algorithm to the sub-volume and render the iso-surface. The partial image generated from the sub-volume is merged with the partial image stored in the z-buffer. Since iso-surface extraction and polygon rendering are both associative and commutative operations, only one z-buffer is needed to store the partial image produced. In the ADR global combine phase, partial images are merged across the processors to generate the final image. The partial image in a processor is partitioned into P equal-sized strips, where P is the number of processors in the ADR back-end. Each processor sends strip k of the local partial image to processor k. When a processor receives a strip, the processor merges the received strip with its local strip. At the end of the global combine phase, a processor has a complete strip from the final image. Each processor then sends its local strip to the client for display. The ADR steps in the rendering process are shown in Figure 3.

3.1.2 Experimental Results

In this section we present experimental performance results for the out-of-core iso-surface renderer on a PC cluster running Linux. The PC cluster consists of 1 frontend node and five processing nodes with 800GB of disk storage. Each processing node has a Pentium III 800MHz CPU, 128MB main memory, and two 5400RPM Maxtor 80GB EIDE disks. The processing nodes are interconnected via a 100Mbps switched Ethernet. The front-end node is also connected to the same switch.

Table 1 shows the characteristics of the datasets used in the experiments. The datasets from Dataset-1 to Dataset-7 were generated by a parallel environmental simulator called Parssim [2, 46], developed at the Texas Institute for Computational and Applied Mathematics (TICAM) at the University of Texas. Parssim simulates the reactive transport of chemical species in underground waterways and reservoirs. The datasets from Dataset-4 to Dataset-7 were generated from simulations of the same region with different physical parameters (e.g., permeability of the reservoir). Each dataset contains the simulation output for fluid flow and the transport of multiple chemical species over many time steps on a rectilinear mesh. The number of chemical species at a grid point for each dataset is shown in the table. In addition, three velocity values, each corresponding to one of the spatial dimensions, are simulated and stored at each grid point. The size of the data generated at a single simulation time step varies from 1.25MB to 10.62MB. Thus, the size of an entire dataset ranges from 3.9GB to 62GB. We also scaled the grid used for Dataset-1 at each time step by a factor of up to 256 (but only for five time steps) so that the largest scaled dataset contains 2.4GB data per time step. The scaled datasets represent application scenarios in which the data for a single time step does not fit into aggregate system memory. The values at grid points in the scaled datasets were interpolated from the values in the original grid in the y- and z-dimensions. In the experiments, the grid at each time step was partitioned into equal sub-volumes in three dimensions. The grid was first divided into slabs in the x-dimension, each slab was partitioned in the y-dimension, and each of those partitions was further divided in the z-dimension. Each sub-volume corresponds to an ADR data chunk. The data chunks were declustered across all the disks in each back-end configuration. The data chunks in Dataset-1 to Dataset-7 were distributed across the disks in round-robin order, in the order they were created during the partitioning of the grid. The data chunks in the scaled datasets were assigned to disks using a Hilbert curve-based declustering algorithm [18]. The mid-point of the bounding box of each sub-volume was used to generate a Hilbert curve index. The sub-volumes were sorted with respect to this index and distributed to disks in round-robin order. Table 1 also shows the characteristics of the scaled datasets. In the table, xS denotes the

Table 1: Characteristics of the datasets used in the experiments.

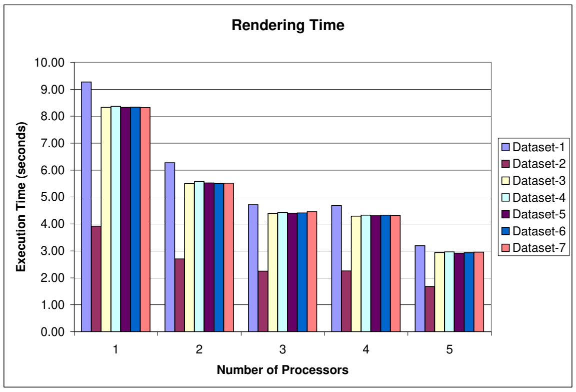
Dataset	Grid dimensions			number of	per time step		total	
name	X	Y	Z	chemicals	# chunks	size (MB)	# time steps	size (GB)
Dataset-1	302	32	32	4	150	10.62	5983	62.08
Dataset-2	32	32	32	4	125	1.25	5600	6.81
Dataset-3	32	42	32	28	150	7.20	495	3.48
Dataset-4	32	42	32	28	150	7.20	523	3.68
Dataset-5	32	42	32	28	150	7.20	741	5.21
Dataset-6	32	42	32	28	150	7.20	735	5.17
Dataset-7	32	42	32	28	150	7.20	554	3.90
							Total:	90.33
Scaled Datasets								
Orig	302	32	32	4	150	10.62	5	0.05
x4	302	63	63	4	384	40.17	5	0.20
x16	302	125	125	4	1536	160.68	5	0.78
x64	302	249	249	4	6144	642.71	5	3.14
x256	302	497	497	4	23814	2561.65	5	12.51
							Total:	16.73

dataset scaled by a factor of S, and $Orig$ denotes the original dataset. In all experiments, the output is a 512x512 RGB image.

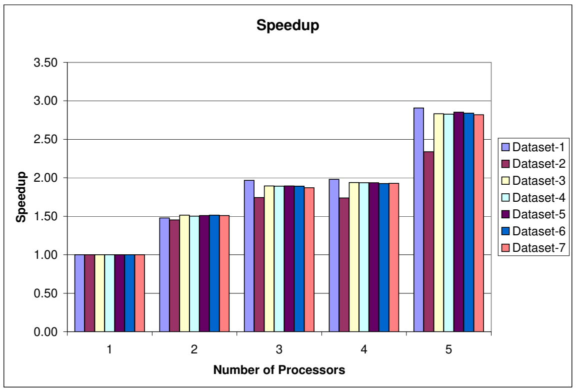
The performance results are shown in Figures 5–7. The timing values in the figures are averages from multiple rendering queries and represent the rendering time at the back-end.

Figures 5(a)–(d) show the total execution time, speedup values, and the breakdown of execution time for the four ADR query processing phases (see Section 2.3), for rendering datasets from a fixed viewing direction over 20 time steps. Execution time decreases as the number of processors is increased, obtaining a speedup of 2.9 on five processors for the largest dataset. This is mainly because of the load imbalance in performing the computation, and the communication overhead incurred in the global combine phase. Although each processor is assigned an equal number of chunks, the amount of processing required for a chunk depends on how many voxels in the chunk intersect with the iso-surface and the number of polygons generated from the extracted iso-surface. The breakdown of query execution time into the four ADR query processing phases is displayed in Figures 5(c) and (d). The initialization and output handling phases take negligible time compared to the reduction and global combine phases. As is expected, the percentage of time spent in the global combine phase increases as the number of processors is increased. In the global combine phase, the number of inter-processor communications per processor is $O(P-1)$, and the volume of communication is $O(\frac{(P-1)I^2}{P})$, where I^2 is the resolution of the screen. Therefore, the communication overhead increases, whereas the per processor computation time decreases as the number of processors is increased. The communication overhead can be decreased by communicating only the "active" pixels that are mapped to by a point on the iso-surface [23]. Figure 6 displays the execution time and speedup values for rendering a single time step from different viewing directions as the number of processors varies. A single time step in each dataset was rendered from six different viewing directions, and each query was repeated five times. We obtained a speedup of 3.4 on five processors.

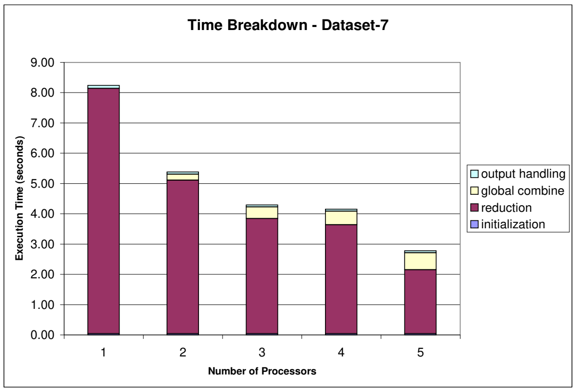
Figures 7(a)–(f) show the performance results for the scaled datasets. Log-scale is used for the y-axis in Figures 7(a), (d) and (f) so that timing results for small datasets can be displayed. As is seen in Figure 7(a), it takes approximately 1600 seconds to render one time step for the largest dataset (denoted by x256 in the figures) on one processor, whereas the rendering time decreases to 400 seconds on 5 processors. The



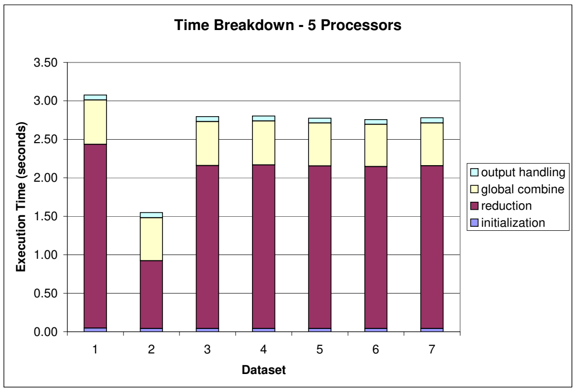
(a)



(b)

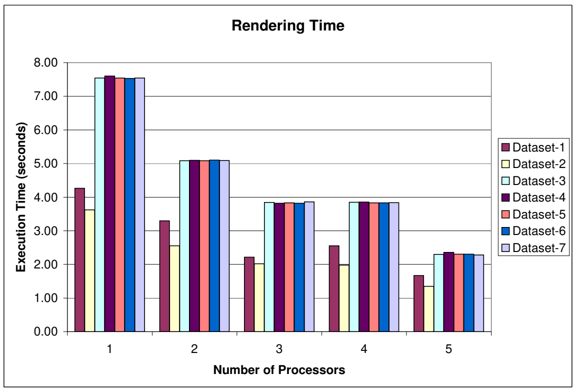


(c)

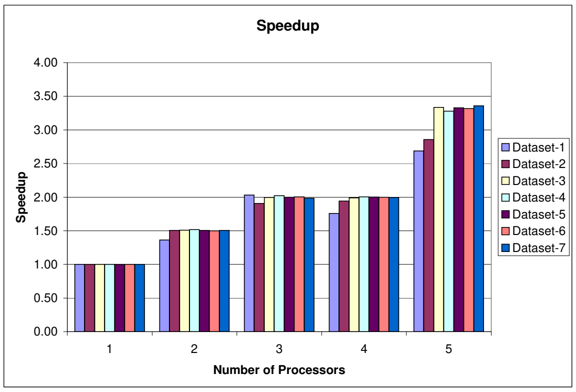


(d)

Figure 5: Performance results for rendering the datasets from a fixed viewing direction over 20 times steps. (a) Total execution time. (b) Speedup values. (c) Breakdown of the execution time for Dataset-7. (d) Breakdown of the execution time on five processors over all datasets.

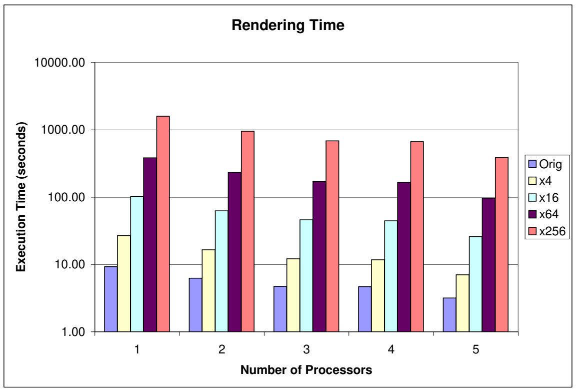


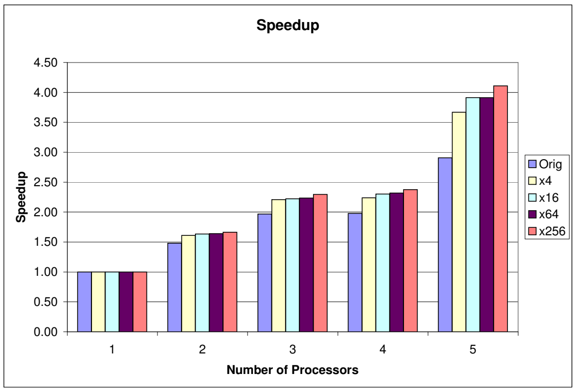
(a)



(b)

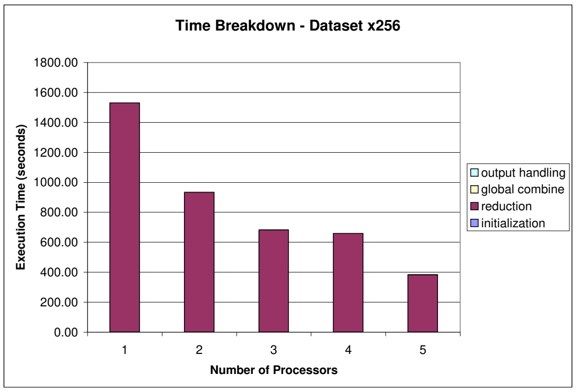
Figure 6: Total execution time and speedup for rendering a single time step from 6 different viewpoints. (a) Total execution time. (b) Speedup values.

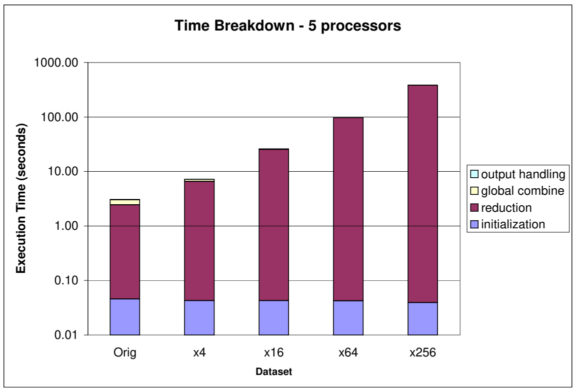




(a)

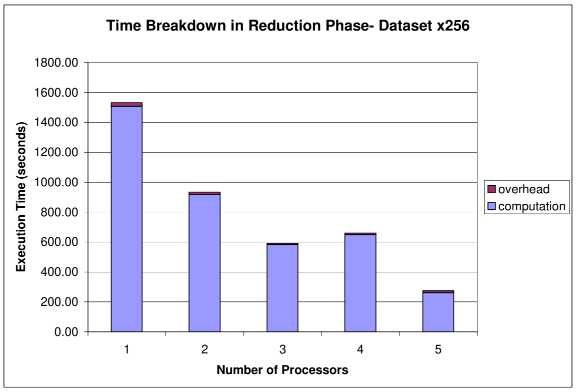
(b)

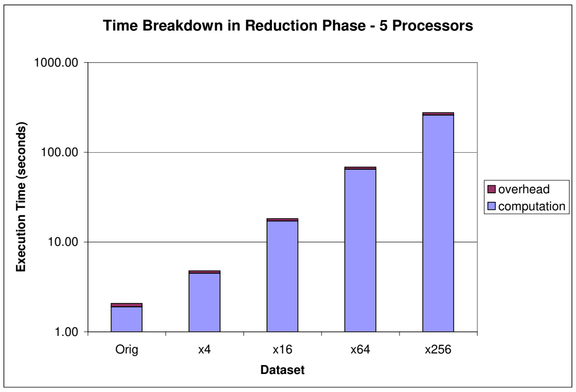




(c)

(d)





(e)

(f)

Figure 7: Performance results for the scaled datasets. The times are average from rendering the datasets over five time steps. (a) Total execution time. (b) Speedup values. (c) Breakdown of the execution times into processing phases for Dataset-x256. (d) Execution times for processing phases on five processors (e) Breakdown of execution time in the reduction phase for Dataset-x256. (f) Breakdown of execution time in the reduction phase on five processors. Log-scale is used for the y-axis in figures (a), (d), and (f).

speedup values are shown in Figure 7(b). As is seen in the figure, we obtain a speedup of 4 on 5 processors. The speedup for the larger scaled datasets is better than that for the datasets with smaller grids. Computation time increases as the grid size increases, whereas the communication overhead in the global combine phase remains constant, since the volume of communication depends on the size of the output image. As is seen in Figures 7(c) and (d), almost all of the processing time in the back-end is spent in the reduction phase. The percentage of time spent in the other three phases (i.e., initialization, global combine and output handling) is negligible compared to that of the reduction phase. The breakdown of execution time in the reduction phase is displayed in Figures 7(e) and (f). In these figures, *computations* corresponds to the execution time of the VTK functions for extracting and rendering the iso-surface from in-core data, after a data chunk is retrieved from disk and is available in memory. The remaining time in the reduction phase is denoted by *overhead*. As was discussed in Section 2.3, data chunks are retrieved from disks via non-blocking I/O functions to overlap I/O with computation, and the ADR runtime system manages buffer space and performs scheduling of I/O, network, and computation operations. Thus, in Figures 7(e) and (f), *overhead* includes non-overlapped I/O and other overheads incurred by the runtime system. The overhead is very small compared to the computation time; it is approximately 2% of the computation time, on average. Our results show that most of the I/O is overlapped with computation, and very little overhead is incurred by the runtime system. Therefore, good performance can be achieved by customizing ADR using functions that operate on in-core data for out-of-core rendering of large datasets.

3.2 Out-of-core Ray-casting Volume Rendering of Rectilinear Meshes

In this section, we describe a ray-casting based volume rendering algorithm for visualizing out-of-core datasets on rectilinear meshes. In ray-casting, a ray is cast from each pixel on the image plane and traversed in the volume. Color and opacity values are computed at sample points along the ray in each volume element the ray intersects. All the color and opacity values computed along a ray are composited to generate the color of the pixel from which the ray is cast. Color and opacity values along the ray must be composited in either front-to-back or back-to-front order. The composition operation is associative, but not commutative. That is, a partial image can be generated from a subset of consecutive sample points on the ray. Partial images generated from different subsets of consecutive sample points can be composited to form the final image. However, the composition of the partial images must be done in front-to-back or back-to-front order to ensure correctness.

3.2.1 Out-of-core Algorithm

In the following discussion, without loss of generality, we assume that a volume dataset with N^3 points is composed of N $N \times N$ slices (or 2-dimensional grids). In the out-of-core algorithm, each $N \times N$ slice is partitioned into P strips, where P is the number of processors. Strip k in every slice is assigned to processor k in the machine. A strip corresponds to a data chunk in ADR. In a processor, strips are scattered across the local disks attached to the processor so that the strip in slice i is stored on local disk i **mod** D, where D is the number of local disks. Figure 8 shows the partitioning of a volume across four processors, and the distribution of strips in processor 0 across two local disks (D0 and D1), attached to the processor.

The index for a volume data at a single time step is an array of size P. Array entry k corresponds to processor k in the machine, and stores the bounding box and size of the strip k. Since all the strips assigned to a processor are the same size, the size of a single strip can be used to calculate the file offset for any of the strips stored in the processor. As in the out-of-core iso-surface rendering, an array, each entry of which corresponds to a time step, can be used to access the index for a single time step. The number of slices, the resolution of each slice, and the number of disks per processor are also stored as meta-data information

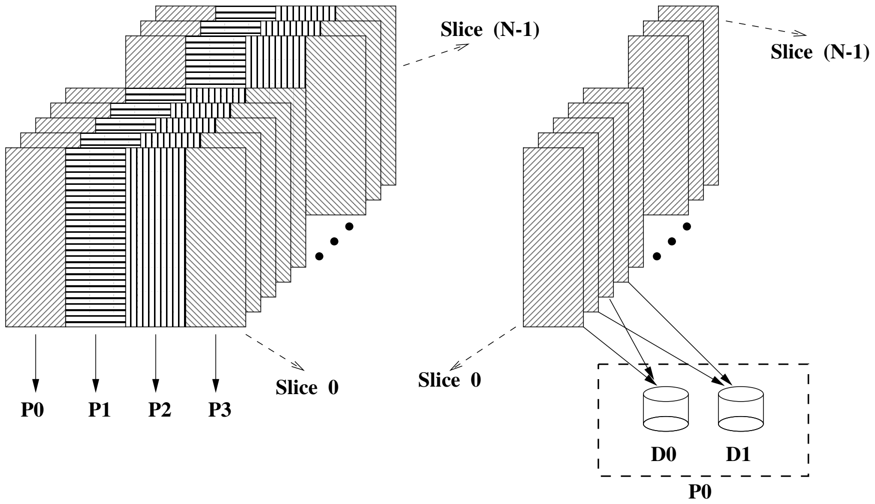
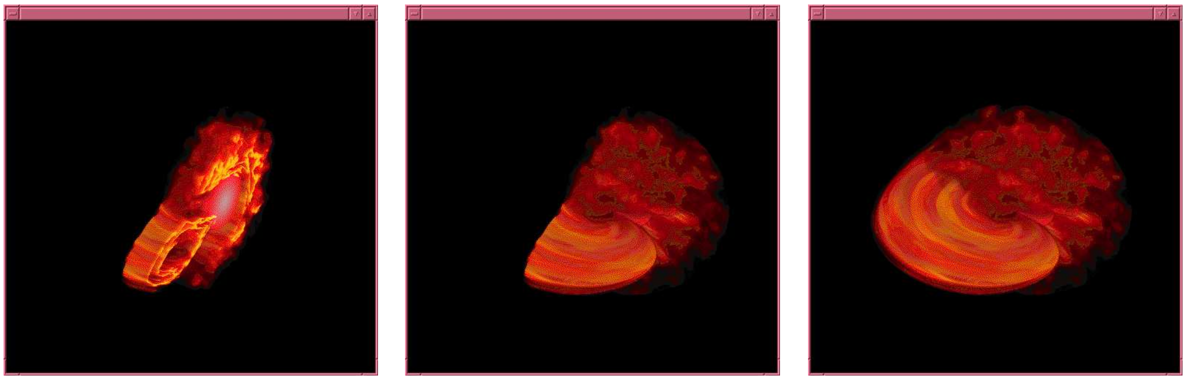


Figure 8: Partitioning of a volume mesh among four processors and declustering of strips across the two disks attached to processor 0.

in the index file. The indexing service of ADR is customized to perform index lookup using this index structure.

In the customized ADR instance, a two-dimensional array is used as the accumulator. Each array element corresponds to a pixel in the output image, and stores opacity and color values computed for the ray cast from the corresponding pixel on the image plane. The volume rendering operations (as a customization of ADR for application-specific processing) are implemented using the volume rendering functions of MPIRE [22], which is a parallel, direct volume rendering system, developed at the San Diego Supercomputer Center. MPIRE has been successfully used in the visualization of large datasets on large scale machines. However, MPIRE requires the entire volume dataset to fit in the overall system memory, which makes it impossible to render large datasets on a small machine. By using MPIRE functions that operate on *in-core* data structures, the ADR customization provides an extension of MPIRE for visualizing out-of-core datasets. The MPIRE ray-casting function, which generates a partial image from an in-core volume data, is used as the mapping and aggregation function executed in the ADR local reduction phase (see Section 2.3). MPIRE also provides a function to composite two partial images. This function is used for merging accumulator elements in the global combine phase.

Note that ray-casting requires color and opacity values for a pixel to be composited in a pre-defined order to produce a correct image. However, as was discussed in Section 2, an aggregation function in ADR is assumed to be commutative and associative to achieve high performance. This means that partial color and opacity values generated from a data chunk, or the data chunk itself, may need to be stored in memory to ensure correct composition order. This could greatly increase the memory requirements during processing, perhaps exceeding the available memory space on a processor for large datasets. Memory overflow could lead to performance problems, because of the limitations of virtual memory systems. In order to avoid memory overflow, a query for rendering the volume is partitioned into smaller queries, each of which encompasses a sub-volume, called a *slab*. A slab is a set of consecutive slices in the entire volume so that the size of the local portion of the slab in a processor is less than the available processor memory. Instead of one big query that encompasses the entire volume, the smaller queries are submitted to ADR one by one. Thus, starting from the foremost slab with respect to the viewing direction, rendering of the entire volume progresses a slab at a time in the back-end, and some of the work for compositing color and opacity values is carried out by the client.



(a) (b) (c)

Figure 9: (a) Partial image rendered in ADR (b) Partial image composited in the client (c) The final rendered image.

In the ADR local rendering phase, each back-end processor renders its local portion of the slab. ADR allows a user-defined grouping of data chunks into one or more clusters during index lookup. When a set of data chunks are grouped into a cluster, the runtime system retrieves all the chunks in the cluster (using asynchronous I/O operations) before calling the user-defined mapping and aggregation functions. Clusters can be retrieved in any order. For ray-casting, each processor collects all the strips in a slab into one cluster. As a result, only one partial image is created in each processor at the end of the local reduction phase. In the ADR global combine phase, the partial images are merged across processors via inter-processor communication. At the end of the global combine phase, each processor has a strip of the entire image, which corresponds to a rendering of the current volume slab. This partial image is sent to the client in the ADR output handling phase. The client is responsible for compositing partial images received from the ADR data server into the final image (see Figure 9). In this way, the final image is incrementally generated at the client, and the composition of partial image from slab k with the partial image in the client can be overlapped with rendering of slab $k + 1$ in the back-end. Also, the user can change the visualization parameters (e.g., viewing angle, transformation function) without waiting until the full image is produced. We should note that since slabs are processed in order of increasing distance from the image plane, partial images are also generated in that order. Therefore, the client only needs to allocate space for the current image and the image retrieved from the server, and can compose the image received from the server with the current image as soon as it is available at the client.

3.2.2 Experimental Results

In this section we present experimental performance results on the same PC cluster described in Section 3.1.2. The dataset used in the experiments has 512 512×512 slices and is 512MB in size. It contains data from a single time step of a simulation of a star with gases swirling around it (see Figure 9). The timing values are averages from rendering the data from 5 different view points into a 512×512 image. The ADR back-end was run on the processing nodes, while the ADR frontend and the client program (a C++ code with a Java-based display) were run on the frontend node.

Figure 10 shows the change in execution time when the number of slices per slab is varied. The execution times shown in the figure are the total and per slab response times measured in the client, which include the time to submit the query and to receive an image from the back-end. The back-end program was run on 5 processing nodes in this experiment. As is seen in the figure, the total rendering time decreases as the number of slices per slab is increased, as is expected. There is a constant overhead for rendering a slab. This overhead includes submitting the query for a slab, initializing and compositing partial images in the client and in the back-end and sending the image from the back-end to the client. As the number of slices per slab increases, the number of slabs to render the image decreases, and so does the total overhead for rendering the entire volume. Figure 10 shows that the algorithm can take advantage of more memory in the back-end nodes by increasing the number of slices per slab. However, as is also seen in the figure, the rendering time per slab increases as the number of slices per slab is increased. As a result, the client receives an image from the back-end less frequently.

We also examined the performance impact of using virtual memory on a processor. Using one processing node to run the back-end, we set the number of slices in a slab so that the size of the slab exceeds the main memory. In that case, it took more than *2 hours* to render a single slab. This is because when a ray is cast from a pixel, the ray is traversed until it leaves the volume (or the slab). Traversal of the ray accesses each slice the ray intersects with, starting from the slice closest to the view plane. Virtual memory pages for the current slice must be brought into memory from disk, possibly replacing the pages for the previous slices. When a ray is cast from the next pixel, it is likely that the pages for the first slice has been swapped out to disk. This causes disk trashing, and decreases performance substantially.

Figure 11 shows the total execution time and speedup when the number of processors is varied. In the experiments, the number of slices per slab was fixed at 60. As is seen in the figure, the execution time decreases almost linearly as the number of processors increases. We obtained a speedup of 3.9 on 5 processors.

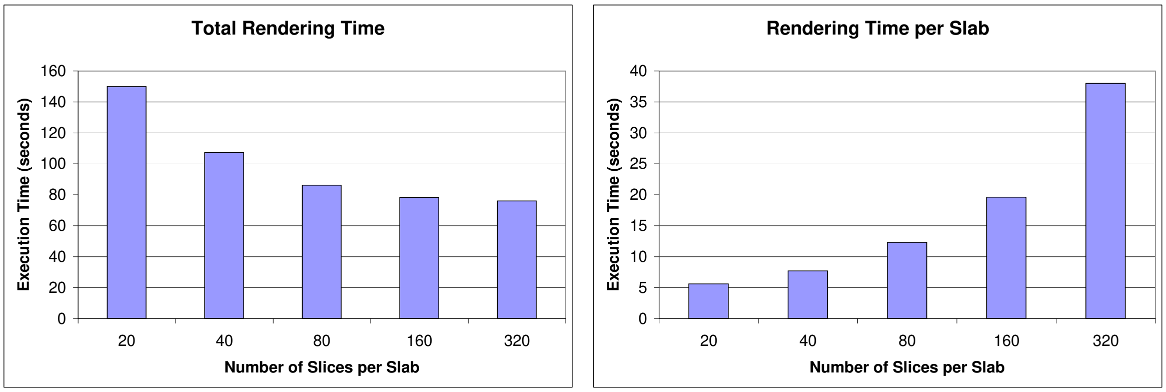
Figure 12 (a) shows the breakdown of the execution time per slab in the back-end into query execution phases (see Section 2.3). Most of the time is spent in the local reduction phase. However, the percentage of time spent in the global combine phase increases as the number of processors is increased, as is expected. Figure 12 (b) shows a breakdown of the execution time in the reduction phase into computation time and overhead. Overhead includes the I/O time and the time spent polling message and I/O queues, issuing I/O operations, and managing space for various buffers in ADR. In the experiments, the slab size was fixed at 60 slices. Most of the time is spent in the computation, and the relative overhead costs decrease as the number of processors increases. This is mainly because each processor reads less data as the number of processors increase. In the 1-processor case, the processor reads 62MB to render a slab, while a processor reads about 12MB in the 5-processor case. Also, the operating system maintains a file cache in memory on each processor. The timings shown in the figures are the average of the rendering times from 5 different views. As the total amount of data retrieved for a processor decreases, it is more likely that the data needed for rendering from the next viewing direction will be accessed from the file cache.

4 Other Applications

The ADR framework is designed to provide support for a wide range of applications. We have used ADR to develop applications in diverse fields, including coupling of simulation codes [25], analysis and processing of satellite datasets [12, 38], volume shape analysis from multi-perspective video sequences [7], and analysis and visualization of microscopy data [1]. In this section, we briefly describe some of these applications.

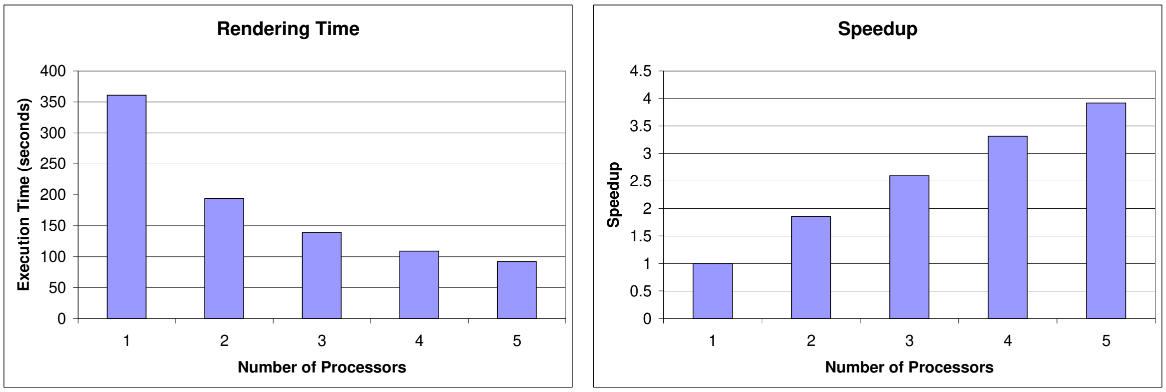
4.1 Satellite Data Processing

Earth scientists study the earth by processing remotely-sensed data continuously acquired from satellite-based sensors, since a significant amount of earth science research is devoted to developing correlations between sensor radiometry and various properties of the surface of the earth. A typical analysis [12] processes satellite data for ten days to a year (for the comparatively low resolution AVHRR sensor, ten days of data is about 4 GB) and generates one or more composite images of the area under study. Generating a



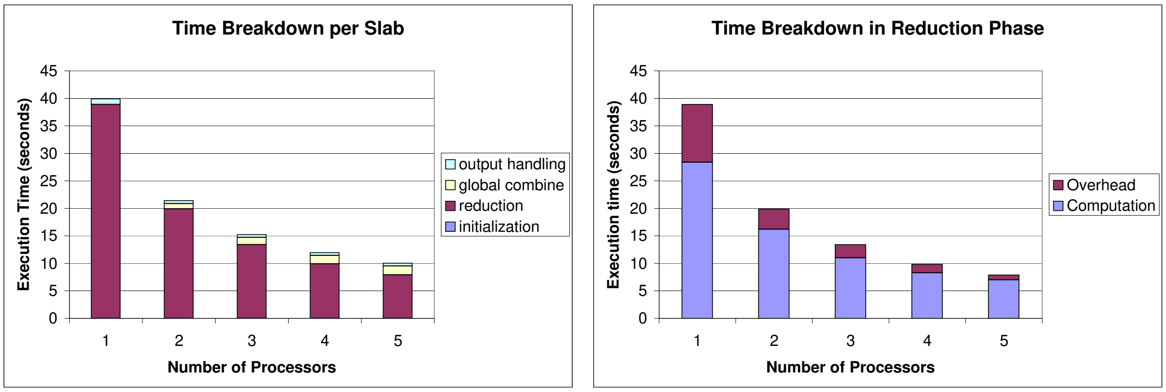
(a) (b)

Figure 10: Execution times when the number of slices in a slab is varied on 5 processors. (a) Total execution time (b) Execution time per slab.



(a) (b)

Figure 11: Total execution time and speedup. The number of slices per slab is 60. (a) Total execution time. (b) Speedup values.



(a) (b)

Figure 12: Breakdown of the execution time per slab in the back-end. (a) Execution times per processing phase. (b) Execution time in reduction phase. The number of slices per slab is 60.

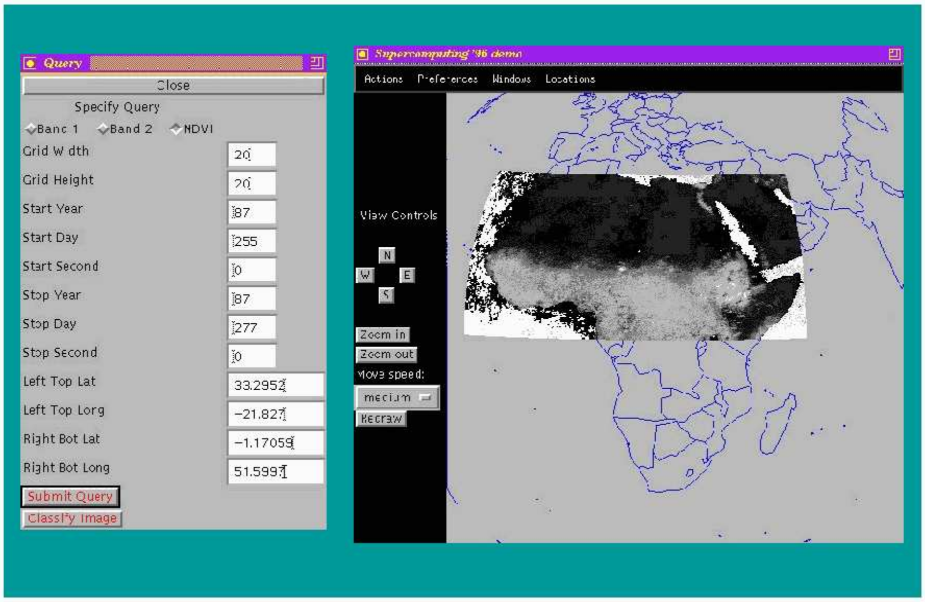


Figure 13: Client interface for visualizing AVHRR datasets.

composite image requires projection of the globe onto a two-dimensional grid; each pixel in the composite image is computed by selecting the "best" sensor value that maps to the associated grid point. A variety of projections are used by earth scientists – the USGS cartographic transformation package supports 24 different projections [43]. An earth scientist specifies the projection that best suits her needs, maps the sensor data using the chosen projection, and generates an image by compositing the projected data (see Figure 13). Sensor values are pre-processed to correct the effects of various distortions, such as instrument drift, atmospheric distortion and topographic effects, before they are used.

4.2 Analysis of Microscopy Data: the Virtual Microscope

The Virtual Microscope [1, 20] is an application we have developed to support the need to interactively view and process digitized data arising from tissue specimens. The Virtual Microscope provides a realistic digital emulation of a high power light microscope. The raw data for such a system can be captured by digitally scanning collections of full microscope slides under high power. An array of approximately 50x50 photomicrographs, each about 1000x1000 pixels, is required to cover an entire slide at high resolution, and each pixel is a three byte RGB color value. Under this scenario, one slide image requires over 7 GB, uncompressed. However, such an image captures only a single focal plane, and many specimens will require capture of from five to thirty focal planes. The digitized images from a slide are effectively a three-dimensional dataset, since each slide can contain multiple focal planes. At the basic level, the Virtual Microscope can emulate the usual behavior of a physical microscope, including continuously moving the stage and changing magnification and focus. The processing for the Virtual Microscope requires projecting high resolution data onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing pixels mapping onto a single grid point, to avoid introducing spurious artifacts into the displayed image. Used in this manner, the Virtual Microscope can support completely digital dynamic telepathology [45]. In

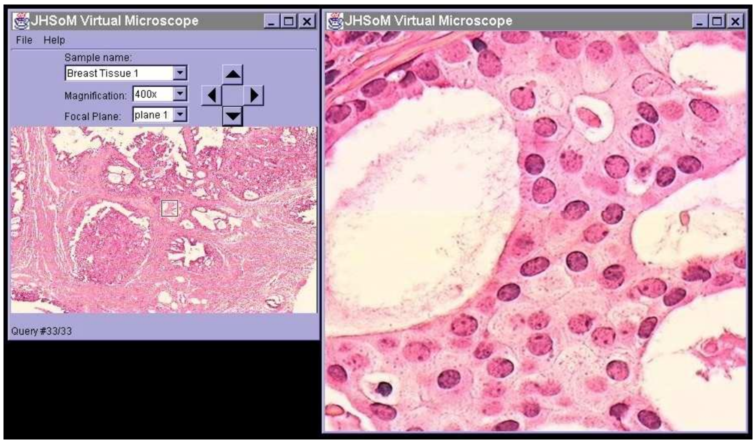


Figure 14: The Virtual Microscope client.

addition, it enables new modes of behavior that cannot be achieved with a physical microscope, such as simultaneous viewing and manipulation of a single slide by multiple users.

The Virtual Microscope application is presently deployed at Department of Pathology at the Johns Hopkins Medical Institutions and is used by MDs to view digitized slides from tissue specimens. The Java client interface is shown in Figure 14.

4.3 Coupling of Simulation Codes for Water Contamination Studies

Powerful simulation tools are crucial to understand and predict transport and reaction of chemicals in bays and estuaries. Such tools include a hydrodynamics simulator [29], which simulates the flow of water in the domain of interest, and a chemical transport simulator [8], which simulates the reactions between chemicals in the bay and transport of these chemicals. For each simulated time step, each simulator generates a grid of data points to represent the current status of the simulated region. For a complete simulation system for bays and estuaries, the hydrodynamics simulator needs to be coupled to the chemical transport simulator, since the latter uses the output of the former to simulate the transport of chemicals within the domain. As the chemical reactions have little effect on the circulation patterns, the fluid velocity data can be generated once and used for many contamination studies. While the grid data for a single time step may only require several megabytes, thousands of time steps may need to be simulated for a particular scenario, leading to very large database storage requirements. The grids used by the chemical simulator are often different from the grids the hydrodynamic simulator employs, and the chemical simulator usually uses coarser time steps than the hydrodynamics simulator. Therefore, running a chemical transport simulation requires retrieving the hydrodynamics output from the appropriate hydrodynamics datasets stored in the database, averaging the hydrodynamics outputs over time, and projecting them into the grid used by the chemical transport simulator, via a computationally complex projection method [15]. We have implemented a prototype parallel database system using ADR to provide optimized storage, retrieval and post-processing (averaging) of the datasets generated by hydrodynamics simulations [25].

5 Related Work

The memory and computing requirements of visualization algorithms have always been a challenge in the field of computer graphics. To cope with large memory and computation requirements for visualizing large datasets, parallel algorithms have been developed on high-performance machines [9, 30, 31, 47, 26]. Those research projects have focused on visualization of in-core datasets.

A number of research projects have focused on algorithms and methods for the exploration and visualization of large, out-of-core datasets [4, 13, 14, 41, 16]. Cox and Ellsworth [16] show that relying on operating system virtual memory results in poor performance. They propose a paged system and algorithms for memory management and paging for out-of-core visualization. Their results show that *application controlled paging* can substantially improve application performance. Chiang and Silva [13] and Chiang et. al [14] propose methods for iso-surface extraction for datasets that cannot fit in memory. They introduce several techniques and indexing structures to efficiently search for cells through which the iso-surface passes, and to reduce I/O costs and disk space requirements. Ueng et. al [41] present algorithms for streamline visualization of large unstructured tetrahedral meshes. They employ an octree to partition the out-of-core dataset into smaller sets, and describe techniques for scheduling operations and managing memory optimized for streamline visualization. Remote visualization of scientific datasets [35, 48], where the server is a visualization program, possibly running on a high-performance machine, and visualization of terrain databases [27, 36] have also been investigated. Arge et. al. [3] present efficient external memory algorithms for applications that make use of grid-based terrains in Geographic Information Systems.

While large scale data visualization systems are in many cases highly optimized, such software is not designed to carry out general patterns of spatial queries and computation. ADR, on the other hand, is a more general framework designed to support mapping and aggregation operations on a wider range of applications.

6 Conclusions

We have presented implementations of ray casting and iso-surface rendering using the Active Data Repository (ADR) for visualizing out-of-core datasets on a cluster of PCs. ADR is an object-oriented framework for developing applications that make use of large scientific datasets on distributed memory machines. The ADR framework makes it possible to implement processing of out-of-core datasets using functions that operate on *in-core* data. The underlying infrastructure provides support for data retrieval and scheduling of operations. Our results show that although ADR is a general framework, good performance can be achieved for diverse applications.

The Active Data Repository is an on-going project. The version of ADR used in this paper employs a query processing strategy called *fully replicated accumulator*. We are also investigating other potential strategies. In our earlier work [11, 24], we examined the performance impact of two other strategies, referred to as *sparsely replicated accumulator* (SRA) and *distributed accumulator* (DA), under various application scenarios. The SRA strategy replicates the accumulator elements only on processors with local input elements that map to those accumulator elements. The DA strategy, on the other hand, assigns the processing for an accumulator element to a single processor. The accumulator elements are partitioned across the processors and each processor carries out aggregation operations on the local accumulator elements. Input elements are communicated to processors that own the corresponding accumulator elements. The replicated accumulator and distributed accumulator strategies are similar to the *sort-last* and *sort-first* parallel rendering approaches. Experimental results show that the relative performance of the strategies depends on the characteristics of the applications (e.g., the distribution of data elements in the underlying multi-dimensional space) and of the available resources of the underlying parallel machine.

Another closely related framework under development is DataCutter [5, 6]. DataCutter provides support for subsetting and processing of datasets across a wide-area network. The programming model in DataCutter, called filter-stream programming, represents components of a data-intensive application as a set of filters. Each filter can potentially be executed on a different host across a wide-area network. Data exchange between any two filters is described via streams, which are uni-directional pipes that deliver data in fixed size buffers. DataCutter services provide support for the execution of filters in a wide-area environment to carry out resource-constrained, pipelined communication and processing. ADR and DataCutter can be used jointly in applications. For example, DataCutter can be used to provide support for an ADR data server to access subsets of datasets across a wide-area network, or can be used by remote clients to further process ADR-generated data products.

We plan to examine the performance impact of the SRA and DA query execution strategies on the out-of-core visualization methods, and implement iso-surface rendering using DataCutter for remote visualization across a wide-area network.

Acknowledgements

We would like to thank Mary F. Wheeler, Steven Bryant and Shuyu Sun for their help in generating the simulation datasets with Parssim, and Jon Genetti and Bernard Pailthorpe for providing the MPIRE codes and the Astronomy dataset for ray casting experiments.

References

[1] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, Nov. 1998.

[2] T. Arbogast, S. Bryant, C. Dawson, and M. F. Wheeler. Parssim: The parallel subsurface simulator, single phase. *http://www.ticam.utexas.edu/∼arbogast/parssim*.

[3] L. Arge, L. Toma, and J. S. Vitter. I/o-efficient algorithms for problems on grid-based terrains. In *Proceedings of 2nd Workshop on Algorithm Engineering and Experimentation (ALENEX '00)*, 2000.

[4] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics*, pages 97–104, San Francisco, CA, USA, Oct 1999.

[5] M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, Mar. 2000. NASA/CP 2000-209888.

[7] E. Borovikov, A. Sussman, and L. Davis. An efficient system for multi-perspective imaging and volumetric shape analysis. In *Proceedings of the Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM'20001)*. IEEE Computer Society Press, Apr. 2001. To appear.

[8] C. F. Cerco and T. Cole. User's guide to the CE-QUAL-ICM three-dimensional eutrophication model, release version 1.0. Technical Report EL-95-15, US Army Corps of Engineers Water Experiment Station, Vicksburg, MS, 1995.

[9] J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *Proceedings of IEEE/ACM 1993 Parallel Rendering Symposium*, 1993.

[10] C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP Symposiums*. IEEE Computer Society Press, Apr. 1999.

[11] C. Chang, T. Kurc, A. Sussman, and J. Saltz. Optimizing retrieval and processing of multi-dimensional scientific datasets. In *Proceedings of the Third Merged IPPS/SPDP (14th International Parallel Processing Symposium & 11th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, May 2000. Also available as University of Maryland Technical Report CS-TR-4101 and UMIACS-TR-2000-03.

[12] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, Apr. 1997.

[13] Y.-J. Chiang and C. Silva. External memory techniques for isosurface extraction in scientific visualization. In J. Abello and J. Vitter, editors, *External Memory Algorithms and Visualization*, volume 50, pages 247–277. DIMACS Book Series, American Mathematical Society, 1999.

[14] Y.-J. Chiang, C. Silva, and W. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of IEEE Visualization '98*, pages 167–174, 1998.

[15] S. Chippada, C. N. Dawson, M. L. Martínez, and M. F. Wheeler. A projection method for constructing a mass conservative velocity field. *Computer Methods in Applied Mechanics and Engineering*, 1997. Also a TICAM Report 97-09, University of Texas, Austin.

[16] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th IEEE Visualization'97 Conference*, 1997.

[17] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the Eleventh International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1997.

[18] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, Jan. 1993.

[19] R. Ferreira, T. Kurc, M. Beynon, C. Chang, A. Sussman, and J. Saltz. Object-relational queries into multi-dimensional databases with the active data repository. *Parallel Processing Letters*, 9(2):173–195, 1999.

[20] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., Oct. 1997.

[21] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD84)*, pages 47–57, Boston, MA, June 1984.

[22] G. Johnson and J. Genetti. MPIRE: Massively Parallel Interactive Rendering Environment. *http://mpire.sdsc.edu*.

[23] T. Kurc, C. Aykanat, and B. Ozguc. Object-space parallel polygon rendering on hypercubes. *Computers & Graphics*, 22(4):487–503, 1998.

[24] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. In *Proceedings of the 1999 ACM/IEEE SC99 Conference*. ACM Press, Nov. 1999.

[25] T. M. Kurc, A. Sussman, and J. Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Mar. 1999.

[26] H. Kutluca, T. Kurc, and C. Aykanat. Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids. *The Journal of Supercomputing*, 15(1):51–93, 2000.

[27] Y. G. Leclerc and S. Q. Lau. TerraVision: A terrain visualization system. Technical Report AIC Technical Note 540, SRI International, 1994.

[28] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface reconstruction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[29] R. A. Luettich, J. J. Westerink, and N. W. Scheffner. *ADCIRC*: An advanced three-dimensional circulation model for shelves, coasts, and estuaries. Technical Report 1, Department of the Army, U.S. Army Corps of Engineers, Washington, D.C. 20314-1000, December 1991.

[30] K. L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of IEEE/ACM 1995 Parallel Rendering Symposium*, 1995.

[31] K. L. Ma and T. W. Crockett. A scalable cell-projection volume rendering algorithm for unstructured data. In *Proceedings of IEEE/ACM 1997 Parallel Rendering Symposium*, 1997.

[32] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, 14(4):23–32, July 1994.

[33] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.

[34] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. Available at http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/LAND_BIO/origins.html.

[35] J. Patten and K.-L. Ma. A graph based interface for representing volume visualization results. In *Proceedings Graphics Interface '98*, pages 117–124. Canadian Information Processing Society, June 1998.

[36] M. Reddy, Y. G. Leclerc, L. Iverson, and N. Bletter. TerraVision II: Visualizing massive terrain databases in VRML. *IEEE Computer Graphics and Applications (Special Issue on VRML)*, 2(19):30–38, 1999.

[37] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Prentice Hall, 2nd edition, 1997.

[38] C. T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, and A. Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, Jan. 1998.

[39] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, 1996.

[40] T. Tanaka. Configurations of the solar wind flow and magnetic field around the planets with no magnetic field: calculation by a new MHD. *Jounal of Geophysical Research*, 98(A10):17251–62, Oct 1993.

[41] S.-K. Ueng, K. Sikorski, and K.-L. Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, Dec. 1997.

[42] U.S. Geological Survey. Land satellite (LANDSAT) thematic mapper (TM). Available at http://edcwww.cr.usgs.gov/nsdi/html/landsat_tm/landsat_tm.

[43] The USGS General Cartographic Transformation Package, version 2.0.2. *ftp://mapping.usgs.gov/pub/software/current_software/gctp/*, 1997.

[44] A. Watt. *Fundamentals of three-dimensional computer graphics*. Addison Wesley, 1989.

[45] R. S. Weinstein, A. Bhattacharyya, A. R. Graham, and J. R. Davis. Telepathology: A ten-year progress report. *Human Pathology*, 28(1):1–7, Jan. 1997.

[46] M. F. Wheeler, T. Arbogast, S. Bryant, C. N. Dawson, F. Saaf, and C. Wang. New computational approaches for chemically reactive transport in porous media. In G. Delic and M. Wheeler, editors, *Next Generation Environmental Models and Computational Methods (NGEMCOM), Proceedings of the U.S. Environmental Protection Agency Workshop*, pages 217–226. SIAM, 1997.

[47] C. M. Wittenbrink. Survey of parallel volume rendering algorithms. In *Proceedings of PDPTA'98*, 1998.

[48] C. M. Wittenbrink, K. Kim, J. Story, A. Pang, K. Hollerbach, and N. Max. PermWeb: Remote parallel and distributed volume visualization. Technical Report HPL-97-34, HP Labs, 1997.