Title:

Large Scale Data Visualization Using Parallel Data Streaming

Author(s): James Ahrens, Kristi Brislawn - Los Alamos National Laboratory
Ken Martin, Berk Geveci, C. Charles Law - Kitware Inc.
Michael Papka - Argonne National Laboratory

Submitted to: IEEE Computer Graphics and Applications.

Large Scale Data Visualization Using Parallel Data Streaming

James Ahrens, Kristi Brislawn
Los Alamos National Laboratory

Ken Martin, Berk Geveci, C. Charles Law
Kitware Inc.

Michael Papka
Argonne National Laboratory

Abstract-- Effective large-scale data visualization remains a significant and important challenge with analysis codes already producing terabyte results on clusters with thousands of processors. Frequently the analysis codes produce distributed data and consume a significant portion of the available memory per node. This paper presents an architectural approach to handling these visualization problems based on mixed dataset topology parallel data streaming. This enables visualizations on a parallel cluster that would normally require more storage/memory than is available while at the same time achieving high code reuse. Results from a variety of hardware and visualization configurations are discussed with data sizes ranging near to a petabyte.

Index Terms-- Parallel Visualization, Large Scale Visualization, MPI, VTK

I. Introduction

Scientists are using computer simulations to resolve models of real world phenomenon, including models of the earth's environment, accelerator physics dynamics, and celestial bodies. With additional computing power and algorithmic advances, these models have been resolved to more detailed levels, increasing our understanding of the world around us. In engineering and product design, simulation continues to replace physical mockups resulting in reduced design cycle times and costs. Key to such applications is the visualization and analysis of simulation results. Simulations are usually run in parallel on clusters of high-bandwidth supercomputers or PCs. The resulting datasets can be so massive in size they require the use of parallel computing resources of similar magnitude in order to effectively visualize them.

While visualization of large datasets is not a new problem, it remains an important and difficult one. The traditional improvements in hardware capabilities continue to make larger datasets possible and more accessible. Improvements in networking software and hardware are promoting growth in networked computing clusters. It is expected that both the use of large dataset visualization and the size of the datasets will continue to grow. As both large and small scale parallel computing resources become commonplace for scientists, so must parallel visualization software that effectively utilizes these resources.

Effective visualization of large datasets is a difficult problem for a number of reasons. Current analysis codes are producing tera-element datasets distributed over thousands of processing nodes. In some cases many time steps are never stored to disk but must be visualized while in memory on the processing nodes. This creates a problem since the visualization must share the already limited resources that the simulation is using. This is compounded by the fact that the visualization could potentially require more storage than the simulation does. Another difficulty is that some traditional visualization algorithms, such as streamline generation or mesh decimation, are not well suited to operating on distributed data or in parallel. Furthermore, visualizations often result in processing mixed dataset topologies even when the simulation dataset is of a uniform topology. An iso-surface of a rectilinear grid is a common example of this issue.

Our solution to these problems was to develop a visualization architecture based on mixed dataset-topology parallel data streaming. Clearly any viable solution must support parallel execution and visualization. Mixed topology parallel data streaming goes beyond this to incorporate data streaming so that the required storage resources for the visualization can be kept significantly smaller than that of the simulation. Additionally it supports such streaming even when the topology of the dataset is changing from one visualization algorithm to the next.

The resulting architecture is implemented within the Visualization Toolkit (VTK) and includes specific additions to support MPI, memory limit based streaming of both implicit and explicit topologies, translation of streaming requests between topologies, and passing data and pipeline control between shared, distributed, and mixed memory configurations.[8] The architecture directly supports both sort first and sort last parallel rendering.

The following sections of this paper will discuss related work, how to stream datasets in a pipeline architecture, how to add support for mixed topologies, and how to handle parallel processing. This paper is not intended to address some known issues in large dataset visualization such as massively parallel IO, effective load balancing, or parallel rendering although their implications are briefly discussed and utilized.

II. Related Work

While data streaming, parallel visualization, and mixed topology visualization are all known techniques, they each can be difficult and combining all three a significant challenge.

A number of out-of-core algorithms have been developed that support efficient streaming of large data.[15,16] The idea behind these approaches is to employ out-of-core or incremental algorithms with a controllable memory footprint. These methods include isosurfaces (modified marching cubes from disk) and related computational geometry work, streamlines, separation and attachment lines, and vortex core lines.[10,13,14,17,18,19,20,21,22,23,24] Typically the algorithm will extract pertinent features (e.g., an isosurface) and incrementally write the output to disk. Feature extraction is then followed by an interactive visualization of the extracted feature. What these algorithms lack is an overall architecture. Typically they are designed to work independently from and to disk storage. Sometimes they can be applied serially but at the cost of constantly reading and writing the data to disk between each algorithm, a poor use of the memory hierarchy.

Systems such as OpenDX, AVS, and SCIRun do provide a pipeline infrastructure and can support parallel execution. OpenDX (formerly IBM Data Explorer) is a data-flow based visualization system providing numerous visualization and analysis algorithms for its users. The OpenDX software architecture relies on a centralized executive to instantiate, allocate memory and execute modules.[1] OpenDX supports threaded data-parallelism on shared-memory multiprocessors and distributed task parallelism. Both mechanisms were designed to support parallelism in the context of a centralized executive. For example, task parallelism is achieved by a remote module that informs the executive it is ready to execute and waits for a signal from the executive before continuing.

SCIRun is a data flow based simulation and visualization system that supports interactive computational steering. SCIRun provides threaded task and data parallelism on shared-memory multiprocessors.[6,9] An extension to SCIRun permits distributed memory task parallelism.[7] SCIRun also uses a centralized executive and in this way is similar to OpenDX.

AVS is another popular data-flow visualization system that provides a similar parallel architecture and support.[11] A prototype extension supported data parallelism on the CM-5.[12] All of these systems provide a tightly integrated programming environment that supports the interactive construction, execution and debugging of programs via a graphical user interface. The existence of a single point of control for program construction and execution (i.e. the GUI) may have lead to the creation of a related centralized executive. Designing an efficient mechanism for controlling a large number of processes from a single centralized executive is difficult.

In contrast to these systems, our approach avoids the use of a centralized executive and therefore provides a more scalable solution. Additionally our approach supports mixtures of task, data and pipeline parallelism on both distributed and shared memory multiprocessors.

Other solutions, such as pV3 and Ensight, encompass a variety of techniques and support large/parallel data but are designed more as turn-key applications.[4] pV3, is an implementation of the Visual3 visualization application in the PVM environment. The application operates on a network of heterogeneous computers that process data in pieces, ultimately sending output to a collector that gathers and displays the results. While successful, pv3 is not designed as a toolkit but more as a custom application. Furthermore, depending on a collector is problematic in the larger data environment. Similarly Ensight is easy to use but lacks the flexibility, and capabilities of our approach.

All of the approaches described above lack the ability to stream data in memory when the dataset topologies are changing. As many visualization techniques can change the topology of the data this is an important consideration. Even when using unstructured grids, which are very general, there are situations where using a structured image is more efficient and best represented as an image and not another unstructured grid.

III. Streaming Data

The ability to stream data through a visualization pipeline offers two main benefits. The first is that visualizations that would not normally fit into memory or swap can be run where otherwise they could not. The second is that visualizations can be run with a smaller memory footprint resulting in higher cache hits, and little or no swapping to disk. To accomplish this the visualization software must support breaking the dataset into pieces and correctly processing those pieces. This requires that the dataset and the algorithms that operate on it are separable, mappable, and result invariant.

1. *Separable.* The data must be separable. That is, the data can be broken into pieces. Ideally, each piece should be coherent in geometry, topology, and/or data structure. The separation of the data should be simple and efficient. In addition, the algorithms in this architecture must be able to correctly process pieces of data.
2. *Mappable.* In order to control the streaming of the data through a pipeline, we must be able to determine what portion of the input data is required to generate a given portion of the output. This allows us to control the size of the data through the pipeline, and configure the algorithms.
3. *Result Invariant*. The results should be independent of the number of pieces, and independent of the execution mode (i.e., single- or multi-threaded). This means proper handling of boundaries and developing algorithms that are multi-thread safe across pieces that may overlap on their boundaries.

Earlier results discuss an architecture for accomplishing this with regularly sampled volumetric data, such as images and volumes.[5] In that architecture consumers of data, such as rendering engines or file writers, make requests for data that are fulfilled using a three-step pipeline update mechanism.

The first step, *Update Information*, is used to determine the characteristics of the dataset. This request is made by the consumer of the data and travels upstream to the source of the data. The resulting information contains the native data type (such as float or short), the largest possible extent expressed as (i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max}), the number of scalar values at each point, and the pipeline-modification time. The native data type and number of scalar values at each point are used in computing how much memory a given piece of data would require. The largest possible extent is typically the size of the dataset on disk. This is useful in determining how to break the dataset into pieces and where the hard boundaries are (versus the boundaries of a piece). The pipeline-modification time is used to determine when cached results can be used.

Many algorithms in a visualization pipeline must modify the information during the *Update Information* pass. For example a 2X image magnification algorithm would produce a largest possible extent that is twice as large as its input. A gradient algorithm would produce three components of output for every input component.

The second step, *Propagate Update Extent*, is used to propagate a request for data (the update extent) up the pipeline (to the data source). As the request propagates upstream, each algorithm must determine how to modify the request. Specifically, what input extent is required for the algorithm to generate its requested update extent. For many algorithms this is a simple one to one mapping but for others, such as a 2X magnification or gradient computation using central differences, the required input extent is different from the requested extent. This is the requirement that the algorithms be mappable. A side effect of the *Propagate Update Extent* pass is that it returns the total memory required to generate the requested extent. This enables streaming based on a memory limit. A simple streaming algorithm is to propagate a large update extent and if that requires exceeds the user specified memory limit, then to break the update extent into smaller pieces until it does fit. This requires that the dataset be separable. More flexible streaming algorithms can switch between dividing a dataset by blocks or slabs and by what axis.

The final step, *Update Data*, causes the visualization pipeline to actually process the data and produce the update extent that was requested in step two. These three steps require a significant amount of code to implement but surprisingly their CPU overhead is negligible. Typically the performance speedup provided by better cache locality more than compensates for the additional overhead. The exception is when boundaries cells are recomputed multiple times because they are shared between multiple pieces. This is typical in neighborhood-based algorithms and creates a tradeoff between piece size (memory consumption) and recomputing shared cells (computation).

This entire three-step process is initiated by the consumer of the data such as a writer that writes to disk or a mapper that converts the data into OpenGL calls. In both these cases the streaming is effective because the entire result is never stored in memory at one time. It is either written to disk in pieces or sent to the rendering hardware in pieces. It is also possible to stream in the middle of a visualization pipeline if there is an operation that requires a significant amount of input but produces a fairly small output.

The use of streaming within VTK is simple. Consider the following C-code example. An instance of an analytical volumetric source is created called source1. It is then connected to a contour filter that is then connected to a mapper. A 50,000 Kilobyte memory limit is set on the mapper which will initiate streaming if the memory consumption exceeds that limit. Finally an actor, renderer, and render window, are created and the resulting image is rendered. The only change made to this program to support streaming is the call to SetMemoryLimit on the mapper.

```
int main( int argc, char* argv[] )
{
  // The pipeline
  //  source
  vtkMySource* source1 = vtkMySource::New();
  source1->SetStandardDeviation( 0.5 );

  // Iso-surfacing
  vtkContourFilter* ctf = vtkContourFilter::New();
  ctf->SetInput(source1->GetOutput());
  ctf->SetValue(0, 220);

  // create the mapper and set a memory limit
  vtkPolyDataMapper* mapper =
    vtkPolyDataMapper::New();
  mapper->SetInput(ctf->GetOutput());
  mapper->SetMemoryLimit(50000);

  // create actor, renderer etc and then render
  vtkActor* actor = vtkActor::New();
  actor->SetMapper(mapper);
  vtkRenderWindow* rWin = vtkRenderWindow::New();
  vtkRenderer* ren = vtkRenderer::New();
  rWin->AddRenderer(ren);
  ren->AddActor(actor);
  rWin->Render();

  // cleanup and delete code
  ...
}
```

IV. Mixed Topologies

The preceding section described how to stream data but it didn't consider the problems associated with streaming unstructured data or mixtures of structured and unstructured data. There are a number of challenges in streaming unstructured data. First an extent must be defined for unstructured datasets. With regularly sampled volumetric data, such as images, an extent defined as (i_{min}, i_{max}, j_{min}, j_{max}, k_{min}, k_{max}) can be used but this does not work with unstructured data. With unstructured data there are a few options. One is to use a geometric extent such as (x_{min}, x_{max}, y_{min}, *ymax, zmin, zmax*) but it is an expensive operation to collect the cells that fit into that extent and such an extent is difficult to translate into the extents used for structured data if they are not axis aligned (consider a curvilinear grid).

A more practical approach is to define an unstructured extent as piece M out of N possible pieces. The division of pieces is done based on cells so that piece 2 of 10 out of a 1000 cell dataset would contain 100 cells. The approach for streaming based on a memory limit is the same as for structured data except that instead of splitting the data into blocks or slabs, the number of pieces, N, is increased. This fairly basic definition of a piece dictates that there isn't any control over what cells a piece will contain, only that it will represent about 1/N of the total cells of the dataset.

This raises the issue of how to support unstructured algorithms that require neighborhood information. The solution is to use ghost cells, which are not normally part of the current extent, but are included because they are required by the algorithm.[25] To support this we extend the definition of an unstructured extent to be piece M of N with G ghost levels. This requires that any source of unstructured grid data be capable of supplying ghost cells. There is a related issue in that some unstructured algorithms, such as contouring, operate on cells while others, such as glyphing, operate on points. Points on the boundary between two different extents will be shared resulting in duplicated glyphs when processed. To solve this we indicate what points in an extent are owned by that extent versus which ones are ghost points. This way point-based algorithms can operate on the appropriate points and yet still pass other points through to the cell based algorithms that require them. In the end both ghost cells and ghost points are required for proper processing of the extents.

Consider Figure 1, which shows one piece of a sphere. The requested extent is shown in red and two ghost levels of cells are shown in green and blue. The points are colored based on their ownership so that all red points are owned by the requested extent, while the green and blue points indicate ownership of the points by other extents. Note that some cells use a mixture of points from different extents.

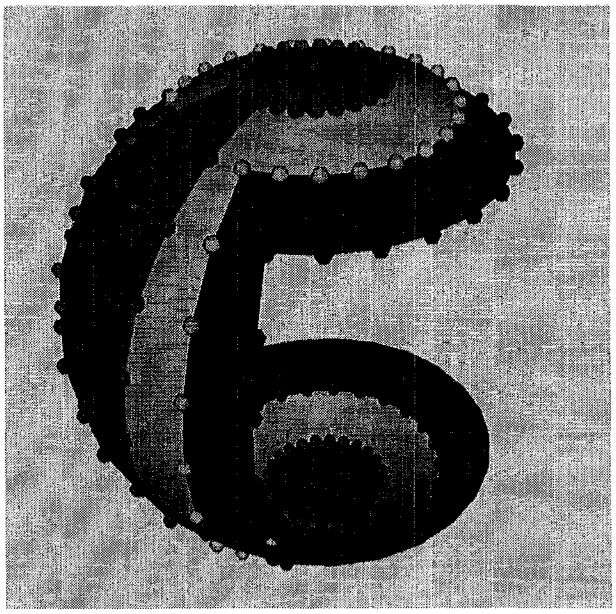


Figure 1.

Now that extents have been defined for both structured and unstructured data, the conversion between them must be defined. For most operations that take in structured data and produce unstructured, a block-based division can be used to divide the structured data into pieces until there are N pieces as requested. If ghost cells are required the resulting extent of the block can be expanded to include them. If ghost point information is required it can be generated algorithmically based on the largest possible extent and some convention regarding what boundary points belong to which extent.

Converting an extent from unstructured to structured could be done in a similar manner except that it is inappropriate for most algorithms that convert unstructured data to structured. Consider a gaussian-splatting algorithm that takes an unstructured grid and resamples it to a regular volume. To produce one part of the resulting volume requires all the cells of the unstructured grid that would splat into that extent. With our definition of an unstructured extent, there is no guarantee that the cells in an extent are collocated or topologically related. So to generate one extent of structured output requires that all of the unstructured data be examined. While this could be done within a loop, our current implementation requires that when translating from a requested structured extent to an unstructured extent, the entire unstructured input is requested.

V. Supporting Parallelism

Most large-scale simulations make use of parallel processing and often the results are distributed across many processing nodes. This requires that the visualization algorithms be capable of operating in such an environment. Supporting parallelism requires some of the same conditions as streaming such as data separability and result invariance. It also requires asynchronous execution, data transfer and collection.

Data transfer is done by creating input and output port objects that can communicate between filters (i.e. algorithms) in different processes. Asynchronous execution is required so that one process is not unnecessarily blocked waiting for input from another process. Consider the pipeline in Figure 2. In this example Filter3 has two inputs. Its first input, Filter1, is in another process so it is requires an input and output port for managing the cross process communication. Before Filter3 executes it must make sure both of its inputs have generated their data. A naïve approach would be to simply ask each input to generate its data in order. The problem is that while Filter3 is waiting for Filter1 to compute its data, Filter2 would be idle. To solve this two modifications are made to the three-step update process. The first modification is to add a non-blocking method to the update process called *Trigger Asynchronous Update.* This method is used to start the execution of any inputs that are in other processes. Essentially this method traverses upstream in the pipeline and when it encounters a port, the port calls *Update Data* on its input.

The second modification is to use the locality of inputs to determine in what order to invoke *Update Data* on them. The locality of an input is defined as 1.0 if the input is generated within the same process, 0.0 if the input is generated in a different process, and between 0.0 and 1.0 if the input is partially generated in one and partially in another (such as in a long pipeline where half of the algorithms are in one process and half in another.) This locality is computed as part of the *Update Information* call. So in Figure 1, *Trigger Asynchronous Update* would be sent to Filter1 which would cause Filter1 to start executing because it is in a different process. Filter2 would ignore the *Trigger Asynchronous Update* call since there are no ports between it and Filter3. Then Filter3 would call *Update Data* on Filter2 first, since it has the highest locality. Once Filter2 has completed executing, *Update Data* would be called on Port2 which could already have the results in memory if Filter1 (which has been executing since the *Trigger Asynchronous Update* call) has completed executing.

In addition to the above infrastructure changes, process initialization and communication calls have been encapsulated into a class so that the user does not have to directly deal with them. Concrete subclasses have been created for distributed-memory and shared-memory processes using MPI and pthreads. Likewise a sort-last parallel rendering class was written that uses inter-process communication to collect and then composite parallel renderings into a final image. Centralized rendering is supported by collecting the polygonal data together using ports between processes connected to an append filter in the collection process. Parallel rendering could also be implemented using polygon collection and then parallel rendering such as WireGL or a future parallel sort-middle approach.[2]

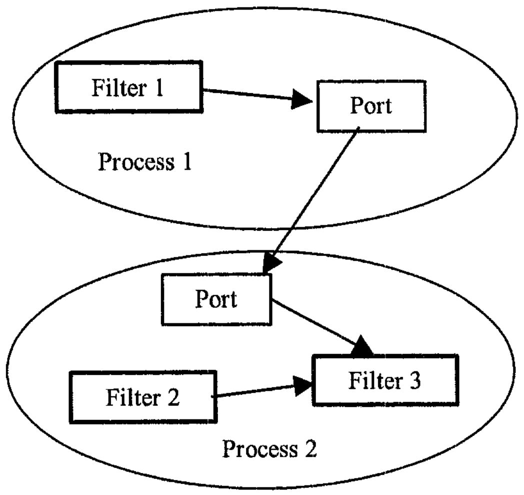


Figure 2

Given the above parallel data streaming architecture, a data parallel program can be created by simply writing a function that will be executed on each processor. Inside that function each processor will request a different extent of the results based on its processor ID. Each processor can still take advantage of data streaming if its local memory is not sufficient, allowing this architecture to process extremely large scale visualizations.

Consider modifying the earlier C-code example to support data parallelism and streaming. First we define a function called process that contains the bulk of the pipeline creation and rendering. This function will be invoked by a vtkMultiProcessController that encapsulates the setup and initialization of the processes. In this example the vtkMPIController subclass of vtkMultiProcessController is used. It is passed into the function as an argument and it provides information such as the process ID, and total number of processes. The visualization pipeline is created as usual but the requested piece (M) and total number of pieces (N) are set on the mapper. This way the mapper of each process will only create its piece of the total N pieces. The memory limit is still set in case generating piece M of N requires excessive memory. Then the request can be broken down into smaller subpieces by the mapper. The actor and renderer are created as usual and then an instance of the TreeComposite class is created and the render window assigned to it. This class encapsulates the sort-last parallel rendering technique. Then a Render call is made to the renderer that will start the rendering process, streaming, and finally the tree-compositing. The main() function creates an instance of vtkMPIController which is one of the subclasses of vtkMultiProcessController, assigns a function for it to execute and then executes it.

```
Void process(vtkMultiProcessController* ctrl,
             void* arg)
{
  int myId = ctrl->GetLocalProcessId();
  int numPrcs = ctrl->GetNumberOfProcesses();

  // The pipeline
  //  source
  vtkMySource* source1 = vtkMySource::New();
  source1->SetStandardDeviation( 0.5 );

  // Iso-surfacing
  vtkContourFilter* ctf = vtkContourFilter::New();
  ctf->SetInput(source1->GetOutput());
  ctf->SetValue(0, 220);

  vtkPolyDataMapper* mapper =
    vtkPolyDataMapper::New();
  mapper->SetInput(ctf->GetOutput());

  // Set the total number of pieces
  mapper->SetNumberOfPieces(numProcs);
  mapper->SetPiece(myId);
  mapper->SetMemoryLimit(50000);
  vtkActor* actor = vtkActor::New();
  actor->SetMapper(mapper);
  vtkRenderWindow* rWin = vtkRenderWindow::New();
  vtkRenderer* ren = vtkRenderer::New();
  rWin->AddRenderer(ren);
  ren->AddActor(actor);

  // setup the tree composite and render
  vtkTreeComposite *tc = vtkTreeComposite::New();
  tc->SetRenderWindow(rWin);
  rWin->Render();

  // cleanup and delete code
  ...
}

int main( int argc, char* argv[] )
{
  vtkMPIController* controller =
    vtkMPIController::New();
  controller->Initialize(&argc, &argv);

  controller->SetSingleMethod(process, 0);
  controller->SingleMethodExecute();
  controller->Delete();
  return 1;
}
```

VI. Results

The results reported here are based on using an in memory analytic function as a data source. This is designed to mimic visualizing data from a running simulation where the simulation data is in memory. This also avoids dealing with issues of massively parallel I/O which are beyond the scope of this paper. The data is organized as a regular volumetric dataset with a double precision scalar value computed at each point. Three different visualization examples were tested. The first two examples were tested on a cluster of 8 SGI Origin 2000s each with 128 shared-memory processors. The third example was tested on a cluster of eight PCs each with two shared memory processors.

The first visualization example was a data parallel pipeline that computes an isosurface from the volume, computes a gradient magnitude field from the volume, then probes the gradient magnitude field with the isosurface and renders the result using a sort last parallel rendering technique.[3] (see Figures 3 and 4) This example was run with input data sizes of 39 gigabytes, 1.1 terabytes, and 0.9 petabytes on configurations between 1 and 1024 processors. The polygons produced were rendered in software using Mesa.

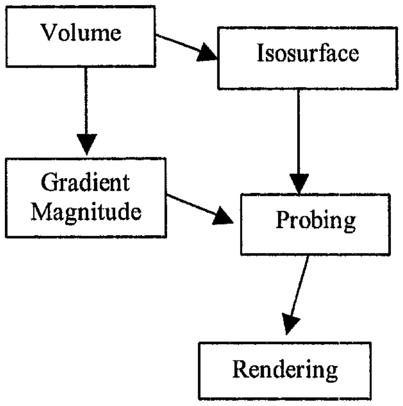
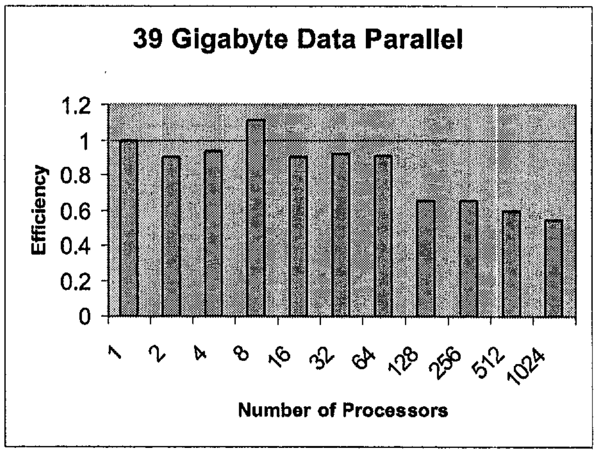


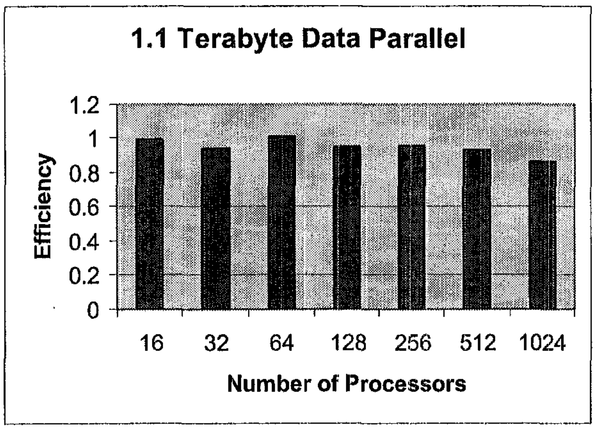
Figure 3

The 39-gigabyte run produced 20 million polygons. Its results are reported in terms of efficiency versus number of processors. The efficiency is a measure of how effectively the additional processors are being utilized. An efficiency of 1.0 represents a linear speedup versus the number of processors. The results are based on wall clock processing time required and include any time required to start the processes and allocate memory for each one. The 39 gigabyte test is small enough that for anything beyond 64 processors the startup time dominates the actual calculation time. Consider that linear scaling would result in a ten second execution on 1024 processors while the time required for MPI to start 1024 processes and for each of them to allocate their memory is on the order of 90 seconds. The results show linear performance up to about 64 processors. Beyond that the calculation is simply to quick to make using more processors worthwhile. If the visualization were to be generated at the end of each time step, so that the process could be kept running, then using 1024 processes would be of value.

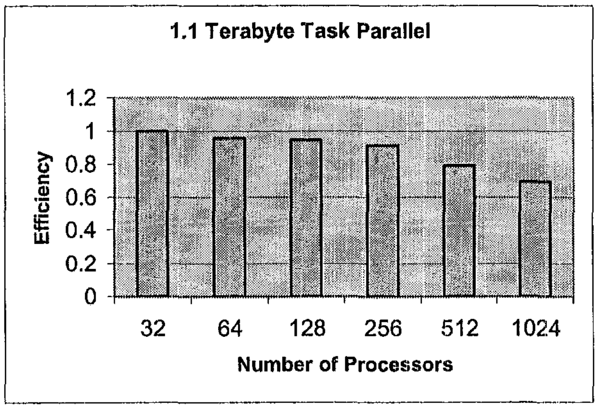


The results of the 1.1 terabyte run are provided for 16 to 1024 processors since running on one to eight processors would be too time consuming. This run produced 190 million polygons and with the larger problem size the results are nearly linear across the entire range. The worst case is the results for 1024 processors that show an efficiency of 0.86 for a 418 second execution time. This is expected due to the process initialization time.

The 0.9 petabyte run was tested on 1024 processors, required 360,000 seconds, and produced 16 billion polygons. It is worth noting that the time required for this run was nearly linear with respect to the time required for the 1.1 terabyte run on 1024 processors. This linearity is due to the streaming of the data. The 0.9 petabyte run requires the same memory footprint as the 1.1 terabyte run.



The second visualization example demonstrates task parallelism. In task parallelism there are multiple independent visualization pipelines. In this case there were three pipelines. The first pipeline is the probed isosurface pipeline as used in the first example. The second pipeline computes a gradient vector field from the input data, it then reduces the resolution and then creates oriented glyphs at each point. The third pipeline extracts a cut plane from the input data and displays it as shown in Figure 5. In a fully data parallel configuration all three tasks would be run on each processor similarly to the data parallel example. For contrast, in this test the tasks were distributed across the processors with the majority of the processors assigned to generating the probed isosurface. So the example is task parallel with each task using data parallelism across the processors it was allocated. The results indicate successful task parallelism with a slightly less than linear speedup due to poor load balancing between the tasks.



The third example considered pipeline parallelism, where one processor performs some of the visualization while another performs the rest of it. This is common in cases where the graphics resources are available to only some of the processors. We simulated this case by running the data parallel example on a cluster of eight Windows 2000 machines connected via gigabit Ethernet. Each machine had two processors and one accelerated OpenGL graphics card. We decided to use the screen for hardware accelerated rendering which limited us to eight hardware renderers even though there were sixteen processors. In this case the hardware rendering consumed less than one percent of the total time.

Simple modifications to the first example allowed the use of both processors on a machine for the computation while only one processor was used to transmit the data to the rendering hardware. Sort-last compositing was used to combine the eight hardware renderings into the final buffer. This resulted in a linear speedup from eight to sixteen processors due to the high performance of the hardware rendering and low cost of the shared memory data transfer. This capability is significant since in many cases the hardware is not homogeneous and standard data parallel approaches will not fully utilize the available resources. In this case the first processor could render the data while the second processors was computing the next piece. For this hardware configuration it allowed the use of all sixteen processors where otherwise only eight would have been used.

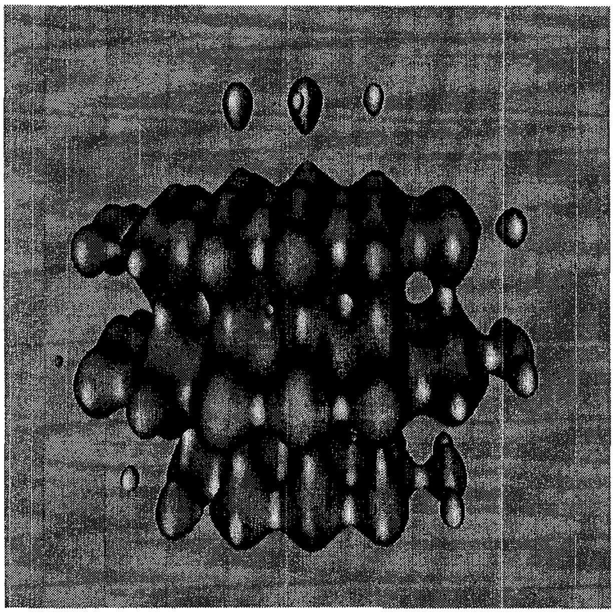


Figure 4

VII. Discussion

While this paper has addressed some difficult issues, there are others that were not addressed that are still being researched. In many simulations with distributed data the ghost cells can only be obtained from other processes. Currently there is no standard mechanism for one process to determine where to find specific ghost cells. Ideally there would be an efficient mechanism so that an algorithm that required ghost cells could determine what process to request them from. Additionally some algorithms, such as streamlines, require parallel specific versions to be written that can pass information concerning when a streamline exits one piece and enters another. These issues are being actively researched in hopes of incorporating such capabilities into the architecture.

This architecture has shown that parallel data streaming can be effectively used to visualize petabyte datasets across hundreds of processors even when the visualization would normally require far more memory than is available. It provided a solution to the challenges associated with performing such streaming on mixed topology datasets and demonstrated data, task, and pipeline parallelism within a software framework that is intuitive and extensible.

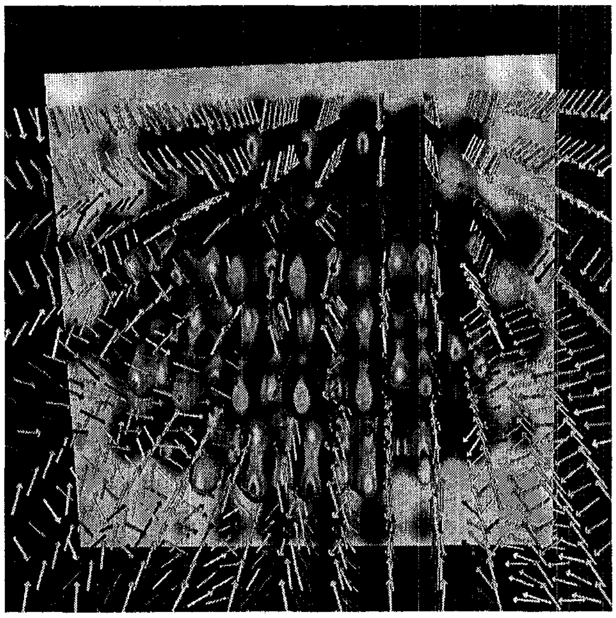


Figure 5

VIII. References

1. G. Abrams, and L. Trenish. An extended data-flow architecture for data analysis and visualization. *Proceedings of Visualization '95*, pg. 263-270. IEEE Computer Society Press 1995.
2. G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan, Distributed Rendering for Scalable Displays, *Proceedings of Supercomputing 2000*
3. H. El-Rewini, T. Lewis, and H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.
4. R. Haimes, and D. E. Edwards. Visualization in a Parallel Processing Environment, American Institute of Aeronautics and Astronautics, Inc., 1997.
5. C.C. Law, W.J. Schroeder, K.M. Martin, and J. Tempkin. A Multi-threaded streaming pipeline architecture for large structured data sets. *Proceedings of Visualization '99*. IEEE Computer Society Press, October 1999.
6. C. R. Johnson, and S.Parker. The SCIRun parallel scientific computing problem solving environment. *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.
7. M. Miller, C. Hansen, and C. Johnson. Simulation steering with SCIRun in a distributed environment. *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
8. W. J. Schroeder, K.M. Martin, and W.E. Lorensen. *The Visualization Toolkit An Object oriented Approach to 3D Graphics*. Prentice Hall, 1996.
9. S.G. Parker, D.M. Weinstein, and C.R. Johnson. The SCIRun computational steering software system. *Modern Software Tools in Scientific Computing* E. Arge, A.M. Brauset, and H.P. Langtangen, editors, pg. 1-40. Birkhauser Press, 1997.
10. P. K. Agarwal, L. Arge, T. M. Murali, and others. I/O-Efficient Algorithms for Contour-Line Extraction and Planar Graph Blocking. In *Proc. ACM-SIAM Symp. On Discrete Algorithms*, 1998 (to appear).
11. C. Upson, T. Faulhaber Jr., D. Kamins and others. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*. 9(4):30–42, July 1989.
12. M. Krogh and C. Hansen. Visualization on massively parallel computers using CM/AVS. In *AVS Users Conference*, 1993.
13. S. Ramaswamy and S. Subramanian. Path Caching: A Technique for Optimal External Searching. In *Proc. ACM Symp. On Principles of Database Sys.*, pp. 25-35, 1994.
14. Y. J. Chiang and C. T. Silva. Interactive Out-of-Core Isosurface Extraction. In *Proc. Of Visualization '98*. IEEE Computer Society Press, October, 1998.
15. M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-Of-Core Visualization. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.
16. M. Cox and D. Ellsworth. Managing Big Data for Scientific Visualization. In ACM Siggraph '97 Course #4 *Exploring Gigabyte Datasets in Real-Time: Algorithms, Data Management, and Time-Critical Design*. August, 1997.
17. T. A. Funkhouser, S. Teller, C. H. Sequin, and D. Khorramabadi. Database Management for Models Larger Than Main Memory. In *Interactive Walkthrough of Large Geometric Databases*, Course Notes 32, Siggraph '95, August 1995.
18. I. Itoh and K. Koyamada. Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Trans. On Visualization and Computer Graphics*. 1(4):319-327.
19. D. Kenwright and R. Haimes. Vortex Identification - Applications in Aerodynamics: A Case Study. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.
20. D. Kenwright. Automatic Detection of Open and Closed Separation and Attachment Lines. In *Proc. Of Visualization '98*. IEEE Computer Society Press, October, 1998.
21. S. Subramanian and S. Ramaswamy. The P-Range Tree: A New Data Structure for Range Searching in Secondary memory. In *Proc. ACM-SIAM Symp. On Discrete Algorithms*, pp. 378-387, 1995.
22. S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and Ordering Large Radiosity Computations. In *Proc. Of SIGGRAPH '94*. pp 443-450, July, 1994.
23. 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*.
24. D. E. Vengroff and J. S. Vitter. Efficient 3-D Range Searching in External Memory. In *Proc. Annu. ACM Sympos. Theory, Comp.*, pp 192-201, 1996.
25. W. Gropp, E. Lusk, and A. Skjellum. Using MPI, Portable Parallel Programming with the Message-Passing Interface, The MIT Press, Cambridge, Mass. 1994.