Large-Scale Data Visualization Using Parallel Data Streaming

James Ahrens and Kristi Brislawn Los Alamos National Laboratory
Ken Martin, Berk Geveci, and C. Charles Law Kitware
Michael Papka Argonne National Laboratory

July-August, 2001

1 Problem Statement

Computer simulations are used in various fields to visualize data and resolve models of real-world phenomenon. Using advanced algorithms and additional computing power, the researchers can study these models to more detailed levels. The amount of data that is used for the computer simulations can be so massive that they are usually run in parallel on clusters of high bandwidth supercomputers to effectively visualize the data set. The traditional improvements in hardware capabilities continue to make larger data sets possible and more accessible. With time the large data set visualization and the size of the data sets will continue to get bigger and efficient.

Data streaming, parallel visualization, and mixed-topology visualization are all known techniques, but it can be difficult and combining all three is a significant challenge. Difficulties we face are :

- Visualizing large data in real-time is challenging due to the massive amount of data, often in terabytes, and the need to share limited computational resources with the ongoing simulation process.
- Traditional visualization algorithms, like streamline generation or mesh decimation, struggle with distributed and parallel data processing. Additionally, visualizations can introduce mixed data-set topologies, even when the original simulation data has a uniform topology, as seen in the case of generating an isosurface from a rectilinear grid.

We do not intend to address issues in large data-set visualization such as massively parallel I/O, effective load balancing, or parallel rendering, although we briefly discuss their implications.

2 Motivation

- Inadequate Architecture for Feature Extraction: Current visualization algorithms lack a coordinated architecture. While they can extract features and perform visualization tasks, they often work independently and write output to disk incrementally. This approach is inefficient and can lead to constant read-write operations in between algorithms, which is not an optimal.
- Centralized Control in Existing Systems: Popular visualization systems like Open Data Explorer (OpenDX), Application Visualization System (AVS), Demand Driven Visualizer (DDV), and SCIRun employ centralized executives to instantiate modules, allocate memory, and execute modules. While this enables task parallelism and data parallelism to some extent, it can be challenging to efficiently control multiple processes from a single centralized executive..
- Demand-Driven Execution Model: Demand Driven Visualizer (DDV) is mentioned as a system that efficiently handles large data sets by requesting only the necessary data. However, it does not yet support a mixture of task, data, and pipeline parallelism on both distributed and shared-memory multiprocessors. This gap highlights the need for more versatile and comprehensive solutions.
- Streaming Data in Memory: Existing approaches do not adequately address the challenge of streaming data in memory when data set topologies change. Many visualization techniques can alter data topologies, making it essential to efficiently handle such changes. Our research aims to tackle this critical consideration in data visualization.
- Flexibility and Capabilities: While some existing solutions, such as pV3 and Ensight, offer support for large or parallel data, they are often designed as turnkey applications and may lack the flexibility and capabilities required for advanced research and customization.

3 Methodology

1) Data Streaming Benifits :
Streaming data through a visualization pipeline offers two primary benefits:

- Allows processing of data that exceeds available memory or swap capacity.
- Reduces memory usage, leading to improved cache utilization and minimal disk swapping.

2) Key Software Requirements :
Effective data streaming necessitates that visualization software supports:

- Data Separability: Algorithms must efficiently break data into coherent pieces, ideally preserving geometry, topology, and data structure.
- Data Mappability: Control over data flow through the pipeline is crucial, enabling size control and algorithm configuration.
- Result Invariance: Results must remain consistent irrespective of the number of pieces or execution mode (single-threaded or multi-threaded).

3) Pipeline Architecture for Volumetric Data :
A three-step pipeline update mechanism is used for regularly sampled volumetric data:

- Update Information: Determines data set characteristics (data type, extent, scalar values, modification time).
- Update Extents: Modifies data requests and calculates memory requirements, supporting streaming based on memory limits.
- Update Data: Executes the visualization pipeline to produce requested updates.

4) CPU Overhead and Performance :
Implementing the three-step process incurs minimal CPU overhead, with cache locality often compensating for the additional processing. Exception: Overhead when recomputing shared boundary cells, typical in neighborhood-based algorithms
5) Unstructured Data Streaming Challenges :
Streaming unstructured data poses challenges:

- Defining extent for unstructured data is complex.
- Options include geometric extents, but they can be expensive and may not align with structured data extents.

6) Handling Unstructured Data with Ghost Cells :
Unstructured data streaming can use ghost cells to address neighborhood-based algorithm requirements.

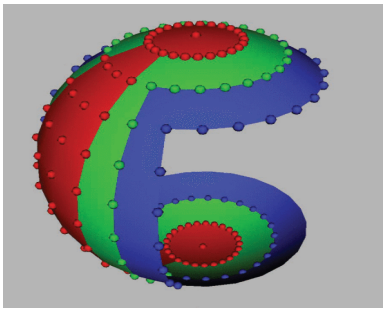

Figure 1: Breaking up a sphere into a piece (red) and ghost-level cells and points (blue and green)

- If this requires ghost cells, the block's resulting extent can be expanded to include them.
- If ghost point information is required, it can be generated algorithmically based on the largest possible extent and on some convention regarding what boundary points belong to which extent.

7) Supporting Parallelism :
Parallel processing is supported by ensuring data separability, result invariance, and using asynchronous execution, data transfer, and collection.

- Data transfer facilitated through input and output port objects for inter-filter communication.
- Asynchronous execution ensures processes are not blocked while waiting for data.

8) Parallel Rendering :
Parallel rendering is achieved through interprocess communication to collect and composite parallel renderings into a final image.

- Centralized rendering is supported by collecting polygonal data.
- The architecture can also implement parallel rendering using polygon collection.

9) Data Parallelism and Streaming :
Data parallel programs can be created by writing a function executed on each processor, with each processor requesting different extents based on its ID.

- Streaming can still be utilized to process large-scale visualizations effectively.

4 Results

The results presented here are based on using an in-memory analytic function as a data source to simulate visualizing data from a running simulation.
The first visualization example was a data-parallel pipeline that computes an isosurface and gradient magnitude and then color-maps the gradient magnitude onto the isosurface and renders the result using a sort-last parallel rendering technique.
The second visualization example demonstrates task parallelism, where there are multiple independent pipelines (3 in our example). In a fully data-parallel configuration, all 3 tasks would be run on each process but in this test, we distributed the tasks across the process

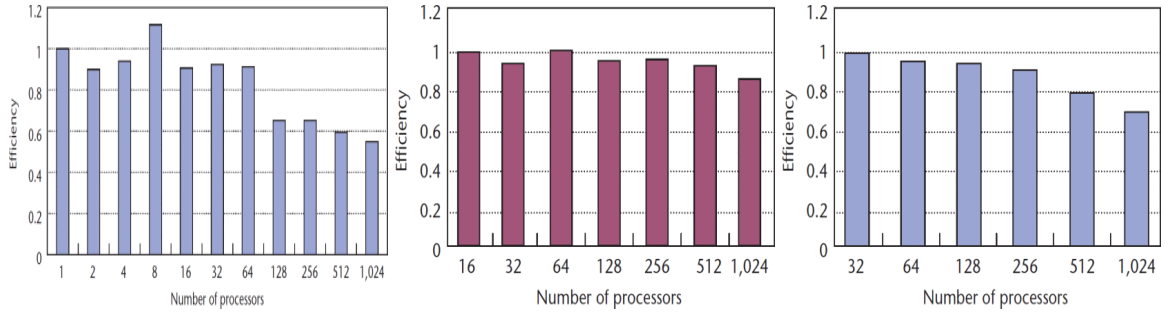

Figure 2: These are the results of 39-Gbyte,1.1-Tbyte using data parallel visualization and 1.1-Tbyte using task parallel visualization respectively

Notably the 0.9-petabyte run's time was nearly identical to the 1.1-terabyte run due to data streaming and similar memory footprints. These findings are valuable for high-performance visualization scenarios, particularly when dealing with large data sets and varying processor configurations.

In our third example, we tested pipeline parallelism with one processor handling part of the visualization while another managed the rest. This was on a cluster of 2000 Windows machines connected via gigabit Ethernet. Hardware rendering, though limited to eight renderers out of 16 processors, consumed less than 1 percent of total time. Simple adjustments to our initial setup allowed both processors on each machine to contribute to the computation, with only one handling data transmission to rendering hardware. Using sort-last compositing to merge the eight hardware renderings, we achieved a linear speedup from 8 to 16 processors, leveraging efficient hardware rendering and shared-memory data transfer. This is particularly valuable when dealing with heterogeneous hardware.
Although this article has addressed some difficult issues, there are still considerable problems . In many simulations with distributed data, the ghost cells can only be obtained from other processes. Currently, there isn't a 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.