A Multi-Threaded Streaming Pipeline Architecture for Large Structured Data Sets

C. Charles Law (*Kitware, Inc.*)
William J. Schroeder (*Kitware, Inc.*)
Kenneth M. Martin (*Kitware, Inc.*)
Joshua. Temkin (*RPI*)

Abstract

Computer simulation and digital measuring systems are now generating data of unprecedented size. The size of data is becoming so large that conventional visualization tools are incapable of processing it, which is in turn is impacting the effectiveness of computational tools. In this paper we describe an object-oriented architecture that addresses this problem by automatically breaking data into pieces, and then processes the data piece-by-piece within a pipeline of filters. The piece size is user specified and can be controlled to eliminate the need for swapping (i.e., relying on virtual memory). In addition, because piece size can be controlled, any size problem can be run on any size computer, at the expense of extra computational time. Furthermore, pieces are automatically broken into sub-pieces and each piece assigned to a different thread for parallel processing. This paper includes numerical performance studies and references to the source code which is freely available on the Web.

1 Introduction

Computer simulation and digital measuring systems are now generating data of unprecedented size. For example, Kenwright [Kenwright98b] describes computational fluid dynamics data sets of sizes ranging up to 600 GByte with larger data (terabytes) foreseen. Machiraju reports computational data sets of similar sizes [Machiraju98]. Measuring systems are generating large data as well. It is anticipated that the Earth Orbiting Satellite (EOS) will generate a terabyte of data *daily*. Volumetric data sources such as CT, MRI, and confocal microscopy generate large volumetric data sets; the addition of time-captured data will multiply the overall sizes of these data sets dramatically.

A primary goal of visualization is to communicate information about large and complex data sets [Schroeder97]. Generally, the benefit of visualization increases as the size and complexity of data increases. However, as data sizes increase, current visualization tools become ineffective due to loss of interactivity; or even fail, as data overwhelms the physical and virtual memory of the computer system. If researchers, engineers, scientists, and users of large data are to take full advantage of advances in computational simulation and digital measurement systems, visualization system must be designed to handle data sets of arbitrary size.

1.1 Why Visualization Systems Fail

Conventional commercial visualization systems such as AVS [AVS89] and IBM Data Explorer [DataExplorer] fail in two important ways when encountering large data. First, interactive control of the visualization process is lost when the time to transmit, process, or render data becomes prohibitively large. (Large may mean millions of cells or primitives, which is relatively small compared to the data sizes quoted previously.) This difficulty causes significant delays in processing results because the ability to rapidly explore data is replaced with a hit-and-miss batch process or other ad hoc methods to reduce data size or extract regions of interest.

While the loss of interactivity is a serious problem, visualizing larger datasets may cause complete system failure. In this second failure mode, the physical or virtual memory address space of the computer is overwhelmed, and the system thrashes ineffectively or crashes catastrophically. The typical response to this problem is to buy larger computers such as a supercomputer, but this solution is prohibitive for all but the wealthiest computer users, and in many cases, may not solve the problem on the largest datasets.

These failures are typically due to one of four problems [Cox97a] [Cox97b]) 1) The data may be too large for local computer memory resulting in excessive thrashing. 2) The data may be too large for local disk (either for storage or paging to virtual memory), making the system unusable. 3) The data may be too large for the combined capacity of remote and local disk. 4) The bandwidth and latency of data transfer causes bottlenecks and results in poorly performing or unusable systems. Creating successful visualization systems for big data requires addressing each of these four problems.

1.2 Goals of Large Data Visualization

We see two fundamental, but opposing, design goals for large data visualization systems.

1. The system must be able to process any size data, on any size computer, without loss of information. The ability of a computer system to treat large data must scale well with processing power and available memory.
2. The system must be allow users to quickly identify important regions in the data, and then enable focused attention on those regions in ever greater detail (to the limit of the resolution of the data).

The quandary for the visualization scientist is that these goals call for the system to be as accurate as possible, and at the same time as interactive as possible. Accuracy is necessary when observing detailed quantitative behavior or comparing data. Interactivity is necessary to understand the overall structure of data or when looking for important features or other qualitative relationships. Often users adopt both goals during a visualization session. Interactive exploration is used to identify features of interest, at which point accurate visualizations are generated in order to understand detailed behavior or data values.

In this paper we focus on the first goal: to process any size data on any size computer, with good scalable characteristics as the computer system grows in memory, computational power, and data bandwidth. We believe this goal is the necessary starting point, since it is necessary to visualize large data before we can visualize it interactively.

2 Approach

Successfully managing large data requires breaking data into pieces, and then processing each piece separately, or distributing each piece across a network for parallel processing. To obtain interactive performance, parallel algorithms must be developed (distributed and/or multithreaded) to achieve the highest possible processing rates, and the total data to be processed must be minimized by employing segmentation algorithms, subsampling, or multiresolution methods.

A subtle but fundamental requirement of a successful system is that it must manage the hierarchy of memory efficiently to achieve maximum data throughput. The hierarchy of memory ranges from tape archival systems, to remote disk, local disk, physical memory, cache, and register memory. The speed of data access can vary by orders of magnitude between each level, and significant computational expense is required to move the data between each level. Therefore, a well designed system will manage the memory hierarchy to avoid moving data between levels, and holding it at the highest possible level until processing is complete.

2.1 Desirable Features

We believe that there are several important characteristics of visualization systems that successfully contend with big data. Some of these include the following.

Graceful Degradation. Ideally, visualization systems should degrade predictably (e.g., linearly) as the data size grows relative to the computer capacity expressed as a function of CPU performance, memory, and data bandwidth. Systems exhibiting this characteristic instill confidence in users, since the desired system performance can be controlled by hardware expenditures—what you pay for is what you get. It also means that no matter the size of the computer or the data, given enough time the system can process the data. Practically what this means is that a single processor PC with 64 megabytes of memory connected to a data source via network or local data bus should generate the same results as a high-end, multiprocessor supercomputer with several gigabytes of memory, the difference being the time taken to complete the visualization.

Minimizes Disk Access. One important lesson learned from conventional virtual memory operating systems is that depending on disk storage for computer memory is undesirable. Most users have experienced running programs which scale gracefully until the system begins swapping, at which point the elapsed time to complete the computation dramatically increases. This is because the time to write or read pages from disk is large compared to the time the CPU takes to perform a single instruction. For example, typical disk access time is on the order of tens of milliseconds (primarily disk head movement). During this same period of time several million operations can be performed by the CPU.

Cached. Modern computer systems are designed with data caches to improve performance. Data caches store pieces of data close to the CPU to minimize delays during read and write operations. Caches dramatically improve performance because frequently used data or code can be accessed faster than less frequently used data or code. Appropriate use of caching when visualizing big data objects can dramatically increase the performance of the system.

Parsimonious. Cox and Ellsworth [Cox97a] [Cox97b] have observed that the amount of data actually generated by a visualization algorithm is small compared to the total amount of data. For example, streamline generation is typically of order $O(N)$, where N^3 is the size of the dataset. Isosurface generation is typically of order $O(N^2)$ since a surface is generated from a volume of data. The implication is that sparse or parsimonious traversal methods can be created which dramatically reduce the amount of data accessed by the system.

Parallel. Parallel processing techniques have demonstrated their ability to greatly accelerate computation, and computer systems are often configured with more than one processor. Visualization systems must take advantage of this capability to deliver timely results. Current visualization systems are parallelized in two ways: either on a per algorithm basis (fine-grained parallelism) or across parallel branches of data flow (coarse-grained parallelism). Both methods of parallelism should be supported.

Hardware Architecture Independent. It is important to develop systems not tied to a particular computer architecture. For example, depending on shared memory parallel processing will fail once data size exceeds shared memory space. Similarly, systems based on a network of heterogeneous computers exclude “typical” engineers and scientists users with a modest single processor system. Visualization systems must be adaptable to a variety of hardware architectures, including the simplest single processor modest memory system of most users.

Demand-Driven. Demand-driven or lazy evaluation systems perform operations on data only when absolutely required. In many cases this avoids unnecessary work. For example, since computing streamlines in a grid requires only

a portion of the original data, loading and processing only the data necessary to perform particle advection can reduce computational and memory costs significantly. Of course, the advantages of caching described earlier point to the fact that lazy evaluation must be tempered by careful consideration of the granularity of data access and evaluation.

Component Based. Commercial visualization systems such as AVS and IBM Data Explorer clearly demonstrate the benefit of component based systems. These systems enable users to run-time configure visualizations by connecting objects, or components, into data flow networks. The network is then executed to generate the desired result. Benefits of this approach include adaptability and flexibility to changing data and visualization needs. The power of component-based systems is that the components are general and can be reused in different applications. Such reuse can be a significant advantage over customized applications, since the effort to maintain and tune components has immediate impact across all applications using them. Tailored applications often suffer from a limited user base (which has impact on long-term survivability of the application), and are often difficult to modify-it is often not possible to drop in a new component to replace an older one.

Streaming. Many visualizations consist of a sequence of operations. For example, in our work we routinely use combinations of data subsampling (extract portion of data), isosurfacing, decimation (reduce mesh size), Laplacian smoothing, and surface normal generation to create high quality images of 3D contour surfaces. These operations are typically implemented by applying a pipeline of reusable components to a stream of data. In conventional use, the size of the data stream is controlled by the size of the input data. When the data is big, however, we would prefer to control the size of the data stream based on run-time configurable memory limits, perhaps on a component-by-component basis.

Other important features of a good software design such as efficiency, ease of use, extensibility, and robustness are also important and assumed.

2.2 Previous Approaches

Two general approaches have been used to process large data sets: use of out-of-core algorithms and design of large data architectures. Multiresolution methods form a third approach, and are typically used in visualization systems with a primary goal of interactivity.

2.2.1 Large Data Algorithms

Feature extraction has been a successful technique for processing large data sets. 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) [Chiang98] [Itoh95] and related computational geometry work [Agarwal98] [Rama94] [Sub95] [Teller94] [Vengroff96] [Funk95], streamlines [Ueng98], separation and attachment lines [Kenwright98], and vortex core lines ([Kenwright97]). 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.

There are two difficulties with this approach. First, the approach presumes that the extracted features are sufficiently small enough to fit into system memory—a poor assumption as data sizes increase in size. Eventually, the extracted features will be large enough that they will not fit into memory. Second, the extraction of features depends on I/O from disk. Data that must be processed by a series of filters must be read and written to and from disk several times, a poor use of the memory hierarchy, and likely to result in poorer performing visualization systems.

2.2.2 Large Data Architectures

Another approach is the design of architectures to directly support large data visualization. For example, researchers at NASA Ames replace the virtual memory system, using an intelligent paging scheme that recognizes the structure and boundaries of data [Cox97a] [Cox97b]. This approach has demonstrated significant improvements over standard virtual memory. It is best suited for the application of a single algorithm. We believe this approach is not as effective for supporting a pipeline of filtering operations. Using the virtual paging scheme, memory is repeatedly swapped as each algorithm processes the entire dataset, one after the other. Instead, we believe that ingesting a *piece* (subset of the entire dataset) of data, and then processing the entire piece—*without any swapping*—through the pipeline can achieve dramatically better results. This approach, which we call *data streaming*, has the benefit that data remains in higher-performing memory as it is processed.

Another successful system was developed by Haimes. This system, 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 a general application solution since it is a custom application, and offers no reusable components. Furthermore, depending on a collector is problematic in the larger data environment. This solution also seems vulnerable to network bandwidth and latency limitations.

3 Multi-Threaded Data Streaming

The key to the structured data streaming pipeline is the ability to break data into pieces. By controlling the piece size carefully, we can avoid swapping and insure that the data is processed in physical memory (or better). The

piece size is set at run-time depending on the size of computer memory, the number of filters in the visualization pipeline, and the size of data. In addition, using the same process to break data into pieces, we can break pieces into sub-pieces for the purpose of multi-threading, each processor taking one sub-piece. Thus our approach naturally supports shared-memory parallel processing.

The overarching goal of this work was to create a multi-threaded, streaming architecture for the structured points (i.e., images and volumes) portion of the *Visualization Toolkit (VTK)* [Schroeder97]. Limiting the problem to structured data greatly simplified the design. In the future we plan on extending our approach to unstructured data.

3.1 Key Principles

The following three principles guided the design of the multi-threaded data streaming architecture for structured data.

1. *Data Separability*. 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 Input*. 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.

Structured data is readily separable— the topological *i-j-k* coordinate scheme naturally breaks data into coherent pieces and was employed as the separation mechanism in the architecture. Most structured (i.e., imaging) algorithms are mappable since the input pixels required to produce an output pixel are known. And finally, careful design of the algorithms enables proper treatment of the boundary of each piece, thereby insuring that the output remains invariant as the number of pieces changes.

3.2 Architectural Overview

As previously mentioned, the basic idea behind our design is to control the amount of data passing through the pipeline at any given time. To do this, we replace the typical visualization pipeline architecture shown in Figure 1(top) with that shown in Figure 1(bottom). As shown in the figure, we augment the usual process object/data object pair with a third object—a data cache. The purpose of the cache is to manage allocation and access to the data. Furthermore, the cache negotiates with its upstream and downstream filters to configure the pipeline for execution. The negotiation process considers available memory (memory limits are set on a per cache basis), requested memory, algorithm kernel size (to determine whether boundary padding is required), and the size of the input and output data. In addition, each algorithm (i.e., filter) expects to operate on a piece of data rather than the entire data set. In fact, because the filters are designed to operate on pieces of data, it is possible to break a piece into sub-pieces and assign them to separate processor threads for parallel processing with little to no changes in the code.

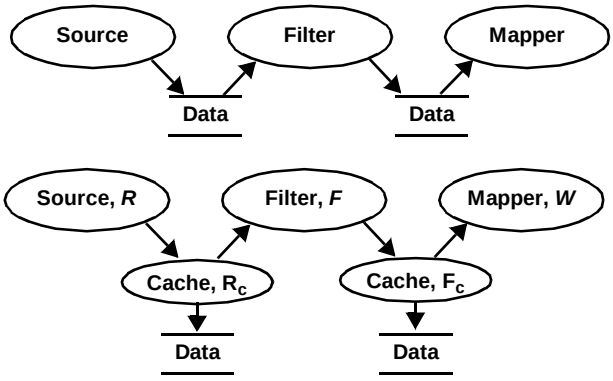

Figure 1. A conventional visualization pipeline (top) compared to a streaming pipeline (bottom).

3.3 Configuring the Pipeline

We will use Figure 1(bottom) to illustrate the streaming architecture. Three process objects (a source *R*, filter *F*, and mapper object *W*), associated caches, and data objects are shown. In this example assume that the mapper is generating output data that may be written to disk or rendered, and further assume that the input data size is greater than a user-specified memory limit.

When the request comes to write the data, a two stage process is initiated (the pipeline is demand driven, so the pipeline only executes when data is requested). In the first stage, the pipeline configures itself for streaming. The writer object requests a portion of data of size *S* from its input cache F_C. F_C has a memory limit S_C != *S*. Assume in this example that F_C grants a request of size S_C with $S_C < S$. *W* accepts this grant and configures itself to process data in pieces of size S_C. The configuration process continues with filter *F*. The filter is limited to generate an output of size S_C (its cache limit), and has knowledge of the algorithm it implements such as kernel size (how many input pixels are required to generate an output pixel) and the relative difference in size between input and output (e.g., vtkImageMagnify can change output size by integer multiples of input size). This information is translated into a request for input of a particular size S_R from cache R_C. The negotiation process is repeated and filter *F* configures itself to operate on pieces of appropriate size, S_R. Finally,

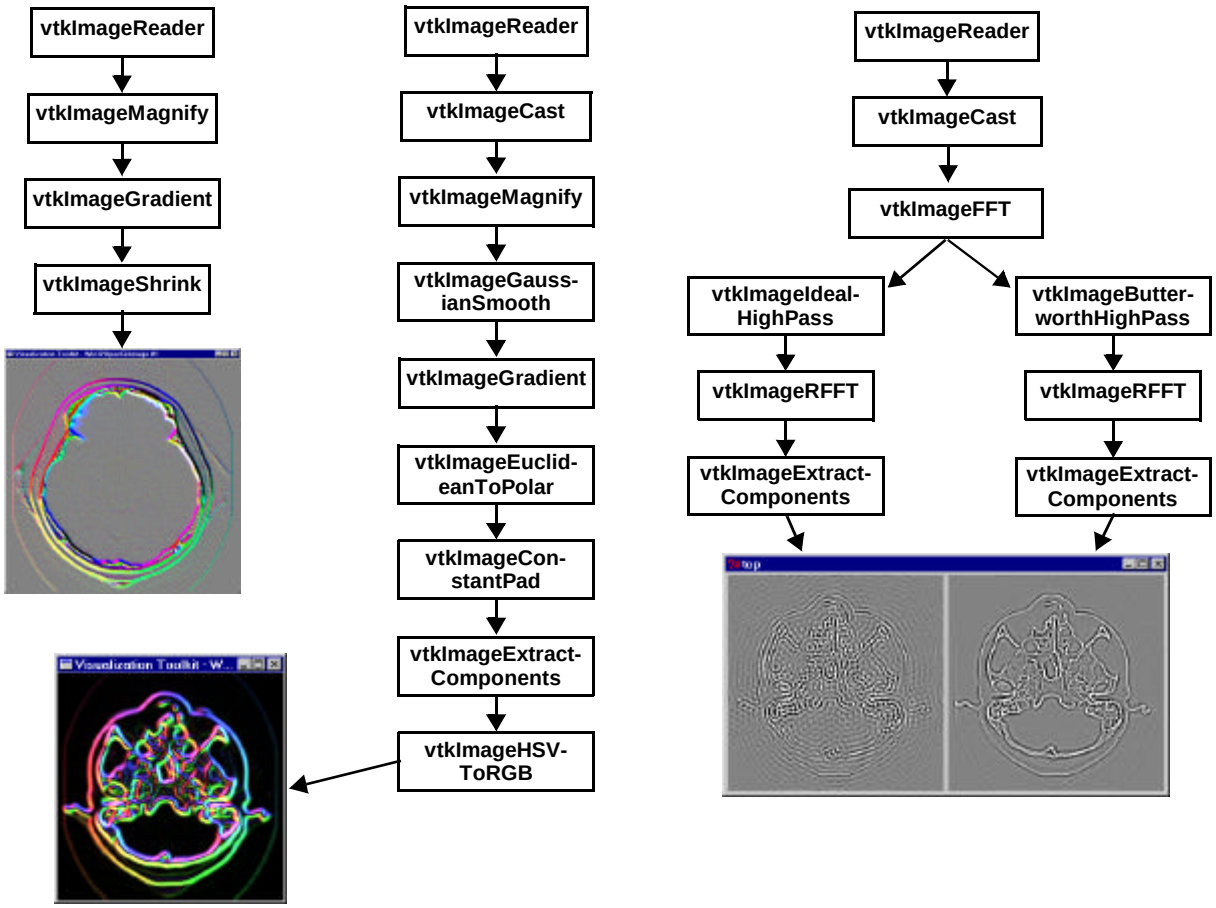

Figure 3. Three pipelines used to test the streaming architecture. The vtkImageCastFilter in pipelines #1 and #2 is used to control the size of the input data. Note that while the images show just a single slice, the entire input volume is processed.

the reader configures itself to generate pieces of size S_R which completes the first stage of the configuration process.

In the second stage, the pipeline begins execution and begins streaming data through it. Note that each process object may operate on different number of pieces depending on the results of the negotiation process. In the worst case, filters like image histogram may have to revisit their input several times (in pieces) to generate a piece of output. This is because the histogram filter requires visiting all input pixels to generate the correct results for a single output pixel.

3.4 Multithreading

Adding multi-threading to the pipeline configuration process is straightforward. Once the initial piece size is negotiated for each filter, pieces are further subdivided into sub-pieces depending on the number of available processors. Then each thread operates on a sub-piece in exactly the same way that pieces are operated on. For best performance each piece should fit in main memory and then each sub-piece will be 1/N the size of the piece, where N is the number of processors.

3.5 Handling Boundaries

Many algorithms require a kernel of data around an input data location to generate a single output value. For example, a Gaussian smoothing operation may take a 2x2 or 3x3 input kernel to generate an output data value. Such algorithms impact the process of generating pieces (and sub-pieces) since the pieces must be enlarged to support algorithm execution. This means that pieces may be generated with overlap, as shown in Figure 2. As long as the overlap size relative to the piece size is relatively small,

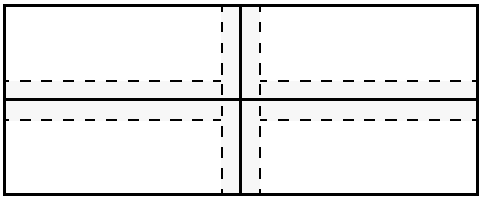

Figure 2. Pieces may be created that overlap one another. The overlap depends on algorithmic kenel size.

the impact on performance is minimal. Also, such overlap remains thread-safe, since separate threads can read from the same memory location—only when writing to the output that overlap must be eliminated.

3.6 Object-Oriented Implementation

The structured data pipeline in the VTK visualization system contains several dozen filters. One of our concerns for implementing this architecture—which is much more complex than the typical visualization system—is that a complex implementation might result in a brittle or unmaintainable system. Fortunately, we were able to embed the complexity of the architecture into three superclasses, including an abstract superclass from which most all filters are derived. Subclasses (i.e., filters) remain relatively simple to create, with the usual care required with multi-threaded implementations.

The three superclasses encapsulating the stream architecture are:

1. vtkImageSource — the superclass for vtkImageFilter and all other process objects; synchronizes pipeline execution.
2. vtkImageFilter — the superclass for most structured filters. It performs the breaking of data into pieces and coordinates multithreading.
3. vtkImageCache — manages the interface between the data objects and the process objects.

Source code for the C++ implementation is available in VTK Version 2.2 (and later versions) from http://www.kitware.com/vtk.html.

4 Results & Discussion

To demonstrate the effectiveness of the streaming architecture, we evaluated it against the three different pipelines shown in Figure 3. Each pipeline was run on a volumetric dataset of size 256x256x93 by 2 bytes, for a total input data size of 12.19 MBytes. Once read into the pipeline, the data was further expanded by the addition of an vtkImageCast filter (transformed the 2-byte short data to 4-byte floats, i.e., doubled the data size) and/or was passed through a vtkImageMagnify filter to further increase data size by 3x3x1 (i.e., a nine-times increase in data size). In addition, the filters were configured to retain their output data as the pipeline executed, and some filters (such as vtkImageGradient) expand their data during execution (vtkImageGradient by a factor of three because a scalar is expanded to a 3-vector). The total data processed by Pipeline #1 is 780 MBytes; by Pipeline #2: 3.76 GBytes; and by Pipeline #3: 475.4 MBytes.

4.1 The Effect of Swapping

The first numerical experiment demonstrates the effect of swapping on performance. Applications that depend on virtual memory suffer severe penalties because the speed of virtual memory is significantly less than physical memory. By controlling the size of data pieces streaming through the pipeline, we can avoid swapping and insure that data is processed only in physical memory.

Pipeline #1 was used to perform the experiment on a small system running Windows/NT with 128 MBytes of physical memory. While the input data is only 12.19 MBytes, the vtkImageMagnifyFilter expanded the data size by a factor of nine, followed by vtkImageGradient, which expanded the data by another factor of three, for a maximum data size of 329 MBytes. The experiment varied the cache size from 10 KBytes to 330 MBytes. (At 330 MBytes we depend entirely on system virtual memory.) We also ran the same data on a two processor system to see the effect of multiprocessing. The results are shown below.

Cache Size (MByte)	Elapsed Time (1 Processor)	Elapsed Time (2 Processors)
330	3934	1740
100	565	327
70	128	59
50	96	58
25	98	50
10	105	55
5	110	59
2	111	60
1	113	61
0.10	134	89
0.03	194	173
0.01	485	553

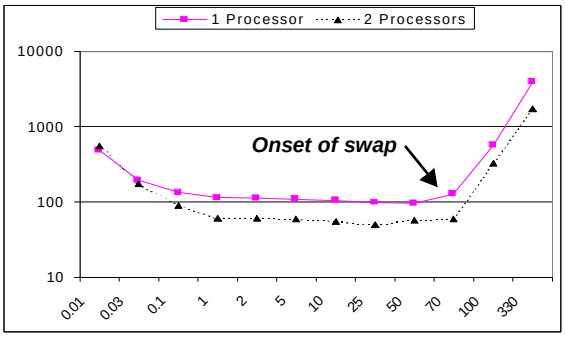

Figure 4. The effect of cache size on elapsed time for Pipeline #1.

The results, plotted in Figure 4, clearly demonstrates the effect of swapping. The best performance for a single processor system occurred when the cache size was set to 50 MBytes, which was 41 times faster than the results obtained when depending on virtual memory. Even when the cache size was set to a tiny 10 KBytes, we observed better performance than was obtained with virtual memory. There is a noticeable penalty as the cache becomes very small, since the overhead of breaking data into pieces

affects performance. The performance for two processors showed similar results, although the effect due to small cache size was greater because each piece of data is divided in half and assigned to each of the two processors.

Another benefit of the streaming architecture is that we can process data whose size is greater than the physical address space of the computer. For example, using the same 32-bit NT, dual-processor system, we were able to process a peak data size of 25 GByte (vtkImageMagnify increased data size by a factor of 676) with a cache size of 50 MByte in approximately 3000 seconds. (This is because each piece is smaller than physical address space and we never need to allocate contiguous memory for the entire data set.)

4.2 Multi-Threading and Cache Size

In the second numerical experiment, we compared the effect of varying cache size from 750 MByte down to 7.5 MByte for each of the three pipelines. In addition, the number of processors is varied between one and eight. The computer system is a large 8-processor Infinite Reality SGI (R10000) with 3.5 GByte of physical memory. Because of the large physical memory and the cache size limit, the system did not swap. Therefore, the difference in elapsed time were due to the effects of more processors (multi-threading) or the overhead of processing data in pieces.

Cache Size	1 Proc.	2	4	8
750 MByte	19.81	12.07	7.99	6.65
375 MByte	19.79	12.08	8.03	6.61
75 MByte	19.83	12.14	8.07	6.67
7.5 MByte	17.54	13.29	11.10	11.15

Figure 5. Elapsed time as number of processors and cache size is varied for Pipeline #1.

Cache Size	1 Proc.	2	4	8
750 MByte	130.6	70.6	41.23	23.51
375 MByte	133.0	70.0	40.95	24.75
75 MByte	129.8	72.2	41.87	28.60
7.5 MByte	138.0	82.2	67.98	64.60

Figure 6. Elapsed time as number of processors and cache size is varied for Pipeline #2.

Cache Size	1 Proc.	2	4	8
750 MByte	181.6	96.81	52.16	29.12
375 MByte	181.4	96.66	52.58	29.78
75 MByte	178.9	96.66	52.25	30.43
7.5 MByte	234.6	131.2	71.56	52.73

Figure 7. Elapsed time as number of processors and cache size is varied for Pipeline #3.

We noticed several interesting features in these results. First, with a single processor, the effect of cache size (breaking data into pieces) was small. In some cases the reduction in cache size actually reduced the elapsed execution time of execution, probably because the data better fit into machine cache. The major exception to this was Pipeline #3. This pipeline is different from the other two in that a branch exists in the pipeline. When generating results, the pipeline evaluates first one branch, and then the other. As the cache at the point of pipeline junction (i.e., vtkImageFFT) is reduced in size, less reusable data is cached. The net result is that for small cache sizes the two filters upstream of vtkImageFFT execute two times: first for the left branch and then for the right branch. (Performance could be improved by increasing the cache size at the point of branching.)

Another striking feature is the effect of reducing cache size combined with adding additional computational threads. Since with single processor systems we observed that the effect of streaming was relatively small, we surmise that the overhead of creating and joining threads becomes significant as the size of the cache becomes smaller. (Note: in the eight-processor case, each piece of data is processed by eight threads in each filter. Therefore, as the piece size becomes smaller, each thread works on less data, so the overhead of thread management becomes proportionally larger.)

5 Conclusions & Future Work

We have successfully designed and implemented an object-oriented architecture that can process structured data of arbitrary size on computers whose physical memory is much smaller than the data size. The architecture achieves this capability by breaking data into pieces as a function of a specified cache (or memory limit) size. In addition, the architecture supports multi-threading automatically without requiring reconfiguration of execution scripts. We found that the effect of streaming (breaking data into pieces) is small for uniprocessor systems, and that the cost of thread management becomes larger as the piece size is reduced. We also demonstrated the capability of systems to process data whose size is much greater than the size of physical memory.

Our ultimate goal is to incorporate the streaming architecture into VTK's unstructured visualization pipeline. This is a difficult task for several reasons. First, the data type changes as it passes through the pipeline; e.g., a structured grid when isosurfaced becomes a polygonal (triangle) mesh. Second, it is difficult to map the input to the output. For example, it is not known beforehand which cells, when isosurfaced, will generate surface primitives, and how many resulting primitives are generated. Third, it is difficult to break data into pieces since there are no natural boundaries in unstructured data. And finally, many algorithms are global in nature. Connectivity and streamlines require data in an unpredictable manner, or in its entirety,

in order to execute.

Incorporating the streaming architecture into the unstructured pipeline may require changes to algorithms and accepting certain compromises. For example, global algorithms such as streamline generation may be recast (use algorithms with local kernels such as LIC [Cabral93]), or decimation [Schroeder92a] may occur in patches with boundary seams visible (not results invariant). It may also be that the architecture is extended to support multipass streaming where filters retain information between each pass.

The computing environment of the future will consist of a heterogeneous mixture of single- and multi-processor computing systems arranged on a high-speed network. While the architecture described here supports a (local) multi-processor, shared memory approach, it is readily extensible to the distributed environment. In a distributed environment, data can be broken into pieces (using the same approach described here) and assigned to systems across the network. Future plans call for distributed support to be built directly into VTK.

6 Acknowledgment

This work was partially supported by the NSF Award #9872147. Thanks to Bill Lorensen for his insightful advice, to James Miller for assisting us in the numerical studies, and our colleagues at GE CRD and GE Medical Systems.

7 References

[Agarwal98] 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).

[AVS89] 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.

[Cabral93] B. Cabral and L. Leedom. Imaging Vector Fields Using Line Integral Convolution. *Computer Graphics (SIGGRAPH '93 Proceedings).* Vol. 27. pp. 240-247.

[Chiang95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia,and others. External-Memory Graph Algorithms. In *Proc. ACM-SIAM Symp. On Discrete Algorithms* pp. 139-149, 1995.

[Chiang97] Y.-J. Chiang and C. T. Silva. I/O Optimal Isosurface Extraction. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.

[Chiang98] Y.-J. Chiang and C. T. Silva. Interactive Out-of-Core Isosurface Extraction. In *Proc. Of Visualization '98*. IEEE Computer Society Press, October, 1998.

[Cox97a] 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.

[Cox97b] 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.

[DataExplorer] *Data Explorer Reference Manual.* IBM Corp, Armonk, NY, 1991.

[Funk95] 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 Fatabases*, Course Notes 32, Siggraph '95, August 1995.

[Itoh95] 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.

[Kenwright97] D. Kenwright and R. Haimes. Vortex Identification - Applications in Aerodynamics: A Case Study. In *Proc. Of Visualization '97*. IEEE Computer Society Press, October, 1997.

[Kenwright98] D. Kenwright. Automatic Detection of Open and Closed Separation and Attachment Lines. In *Proc. Of Visualization '98*. IEEE Computer Society Press, October, 1998.

[Kenwright98b] D. Kenwright. Presentation to Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Institute, 1998.

[Machiraju98] E. Machiraju, A. Gaddipati, R. Yagel. Detection and Enhancement of Scale Coherent Structures Using Wavelet Transform Products. *Proc. of the Tech. Conf. on Wavelets in Image and Signal Processing* SPIE Annual Meeting, San Diego CA, 1997.

[Machiraju93] .R. Machiraju and R. Yagel, Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors. *SUPERCOMPUTING'93*, Portland, Oregon, November 1993, pp. 699-708.

[Rama94] S. Ramaswamy and S. Subramanian. Path Cathcing: A Technique forOptimal External Searching. In *Proc. ACM Symp. On Principles of Database Sys*., pp. 25-35, 1994.

[Schroeder92a] W.J. Schroeder, J. Zarge, and W.E. Lorensen. Decimation of Triangle Meshes. *Computer Graphics (SIGGRAPH `92)*, 26(2):65-70, August 1992.

[Schroeder97] W.J. Schroeder, K.M. Martin, and W.E.Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[Sub95] 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.

[Teller94] 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.

[Ueng98] 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.*

[Vengroff96] 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.