Parallel Dataflow Scheme for Streaming (Un)Structured Data

Huy T. Vo, Daniel K. Osmari, Brian Summa, João L.D. Comba, Valerio Pascucci and Cláudio T. Silva

UUSCI-2009-004

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

May 27, 2009

Abstract:

We propose new techniques for exploiting multi-core architectures in the context of visualization dataflow systems. Recent hardware advancements have greatly increased the level of parallelism available in these architectures, and it is expected that this trend is likely to continue in the future. Existing dataflow systems have a number of limitations: they are written for a standard CPU programming; they mostly support a limited type of multi-threading execution; and while some systems (e.g., ParaView, VisIt) include streaming and tiling methods for parallelism, due to the monolithic data and computation model, they are not optimally suited for the higher levels of parallelism available in highly parallel architectures of the future. Ideally, visualization systems should be built on top of a parallel dataflow scheme that is able to better utilize CPUs and assign resources adaptively to pipeline elements. Another important goal is to natively supporting streaming data structures. We propose the design of a flexible dataflow architecture aimed at addressing many of the shortcomings of existing systems, including a unified execution model of both demand-driven and event-driven; a resource scheduler, that can automatically make decisions on how to allocate computing resources; support for more general streaming data structures, including unstructured elements. We have implemented our system on top of VTK, and we provide backward compatibility and in this paper, we provide experimental evidence of performance improvements on a number of application.

Parallel Dataflow Scheme for Streaming (Un)Structured Data

Huy T. Vo, Daniel K. Osmari, Brian Summa, João L.D. Comba, Valerio Pascucci and Cláudio T. Silva

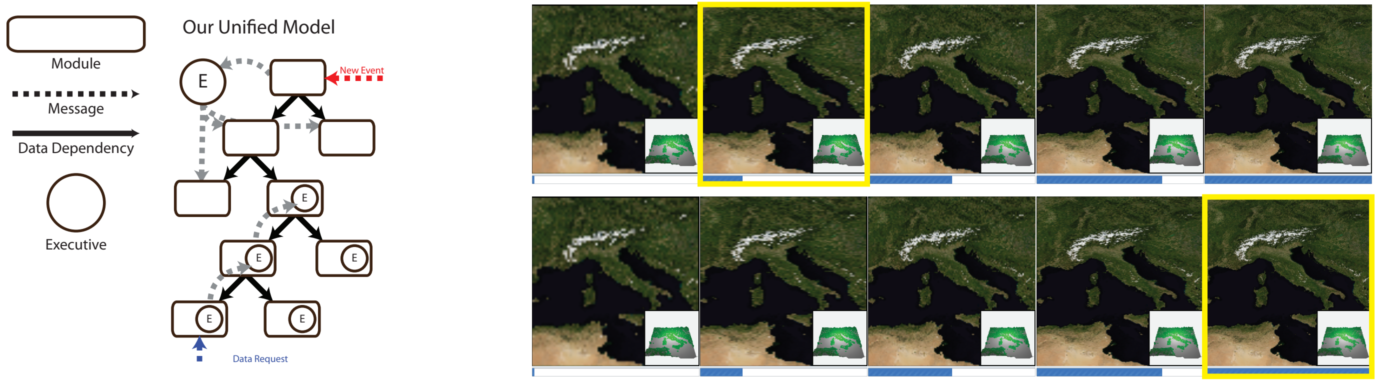

Fig. 1. Left: Our new unified data-flow model with a new control strategy. Right:Comparison of a progressive rendering pipeline executed in VTK using the default execution model (top) and ours (bottom). The progressive renderer increases the level of detail from left to right. We have achieved approximately 3 times speed-up on a 4-core machine. The same pipeline running in our system were able to finish the rendering at the finest level 5 while the VTK one only reaches the level 2.

Abstract—
We propose new techniques for exploiting multi-core architectures in the context of visualization dataflow systems. Recent hardware advancements have greatly increased the level of parallelism available in these architectures, and it is expected that this trend is likely to continue in the future. Existing dataflow systems have a number of limitations: they are written for a standard CPU programming; they mostly support a limited type of multi-threading execution; and while some systems (e.g., ParaView, VisIt) include streaming and tiling methods for parallelism, due to the monolithic data and computation model, they are not optimally suited for the higher levels of parallelism available in highly parallel architectures of the future. Ideally, visualization systems should be built on top of a parallel dataflow scheme that is able to better utilize CPUs and assign resources adaptively to pipeline elements. Another important goal is to natively supporting streaming data structures. We propose the design of a flexible dataflow architecture aimed at addressing many of the shortcomings of existing systems, including a unified execution model of both demand-driven and event-driven; a resource scheduler, that can automatically make decisions on how to allocate computing resources; support for more general streaming data structures, including unstructured elements. We have implemented our system on top of VTK, and we provide backward compatibility and in this paper, we provide experimental evidence of performance improvements on a number of application.

Index Terms—Dataflow architectures, Streaming data structures, Parallel processing.

1 Introduction

The dataflow model is widely-used in visualizations systems, including AVS [26], SCIRun [19], and VTK-based systems such as Paraview [14], VisIt [8] and VisTrails [5]. Existing dataflow architectures have been designed primarily to work on either a single CPU or a small collection of CPUs, such as a small SMP workstation. Recent hardware advancements have greatly increased the level of parallelism available in these architectures, and it is expected that this trend is likely to continue in the future. Equally important is that to a large extent, existing architectures assumes that all the data structures are maintained in memory, and that potentially multiple copies of the data might be stored as it is processed by the different dataflow modules. Most, if not all designs, have completely ignored streaming data structures, which sometimes are supported for simple data types (e.g., regular 2-D images or regular 3-D volumes). The support for unstructured and hierarchical data structures is either non-existent or fairly naive.

We propose the design of a flexible dataflow scheme aimed at addressing many of the shortcomings of existing systems. Our system supports a unified execution model of both demand-driven and event-driven that allows each module in our pipeline to *pull* as well as *push* data. It also includes a resource scheduler, that can automatically make decisions on how to allocate computing resources (e.g, the number of threads to use for a particular module). Another novel feature is the support for more general streaming data structures, including support for unstructured elements (e.g., tetrahedral elements), and the fact that we use streaming not only as a way to decreasing memory overhead, but also as a way to obtain extra performance. We have implemented our system on top of VTK, and we provide backward compatibility. This facilitates integration into the many systems that support VTK.

Contributions of our flexible parallel dataflow framework:

- A new scheme for executing pipelines that easily exploits multi-core hardware.
- A unified data-flow model that easily integrates pull (demand-driven) and push (event-driven) policies into a single pipeline.
- A data-flow control strategy that combines the benefits of both the distributed and centralized execution controls.
- A system that includes new support for streaming structured and unstructured data.
- An new adaptive scheduling strategy for dynamic load balancing.

- *Huy T. Vo, Brian Summa, Valerio Pascucci and Cláudio T. Silva are with the Scientific Computing and Imaging Institute, University of Utah. USA, E-mail: [hvo,bsumma,pascucci,csilva]@sci.utah.edu*
- *Daniel K. Osmari and João L.D. Comba are with the Instituto de Informática, UFRGS, Brazil, E-mail: [dkosmari,comba]@inf.ufrgs.br*

- A complete multi-core implementation and seamless integration into a widely-used visualization system.

2 Related Work

Parallel rendering on a variety of architectures has been the focus of a large body of work [4, 17, 22, 25]. Even a cursory review is beyond the scope of this paper, therefore we point the reader to [7] for a complete introductory survey. For this paper, we focus our discussion on visualization dataflow systems. Following the pioneering work of Haber and McNabb [10], many leading visualization systems (e.g., [1,5,8,15,19,26]) have been based on the notion of a dataflow network, where nodes (often called *modules*) are processing elements and arcs represent data dependencies. Several different methods have been developed for implementing this framework; for instance, a typical technique is to implement the system as a demand-driven workflow where modules respond to *update requests*. Following a request modules perform their computations and pass the processed data downstream until it reaches the display modules [9].

This *component framework* methodology for developing visualization libraries encourages code reuse, while enabling the development of extremely flexible and powerful visualization systems. As an example, consider the de-facto standard visualization API, Kitware's Visualization ToolKit (VTK) [15]. VTK is a highly successful visualization system used by thousands of researchers and developers around the world. VTK was originally developed as teaching software as part of collaboration among three GE researchers, but it quickly grew in functionality and complexity. The underlying architecture has undergone substantial modifications over the years to allow for the handling of larger and more complex datasets (e.g., time-varying, AMR, unstructured, high-order).

VTK's execution model has a number of limitations with respect to parallelism. First, it only supports concurrency execution at a module level, which means that no matter how many threads an algorithm is implemented to run in, the whole network has to be updated serially (that is, one module at a time). Moreover, by default, only a small subset of VTK, such as those inherited from *vtkThreadedImageAlgorithm*, can run multi-threaded. This poses a limitation on the performance and scalability for many applications. While much effort [6, 16] has been put into extending VTK's execution pipeline over the years, it is still challenging and problematic to build highly-parallel pipelines using the existing pipeline infrastructure. Part of the reason is that VTK makes use of a demand-driven execution model while some pipelines, in particular those that need streaming and/or time-dependent computations fit more naturally in an event-driven architecture.

ParaView [14] is a leading parallel rendering tool built on top of VTK. It is designed for data-parallelism, where pipeline elements can be instantiated more than once and executed in parallel with independent pieces of the data. Specifically, a typical parallel execution in ParaView involves a pipeline instantiating processes based on the data input, then, relies on MPI to distribute them on cluster nodes to finish the computation. However, ParaView does not support a hybrid MPI/multi-threading model; MPI is also used for creating multiple processes on the same node with multi-core architecture. This may impose substantial overhead due to the additional expense of interprocess communication over thread messages. On the other hand, task scheduling is performed only once; thus, it is not possible to parallelize a pipeline with more than one sub-network (but not the entire network) that is suitable for concurrent computation.

Ahrens et al. [2,3] proposed parallel visualization techniques for large datasets. Their work included various forms of parallelism including task-parallelism (concurrent execution of independent branches in a network), pipeline-parallelism (concurrent execution of dependent modules but with different data) and data parallelism. Their goals included the support for streaming computations and support for time-varying datasets. Their pioneering work led to many improvements to the VTK pipeline execution model and serve as the basis for ParaView. A related system is VisIt [8];, introduced the notion of *contracts* between modules, which are data structures that allow modules to negotiate with other modules how to transfer data. This method has proven to be very useful for optimizing many operations on the dataflow network. Recent versions of the VTK pipeline incorporate many ideas that were originally developed for VisIt and ParaView.

SCIRun [19] is another parallel dataflow system targeting scientific environment. Unlike ParaView, the original design targeted multicomputers with shared memory. SCIRun supports threaded execution of its network by laying down the network in execution order and running independent modules in parallel if possible. However, this system has no resource management. Each module is assumed to run on a fixed number of threads, usually just one. SCIRun is also able to run on a distributed environment with data-parallelism but pipeline-parallelism is not supported. Other major visualization systems, such as IBM Data Explorer [1] and AVS [26], operate in similar fashion as SCIRun. A common design in the systems , such as those listed as above, is separation of the execution strategy of a module from its algorithm and the data object that it is operating on. This execution strategy is often called the *executive* and is responsible for calling an algorithm to execute on a certain set of input. There are two models for assigning executives in a dataflow system. While centralized executives manages the execution strategy for a set of module or a pipeline, local executives only operate on a single module.

A major difference between the later systems and VTK (and ParaView) is the use of centralized (or explicit) executives instead of local (or implicit) executives. While it is easier to perform pipeline network analysis and distributed scheduling with centralized executives, it requires each module to connect to its centralized executive, which can sometimes be sitting on a different machine. In contrast, although local executives allow modules to execute autonomous from each other, it is difficult for a pipeline using local executives to employ an event-driven execution model or a more complicated scheduling strategy. Thus, a system that supports both centralized and local executives as well as demand and event driven execution model, as our system, could avoid much of the mentioned problems while giving more flexibility to developers.

The last ten years saw the development of a large number of streaming and out-of-core visualization algorithms (see survey by Silva et al [24]). These include a number of cache-oblivious techniques [29] that have been shown to provide a memory-system agnostic way to obtain efficiency throughout complex memory hierarchies. One of the key developments in this area is the introduction of streaming meshes by Isenburg and Lindstrom [12]. Streaming and cache-oblivious algorithms have been getting more mature and can be used in many areas of visualization because of their ability to work with data that is too large to fit in main memory. There are techniques for handling surfaces, volumetric grids, tetrahedral meshes and reeb graphs [13,21,28].

Up to now, existing dataflow systems have not exploited these types of data structures and algorithms. In particular, VTK only supports streaming structured grids (i.e., regular data). This is partly due to the complications that handling such data structures introduce on the demand-driven execution model. The system requires the addition of "many new requests" in order to support streaming; this makes it problematic, time-consuming, and over complicated for developers to implement streaming algorithms. They are much more easily implemented in a event-driven model.

Another advantage of using event-driven streaming execution is the lessening on the *mappable* property of streamable data structures [16], where we must be able to map any portion of the output data to the pieces of the input. This is often not feasible in many applications where the output is only known at the end of the stream. In our system, any module that is part of an event-driven network does not have to provide data mapping capability.

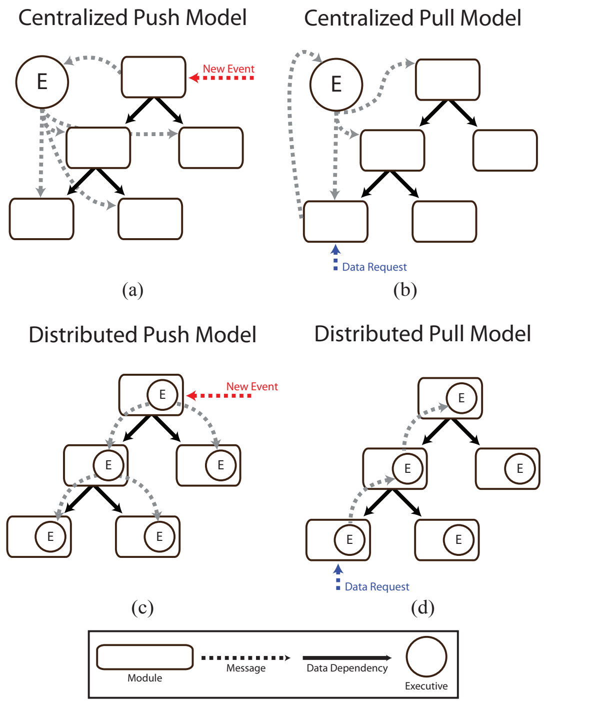

Fig. 2. Classes of pipeline models (a) a centralized push model handling a new event acting on the highest upstream module in the pipeline (b) a centralized pull model handling a data request on the bottom-left module in the pipeline (c) a distributed push model handling a data request at the highest upstream module in the pipeline (d) a distributed pull model handling a data request on the bottom-left module in the pipeline

3 Background

In this section, we will discuss some background material necessary for understanding our method and its contributions.

As we have defined previously, a pipeline consists of a series of modules executing on a given set of data. The input data to these modules may or may not be independent of other modules in the pipeline. Therefore, when a module executes, a level of coordination is required between the modules and their data dependencies. In the simplest form, this can be implemented in the algorithm at the module level. However, as dataflow systems become more complex, this coordination is typically assigned to a separate component called the *executive*. The executive is responsible for all coordination, instructing the module when and on which data it should operate. To handle this coordination, the component holds pipeline information, such as upstream and downstream modules, and processes requests to the module from the pipeline. In this approach, algorithms can be implemented based purely on the computational task required, without consideration for the topology of the pipeline.

In existing systems, executives are usually deployed as either centralized or distributed. A centralized executive operates globally and is connected to every module in the pipeline. Therefore, there is one central process tracking all changes to the network and handling all requests and queries to the pipeline. This approach gives the system the advantage of having control over the entire execution network, thereby making it easily distributed across multiple machines. However, this centralized control leads to high overhead in the coordination and reduces the scalability of such systems, especially when operating on complex pipelines.

Unlike a centralized executive, a distributed executive is a lightweight and simple approach. In this scheme, each module contains its own executive, which is only connected to its immediate upstream and downstream neighbors. This approach does not suffer the same scalability issues of the centralized scheme. However, this myopic view of

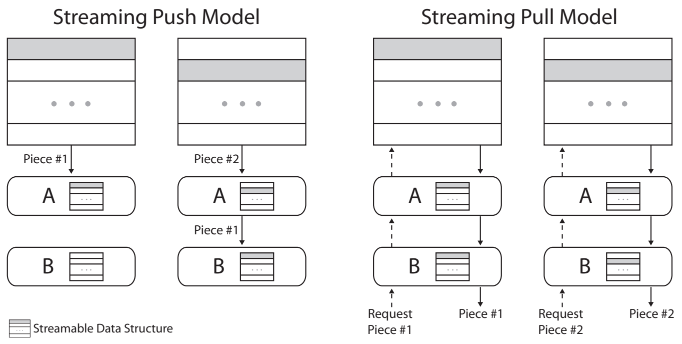

Fig. 3. Two different examples of a streaming pipeline. The two left diagrams show a streaming push model over two steps on a streamable data structure. First, the data denoted in dark gray is sent to module A. After module A has completed execution the stream data is passed downstream to module B. Module A then receives a new section of stream data. On the right two diagrams show a streaming push model over two steps on the same streamable data structure. When B receives a request for the section of stream data denoted in dark gray, the request is sent upstream to the source. This section is then sent and processed for each module downstream to the request. The next section requested follows similarly.

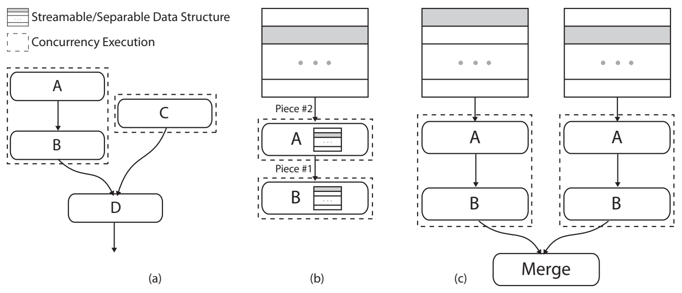

Fig. 4. Different types of parallelism above module level (a) task-parallelism (b) pipeline-parallelism and (c) data-parallelism

the network makes distributing resources or coping with more complicated executing strategies a much more difficult problem.

There are two execution models currently employed in existing dataflow systems: *push* and *pull* pipelines. In [23], these are named *event-driven* and *demand-driven* respectively. Each model has its own advantages and disadvantages. In a pull pipeline, a module is executed only when one of its outputs is requested. See Figure 2 (b) and (d). If a module's input is not up-to-date, it will *demand* or *pull* the corresponding upstream module for output. Therefore only modules that are needed for computing the data request will be executed, thus avoiding wasteful calculations. However, since a request is dependent on all upstream modules to be updated before starting computation, the module that initiates the request will be locked longer than its computing time. In an *push* or *event-driven* pipeline, modules are updated as soon as an *event* occurs, e.g. a change in its input. See Figure 2 (a) and (c). All modules that are dependent on the requested module's output are also computed. This results in a shorter locking period, which is equivalent to the computation time for each module. Nevertheless, this approach has a key disadvantage of redundancy, generating data and computations even when they will not be used.

In a system that does not support any level of parallelism, modules are only allowed to be updated sequentially starting from the sources. For example, if computed in this way the pipeline of Figure 4(a) would run in the order of ABCD. As you can see from this simple example, there would not be any performance gain when executing such a pipeline sequentially on a multi-core machine.

With the proliferation of multi-core machines, parallelism in dataflow systems has become increasingly desirable. Therefore it has been the focus of a large body of research including the work of Ahrens et al. [2] in visualization. Given this hardware, computing a pipeline without parallelism, i.e. sequentially, is not the optimum scheme for maximizing performance unless parallelism can be exploited at the module level. We call this case *module-parallelism*. For instance, VTK currently follows this model where each pipeline is computed sequentially but each module has the ability to use all available cores. As an example, consider the pipeline outlined in Figure 4(a) computed on a machine with multiple cores with a scheme that exploits module-parallelism. Even though the pipeline is executed sequentially (in the order ABCD), each module will run on all available cores during their execution. This increases the utilization of computation resources by running all resources on a single module (which typically only contains small computations). Though this has the unfortunate effect of increased overhead due to memory access collisions.

A second form of parallelism is *task-parallelism*. This scheme allows independent branches of the networks to run in parallel each using their own thread. See Figure 4(a). In this example, modules A and B are independant of module C, therefore A and B are allowed to be run concurrently with C. Branches that share common upstream modules are not allowed to run in parallel because a typically modules are not reentrant. Updating a common upstream module simultaneously from two different threads may cause program failure.

In a system that supports *pipeline-parallelism*, pipelines are partitioned into sub-networks operating on different tasks. During execution, each network would be allocated their own threads and each is computed concurrently on streams of the data. However, pipeline-parallelism is only applicable to separable or streamable data structures. In addition, partitioning of the pipeline into sub-networks must be proportional to the computing resources (e.g. the number of cores) in order to maximize the performance. This is not scalable since this division is typically done by hand. Figure 4(b), is an example of streaming pipeline parallelism between two modules.

The last common form of parallelism is *data-parallelism*. In this scheme, the data is divided into independant pieces to be processed in parallel. At the end of execution, data is collected and merged into a single result. This is an extremely useful method for large datasets that must be processed out-of-core and and is highly scalable to clusters of machines. However, this performance gain in maintained only if there is one merge operation in the end. Otherwise, data must be redistributed multiple times causing a large degradation of performance. Fortunately, a single merge is the usual case for rendering pipelines. Figure 4(c) illustrates the use of data-parallelism, where the pipeline AB is duplicated multiple times to work on the separate pieces of the streaming data. ParaView is a common system that follows this model of parallelism.

As outlined above, streaming algorithms can have a large impact on parallel computation schemes. Theoretically, streaming algorithms can be implemented in dataflow systems using either push or pull models. However, due to its nature of constantly supplying data, a push model is preferred. Figure 3 illustrates this simple approach. A typical streaming pipeline would consist of a streamable data structure acting as a source constantly supplying data downstream to processing modules. If the system supports pipeline-parallelism, modules can be executed concurrently with different pieces of data as well.

For pull models, streaming can also be applied, however, at the cost of pipeline-parallelism. As shown in the right diagram of Figure 3, instead of flowing data down the pipeline constantly, the streaming source must wait for requests from modules downstream. Since each module often can only process one request at a time (not re-entrant), the whole pipeline must be blocked for the sink module, B.

Besides streaming architectures, streamable data structures have also been a major hurdle for efficiently porting streaming algorithms into dataflow system. Currently, only structured data is supported. Data is usually broken into sections with a ghosted region (overlapping area between sections) and sent down the pipeline. Only VTK-based systems support streamable data structures at this time. In order to work properly under a pull model, a strict set of constraints on the data is required. The data must be *separable* (can be broken into sections), *mappable* (a portion of the input can be computed from the output), and *result invariant* (the result must be invariant to the number of pieces and threads). However, many streaming algorithms in practice cannot guarantee these constants. For example, some streaming algorithms may not be mappable without performing the actual computation. Though, the mappable constraint may be relaxed if a system uses a push model.

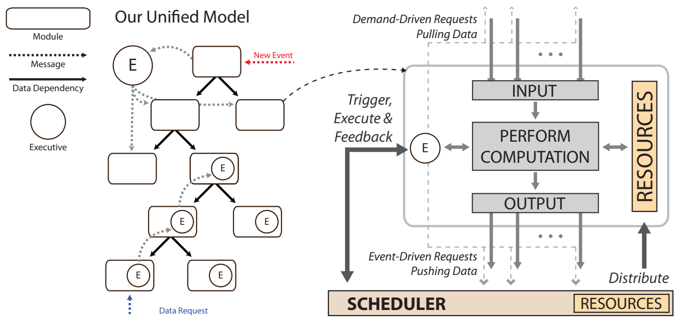

Fig. 5. An overview of our system architecture

4 Parallel Dataflow Architecture

In this section, we discuss the design of a flexible dataflow architecture that can run efficiently on multi-core machines. A diagram outlining an overview of our system is shown in Figure 5.

4.1 Execution Model

Similar to other models, coordination of modules is handled through the use of executives. However, we borrow from the best of all previous models to construct in a new unified model that is efficient, flexible and highly scalable. Rather than being constrained by just one, the executives in our system are allowed to follow both push and pull strategies. Our executives also support both centralized and distributed executive strategies.

Modules have the ability to make a `PULL` request to ask for one or more of its upstream modules to be brought up to date up-to-date. Likewise, a module can also push data downstream using the command `PUSH`. This simply transfers modified module data downstream. The module will lock until its downstream modules are finished either copying the data or taking control of the ownership of the data.

This multi-request unified strategy can have obvious positive consequences for some very common pipelines. For example, the ability to handle both pull and push requests is useful in visualization in the implementation of a view-dependant progressive viewer. Whenever a viewport is changed, the viewer could request its upstream renderer to produce new images by issuing the `PULL` command providing interactive feedback of coarse images. However, when the user stops interacting, the data stream source can `PUSH` data from finer levels upstream to the viewer providing the viewer with the full-resolution data. See Section 5 for an implementation of such an system.

The unified implementation also supports both centralized and distributed executives. By default, each module has its own executive following a typical distributed model. However, our system allows users the ability to mark any sub-network in the pipeline to be controlled by a centralized executive. This centralized executive treats the sub-network as a compound module. This unified executive model allows our system to retain the lightweight distributed scheme, but also adds benefits of global scheme in a highly scalable way. Figure 5 shows an example of our unified execution model both making a push and pull request on a pipeline using a mixed centralized and distributed model.

4.2 Scheduler

To achieve this flexibility in the executive and to maximize performance, a level of control must be delegated to another lightweight process in our model. Since our model uses executives operating locally and centralized, resource management is now handled by an external resource *scheduler*. While the responsibility of the executive is still to ensure the order of execution of the whole pipeline, the scheduler will control when they are executed and with how many resources are allocated to the algorithm. Thus, the scheduler can be thought of as a queue of executives waiting to run whenever resources are available. To handle this resource management, our localized executive strategy is extended to contain some internal knowledge of the resources it has been allocated. A module stores both the requested amount of resources and the allocated amount. For performance, these may not necessary be the same. A module, with this knowledge, complies with the allocated amount. This allows our system to maximize performance globally. Below are the steps outlining how a centralized scheduler interacts with the rest of the pipeline:

1. If triggered by a module asking for its output or to be updated (through its executive)
2. Adds the executive the scheduler queue
3. Computes the layout of the network for computation
4. Estimates and distributes resources using a scheduling *strategy*
5. Executes queued request(s), each module when completed will release resources and return statistics
6. Reapplies the scheduling strategy for every successful request from the orignal module
7. Becomes inactive when there are no more requests

Scheduling strategy Our scheme is flexible enough to handle any scheduling strategy. For our testing, we have implemented a heurisics strategy based on time statistics. At the time of rescheduling, our scheduler transverses the whole pipeline starting at the sink modules and distributes resources among the branches. Since module can only be executed if its inputs are up-to-date, the scheduler minimizes the difference in input computation time for each module. At run-time, modules are scheduled and allocated with resources proportional to the accumulated computation time from its source modules in the pipeline. If a module has more than one source, the scheduler distributes resources proportionally to the arrival time of the previous request. For example, in Figure 3 if the data flowing from module A to D in required twice the computation time as the data flowing from module C to D, then the branch containing module A and B will receive twice the resources as the branch containing C. In a single branch, sub-pipelines that can be executed concurrently with resources distributed evenly. The scheduling can be summarized as:

```
Module.Time: the accumulated time from a source
function ScheduleResource(Module A, Resources Total):
    UpstreamModules = FindUpstreamModulesFrom(A);
    if UpstreamModules is empty:
        A.AssignResource(Total)
        return

    TotalLastUpdateTime = 0
    for module in UpstreamModules:
        TotalLastUpdateTime += module.Time

    for module in UpstreamModules:
        ScheduleResource(module, Resource *
                         module.Time /
                         TotalLastUpdateTime)
```

The above scheduler can address both task-parallelism and pipeline-parallelism. Data-parallelism can be added by manually duplicating pipeline elements.

4.3 Data Connection

Data duplication is avoided whenever possible since copying and allocating memory could substantially degrade the whole pipeline performance. This is the case especially in a shared-memory systems, where multiple cores have the ability to access memory simultaneously. As a result, modules are encouraged to pass pointers downstream and perform processes in-place. To handle this functionality, every output data object in our system contains a flag indicating if it can be changed in-place by downstream module. This does not account for the fact that there can be more than one downstream module using the data, therefore our system will duplicate and create additional instances of the same object as needed.

If modules are modifying data in-place, the downstream modules are allowed to execute as soon as the data is ready. This allows a lock-free execution strategy. However, locking can be implemented by simply allowing a module to retain ownership of the data until it is finished processing. If all modules in a pipeline are internally locking their data, the scheduler gracefully handles this situation by alternatively executes modules at different levels in the pipeline.

4.4 Streaming Computation

As discussed previously, the primary reason streaming visualization algorithms have not populated dataflow systems in the past is the due to a lack of a general streaming framework. As a contribution to this paper, we have introduced this missing element to our system and addressed the two primary concerns: streaming execution and streamable data structures.

Since a streaming algorithm naturally fits into the methodology of a push model where data is contiguously flowing downstream, we have built two abstract modules into our system to work in tandem; *StreamingGenerator* and *StreamingMerger*. These modules denote the start and end respectively of a sub-network that will execute in a streaming, push fashion. In the sub-network the stream generator is the data source, responsible for pushing data downstream to be processed. The stream merger marks the end of the streaming sub-network and merges the streaming into one final output. Other modules outside the streaming subnetwork can use the merger module as an interface for pull requests. In this case, the merger module will send a request upstream to its generators creating a streaming, push data access. Therefore, we exploit the advantages of a streaming, push model while maintaining the functionality of a pull request. These new sub-networks interact seamlessly with the rest of our unified model.

Our system extends streamable data structures to beyond the standard structured grids by generalizing the streaming mesh format. The streaming mesh format is originally designed for triangular meshes by interleaving its geometry with connectivity. This introduces a notion of finalized and unfinalized vertices. A vertex is finalized if it will not be used by any other element later in the stream, thus, it is safe to remove it from the buffer. Our generalized streamable data structure is considered as a single stream that can be segmented with overlapping regions. The dimension of overlapping regions are defined by the finalization of the stream elements itself, i.e. unfinalized elements cannot be processed and will remain in the buffer. However, we have also extended the definition of finalization. Instead of allowing the data to decide which elements are finalized, the algorithm is also allowed to flag elements as unfinalized, e.g. an image filter may set a neighborhood outside the processing image as unfinalized.

Streaming techniques are often used in other dataflow systems to reduce the memory footprint and increase cache coherency. They are

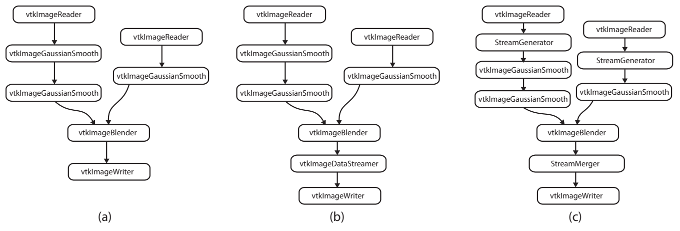

Fig. 6. Simple Gaussian smooth pipeline (a) VTK default image pipeline (b) VTK streaming image pipeline (c) a pipeline using our scheme

rarely used to increase performance of streaming computation by concurrent execution. In our scheme, we allow the scheduler to operate on the push sub-network thereby maximizing pipeline-parallelism.

4.5 Framework Implementation

We implement our framework on top of VTK to inherit a robust software infrastructure with existing algorithm implementations for testing and comparison purposes. We replace VTK's default streaming demand-driven pipeline executive with our unified executive. The executive interacts with the scheduler and can be centralized to support a push model. In fact, our new executive class is inherited from `vtkStreamingDemandDrivenPipeline`, thus, should be backward compatible with all VTK pipelines with the added advantage of the multi-threaded scheduler.

In this new executive, we also implement a light interface to allow subclasses of `vtkAlgorithm` to quickly interact with the unified model. In particular, new functions that have been added to this model were: `PULL(port, connection)` that is a blocking call to acquire data from a module connection and `PUSH(port)` that will take data existing at a module port and pass it to all consumers. Since VTK uses distributed executives with demand-driven pipelines, it was very difficult to retrieve global information from our unified executive. We had to build our own request tracing system, as well as execution network manager, on top of this class for directing execution.

In order to support the in-place data manipulation, we have added a new class `vtkFlexibleDataObject` which can smartly handle pointers with reference count for copy-on-demand and locking mechanisms for communication between modules. One of the problems we face with VTK is that it is not thread-safe. This has the unfortunate problem of causing the request passing system to crash if intense care were not taken in implementation. As a result, we had to build our own locking mechanism and use `CallAlgorithm` instead of sending requests.

5 Applications

To demonstrate the performance and flexibility of our method, we have implemented our framework as a new executive execution strategy in VTK [15]. Specifically, we have tested examples of our framework processing structured imaging data and unstructured tetrahedral meshes.

VTK has been the subject of a large body of streaming research and therefore is a apt system for both implementation and comparison to our framework. We have selected imaging as the primary focus for testing since VTK only fully supports multi-threaded processing and streaming in its imaging framework, and therefore would be a proper basis for comparison. While our initial implementation and testing uses VTK, we are by no means limited to its use only.

All tests are performed on a Dual Processor Dual Core Opteron 275 with 8GB of RAM, unless otherwise stated.

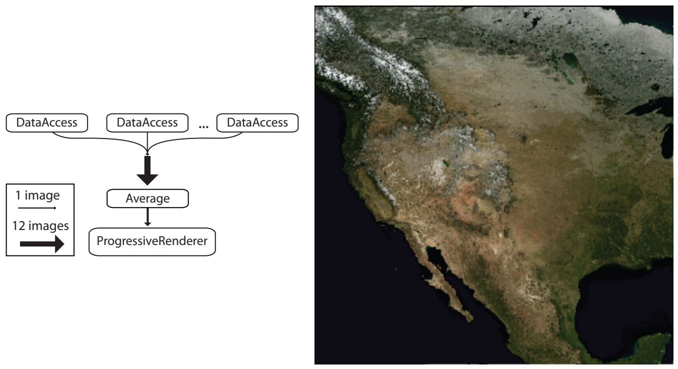

Fig. 7. A pipeline to compute the average of 12 image sources. When a viewing window request is sent, an average of each of the 12 source images is compute for each color channel. After the average is taken, the pixel values are sent downstream to a progressive renderer.

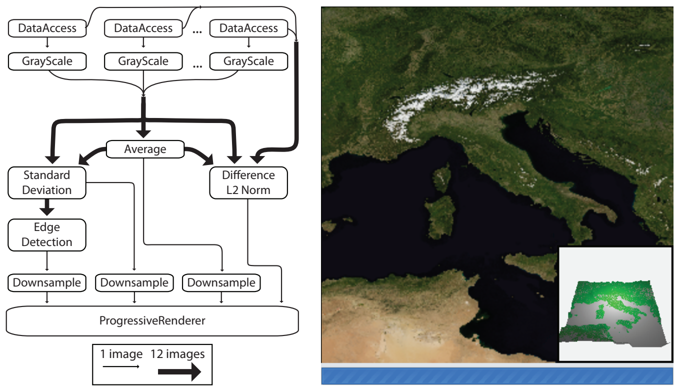

Fig. 8. An asymmetric, deep pipeline to compute several image processing functions. When a viewing window request is sent, each of the 12 source images for the given viewport is converted to its grayscale equivalent. The average is then taken of these values. This average is sent downstream to a module to compute the standard deviation and a module to determine the "closest" source image (L2 norm of the difference from the average. This "closest" is sent to the progressive renderer. The standard deviation is sent to an edge detection algorithm. The edge map, the standard deviation map, and the average are sent to the progressive renderer. The average is rendered as a height field on a quad mesh, while the standard deviation and edge map are textures on the mesh. All three are down-sampled before being rendered.

5.1 Multi-core Image Processing

The VTK image processing pipeline is capable of multithreaded processing, but only at the module level. We compare this existing functionality to the full pipeline parallelism of our framework. Using VTK's default multithreaded pipeline enables the system to maximize performance by utilizing all available cores on a per module basis. For the processing of massive imagery, this performance gain is outweighed by the necessary high memory footprint and poor data locality in each thread. For such images, VTK provides the ability to perform out-of-core streaming of image data (using the vtkImageDataStreamer class), which alleviates the problems outlined above for the standard system. Specifically, it requires a smaller memory footprint and retains high cache coherency. This functionality is unfortunately demand-driven, which can block pipeline parallelism causing poor performance for large numbers of threads on small portions of data. Our our system pipeline does not suffer from the problems inherent in VTK's default or streaming pipeline and can exploit parallelism, low memory requirements, and high locality.

To test the performance of the system on imaging pipelines, we have constructed three simple, yet computationally expensive, examples.

50MP Image Inputs			
Threads	VTK Default	VTK Streaming	Our Streaming
1	103.8s	21.6s	21.9s
2	11.6s	12.8s	9.7s
4	8.6s	9.2s	7.4s
8	8.4s	9.0s	6.2s
200MP Image Inputs			
Threads	VTK Default	VTK Streaming	Our Streaming
1	468.4s	280.7s	281.1s
2	171.2s	58.6s	46.3s
4	100.3s	37.5s	31.4s
8	45.5s	37.1s	25.5s

Table 1. Comparison of image processing between VTK and our system for the simple gaussian pipeline. See Figure 6 for diagrams of each pipeline.

Gaussian Smooth Pipeline VTK's gaussian smooth module can be configured to maximize performance by using all available cores. For testing, we use two synthetic images of 50 megapixels and 200 megapixels in size. Two copies of each image are loaded and processed in a simple pipeline, where one image is processed by the Gaussian module once and the other copy twice. See Figure **??** for a diagram of the pipleine.. As we show in the companion video and in Table 1, our framework offers significant improvement over VTK's default and streaming frameworks with `vtkImageDataStreamer`. Both pipelines have been tested independantly on an 8-core Opteron 3.0GHz machine with 16GB of RAM. The number of threads for VTK's image algorithm is controlled in our test program by using `vtkMultiThreader::SetGlobalNumberOfThreads`. Since these imaging classes are derived from `vtkThreadedImageAlgorithm`, they are designed to run threaded. Therefore, when setting the module to run on a single thread, some degradation in performance may occur.

12 Month Average Pipeline For this example, we show the per-pixel average of 12 months of satellite imagery (1 image per month) from NASA's Blue Marble Project [18]. Each image from this data set is 3.7-gigapixel, therefore we must employ out-of-core data access. For this implementation, we have chosen the hierarchical z-order scheme as outlined in [20]. In practice, we have found this method well-suited for our access needs since it inherently provides a hierarchical structure and exhibits good data locality in both dimensions. This allows our system to have fast data access and intelligent partitioning of the image for processing.

We demonstrate the performance of our system versus VTK's streaming system by processing the per-pixel average of the 12 months of data. See Figure [**?**] for a diagram of the pipeline. This average is view dependent, therefore the system only needs to process the pixels visible on the screen. Each module operates on a hierarchical resolution from our data access and data is displayed progressively as it is available. Even with this simple operation and reduction to visible pixels, our fully parallel system significantly outperforms VTK's current framework achieving a more than 3 times speedup. See and the companion video for these results.

Multi-visualization Pipeline For this example, we have deepened the 12 month average pipeline to incorporate more image processing modules, increase the data dependency between them, and increase the asymmetry of the pipeline. Like the previous example, we are accessing 12 images, one for each month from NASA's Blue Marble Project [18]. Also, we have employed the same data access scheme from the previous example and all operations are purely view dependent on a per-pixel basis.

The first stage of the pipeline involves the conversion of our data sources from 8-bit RGB to their grayscale floating-point representation. After the image is converted, a per-pixel average is computed for all images. This average is streamed to a module that computes the standard deviation. The average is also streamed to another module, which computes the image that is closest in terms of the L2 norm of the difference between the original data and the average. This will give the user the best representative month for the given viewing window. The standard deviation is fed to an edge detection module. This will give the user the areas of greatest change in deviation from the average. Finally, the standard deviation, the edges of the standard deviation, the average, and the pixel data from the best representative month are streamed to the progressive renderer for visualization. The standard deviation, standard deviation edge map, and average are also down-sampled in this process for display. The average is rendered as a height field quad mesh with the standard deviation and edge map as a texture on the mesh. Each module operates on a resolution at a time of the image hierarchy given by our data access from coarse to fine. Data is rendered in a progressive manner as it is completed. See Figure [**?**] for a diagram of the pipeline. We were able to achieve a speed up of over two times with our system.

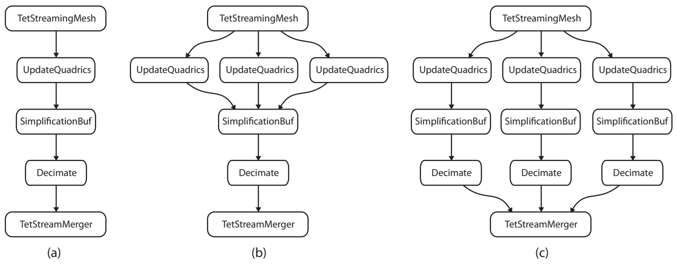

Fig. 9. Streaming simplification of tetrahedral meshes under our system (a) stream with no concurrency (b) data-parallelism (c) complete parallelism

5.2 Streaming Tetrahedral Simplification

To test and demonstration our unstructured streaming capabilities, we implemented the streaming tetrahedral mesh simplification technique of Vo et al. [28]. Given the current infrastructure of VTK without our scheme, this would not be possible. There is no streamable data structure for unstructured grids in VTK. In order to implement streaming simplification in VTK, a mapping of a portion of the output to the portion of the input meshes is necessary. This can only be done after the actual computation. Finally, VTK's streaming pipeline only supports streamable data with a pre-determined number of sections, while the algorithm only defines the end of a stream on the fly.

In our system with the generalized streaming scheme extension, we are able to construct and execute the corresponding pipeline as shown in Figure 9. TetStreamingMesh and TetStreamMerger are subclasses of StreamGenerator and StreamMerger respectively. The streaming algorithm consists of 3 main processing units: UpdateQuadrics builds the quadric metrics for vertices of the meshes, SimplificationBuf combines new streams of data into the current buffer and readies the data to go be processed by Decimate, which performs an edge collapse operation. The system also exploits several locations of data-parallelism in the pipeline.

Figure 9a shows pipeline with no added concurrency execution except for the pipeline-parallelism from our scheduler. The original version of the application is highly optimized for a single module. One would assume a degradation in performance from the optimized version, if the single module is executed in sections without any changes to the code. Due to the increase in performance inherent to our system, it can negate this degradation and achieve a similar benchmark.

To exploit data-parallelism, we can change the pipeline to allow our streaming source to send data to multiple modules. This type of data-parallelism is possible due to the fact that TetStreamingMesh utilizes the finalization property of streaming meshes to protect boundary cells across pieces. Figure 9b shows a manual tweak to the pipeline to cre-

Streaming Simplication of Tetrahedral Meshes					
Models (Tets)		Original	Streaming	Quadric Inst.	Pipeline Inst.
Torso	1.0M	8.5s	8.4s	7.9s	3.0s
Fighter	1.4M	10.1s	9.5s	8.4s	3.6s
Rbl	3.8M	37.4s	35.6s	31.5s	8.9s
Mito	5.5M	52.0s	53.1s	44.9s	10.2s

Table 2. Running time for completely simplifying models to 10% of its original resolution

ate a data-parallel pipeline with three UpdateQuadrics. The three modules are working on different portions of the meshes. However, as we see in Table 2 the building of quadrics is not the main bottle neck of this application. Therefore we still do not gain much in performance. Nonetheless, there is still a slight improvement.

In Figure 9c, the we have converted the pipeline to have complete parallelism; duplicating all three processes into three concurrent executions. As we can see in Table 2 there is a significant improvement over the original pipeline due to the parallelism. Unfortunately such an extremely parallel implementation of this algorithm can reduce the quality of the simplified mesh since there are too many boundary triangles to preserve.

Even though an optimal parallel, streaming pipeline for this particular algorithm was not found in testing, we feel that this provides an example of our system's ability to facilitate experimentation with streaming and parallelism with little effort.

6 Conclusion and Future Work

In this paper, we propose new techniques for exploiting multi-core architectures in the context of visualization dataflow systems. Specifically, we offer a robust, flexible and lightweight unified data-flow control scheme for visualization pipelines. This unified scheme allows the use of pull (demand-driven) and push (event-driven) policies in a single pipeline. The new unified scheme also combines the positive attributes of both centralized and distributed executive strategies. Moreover, we offer a system that includes new support for streaming of structured and unstructured data. As our results in the previous section show, along with the companion video, our new parallel execution strategy offers significant benefits over a both multi-core, serially-executed visualization pipelines and pipelines that are computed in streaming modes.

In the immediate future, we plan to expand our new scheme to larger machines. Although we have shown significant improvements on 4-8 cores, we feel that this is only a lower bound on the performance increase possible. We have designed this scheme with scalability as a primary consideration. In the long run, we feel this can be expanded to use all available processing resources, including GPUs. In existing dataflow systems, GPUs are relegated to back-end rendering tasks (based on OpenGL). Despite their proven superiority in terms of raw performance, it is not possible to use available GPUs to perform any of the computations in existing dataflow architectures. In fact, using GPUs to perform dataflow computations is not trivial since a modern GPU requires on the order of 1000 to 10,000 threads to achieve peak performance, and the design of the existing supported data structures makes this very difficult. Obviously, exploiting multiple GPUs either in a single machine or in the cluster of machines is not feasible with current architectures. To design a dataflow architecture that treats all the processing elements in a system as first rate processing elements, including CPUs, GPUs, and potentially other types of processing elements is a challenging and noteworthy goal.

References

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *VIS '95: Proceedings of the 6th conference on Visualization '95*, page 263. IEEE Computer Society, 1995.

[2] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics & Applications*, 21(4):34–41, July/Aug. 2001.

[3] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Technical Report LAUR-00-1620, Los Alamos National Laboratory, 2000.

[4] J. Allard and B. Raffin. A shader-based parallel rendering framework. *Visualization Conference, IEEE*, 0:17, 2005.

[5] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. VisTrails: Enabling interactive, multiple-view visualizations. In *Proceedings of IEEE Visualization*, pages 135–142, 2005.

[6] J. Biddiscombe, B. Geveci, K. Martin, K. Moreland, and D. Thompson. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1376–1383, Nov./Dec. 2007.

[7] A. Chalmers, T. Davis, and E. Reinhard. *Practical Parallel Rendering*. AK Peters Ltd, July 2002.

[8] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max. A contract-based system for large data visualization. In *Proceedings of IEEE Visualization*, pages 190–198, 2005.

[9] D. Duke, M. Wallace, R. Borgo, and C. Runciman. Fine-grained visualization pipelines and lazy functional languages. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):973–980, 2006.

[10] R. Haber and D. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*. IEEE Computer Society Press, 1990.

[11] P. E. Haeberli. ConMan: A Visual Programming Language for Interactive Graphics. In *Proceedings of SIGGRAPH'88*, pages 103–111, 1988.

[12] M. Isenburg and P. Lindstrom. Streaming meshes. In *IEEE Visualization '05*, pages 231–238, oct 2005.

[13] M. Isenburg, P. Lindstrom, and J. Snoeyink. Streaming compression of triangle meshes. In *Third Eurographics Symposium on Geometry Processing*, pages 111–118, July 2005.

[14] Kitware. ParaView. http://www.paraview.org.

[15] Kitware. The Visualization Toolkit (VTK) and Paraview. `http://www.kitware.com`.

[16] C. C. Law, K. M. Martin, W. J. Schroeder, and J. Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *IEEE Visualization '99*, pages 225–232, Oct. 1999.

[17] S. Marchesin, C. Mongenet, and J.-M. Dischler. Multi-gpu sort last volume visualization. In *EG Symposium on Parallel Graphics and Visualization (EGPGV'08)*, 2008.

[18] NASA, NASA Blue Marble http://earthobservatory.nasa.gov/Features/BlueMarble/.

[19] S. G. Parker and C. R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, page 52, 1995.

[20] V. Pascucci, D. E. Laney, R. J. Frank, G. Scorzelli, L. Linsen, B. Hamann, and F. Gygi. Real-time monitoring of large scientific simulations. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 194–198, New York, NY, USA, 2003. ACM.

[21] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of reeb graphs: Simplicity and speed. *ACM Transactions on Graphics*, 26(3):58:1–58:9, July 2007.

[22] R. Rajagopalan, D. Goswami, and S. P. Mudur. Functionality distribution for parallel rendering. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 18, Washington, DC, USA, 2005. IEEE Computer Society.

[23] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. *The Visualization Toolkit*. Prentice-Hall, pub-PH:adr, second edition, 1998. With special contributors Lisa Sobierajski Avila, Rick Avila, and C. Charles Law. Includes CD-ROM with vtk-2.0. The most recent release is available on the World-Wide Web at `http://www.kitware.com/vtk.html`.

[24] C. T. Silva, Y.-J. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization 2002, Tutorial #4*, 2002.

[25] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl. Large volume visualization of compressed time-dependent datasets on gpu clusters. *Parallel Comput.*, 31(2):205–219, 2005.

[26] C. Upson et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and*

Applications, 9(4):30–42, 1989.

[27] W. J. van der Laan, A. C. Jalba, and J. B. T. M. Roerdink. Multiresolution mip rendering of large volumetric data accelerated on graphics hardware. In *EuroVis07 - Eurographics / IEEE VGTC Symposium on Visualization*, pages 243–250, May 2007.

[28] H. T. Vo, S. P. Callahan, P. Lindstrom, V. Pascucci, and C. T. Silva. Streaming simplification of tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):145–155, Jan./Feb. 2007.

[29] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Transactions on Graphics*, 24(3):886–893, Aug. 2005.