Eurographics/ IEEE-VGTC Symposium on Visualization 2010
G. Melançon, T. Munzner, and D. Weiskopf
(Guest Editors)

Volume 29 (2010), Number 3

Streaming-Enabled Parallel Dataflow Architecture for Multicore Systems

Huy T. Vo[1], Daniel K. Osmari[2], Brian Summa[1], João L. D. Comba[2], Valerio Pascucci[1], and Cláudio T. Silva[1]

[1]SCI Institute, University of Utah, USA
[2]Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brasil

Abstract

We propose a new framework design for exploiting multi-core architectures in the context of visualization dataflow systems. Recent hardware advancements have greatly increased the levels of parallelism available with all indications showing this trend will continue in the future. Existing visualization dataflow systems have attempted to take advantage of these new resources, though they still have a number of limitations when deployed on shared memory multi-core architectures. Ideally, visualization systems should be built on top of a parallel dataflow scheme that can optimally utilize CPUs and assign resources adaptively to pipeline elements. We propose the design of a flexible dataflow architecture aimed at addressing many of the shortcomings of existing systems including a unified execution model for both demand-driven and event-driven models; a resource scheduler that can automatically make decisions on how to allocate computing resources; and support for more general streaming data structures which include unstructured elements. We have implemented our system on top of VTK with backward compatibility. In this paper, we provide evidence of performance improvements on a number of applications.

Categories and Subject Descriptors (according to ACM CCS): C.1.3 [Processor Architectures]: Other Architecture Styles—Data-flow architectures

1. Introduction

Dataflow pipeline models are widely-used in visualization systems, including AVS [Ups89], SCIRun [PJ95], and VTK-based systems such as Paraview [Kita], VisIt [CBB*05], VisTrails [BCC*05] and DeVIDE [BP08]. Despite recent advancements in parallel architecture, most systems still support only a single CPU or a small collection of CPUs such as a SMP workstation. Even the current parallel implementations fail to take advantage of the available shared-memory architecture to increase performance. It is expected that this trend towards higher levels of parallelism is likely to continue, therefore it is vital that visualization systems exploit these new architectures. Most current systems also assume, to a large extent, that the data can be maintained in memory, or worse, that multiple copies of the data could be stored across different dataflow modules. These assumptions can cause scalability problems when dealing with large data. Streaming data structures are often used in this case, though most current systems do not include this support. Even the systems that do so, only support simple data types (e.g., regular 2-D images or regular 3-D volumes). The support for unstructured and hierarchical data structures is either nonexistent 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 for both demand-driven and event-driven models. It also includes a resource scheduler that exploits the shared memory architecture to dynamically allocate computing resources (i.e. the number of threads to use with a particular module) for optimal performance. We also demonstrate the flexibility of our system by integrating support for general streaming data. This allows our system to scale to massive data. Our implementation is on top of a popular visualization toolkit (VTK) and provides backward compatibility. Due to VTK's wide acceptance in the scientific community, our system has the potential to provide an immediate and significant impact to the field.

Specifically, our contributions are the following:

	Executive			Parallelism				Scheduler		Streaming
	Scope	Policy	Async. Update	Data	Task	Pipeline	Memory	Resource Mngr.	Load Balancing	
VTK	Dist.	Pull	X				Shared			Serial
ParaView	Dist.	Pull	X	X			Dist.			Serial
VisIt	Dist.	Pull		X	X		Dist.		X	Serial
DeVIDE	Cent.	Pull/Push								Serial
SCIRun	Cent.	Push			X		Shared			N/A
VisTrails	Cent.	Pull								N/A
Ours	Dist.	Pull/Push	X	X	X	X	Shared	X	X	Parallel

Table 1: *Summary of Current Visualization Dataflow Systems*

- A new scheme for executing pipelines on multi-core hardware.
- A unified data-flow model integrating pull and push policies into an API that allows for flexible and dynamic execution strategies
- An adaptive scheduling strategy for dynamic load balancing
- A data-flow control strategy that combines the benefits of both the distributed and centralized execution controls
- A streaming framework built on top of our system that adds support for both structured and unstructured data
- A complete 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 [RGM05, SMW*05, AR05, MMD08]. Even a cursory review is beyond the scope of this paper, therefore we point the reader to [CDR02] for a complete introductory survey. For this paper, we focus our discussion on visualization dataflow systems, and in Table 1 we summarize the feature set for the systems that will be discussed in this section. Following the pioneering work of Haber and McNabb [HM90], many leading visualization systems mentioned in the previous section (e.g., [Kitb, Ups89, AT95, PJ95, CBB*05, BCC*05, BP08]) have been based on the notion of a dataflow network, which is often called a *pipeline*. *Modules* (nodes of the network) are processing elements while connections between them represent data dependencies.

Kitware's Visualization ToolKit (VTK) [Kitb] is considered to be the de-facto standard visualization API and is used by thousands of researchers and developers around the world. 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). However, its 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 [LMST99, BGM*07] 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. This is partly due to the fact 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.

Ahrens et al. [ALS*00, ABM*01] 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 [Kita].

ParaView is designed for data-parallelism only, 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 one or multiple processes based on the data input. ParaView must then rely on MPI to distribute the processes to cluster nodes to finish the computation. However, ParaView does not support a hybrid MPI/multi-threaded model; for multi-core architectures MPI is also used for creating multiple processes on the same node. This may impose substantial overhead due to the additional expense of inter-process communication over thread messages.

A related system is VisIt [CBB*05]; an interactive parallel visualization and graphical analysis tool for viewing scientific data. Though the pipeline topology in VisIt is fixed and relatively simple, it introduced the notion of *contracts* between modules, which are data structures that allow modules to negotiate with other modules on 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.

DeVIDE [BP08] is a cross-platform software framework for the rapid prototyping, testing and deployment of visualization and image processing algorithms. It was one of the first systems to fully support a hybrid execution model

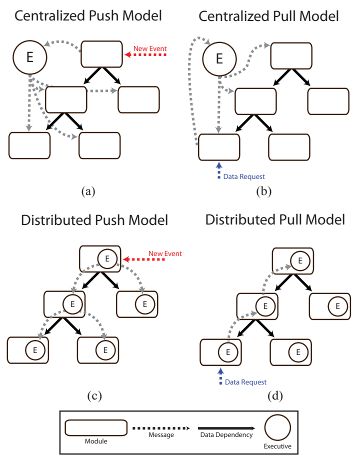

Figure 1: *Executives are classified based on their updating scope and policy (a) a centralized push model handling (b) a centralized pull model handling (c) a distributed push model handling a data request (d) a distributed pull model*

for demand- and event-driven updating policies. However, it does not target the parallel/high performance perspective of pipeline execution.

The last ten years has seen the development of a large number of streaming and out-of-core visualization algorithms (see survey by Silva et al [SCESL02]). These include a number of cache-oblivious techniques [YLPM05] that provide a memory-system agnostic way to obtain efficiency throughout complex memory hierarchies. One key development in this area is the introduction of streaming meshes by Isenburg and Lindstrom [IL05]. Streaming and cache-oblivious algorithms are used in many areas of visualization [ILS05, VCL*07, PSBM07] due to their ability to work well with data that is too large to fit in main memory.

Up to now, existing dataflow systems have not fully exploited streaming 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 an event-driven model.

We present a flexible dataflow scheme aimed at resolving many of the shortcomings of existing systems outlined in this section. Like DeVIDE, our system supports a unified execution model for both demand-driven and event-driven updating policies. However, a major difference, as shown in Table 1, is that we allow asynchronous updates of a pipeline or its subset of modules through our API calls, where as in DeVIDE, and other workflow systems such as VisTrails, a pipeline must be scheduled to update in sync. Our system also provides a resource scheduler to exploit the shared memory architecture and a dynamic allocation of computing resources for optimal performance. We also demonstrate the flexibility of our system by novelly supporting general streaming data including unstructured elements (e.g. tetrahedral elements). Streaming not only decreases memory overhead, but also increases performance. Our system has been integrated with VTK and provides backward compatibility. This seamless extension to a widely used toolkit in the scientific community allows our system to have immediate and significant impact to the field.

3. Visualization Dataflow System Design

The input for a module when it executes may or may not be independent of other modules in the pipeline. Thus, a level of coordination is required between the modules and their data dependencies. In the simplest form, this is implemented statically 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. In this approach, algorithms can be implemented based purely on the computational task required without consideration for the topology of the pipeline.

Executives can be classified based on their updating scope, centralized vs. distributed, and policy, pull vs. push (or demand- vs. event-driven as stated in [SML98]). A *centralized* executive operates globally and is connected to every module to track all changes to the network, as well as handling all requests and queries in the pipeline. This approach gives the system the advantage of having control over the entire execution network, thereby allowing it to be 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. In the *distributed* executive 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 the network makes distributing resources or coping with more complicated executing strategies a much more difficult problem. With respect to an updating policy, in a *pull* model a module is executed only when one of its outputs is requested. If a module's input is not up-to-date, it will demand or pull the corresponding upstream module for data. Therefore only modules that are needed for the actual computation will be executed, 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 contrast, for a *push* model modules are updated as soon as an event occurs, e.g. a change in its input. All modules that are dependent on the requested module's output are then 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. Modules compute and generate data even when they will not be used. Figure 1 illustrates the classifications for executives outlined in this section.

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 2. Our system is based on a distributed executive scheme with a centralized resource-aware scheduler to efficiently distribute resources. Each local executive can perform both the `Pull` and `Push` functions . Each module also contains a resource specification indicating the number of threads its algorithm can utilize at run-time. This information is used by the scheduler.

4.1. Execution Model

In our framework, a pipeline execution starts with an explicit update request of a module. Depending on a module's update policy, i.e. pull or push, its upstream or downstream modules will then be executed accordingly. This process repeats until all modules are updated. However, instead of statically assigning each module with a fixed update policy like other systems, we allow the policy to be dynamically set. This can be done at the module implementation level with the supplied API's `Pull(M, R)` and `Push(M, R)` functions. `M` and `R` are optional arguments where `M` indicates the list of target modules to be updated and `R` is any additional information that needs to be passed. By default, `M` is set to all immediate upstream and downstream modules for `Pull` and `Push` respectively.

For example, consider the simple pipeline in Figure 3. Data is read through the `DataAccess` module and passed to the `DataProcess` module before being rendered by the `Viewer` module. Although simple, this pipeline is common in a progressive rendering system, where changes to the viewport caused by user interaction require a series of data requests ranging from the coarsest to finest level-of-detail (LOD). These new requests would be made to the `DataAccess` module and can be implemented using a pull policy as follows:

```
renderRequested()
  LOD = C // the lowest resolution of LOD
  while (LOD>0)
    Pull(<Viewer>, LOD)
    LOD = LOD - 1
```

Often certain applications would like to refresh a display whenever new data is available (e.g. viewing data as it is downloaded from an external device). In this case, a push policy can be easily introduced to the pipeline to trigger a new data event:

```
newDataArrived()
  Push(<DataAccess>)
```

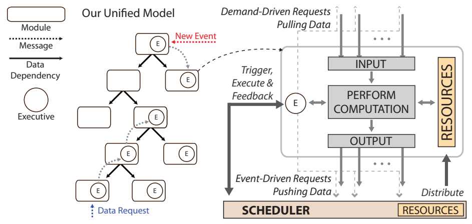

Figure 2: *An overview of our system architecture*

DataAccess
DataProcess
Viewer

Figure 3: *A simple rendering pipeline.*

In the case of streaming data, `Push` offers higher efficiency due to its support for both task and pipeline parallelism. When `Pull` is called, the function only returns after all the upstream modules are updated. This, in effect, locks the modules in a dataflow to a selected piece of the stream data. On the other hand, `Push` will return as soon as the scheduler determines that there are idling threads from the available resources. This allows a module to load new data after it sends its data downstream. Therefore multiple `Push` calls made sequentially can operate independently on separate stream-data-blocks.

Data duplication is avoided whenever possible since copying and allocating memory could substantially degrade the whole pipeline performance. This can be especially detrimental in a shared-memory system, where multiple cores have the ability to access memory simultaneously. In order to prevent write-before-read issues, modules are locked for scheduling upon entering its execution loop and stay locked until all output data has been flagged for release by downstream modules. By default, at the end of each `compute()` method, a module releases its input automatically. However, we allow API users to override this default by manually releasing the data earlier or delaying release by using the `ReleaseInputs()` method. When computing heavy modules, it would be advantageous to copy data locally and release the input to allow upstream modules to process new data. For instance, in the same example on Figure 3, if both reading the data in `DataAccess` and processing the data in `DataProcess` are time-consuming, `DataProcess` can copy its input locally and release, allowing `DataAccess` free to read the next data block.

```
DataProcess::compute()
  // copy input data to local memory
  ...
  this->ReleaseInputs();
  // process the copied data in local memory
  ...
```

4.2. Scheduler

The scheduler is responsible for both scheduling and distributing computing resources (threads) to modules in a pipeline. When a module executive is asked to execute its algorithm, instead of performing the computation right away,

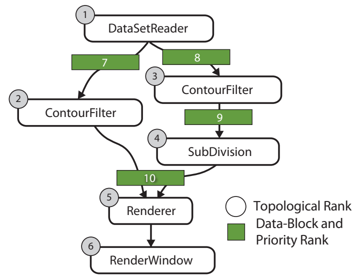

Figure 4: *Streaming priority assignments by our scheduler*

it submits the execution to the scheduler's queue. The scheduler, depending on the number of available threads, will execute the algorithm at an appropriate time with the appropriate resources.

When executing modules of a network concurrently, a scheduler with a simple FIFO queue will not guarantee the order and data dependencies of the pipeline. For our scheduler, we use a priority queue partially keyed by the module's topological order. This also ensures that there is only one update for a single request in a push model. For example, in the pipeline in Figure 4, a regular FIFO queue would push modules (1), (2), (3), (5), (4) and then (5) again, but with the priority queue the execution of (5) would be postponed until after (4) completes.

However, topological order alone still has problems in regards to streaming. If multiple threads are available, relying primarily on topological order may run the risk of all threads being allocated to the modules loading the data, counteracting the benefits of streaming. For example, if (1) is a streaming data reader, after it processes the first data piece, it passes the data down to the contour filters (2) and (3) which are now in the scheduling queue. Then, the reader will move to the second piece of data, putting itself again on the queue. Because (1) has lower order than (2) and (3) it will be executed again. Our solution is to use not only the topological order as priority key but also use the data block number, e.g. streaming piece. Internally, if modules don't specify the data block number as they submit an execution to the queue, a global counter is used. With this approach, the scheduler will attempt to move a single data-block as far down the network as possible before processing the next piece.

Scheduling strategy Our flexible scheme handles any scheduling strategy. For testing, we have implemented a heuristic strategy based on time statistics. At the time of rescheduling, the scheduler transverses the whole pipeline starting at the sink modules and distributes resources among the branches. Since a 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 runtime, 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. 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. Streaming Computation

Streaming algorithms are inherently useful in visualization pipelines, though they are still under-represented in current dataflow systems due to the lack of a general streaming framework. Our system intrinsically supports streaming. Since both `Pull()` and `Push()` only return when target modules are able process more data, streaming algorithms can simply be expressed as a sequential program. Below are two scenario usages of streaming:

With `Pull()`:

```
for (i=0; i<numPieces; i++)
  R = i // Set the piece number
  this->Pull(<Upstream Modules>, R)
```

With `Push()`:

```
while (!this->EndOfStream())
  this->ReadData()
  this->Push()
```

There are two basic differences between the push and pull models for streaming: (1) a push is triggered at source modules while a pull is triggered from sinks and (2) only the push model can take advantage of pipeline-parallelism since the pull model requires that all upstream modules be locked during an update. Therefore, even though both models are easy to use with streaming, push is encouraged since it can achieve higher performance at the cost of more memory usage.

Our system also extends streamable data structures beyond the standard structured grids by generalizing the streaming mesh format. The streaming mesh format was originally designed for triangular meshes by interleaving its geometry with connectivity. It introduced the 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 just allowing the data structures to decide which elements are finalized, the algorithm is also allowed to flag elements as unfinalized. For example, an image filter may set a neighborhood outside the portion being processed as unfinalized. The interface for this class of streamable data structure consists of two main methods that can be subclass-ed into other needs:

```
class StreamableData:
  void     setData(POINTER *data)
  POINTER *getData(pos)
  void     next(pos);
  void     finalizeData(pos)
```

where `pos` is the relative position of the data to the current position of the stream, e.g. `pos=0` is the current position. `setData()` and `getData()` are used to set and retrieve the data associated with a position. `next()` shifts the current stream position, which can be treated as moving the current sliding window of the stream. `finalizeData()` flags a certain piece of data as finalized and that it can be discarded to free memory.

4.4. Framework Implementation

We have implemented our framework on top of VTK, inheriting a robust software infrastructure along with existing algorithms for testing. We have also added three new classes into VTK's Filtering package without any other modifications to the existing source code: *vtkComputingResources*, *vtkExecutionScheduler*, and *vtkThreadedStreamingPipeline*.

vtkComputingResources holds information on computing resources, i.e. the minimum, maximum and the preferable number of threads. Each instance of vtkAlgorithm may include a vtkComputingResources object if it can run with more than one thread.

vtkExecutionScheduler is responsible for scheduling executions as well as distributing threads to pipeline modules. There is a static global scheduler for the whole system, however, our framework permits the existence of multiple instances of `vtkExecutionScheduler`. Each instance can work separately using its own specification of `vtkComputingResources` indicating how many threads it is managing.

These classes are not designed to be used directly by module developers, though they are the building blocks for the implementation of `vtkThreadedStreamingPipeline`.

`vtkExecutionScheduler::Schedule()` takes a collection of executives as input and schedules their execution. This method first creates a dependency graph from the input modules, then assigns a topological order to them. Since functions can be called while modules are currently being executed, the newly created dependency graph could be merged with the current running dataflow network if it exists. The combined graph is then placed in the priority queue. However, no module execution will be explicitly triggered by this function. Instead, the scheduler's secondary thread checks the queue and decides which, if any, modules need to be executed. Before a module becomes active for execution, this secondary thread would also assign the number of threads allocated for the module based on the default scheduling strategy.

vtkThreadedStreamingPipeline inherits from the `vtkCompositeDataPipeline` class and is therefore fully backwards-compatible with original VTK pipelines. For our framework, we reimplemented the `ForwardUpstream` method and added `Pull()` and `Push()` functions to interface with our execution model. Note that all of our systems multi-threaded features can be turned on/off through a global flag set by the method `SetMultiThreadedEnabled()` of this class.

As previously discussed, both `Pull()` and `Push()` can accept optional arguments `M` and `R`. In VTK, these are set be a subclass of `vtkCollection` and `vtkInformation` respectively.

When `Pull()` is called on a module, it performs a search on the dataflow network to collect all of the upstream modules on which the module depends. It then passes them to the `Schedule` method of the global scheduler. A call to `WaitUntilDone()` is also made to guarantee that the control only returns when all the scheduled upstream modules have been executed.

On the other hand, `Push()` does not need to look beyond its immediate downstream modules to pass to the scheduler. After the modules are passed, a call to `WaitUntilRelease()` is then made. This will block the control until the scheduler allocates the resources to get more data.

5. Applications

VTK has been the subject of a large body of streaming research and therefore is an 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. While our initial implementation and testing uses VTK, our framework by is general by design and can be easily extended to other systems. All tests were performed on a machine consisting of 2 Intel Nehalem Xeon w5580 processors with a total of 8 cores and 24GB of DDR3 RAM. This is the maximum number of cores that we can get for a shared-memory system using the fast DDR3 memory.

5.1. Multi-core Image Processing

The VTK image processing pipeline is capable of multi-threaded 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

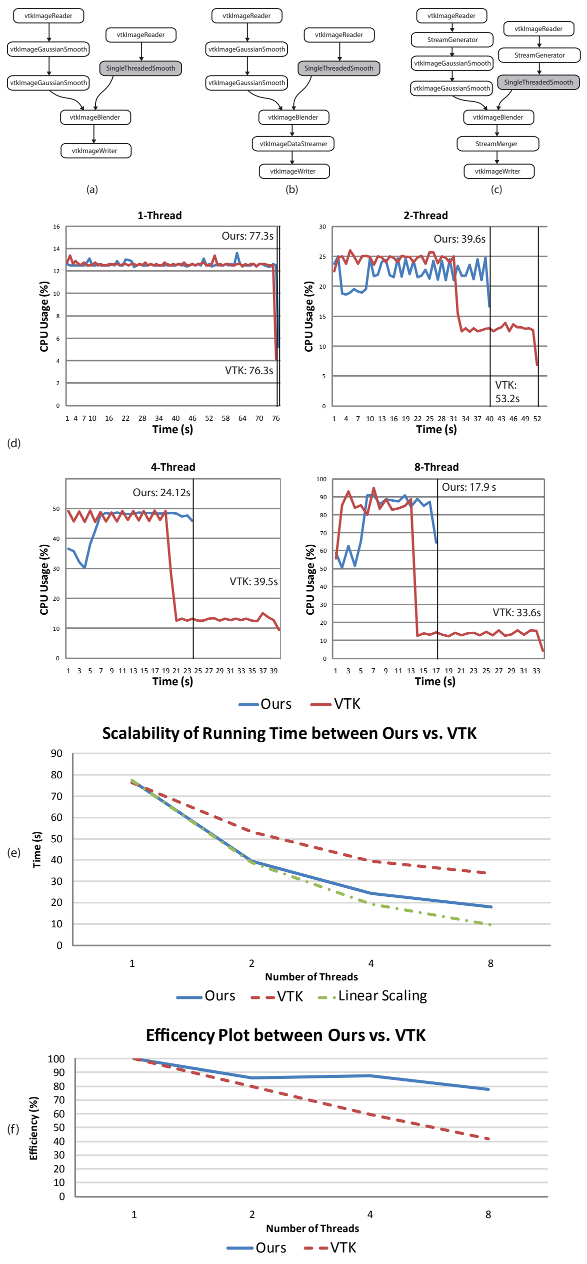

Figure 5: *The Gaussian smooth pipeline and its performance analysis (a) VTK pipeline without streaming; (b) the same pipeline with streaming (c) our streaming pipeline; (d) CPU usage 1-8 threads; (e) Strong scaling (f) Efficiency*

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.

	1 thread		2 threads		4 threads		8 threads	
	Time	Eff.	Time	Eff.	Time	Eff.	Time	Eff.
VTK	76s	100%	53s	80.0%	39s	59.4%	33s	41.8%
Ours	77s	100%	39s	85.9%	24s	87.6%	17s	77.7%

Table 2: *Running time and efficiency ratio of CPU usage between VTK and our system for a simple Gaussian pipeline.*

Specifically, it requires a smaller memory footprint and retains high cache coherency. This functionality is unfortunately demand-driven, which can block pipeline-parallelism, and is only applicable for subclasses of `vtkThreaded-ImageAlgorithm`. Our pipeline does not suffer from the problems inherent in VTK's default or streaming pipeline, exploits parallelism, has 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.

Gaussian Smooth Pipeline VTK's imaging modules, such as Gaussian smooth and blender, can be configured to run multi-threaded using all available cores on a machine. Thus, it is possible to achieve maximum performance with pipelines that only contain these types of modules. Unfortunately, in practice these modules will only be a small portion of a typical pipeline. For testing, we construct a simple smoothing pipeline that consists of both threadable and unthreadable modules. The pipeline takes two images, then smoothes and blends them together. See Figure 5(a,b,c) for a diagram of the pipeline. Here, `SingleThreadedSmooth` cannot utilize more than one thread to increase performance. Using the default VTK model of serial execution of modules, the performance would not be optimal in a multi-threaded environment since there would be idling threads when `SingleThreadedSmooth` is running. In this case, the more threads available to the system, the worse its efficiency is. On the other hand, our framework can handle this situation by promoting task parallelism, i.e. having `SingleThreadedSmooth` run concurrently with the others. This difference is shown in the CPU usage graph in Figure 5d. VTK first runs all multi-threaded image filters, then executes the single threaded smoothing module only when the multi-threaded modules have finished. In contrast, our framework after a time-collecting phase load balances and keeps all cores busy. The strong scaling test in Figure 5(e and f) clearly shows our execution model is superior to VTK's default threaded model. Table 2 provides the actual running details including the efficiency ratio of CPU usage. The tests were used with two synthetic images of 200 megapixels in size.

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 [NASin]. Each image from this data set is 3.7-gigapixel, therefore we must employ out-of-core data access. For this implementation, we use the ViSUS library which is based on the hierarchical z-order scheme as outlined in [PLF*03]. In practice, we have found this method inherently provides a hierarchical

	1 thread	2 threads	4 threads	8 threads
12-Month Avg.	20.5s	10.3s	5.4s	2.7s
Multi-View	88.1s	40.7s	18.9s	18.7s

Table 3: *Timing for ViSUS*

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 6 top 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 near-optimal scalability with an 8 times speedup when moving from 1 to 8 cores. The strong scalability graph and the numbers can be found in Figure 6 bottom and Table 3.

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 [NASin]. 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 6 middle for a diagram of the pipeline. Since there are only 4 parallel independent execution paths in this pipeline, our system was able to scale in performance to only 4-core.

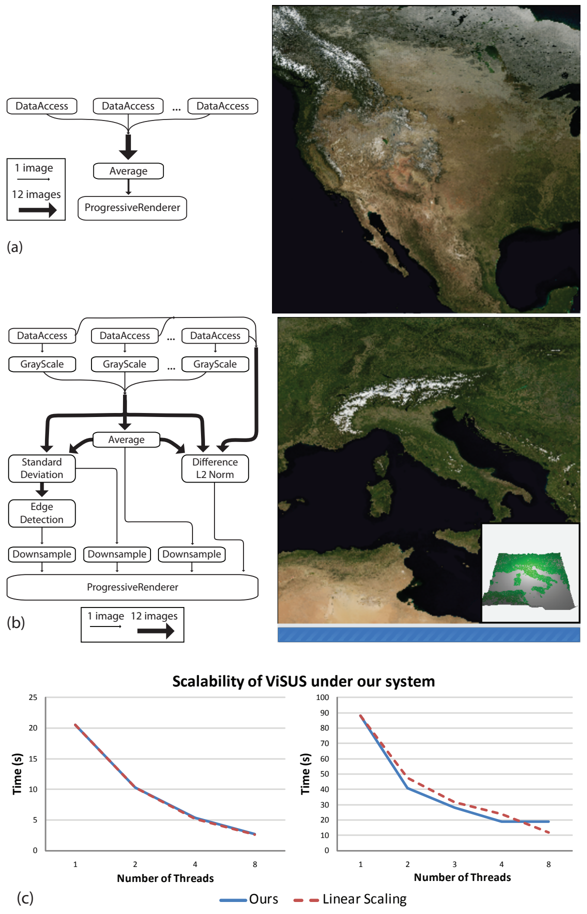

Figure 6: *Two ViSUS pipelines performing: (a) the 12-Month Average, (b) Multi-View selective rendering and (c) the scalability plot of their performances.*

After that, the performance stays at that optimal peak. Table 3 illustrates these results with timing numbers. In the scalability plot in Figure 6 bottom, we observe a slight super linear speedup probably due to a coherent disk cache when multiple threads access data simultaneously.

5.2. Streaming Tetrahedral Mesh Simplification

To test and demonstration the flexibility in extending our framework to include unstructured streaming capabilities, we implemented the streaming tetrahedral mesh simplification technique of Vo et al. [VCL*07]. 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 predetermined number of sections, while the algorithm only defines the end of a stream on the fly.

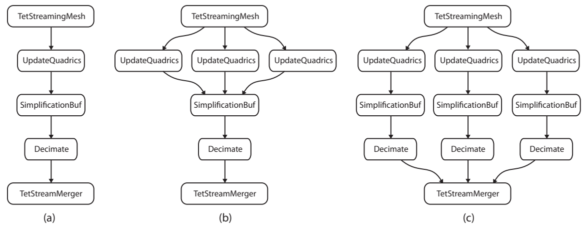

Figure 7: *Streaming simplification of tetrahedral meshes under our system (a) stream with no concurrency (b) data-parallelism and (c) complete parallelism*

In our system with the generalized streaming scheme extension, we are able to construct and execute the corresponding pipeline as shown in Figure 7. 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 be processed by Decimate, which performs an edge collapse operation. The system also exploits several locations of data-parallelism in the pipeline.

Figure 7a shows a 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 7b shows a manual tweak to the pipeline to create a data-parallel pipeline with three UpdateQuadrics. The three modules are working on different portions of the meshes. However, as we see in Table 4 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 7c, we have converted the pipeline to have complete parallelism; duplicating all three processes into three concurrent executions. As we can see in Table 4 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.

Streaming Simplication of Tetrahedral Meshes					
Models (Tets)		Original	Streaming	Quadric Duplicates	Pipeline Duplicates
Torso	1.0M	5.8s	5.8s	5.1s	1.3s
Fighter	1.4M	7.5s	6.7s	5.4s	1.6s
Rbl	3.8M	29.7s	26.1s	22.3s	6.2s
Mito	5.5M	36.4s	28.8s	27.6s	7.1s

Table 4: *Running time for completely simplifying models to 10% of its original resolution*

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 is flexible enough to support a general streaming data structure. 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.

Although we have shown significant improvements on a state-of-the-art 8-core machine, 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 running in distributed mode. 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. Once the system is expanded to use both CPUs and GPUs on a single machine, the flexible design of interconnects across modules would allow us to proceed to execute pipelines on a cluster with minimum efforts. However, a new scheduling strategy must be implemented in order to take full advantages of both shared and distributed architecture, i.e., minimizing data transfers. 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

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics & Applications 21*, 4 (July/Aug. 2001), 34–41.

[ALS*00] AHRENS J., LAW C., SCHROEDER W., MARTIN K., PAPKA M.: *A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets*. Technical Report LAUR-00-1620, Los Alamos National Laboratory, 2000.

[AR05] ALLARD J., RAFFIN B.: A shader-based parallel rendering framework. *Visualization Conference, IEEE 0* (2005), 17.

[AT95] ABRAM G., TREINISH L.: An extended data-flow architecture for data analysis and visualization. In *VIS '95: Proceedings of the 6th conference on Visualization '95* (1995), IEEE Computer Society, p. 263.

[BCC*05] BAVOIL L., CALLAHAN S., CROSSNO P., FREIRE J., SCHEIDEGGER C., SILVA C., VO H.: VisTrails: Enabling interactive, multiple-view visualizations. In *Proceedings of IEEE Visualization* (2005), pp. 135–142.

[BGM*07] BIDDISCOMBE J., GEVECI B., MARTIN K., MORELAND K., THOMPSON D.: Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (Nov./Dec. 2007), 1376–1383.

[BP08] BOTHA C. P., POST F. H.: Hybrid scheduling in the devide dataflow visualisation environment. In *SimVis* (2008), pp. 309–322.

[CBB*05] CHILDS H., BRUGGER E. S., BONNELL K. S., MEREDITH J. S., MILLER M., WHITLOCK B. J., MAX N.: A contract-based system for large data visualization. In *Proceedings of IEEE Visualization* (2005), pp. 190–198.

[CDR02] CHALMERS A., DAVIS T., REINHARD E.: *Practical Parallel Rendering*. AK Peters Ltd, July 2002.

[DWBR06] DUKE D., WALLACE M., BORGO R., RUNCIMAN C.: Fine-grained visualization pipelines and lazy functional languages. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 973–980.

[Hae88] HAEBERLI P. E.: ConMan: A Visual Programming Language for Interactive Graphics. In *Proceedings of SIGGRAPH'88* (1988), pp. 103–111.

[HM90] HABER R., MCNABB D.: Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing* (1990), IEEE Computer Society Press.

[IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *IEEE Visualization '05* (oct 2005), pp. 231–238.

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Streaming compression of triangle meshes. In *Third Eurographics Symposium on Geometry Processing* (July 2005), pp. 111–118.

[Kita] KITWARE: ParaView. http://www.paraview.org.

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

[LMST99] LAW C. C., MARTIN K. M., SCHROEDER W. J., TEMKIN J.: A multi-threaded streaming pipeline architecture for large structured data sets. In *IEEE Visualization '99* (Oct. 1999), pp. 225–232.

[MMD08] MARCHESIN S., MONGENET C., DISCHLER J.-M.: Multi-gpu sort last volume visualization. In *EG Symposium on Parallel Graphics and Visualization (EGPGV'08)* (2008).

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

[PJ95] PARKER S. G., JOHNSON C. R.: SCIRun: a scientific programming environment for computational steering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)* (1995), p. 52.

[PLF*03] PASCUCCI V., LANEY D. E., FRANK R. J., SCORZELLI G., LINSEN L., HAMANN B., GYGI F.: Real-time monitoring of large scientific simulations. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing* (New York, NY, USA, 2003), ACM, pp. 194–198.

[PSBM07] PASCUCCI V., SCORZELLI G., BREMER P.-T., MASCARENHAS A.: Robust on-line computation of reeb graphs: Simplicity and speed. *ACM Transactions on Graphics 26*, 3 (July 2007), 58:1–58:9.

[RGM05] RAJAGOPALAN R., GOSWAMI D., MUDUR S. P.: Functionality distribution for parallel rendering. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers* (Washington, DC, USA, 2005), IEEE Computer Society, p. 18.

[SCESL02] SILVA C. T., CHIANG Y.-J., EL-SANA J., LINDSTROM P.: Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization 2002, Tutorial #4* (2002).

[SML98] SCHROEDER W. J., MARTIN K. M., LORENSEN W. E.: *The Visualization Toolkit*, second ed. Prentice-Hall, pub-PH:adr, 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`.

[SMW*05] STRENGERT M., MAGALLÓN M., WEISKOPF D., GUTHE S., ERTL T.: Large volume visualization of compressed time-dependent datasets on gpu clusters. *Parallel Comput. 31*, 2 (2005), 205–219.

[Ups89] UPSON ET AL C.: The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications 9*, 4 (1989), 30–42.

[VCL*07] VO H. T., CALLAHAN S. P., LINDSTROM P., PASCUCCI V., SILVA C. T.: Streaming simplification of tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics 13*, 1 (Jan./Feb. 2007), 145–155.

[vdLJR07] VAN DER LAAN W. J., JALBA A. C., ROERDINK J. B. T. M.: Multiresolution mip rendering of large volumetric data accelerated on graphics hardware. In *EuroVis07 - Eurographics / IEEE VGTC Symposium on Visualization* (May 2007), pp. 243–250.

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. *ACM Transactions on Graphics 24*, 3 (Aug. 2005), 886–893.