DESIGNING A PARALLEL DATAFLOW ARCHITECTURE FOR STREAMING LARGE-SCALE VISUALIZATION ON HETEROGENEOUS PLATFORMS

by

Huy T. Vo

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

August 2011

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Huy T. Vo**

has been approved by the following supervisory committee members:

Cláudio T. Silva	, Chair	**03/18/2011** Date Approved
Juliana Freire	, Member	**03/18/2011** Date Approved
Charles D. Hansen	, Member	Date Approved
Valerio Pascucci	, Member	**03/18/2011** Date Approved
João L.D. Comba	, Member	**03/18/2011** Date Approved

and by **Al Davis** , Chair of

the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Dataflow pipeline models are widely used in visualization systems. 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 for systems that are specifically tuned towards parallel visualization, their execution models only provide support for data-parallelism while ignoring task-parallelism and pipeline-parallelism. With the recent popularization of machines equipped with multicore CPUs and multi-GPU units, these visualization systems are undoubtedly falling further behind in reaching maximum efficiency.

On the other hand, there exist several libraries that can schedule program executions on multiple CPUs and/or multiple GPUs. However, due to differences in executing a task graph and a pipeline along with their APIs being considerably low-level, it still remains a challenge to integrate these run-time libraries into current visualization systems.

Thus, there is a need for a redesigned dataflow architecture to fully support and exploit the power of highly parallel machines in large-scale visualization. The new design must be able to schedule executions on heterogeneous platforms while at the same time supporting arbitrarily large datasets through the use of streaming data structures.

The primary goal of this dissertation work is to develop a parallel dataflow architecture for streaming large-scale visualizations. The framework includes supports for platforms ranging from multicore processors to clusters consisting of thousands CPUs and GPUs. We achieve this in our system by introducing the notion of *Virtual Processing Elements* and *Task-Oriented Modules* along with a highly customizable scheduler that controls the assignment of tasks to elements dynamically. This creates an intuitive way to maintain multiple CPU/GPU kernels yet still provide coherency and synchronization across module executions. We have implemented these techniques into *HyperFlow* which is made of an API with all basic dataflow constructs described in the dissertation, and a distributed run-time library that can be used to deploy those pipelines on multicore, multi-GPU and cluster-based platforms.

To my mother, my endless source of support...

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Prof. Cláudio Silva deeply for offering me tremendous guidance and support throughout both my undergraduate and graduate years. I consider myself to be very fortunate to get an opportunity to work with him during the past seven years. He has given me countless experiences to become a researcher that I could not have gained anywhere else. He has been greatly patient with me and my “random disappearing”. I hope that I did not give him too many troubles.

I would like to thank all of my committee members: Prof. Cláudio Silva, Prof. Juliana Freire, Prof. Charles Hansen, Prof. Valerio Pascucci and Prof. João Comba, for their valuable comments and insightful discussions to make this dissertation possible.

I would like to thank Prof. Cláudio Silva, Prof. Juliana Freire and the entire VisTrails team for giving me an opportunity to work with such a talented group of researchers. It has been my honor.

I would like to thank Louis Bavoil, Steve Callahan, and Carlos Scheidegger, whom I have learned so much from during my early years in the VGC group. I also want to thank Brian Summa, Daniel Osmari, Luiz Scheidegger and Jonathan Bronson for working hard with me through many projects.

I would like to thank Nick Rathke and the IT team at SCI for picking up his phone at midnight and Liz Jordan for granting me sodas after hours to keep me up during deadlines.

I would like to thank Emanuele Santos, Linh Ha, Hoa Nguyen, Anh Vo, Huong Nguyen, Trang Pham, and Khiem Nguyen for supplying me with many home-made meals. They were delicious. Without them, I would have only lived on instant noodles.

I would like to thank Gustavo and Manu for many running sessions altogether. I also want to thank Luciano Barbosa, Lauro Lins, Karane Vieira, Tiago Quieroz and the Brazillian “gang” for great friendships.

I would like to thank D.A., N., L., C., V., T., H., and S.B for being there for me.

I would like to thank my buddy Wang Bung Bu just because he is my best friend.

With all my heart, I would like to show my deepest gratitude to my family: my dad, my mom, my sister and recently my brother-in-law and my niece. My parents have been supporting me unconditionally with love and much more for the past 26 years. I hope my dissertation could somehow express my appreciation, though only a fraction of it, to them.

CHAPTER 1

INTRODUCTION

The recent popularization of commodity multicore CPUs and multi-GPU units has opened many alternatives for the design and implementation of efficient visualization and data analysis algorithms. To do this, one typically needs to manually design a solution that distributes the processing work among CPU cores and GPU units, an approach that can be very cumbersome in terms of development time. The ability to combine the processing power of CPUs and GPUs in a more automatic and extensible fashion requires algorithms to encapsulate tasks into high-level modules capable of being executed in two or more kinds of fundamentally different processing elements. This raises an immediate challenge, since CPUs and modern GPUs behave very differently from one another. Also, this heterogeneous integration requires an underlying infrastructure to control execution and data-transfer between modules efficiently.

In order to assist developers in these new architectures, libraries and frameworks have been created with capabilities of abstracting native threads and GPU stream processors in execution as well as decomposing programs into task graphs, that can be mapped efficiently to different processing elements at run-time. Among these are Intel Threading Building Blocks [42], AMD Stream SDK [5], HMPP [17], StarPU [6]. StarPU and AMD Stream SDK are also capable of scheduling executions on both CPUs and GPUs simultaneously.

On the other hand, existing pipeline models used in scientific and visualization applications, such as AVS [86], SCIRun [73], and VTK-based systems such as Paraview [50], VisIt [19], VisTrails [8] and DeVIDE [11], already decompose computing tasks into independent modules and transparently manage data communication inside a pipeline. However, these systems were designed around the assumption of a homogeneous processing model, based only on multicore CPUs. Even the current parallel implementations fail to take advantage of the available shared-memory architecture to increase performance.

Unfortunately, these systems cannot utilize existing parallel frameworks directly because of their differences in execution models. Current parallel frameworks make use of task graph execution, which is a *workflow*-based architecture, whereas visualization systems are mostly *dataflow*-based. Since workflows require more constraints in pipeline topology as well as execution mechanism than dataflow, simply transforming pipelines in these systems into task graphs will not work.

Moreover, 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. A full revision of the streaming data structures and algorithms is necessary, besides the execution model, in order to extend current systems to use the processing power of both CPUs and GPUs.

1.1 Dissertation Objectives

In this dissertation, we present a parallel dataflow architecture that treats all processing elements in a heterogeneous system as first rate computational units, including multicore CPUs and GPU units, along with a number of large-scale visualization experiments that have helped us determine crucial requirements for such framework:

- **Parallel Constructs:** the framework supports building pipelines with all three fundamental types of parallelism: *task-*, *data-* and *pipeline-* parallelism, and providing methods for specifying which of them should be incurred on demand.
- **Heterogeneous Deployment:** pipelines built with the framework should be able to run on machines with different configurations ranging from a single machine with multiple CPUs and/or GPUs to a cluster of machines.
- **Streaming:** large datasets are supported through the use of streaming data structures with all types of parallelism.
- **Extensible API:** the framework API has to be flexible enough such that new hardware configurations can be added without difficulty.

As part of this dissertation, we have built HyperFlow, a parallel dataflow architecture for efficient parallel executions on heterogeneous systems. It includes an adaptive execution scheduler that assigns tasks for different processing elements such as CPUs and GPUs, thus minimizing load balancing problems without added effort by pipeline developers. HyperFlow also supports classical parallel constructions such as Map/Reduce templates, among others.

1.2 Dissertation Structure

The rest of the dissertation is organized as follows. Chapter 2 reviews current parallel programming environments and visualization systems. It also provides a background on dataflow architecture. Next, Chapter 3, 4 and 5 describe our experiments with large-scale visualizations in three applications: streaming simplification of tetrahedral meshes, interactive rendering of large unstructured grids and visualizations with MapReduce, respectively. Chapter 6 presents our initial design of a parallel dataflow architecture for multicore machines. Chapter 7 introduces HyperFlow, our design for multiple CPUs and GPUs systems. Then, Chapter 8 discusses options for extending HyperFlow to distributed environments. Finally, Chapter 9 concludes the dissertation along with future work.

1.3 Remark

Many of the results presented in this dissertation have been published in journals and/or proceedings of conferences. In particular, Chapter 3, Chapter 4 and Chapter 6 are based on [87], [88], and [89], respectively.

CHAPTER 2

BACKGROUND AND RELATED WORK

Parallel rendering and visualization has been the focus of a large body of work [75, 83, 4, 62, 61]. Even a cursory review is beyond the scope of this chapter. Therefore we point the reader to [18, 47] for a complete introductory survey. In this chapter, we will review design choices of dataflow architecture and provide an overview of current visualizations systems.

2.1 Dataflow Architecture

Following the pioneering work of Haber and McNabb [38], many leading visualization systems mentioned in the previous section (e.g., [51, 86, 73, 19, 8, 11]) 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. Inputs for a module may or may not be independent of the others. Thus, a level of coordination is required between the modules and their data dependencies during executions. 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.

2.1.1 Executive Policy and Scope

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 [78]). 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 as 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 2.1 illustrates the classifications for executives outlined here.

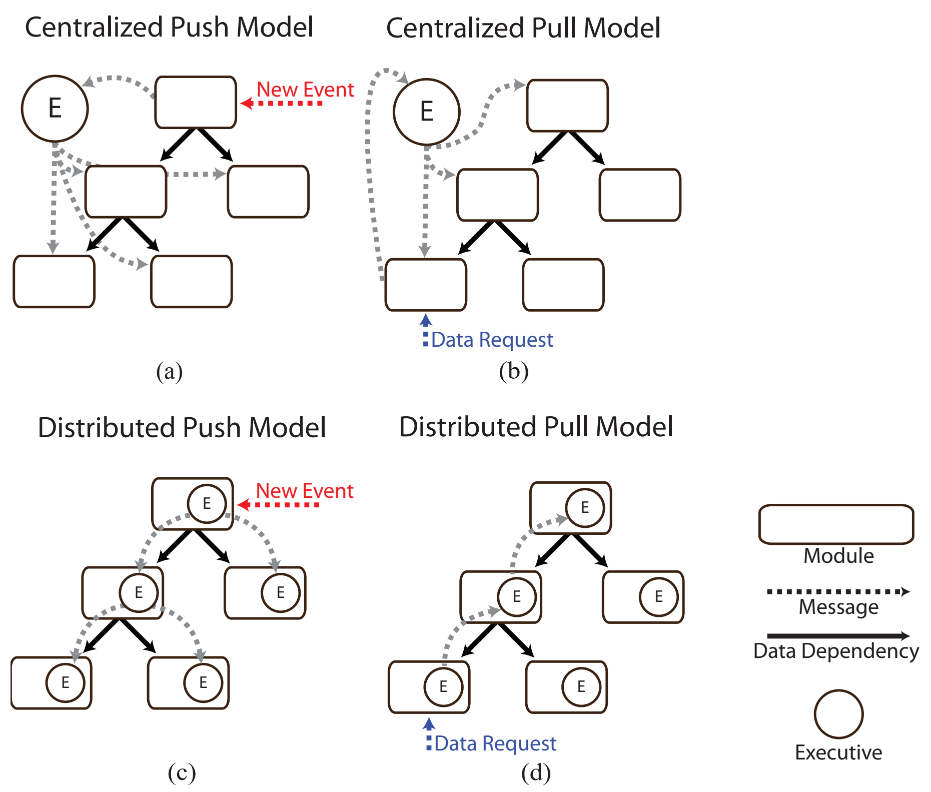


Figure 2.1: **Executive Classification** Executives are classified based on their updating scope and policy (a) a centralized push model handling (b) a centralized pull model handling (c) a distributed push model handling a data request (d) a distributed pull model

2.1.2 Parallelism

There are three fundamental types of parallelism in dataflow execution. The first form of parallelism is *task-parallelism*. This scheme allows independent branches of the networks to run in parallel each using its own thread. Task-parallelism requires pipelines with many independant modules in order to maximize concurrent executions. Figure 2.2a shows an example where modules A and B are independant of C. Therefore A and B are allowed to be run concurrently with C.

The second form of parallelism is *pipeline-parallelism*: pipelines are partitioned into subnetworks operating on different tasks. During execution, each network would be allocated its 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 subnetworks 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 2.2b, is an example of streaming pipeline parallelism between two modules.

The last common form of parallelism is *data-parallelism*. In this scheme, the data are divided into independant pieces to be processed in parallel. At the end of execution, data are 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 is maintained only if there is one merge operation in the end. Otherwise, data must be redistributed multiple times causing a large degradation of performance. In practice, a single merge is the usual case for rendering pipelines. Figure 2.2c illustrates the use of data-parallelism, where the pipeline AB is duplicated multiple times to work on the separate pieces of the streaming data.

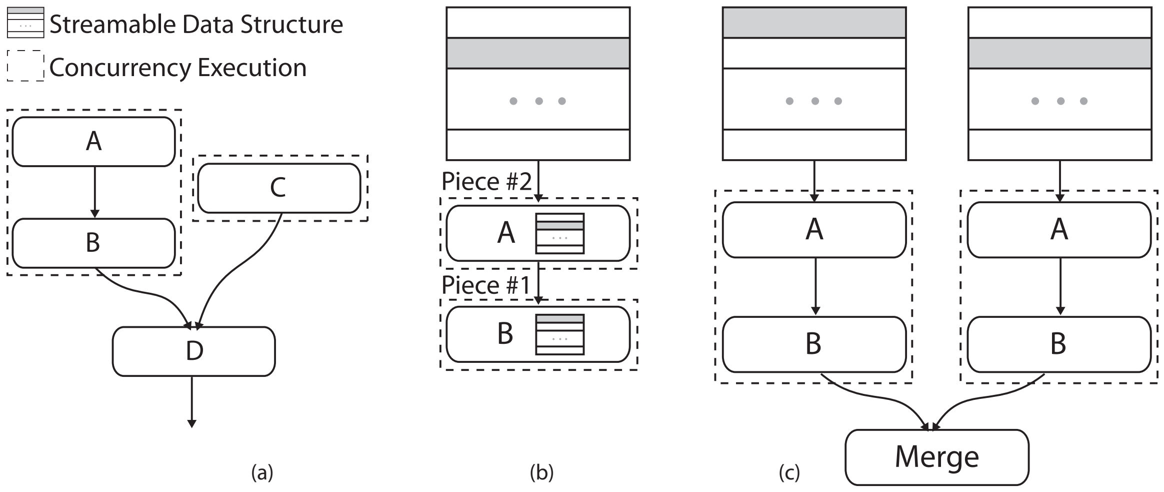


Figure 2.2: **Dataflow Parallelism** The three fundamental types of parallelism in the dataflow architecture (a) Task Parallelism (b) Pipeline Parallelism and (c) Data Parallelism.

2.1.3 Dataflow vs. Workflow

The term *workflow* is sometime used interchangeably with *dataflow* in many literatures. Though they both refer to a set of processes with data paths describing how data should be moved among them, they have two main differences in *pipeline topology* and *execution strategy*.

Pipeline Topology: a workflow is required to be a directed acyclic graph (DAG) while dataflow is not. As in Figure 2.3a, a pipeline expressed in dataflow is allowed to have feedback connections, *e.g.,*from the Viewer to the Fetch module. In a workflow, the same feedback connection has to be implemented and triggered manually by the developer at run-time.

Execution Strategy: in a workflow, the execution order of modules is usually fixed based on the dependency of data connections. Each time a workflow is executed, every module is also executed exactly once. This is considered to be a *task-driven* approach comparing to the data-driven one of dataflow, where only modules that receive input data, either as a result of an upstream execution or programmatically commands, will be triggered for execution. Figure 2.3b illustrates this difference. On the left, the execution order for the shown workflow is fixed: Fetch, Filter A, Filter B and Viewer. On the right, depending on whether module Filter A and Filter B produce any output data, the Viewer module will or will not be executed accordingly.

2.2 Visualization Systems

Kitware's Visualization ToolKit (VTK) [51] 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

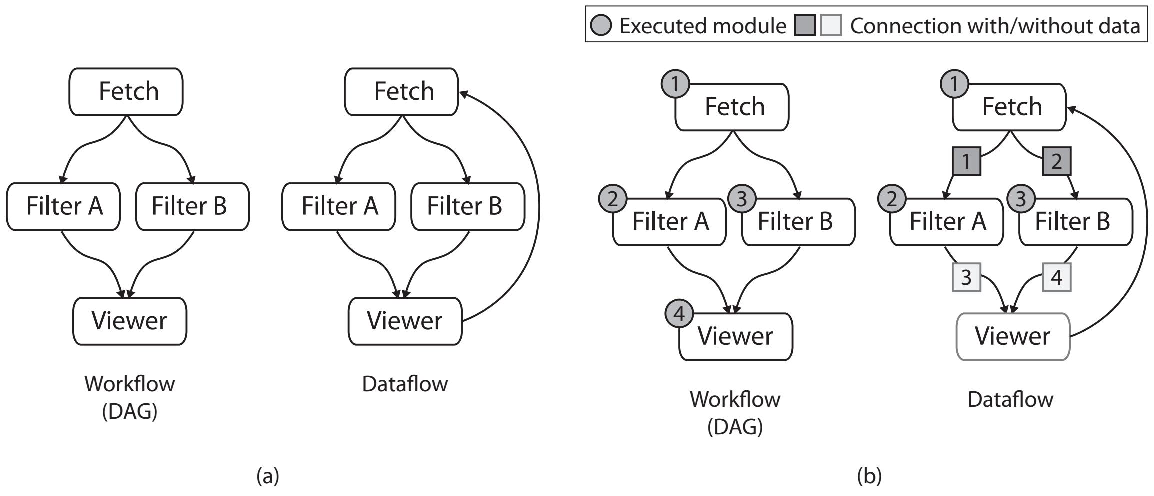


Figure 2.3: **Dataflow vs. Workflow** Two main differences between dataflow and workflow: (a) pipeline topology and (b) execution strategy.

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 multithreaded. This poses a limitation on the performance and scalability for many applications. While much effort [54, 9] 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. [2, 1] 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 [50].

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/multithreaded model; for multicore architectures MPI is also used for creating multiple processes on the same node. This may impose substantial overhead due to the additional expense of interprocess communication.

A related system is VisIt [19]; 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 [11] 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 for demand- and event-driven updating policies. However, it does not target the parallel/high performance perspective of pipeline execution.

Table 2.1 summarizes the feature set for the systems discussed above.

2.3 Heterogeneous Programming Frameworks

There exist several proposals of programming languages and environments to allow combined computation on CPUs and GPUs, such as the Brook [12] programming language, designed for general-purpose computation using GPUs[72]. At the time of its proposal, GPUs were programmable mainly through graphics APIs. Brook introduces a streaming architecture that exposes the GPU as a parallel programmable unit, composed of kernels (programs) that operate on streams (data). Similar to Brook, the Scout [64] programming language was designed with specific data constraints and a target application in the field of visualization and data exploration. Only regular grid structures are supported in Scout, which are directly mapped to OpenGL textures. Glift [55] is an abstraction layer and generic template library for representing complex random-access data structures (*e.g.,* Stacks, Quadtrees and Octrees) on the GPU. Glift components were designed to support data access both on the CPU and GPU using a streaming computation model. With the introduction of the CUDA [70] programming language by NVIDIA (and subsequently OpenCL[53]), the full infrastructure of the GPU was made available for general purpose computation in a single system.

Support for heterogeneous systems is described in several works. Instead of forcing applications to be rewritten into a streaming processing format, HMPP [32, 10] is a programming environment that enhances CUDA programs with a set of compiler directives, tools and software runtime that support multicore CPU parallel programming. HMPP offers a dynamic linking mechanism that allows the GPU to be used as a co-processor in a way that preserves legacy code. The CUDASA programming language [68] extends CUDA to allow distributed computation on multiple GPUs in a local system or across machines on a network. CUDASA introduces abstraction layers with different levels of parallelism. In particular, a bus layer allows manual task delegation to CPU and

Table 2.1: Summary of Current Visualization Dataflow Systems

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

GPU cores in arbitrary combinations, while a network layer adds support for clusters of multiple interconnected computers. Shared memory across network nodes relies on the MPI Remote Memory Access (RMA) protocol. It lacks support for improved load balancing and distributed memory accesses. FastFlow [3] introduces a low-level programming framework based on lock-free queues (no memory fences) to support streaming applications. It allows multiparty communication in multicore architectures, with support for cyclic graphs of thread streaming networks, and is specially designed for fine-grained computations.

Some recent work discusses more advanced scheduling algorithms for controlling parallel applications. Jimenez et al. [46] introduce a predictive user-level scheduler for a CPU/GPU heterogeneous system. They discuss and evaluate several scheduling algorithms that improve performance of of functions that have previously been profiled during execution.

Unfortunately, current programming environments either make use of task-graphs or a similar DAG-based approach in their execution models or only providing a low-level scheduling API, and thus, cannot be extended to current dataflow systems without major modifications.

CHAPTER 3

STREAMING SIMPLIFICATION OF TETRAHEDRAL MESHES

In this chapter, we present our experiment with streaming data structures and algorithms in the context of unstructured grid simplification. Simplification techniques have been a major focus of research for the past decade due to the increasing size and complexity of geometric data. Scientific simulations and measurements from fluid dynamics and partial differential equation solvers have produced datasets that are too large to visualize with current hardware. Thus, approximations which maintain a volumetric mesh are necessary to achieve a level of interactivity that is necessary for proper analysis through visualization techniques such as isosurfacing or direct volume rendering. This experiment also shows the importance of streaming in handling large-scale datasets, thus, guiding us to a generalized streaming data structure that can optimize both performance and memory footprint.

Although significant work has been done in simplifying triangle meshes, relatively little has been done with tetrahedral meshes. Most of the work in tetrahedral simplification falls into two categories: edge-collapse methods and point sampling methods. These algorithms assume that the entire mesh can be loaded into main memory. However, due to the high memory overhead of storing the mesh connectivity in addition to the geometry, there are limitations on the size of the dataset that can be simplified in this manner.

We present an algorithm that *streams* the data from disk through memory and performs the simplification on a localized portion of the entire mesh. Our streaming algorithm requires only one pass to simplify the entire mesh. Thus, the layout of the mesh is of great importance to produce high quality results. We perform a reordering of the tetrahedral cells and store them on disk using a streaming tetrahedral mesh format. This format provides concurrent access to coherently ordered vertices and tetrahedra. It also minimizes the duration that a vertex remains in-core, which limits the memory footprint of the simplification.

Our tetrahedral simplification incrementally works on overlapping portions of the mesh in-core (see Figure 3.1). We use the quadric-error metric to perform a series of edge collapses until a target decimation is reached. This results in a simplification algorithm that can efficiently simplify

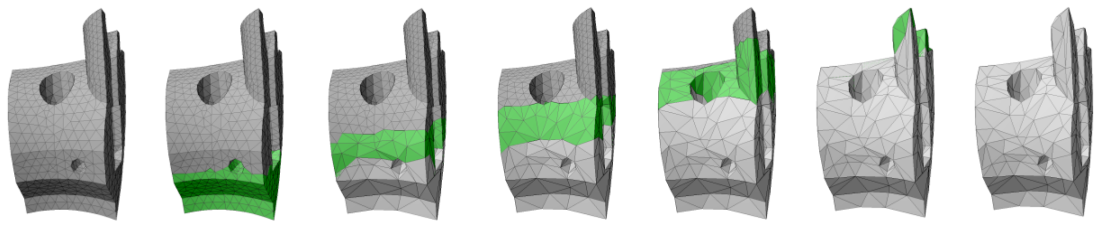


Figure 3.1: **Streaming Simplification** Streaming simplification performed on a tetrahedral mesh ordered from bottom to top. The portion of the mesh that is in-core at each step is shown in green.

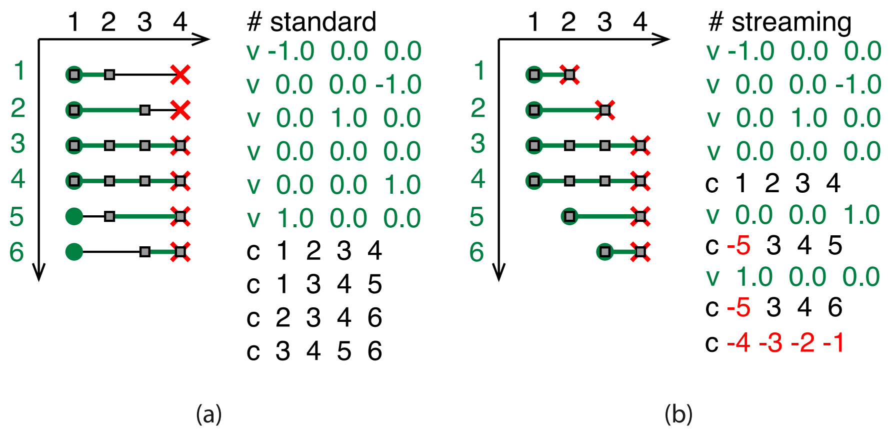
extremely large datasets. In addition, through the use of carefully optimized algorithms, linear solvers, and data structures we show that significant improvements in speed and stability can be achieved over previous techniques.

3.1 Streaming Mesh Layout

Traditional object file formats consist of a list of vertices followed by a list of polygonal or polyhedral elements that are defined by indexing into the vertex list. Dereferencing such a mesh, *i.e.,* accessing vertices via their indices, requires the whole vertex list to be in memory since elements are generally not assumed to reference vertices in any particular order; an element can arbitrarily reference *any* vertex in the list. Furthermore, streaming such a mesh implies buffering all vertices before the first element is encountered in the stream. A logical progression for large meshes is to store them in a streaming mesh representation that interleaves the vertices and elements and stores them in a "compatible" order. This representation allows a vertex to be introduced (added to an in-core active set) when needed and finalized (removed from the active set) when no longer used.

Isenburg and Lindstrom [43] provide a streaming mesh format for triangle meshes that extends the popular OBJ file format. We use their format with straightforward additions to handle tetrahedral meshes. This includes extending the vertices to four values $\langle x, y, z, f \rangle$ that represent the position in 3D space and a scalar value. In addition, we provide a new element type for a tetrahedral cell that indexes four vertices. This format allows us to finalize a vertex when it is no longer in use by using a negative index into the vertex list, which is backwards compatible with the OBJ format. Figure 3.2 shows an ASCII example of the streaming tetrahedral format.

An important consideration with streaming meshes is the ability to analyze the efficiency of a mesh layout. Isenburg and Lindstrom developed several techniques for visualizing properties of the mesh to determine the effectiveness of the layout. We use similar tools to measure tetrahedral mesh quality. An important property is the *front width*, or the maximum number of concurrently active vertices. An *active* vertex is one that has been introduced, but not yet finalized. The width of a streaming mesh gives a lower bound on the amount of memory required for dereferencing (and thus



```
# standard
v -1.0  0.0  0.0
v  0.0  0.0 -1.0
v  0.0  1.0  0.0
v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  1.0  0.0  0.0
c  1  2  3  4
c  1  3  4  5
c  2  3  4  6
c  3  4  5  6
```

```
# streaming
v -1.0  0.0  0.0
v  0.0  0.0 -1.0
v  0.0  1.0  0.0
v  0.0  0.0  0.0
c  1  2  3  4
v  0.0  0.0  1.0
c -5  3  4  5
v  1.0  0.0  0.0
c -5  3  4  6
c -4 -3 -2 -1
```

Figure 3.2: **Mesh Format** Layout diagrams with cell indices on the horizontal and vertex indices on the vertical axis. A vertex is active from the time it is introduced until it is finalized, as indicated by the horizontal lines. (a) A standard format for tetrahedral meshes based on the OBJ format. (b) A streaming format that interleaves vertices with cells. Vertex finalization is provided with a negative index (*i.e.,* relative to the most recent vertex).

processing) the mesh. Another important property of the mesh is the *front span*, which measures the maximum index difference (plus one) of the concurrently active vertices. A low span allows a faster implementation because optimizations can be performed that achieve the best performance when the span is similar to the width. Low span makes for efficient indexing in the file format, bounds the width, and for simplification purposes, ensures that vertices do not become stagnant in the buffer, which would prevent all incident edges from being collapsed. The efficiency of our layout depends on quantifying these properties. Thus we provide analysis on different layout techniques so that we can choose the most streamable layout for a given dataset.

One simple mesh layout is to sort the vertices on a *spatial* direction, in particular one that crosses the most tetrahedra. Wu and Kobbelt [91] use this technique for triangle mesh simplification. This can be accomplished for large meshes by performing an out-of-core sort on the vertices [58] and writing them into a new file. An additional file is created to contain a mapping of the old ordering to the new one. Next, the tetrahedral cells are written to a new file and reindexed according to the mapping file. A sort is then performed on the file containing the tetrahedral cells based on the largest index of each cell. Finally, the vertex file and the cell file are read simultaneously and interleaved into a new file by writing each cell immediately after the vertex corresponding to the cell's largest index has been written. Spatial layouts work especially well when considering meshes that have a dominant principal direction.

Other techniques may be desirable if the mesh does not have a dominant principal direction, such as a sphere. An approach to handle this type of data is to use a *bricking* method similar to the one proposed by Cox and Ellsworth [29] in which the vertices are ordered into a fixed number of small cubes for better sequential access. A similar approach is to arrange the vertices using a Lebesgue space-filling curve (*i.e., z-order*), which provides better sequential access in the average case. This arrangement can be generated by creating an out-of-core octree [22] of the vertices and traversing them in-order. The interleaved mesh is then written to a file in the same manner described above for spatial sorting. The results of the layout produced by bricking and *z*-order traversal are similar. When streamed they provide a more contiguous portion of the mesh on average, but the front width and span are typically much larger than sorting spatially.

Another approach used for laying out the mesh on disk is *spectral sequencing*. This heuristic finds the first nontrivial eigenvector (the *Fiedler vector*) of the mesh's Laplacian matrix and was shown by Isenburg and Lindstrom [43] to be very effective at producing low-width layouts. They provide an out-of-core algorithm for generating this ordering for streaming triangle meshes, which we have extended to handle tetrahedra. This method works particularly well for curvy triangle meshes, but tetrahedral meshes are generally less curvy and more compact. Still, in most cases this ordering results in the lowest width, which is ideal for minimizing memory consumption.

A final approach is to create a *topological* layout, which starts at a vertex on the boundary and grows out to neighboring vertices. To grow in a contiguous manner, we use a breadth-first traversal with optimizations to improve coherence [43]. Instead of a traditional FIFO priority queue, we assign priority using three keys. First, the oldest vertex on the queue is used in the same way that it would be in standard breadth-first algorithms. However, if multiple vertices were added to the queue at the same time, a second and third key are used to achieve a more coherent order. The second key is boolean, and gives preference to a vertex if it is the final one in a cell that has not been processed. Finally, the third key is to use the vertex that was most recently put on the queue, which is more likely to be adjacent to the last vertex. These sort keys guarantee a layout that is *compact* [43], such that runs of vertices are referenced by the next cell, and runs of cells reference the previous vertex (*e.g.,* as in Figure 3.2b). In practice, this traversal can be accomplished out-of-core by breaking the mesh into pieces. Using this approach, we were able to minimize the front span of the datasets in all of our experimental cases. This is ideal because having a span and width that are similar allows us to exploit optimization techniques described in the simplification algorithm. Yoon *et al.* [94] propose a layout based on local mesh optimizations which reduce cache misses. This approach applied to tetrahedra would also give a compact representation as with our breadth-first layout and could be used to achieve similar results.

Table 3.1 shows the front width and span for five different datasets (their statistics are specified in Table 3.2) produced by the layout techniques described above. Spectral sequencing proves to be the superior choice when low width and thus memory efficiency is required. Breadth-first layouts

Table 3.1: Analysis of Mesh Layout

Dataset	Spatial Sort		Z-Order		Spectral		Breadth-First	
	Width	Span	Width	Span	Width	Span	Width	Span
Torso	3,118	20,784	7,256	122,174	**2,894**	13,890	5,528	**6,370**
Fighter	**3,894**	110,881	9,382	215,697	3,916	28,638	16,629	**19,523**
Rbl	2,814	5,270	10,232	371,269	**2,291**	21,764	3,206	**3,495**
Mito	19,876	33,524	10,202	642,550	**6,745**	44,190	10,552	**11,498**
SF1	16,898	65,921	48,532	1,958,212	**12,851**	131,152	30,258	**33,378**

Table 3.2: Dataset Statistics

Dataset	Vertices	Tetrahedra
Torso	168,930	1,082,723
Fighter	256,614	1,403,504
Rbl	730,273	3,886,728
Mito	972,455	5,537,168
SF1	2,461,694	13,980,162

are not as memory-efficient, but as we will see can be processed fast. Note that unlike [45] we do not require a face-connected order and we do not require that each vertex be finalized before it can be inserted into the in-core mesh. This allows us to avoid local reordering of the tetrahedra or the vertices.

3.2 Tetrahedral Simplification

3.2.1 Quadric-Based Simplification

To achieve high-quality approximations, we use the quadric error metric proposed by Garland and Zhou [36]. This metric measures the squared (geometric and field) distances from points to hyperplanes spanned by tetrahedra. The volume boundaries are preserved using a similar metric on the boundary faces and by weighting boundary and interior errors appropriately.

The generalized quadric error allows the flexibility of representing field data by extending the codimension of the manifold. Given a scalar function $f : D \subseteq \mathbb{R}^3 \rightarrow \mathbb{R}$ defined over a domain D represented by a tetrahedral mesh, we can represent the vertices at each point $\mathbf{p}$ as $\langle x_p, y_p, z_p, f_p \rangle$, which can be considered a point on a 3D manifold embedded in $\mathbb{R}^4$. Thus, by extending the quadric to handle field data, the algorithm intrinsically optimizes the field approximation along with the geometric position.

The quadric error of collapsing an edge to a single point is expressed as the sum of squared distances to all accumulated incident hyperplanes, and can in n dimensions be encoded efficiently as a symmetric $n \times n$ matrix $\mathbf{A}$, an n-vector $\mathbf{b}$, and a scalar c. It is sufficient to component-wise add these terms to combine the quadric error of two collapsed vertices. Finding the point $\mathbf{x}$ that minimizes this measure amounts to solving a linear system of equations $\mathbf{Ax} = \mathbf{b}$. Once $\mathbf{x}$ is computed, we test whether collapsing to this point causes any tetrahedra to flip [20], *i.e.,* changes the sign of their volume, in which case we disallow the edge collapse. Because $\mathbf{A}$ is not necessarily invertible, it is important to choose a linear solver that is numerically stable.

3.2.2 Streaming Simplification

Combining streaming meshes with quadric-based simplification, we introduce a technique for simplifying large tetrahedral meshes out-of-core. We base our streaming algorithm on [45] and [43], but make several general improvements and provide a list of optimizations that compared to a less carefully engineered implementation results in dramatic speed improvements.

First, unless already provided with streaming input, we convert standard indexed meshes and optionally reorder them for improved streamability. Then, portions of the streaming mesh are loaded incrementally into a fixed-size main memory buffer and are simplified using the quadric-based method. Once the in-core portion of the mesh reaches the user-prescribed resolution, simplified elements are output, *e.g.,* to disk or to a downstream processing module. Thus input and output happen virtually simultaneously as the mesh streams through the memory buffer (see Figure 3.3).

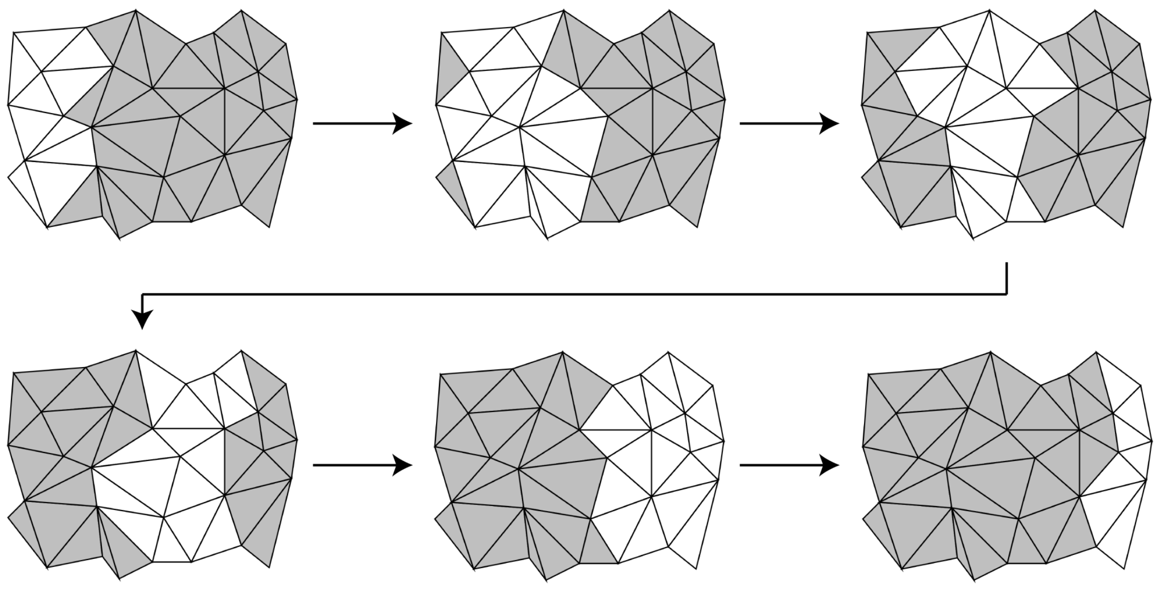


Figure 3.3: **Streaming Simplification Buffer** Example of a buffer moving across the surface of a tetrahedral mesh sorted from left to right. As new tetrahedra are introduced, the tetrahedra that have been in-core the longest are removed from memory.

To ensure that the final approximation is the desired size, two control parameters have been added: *target reduction* and *boundary weight*. Target reduction is the ratio between the number of tetrahedra in the output mesh and the number of tetrahedra in the original mesh. Alternatively, this parameter can be expressed as a target tetrahedral count of the resulting mesh. The boundary weight prevents the shape of the mesh from changing throughout the simplification. We use a fixed value of 100 times the maximum field value in the data for the weight in our experiments. Similar to [36], the scalar field is always normalized to the geometry range before actual simplification begins.

Because we only keep a small set of tetrahedra in memory, we do not know the entire mesh connectivity. Thus, we keep the boundary between the set of tetrahedra that are currently in memory and all remaining elements—tetrahedra that have not yet been read or that have been output—fixed to ensure that the simplified mesh is crack-free. We call this boundary the *stream boundary*, which consists entirely of faces from the interior of the mesh. We can identify the stream boundary faces as they are read in by utilizing the finalization information stored in the streaming mesh. A face of the current in-core mesh is part of the stream boundary if none of its three vertices are finalized. We disallow collapsing any edge that has one or both vertices on the stream boundary.

Due to the stream boundary constraint, if we read in one portion of the mesh, simplify it, and write it out to disk in one phase, our output mesh will have unsimplified areas along the stream boundaries. This results in an approximation that is oversimplified in areas and under-simplified in others. To avoid this problem, we follow the algorithm proposed by Wu and Kobbelt [91]. Their algorithm consists of a main loop in which READ, DECIMATE, and WRITE operations are performed in each iteration. The READ operation introduces new elements until it fills the buffer. Next, DECIMATE simplifies the elements in the buffer until either the target ratio is reached or the buffer size is halved. Finally, in their method the WRITE operation outputs the elements with the largest error to file.

As in [43], we improve upon [91, 45] by ensuring, to the extent possible, that the relative order among input elements is preserved in the output stream, with the caveat that tetrahedra whose vertices have not yet been finalized (*i.e.,* are on the input stream boundary) must be delayed. Therefore, the output typically retains the coherence of the input. An error-driven output criterion, on the other hand, can considerably fragment the buffer and split off small “islands” that remain in the buffer for a long time without being eligible for simplification, and thus unnecessarily clog the stream buffer. Furthermore, such an output stream generally has poor stream qualities, which affects downstream processing. The front width (*i.e.,* number of active vertices), for example, is particularly important for tetrahedral meshes, for which each active vertex affects on average four times as many elements as in a triangle mesh, and therefore more adversely affects memory requirements and processing delay. Furthermore, we relax the requirement that the stream of tetrahedra (triangles) advance in a face (edge) adjacent manner [45], as this is of no particular value to us, and we allow any coherent ordering of mesh elements. Finally, using the more streamable layouts and simpler streaming mesh formats and API from [43], we gain considerably in performance and memory usage over [45].

3.3 Implementation Details and Optimizations

Since our method processes different mesh portions of bounded size sequentially, a statically allocated data structure is more efficient than dynamic allocations, which collectively increase the memory footprint. The size for this buffer should be $O(width)$ depending on the width of the input mesh. However, in practice, we are able to simplify even a 14 million tetrahedra dataset using only 20MB of RAM (see Section 3.4).

In our implementation, we extended Rossignac's corner table [76] for triangle meshes to tetrahedral meshes. The original corner table requires two fixed-size arrays V and O indexed by corners (vertex-cell associations) c, where $V[c]$ references the vertex of c and $O[c]$ references the "opposite" corner of c.

In the case of tetrahedral simplification, the most common query is to find all tetrahedra around a vertex. Therefore, we replace the O array with a link table L of equal size, which joins together all corners of a given vertex in a circular linked list. We additionally store with each vertex an index to one of its corners.

We store the mesh internally as three fixed-size arrays of vertices, tetrahedra, and corners (*i.e.,* the links L). Each vertex contains a pointer to one corner and the quadric error information $(\mathbf{A}, \mathbf{p}, \varepsilon)$ using a parameterization that explicitly represents the vertex geometry and scalar data in $\mathbf{p}$. In Figure 3.4, only $\mathbf{p}$ and *vidx* are stored out-of-core while the rest are computed on-the-fly. This data structure employs 69 bytes per vertex and 37 bytes per tetrahedra. The quadrics for the tetrahedra are calculated when we read in a new set of tetrahedra, and are then distributed to the vertices. For each finalized vertex, we compute boundary quadrics for all incident boundary faces (if any) that have no other finalized vertices, and distribute these quadrics to the boundary vertices.

Garland and Zhou [36] use a greedy edge collapse method and maintain a priority queue for the edges ordered by quadric error. Forsaking greediness, we obtain comparable mesh quality by using a multiple choice randomized approach [91, 30] with eight candidates per collapse. There are several advantages of using randomized selection. One is that we no longer need a priority queue or explicit representation of edges. Instead an edge can be found by randomly picking a tetrahedron and then randomly selecting two of its vertices. Another advantage is that the randomized technique can be further accelerated by exploiting information readily available through our quadric represen-

VERTEX		
	float[10] **A**	quadric matrix
	float[4] **p**	position and field value
	float ε	quadric error at **p**
	int *idx*	index to input/output stream
	int *corner*	vertex-to-corner index
	bool *deleted*	
	bool *written*	
TETRAHEDRON		
	int[4] *vidx*	vertex indices
	int[4] *link*	corner links
	bool *deleted*	
	int *idx*	position in input stream

Figure 3.4: **QEM Data Structure** Data structures used for our quadric-based simplification.

tation. Table 3.3 illustrates the performance of the randomized approach over the priority-queue based approach. Both algorithms simplified the models to 10% of their original resolutions. The randomized results were collected as an average of 3 runs on the same input with different random seeds. Intuition would suggest that the results generated by the priority-queue approach would have smaller error because it always picks the edge with the smallest error to collapse at each step. However, this is not optimal in many cases. A series of minimal edge collapses can lead to a locked state where the edges with the smallest error cannot be collapsed without flipping tetrahedra. In practice, the problem is more likely to occur in the homogeneous regions of a dataset, which contains edges with zero error. The priority-queue approach will greedily simplify this region as much as possible first, leaving it in a locked state. As a result, many neighbor regions (containing edges with small error) may not get simplified because it would violate the flipping constraint. On the other hand, the randomized approach tends to spread the simplification over the whole model, resulting in significantly less locked state. This explains why the maximum error of the Torso dataset using the priority queue approach was very high compared to the randomized approach. In general, the randomized approach produces comparable quality to a priority queue, while demonstrating superior performance.

Before we output a tetrahedron, we must ensure that its four vertices are output first. Once a vertex is output, we mark it as not being collapsible in future iterations. To enhance the performance, we use a lazy deletion scheme, where all vertices and tetrahedra to be deleted are initially marked. At the end of each WRITE phase, we make a linear pass through all vertices and tetrahedra to remove marked elements and compact the arrays. Since we do not allocate additional memory during simplification, keeping deleted vertices and tetrahedra does not increase the memory footprint.

Table 3.3: Priority-Queue (P) vs. Randomized (R) Approach

Model		Mean	RMS	Max	Time
Torus	P	0.08%	.0.13%	0.56%	0.31s
	R	0.06%	0.10%	0.59%	0.13s
SPX	P	1.37%	2.56%	22.60%	0.52s
	R	1.53%	2.19%	15.96%	0.30s
Torso	P	0.00%	0.02%	1.12%	55.98s
	R	0.00%	0.00%	0.01%	24.11s
Fighter	P	0.06%	0.13%	2.28%	79.03s
	R	0.08%	0.16%	3.46%	29.89s

Storing large datasets on disk in ASCII format can adversely affect performance because converting ASCII numbers to an internal binary representation can be surprisingly slow. We have extended the ASCII stream format in Figure 3.2 to a binary representation. Because our program spends over 30% of the time on disk I/O, this optimization results in a nonnegligible speedup. For example, on the SF1 dataset it improves overall performance by 17%.

Since we only maintain a small portion of the mesh in-core, we require a way of mapping global vertex indices to in-core buffer indices. Usually a hash map is used, but with our low-span breadth-first mesh layout, this hash map can be replaced by a fixed-size array indexed using modular arithmetic. We move occasional high-span vertices that cause "collisions" in this circular array to an auxiliary hash [43].

With all of the optimizations described above, our simplifier can run at high speed without any dynamic memory allocation at run time. The performance and memory summary can be found in Table 3.4. The results are for simplifying the Fighter dataset (1.4 M tetrahedra) completely in-core with 1GB of RAM.

Table 3.4: Implementation Improvements

Improvements	Time (sec)	Memory (MB)
Initial Implementation	212.95	310
QEF + CG	132.68	282
Multiple Choice + Corner Table	38.35	130
4D Normal, Floats	35.99	78
Final / Binary I/O	22.10	78

3.3.1 Numerical Issues

Great care has to be taken when working with quadric metrics to ensure numerical stability while retaining efficiency. To minimize quadric errors, a positive semidefinite system of linear equations must be solved, for which numerically accurate but heavy-duty techniques such as singular value decomposition (SVD) [56, 52] and QR factorization [48] have been proposed. However, even constructing, representing, and evaluating quadric errors require that special care be taken. We here outline an efficient representation of quadric error functions that leads to numerically stable operations, improved speed, and less storage.

The standard representation [36] of quadric errors is parameterized by $(\mathbf{A}, \mathbf{b}, c)$, and is evaluated as

$$Q(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x} - 2\mathbf{b}^\mathsf{T}\mathbf{x} + c \tag{3.1}$$

Typically the three terms in this equation are "large," but sum to a "small" value, resulting in a loss of precision. One can show that the roundoff error is proportional to $\|\mathbf{A}\|\|\mathbf{x}\|^2$. Furthermore, in addition to this quadric information, it is common to store the vertex position (and field value) $\mathbf{p}$ that minimizes Q separately. Lindstrom [57] suggested an alternative representation that removes this redundancy:

$$Q(\mathbf{x}) = (\mathbf{x} - \mathbf{p})^\mathsf{T}\mathbf{A}(\mathbf{x} - \mathbf{p}) + \varepsilon \tag{3.2}$$

where $\mathbf{A}$ is the same as in the standard representation and

$$\begin{aligned} \mathbf{A}\mathbf{p} &= \mathbf{b} \\ \mathbf{p}^\mathsf{T}\mathbf{A}\mathbf{p} + \varepsilon &= c \end{aligned}$$

This parameterization $(\mathbf{A}, \mathbf{p}, \varepsilon)$ provides direct access to the minimum quadric error ε and the minimizer $\mathbf{p}$. This not only saves memory but also results in a more stable evaluation of Q, as the roundoff error is now proportional to $\|\mathbf{A}\|\|\mathbf{x} - \mathbf{p}\|^2$, and we are generally interested in evaluating $Q(\mathbf{x})$ near its minimum $\mathbf{p}$ as opposed to near the origin. Another significant benefit of this representation is that it provides a lower bound $\varepsilon_i + \varepsilon_j$ on $Q_i + Q_j$ when collapsing two vertices v_i and v_j. Using randomized edge collapse [91], we can thus often avoid minimizing $Q_i + Q_j$ if the lower bound already exceeds the smallest quadric error found so far. In this chapter, this representation is used explicitly to speed up the algorithm, reduce in-core storage, and improve numerical robustness rather than as a means of compressing quadric information for out-of-core storage.

Our quadric representation also lends itself to an efficient and numerically stable iterative linear solver. To handle ill-conditioned matrices $\mathbf{A}$, we have adapted the well-known conjugate gradient (CG) method [37] to work on semidefinite matrices (see Figure 3.5). As in SVD, we provide a tolerance κ_{max} on the condition number $\kappa(\mathbf{A})$, and preempt the iterative solver when all remaining

```
SOLVE($\mathbf{A}$, $\mathbf{x}$, $\mathbf{b}$, $n$, $\kappa_{max}$)
 1  $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$                              negative gradient of $Q$
 2  $\mathbf{p} = \mathbf{0}$
 3  for $k = 1, \ldots, n$                          iterate up to $n$ times
 4      $s = \mathbf{r}^\mathsf{T}\mathbf{r}$
 5      if $s = 0$ then exit                        solution found?
 6      $\mathbf{p} = \mathbf{p} + \mathbf{r}/s$                            update search direction
 7      $\mathbf{q} = \mathbf{Ap}$
 8      $t = \mathbf{p}^\mathsf{T}\mathbf{q}$
 9      if $st \leq \mathrm{tr}(\mathbf{A})/(n\kappa_{max})$ then exit    insignificant direction?
10      $\mathbf{r} = \mathbf{r} - \mathbf{q}/t$                            update gradient
11      $\mathbf{x} = \mathbf{x} + \mathbf{p}/t$                            update solution
```

Figure 3.5: **Conjugate Gradient Solver** Conjugate gradient solver for positive semidefinite systems $\mathbf{Ax} = \mathbf{b}$. On input $\mathbf{x}$ is an estimate of the solution, $n = 4$ is the number of linear equations, and κ_{max} is a tolerance on the condition number. $\mathrm{tr}(\mathbf{A})$ is the trace of $\mathbf{A}$.

conjugate directions are deemed "insignificant" for reducing Q. The effect of this is similar to zeroing small singular values in SVD. Using our quadric representation, we conveniently initialize the CG solver with the guess $\mathbf{x} = (\mathbf{p}_i + \mathbf{p}_j)/2$. Whereas CG methods are typically used to quickly approximate solutions to very large systems using only a few iterations, our method can be considered "direct" in the sense that we solve for each of the n (*i.e.,* 4) components, although in the Krylov basis rather than in the Euclidean basis as done by the Cholesky method. We never require more than n iterations, and only in the rank-deficient case do we perform fewer than n iterations.

A final word of caution: The computation of generalized quadrics presented in [36] computes **A** = **I** - **N**, whose null space $\mathrm{null}(\mathbf{A}) = \mathrm{range}(\mathbf{N})$ is spanned by the tetrahedron, via subtraction, which due to roundoff error can leave **A** indefinite, *i.e.,* with one or more negative eigenvalues. This causes Q to have a "saddle" shape with no defined minimum, and can cause numerical instability. Instead, we compute a 4D "volume normal" using a generalization of the 3D cross product to 4D.

$$\mathbf{n} = det \begin{pmatrix} \mathbf{e_1} & \mathbf{e_2} & \mathbf{e_3} & \mathbf{e_4} \\ v1_x & v1_y & v1_z & v1_s \\ v2_x & v2_y & v2_z & v2_s \\ v3_x & v3_y & v3_z & v3_s \end{pmatrix}$$

where $\mathbf{e_i}$ is the i-column of the 4×4 identity matrix and **v1**, **v2** and **v3** are three vectors from one of the tetrahedron's vertices to the others. The outer product of this normal with itself gives a positive semidefinite **A** for a tetrahedron.

Because of our attention to numerical stability, with $\kappa_{max} = 10^4$ we are able to use single precision floating point throughout our simplifier, even for the largest meshes. Since the 4D quadric information requires 15 scalars per vertex, this saves considerable memory and improves the speed.

3.4 Results

3.4.1 Stability and Error Analysis

We have described a CG method for solving the linear equations that arise when minimizing the quadric error. The choice of solver is important because degenerate tetrahedra and regions of near-constant field value can cause singularity. For testing purposes, we constructed a dataset by subdividing a tetrahedron into hundreds of smaller tetrahedra by linear interpolating the vertices and field data. Obviously, these small tetrahedra all lie on the hyperplane spanned by the original tetrahedron. Thus they are solutions to the linear equations. We then picked a solution as a target for each collapsed edge. We experimented with several linear solvers as shown in Table 3.5. The results are for simplifying the Fighter dataset (1.4 M tetrahedra) completely in-core. We experimented with Cholesky factorization, the least square QR factorization [37], and CG.

Cholesky with pivoting provides stable solutions for solving $\mathbf{Ax} = \mathbf{b}$ if $\mathbf{A}$ is positive definite. However, in order to solve this linear system when $\mathbf{A}$ has rank-deficiency (the semidefinite case), we must solve the under-constrained least square problem. Unfortunately, solving this using normal equations requires us to be able to perform Cholesky factorizations on matrices with arbitrary dimensions less than 4×4. This defeats the purpose of optimizing our simplification for working only with 4×4 matrices. Thus, our implementation uses SVD to handle rank-deficiency matrices detected by the Cholesky method. As a result, this method yields the fastest solution when $\mathbf{A}$ is positive-definite but it becomes slower compared to CG when handling rank-deficient matrices.

Like Cholesky, QR does not handle the problem of rank-deficiency. Nevertheless, the implementation for solving the least square problem using QR factorization is much simpler than using Cholesky with normal equations since it does not explicitly require a general representation of matrices with arbitrary dimensions. We use the least square version of QR as suggested in [37].

Using all of our solvers, we were able to simplify our subdivided tetrahedron to its original shape with small error in both field and geometry. To compare the correctness of their solutions, we recorded *norm-2* residuals of computed solutions to $15,000$ linear systems using all 3 methods while simplifying the fighter dataset. Denote by e_{qr}, e_{ch} and e_{cg} the sum of all norm-2 residuals of computed solutions $\hat{\mathbf{x}}$ (*i.e.,* $\|\mathbf{Ab} - \hat{\mathbf{x}}\|_2$) using QR, Cholesky and CG, respectively. Given $e_{avg} =$

Table 3.5: Performance of Linear Solvers

Dataset	QR	Cholesky	CG
SPX	0.57s	0.30s	0.35s
Blunt	16.47s	9.88s	7.72s
Fighter	46.11s	30.57s	29.89s

$\frac{1}{3}(e_{qr}+e_{ch}+e_{cg})$, relative errors of solutions computed by QR, Cholesky and CG can be computed as $re_{\{qr,ch,cg\}} = e_{\{qr,ch,cg\}}/e_{avg}$. Our experiment found re_{qr} to have the smallest error with *96%*, followed by re_{ch} with *99%*. Our CG approach obtained a comparable result with re_{ch} of *105%*.

Overall, QR gives the most optimal solution in terms of error, but it is approximately twice as slow as the others. On the other hand, while Cholesky is a good choice for both efficiency and accuracy, its implementation is quite complicated. Therefore, we chose CG over the others for its simplicity and performance while still maintaining comparatively optimal solutions.

To estimate the error in the simplified mesh we use two different methods. The first method is to measure the error on the surface boundary of the mesh using the tool *Metro* [23]. The second method is to measure the error in the field data using a similar approach to Cignoni *et al.* [21]. We sample the domain of the simplified dataset at points inside the domain of the original one. These points are not only vertices of the input but also interpolated ones inside each tetrahedron. The error is then computed by the differences between their scalar values. Our implementation differs because it ignores points outside the domain of the simplified mesh since these points become part of the surface boundary error. Table 3.6 shows these measured error estimations. Field error percentages are in relation to the range of the field and surface error percentages are in relation to the bounding box diagonal. Figure 3.6 shows an example of the quality of the resulting field and Figure 3.7 shows an example of the quality of the resulting surface.

Table 3.6: In-Core and Streaming Simplification Results

Dataset	Number of Tets Input	Number of Tets Output	Time (sec)	Max RAM (MB)	Field Error Max (%)	Field Error RMS (%)	Surface Error Max (%)	Surface Error RMS (%)
In-core								
Torso	1,082,723	108,271	14.88	57	0.012	0.000884	0.120	0.013360
Fighter	1,403,504	140,348	15.46	78	4.845	0.280266	0.038	0.000352
Rbl	3,886,728	388,668	59.10	212	0.020	0.002574	0.025	0.000055
Mito	5,537,168	553,711	47.13	285	0.045	0.007355	0.001	0.000008
SF1	13,980,162	1,398,013	191.69	709	5.626	0.262335	0.036	0.000811
Streaming								
Torso	1,082,723	108,270	19.07	20	0.019	0.000879	0.161	0.001226
Fighter	1,403,504	140,345	20.87	20	4.549	0.299081	0.102	0.000470
Rbl	3,886,728	388,671	95.54	20	0.025	0.002833	0.036	0.000089
Mito	5,537,168	553,716	73.58	20	0.045	0.007614	0.044	0.000009
SF1	13,980,162	1,398,012	246.15	20	23.287	0.472869	0.169	0.004150
SF1	13,980,162	1,398,017	244.42	50	5.834	0.315583	0.050	0.001394

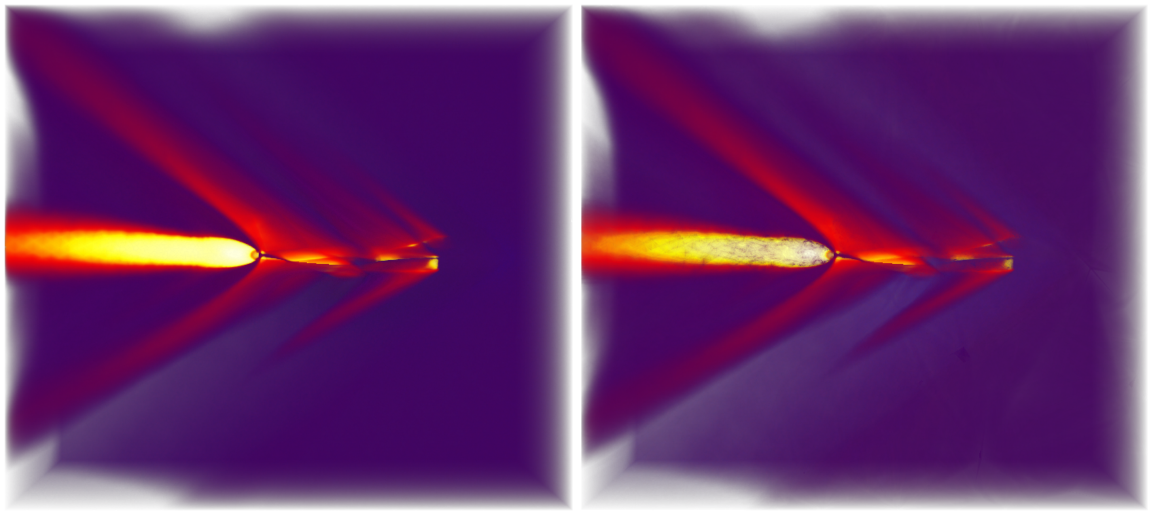


Figure 3.6: **Simplified Scalar Result** Volume rendered images of the Fighter dataset show the preservation of scalar values. The original dataset is shown on the left (1,403,504 tetrahedra) and the simplified version is shown on the right (140,348 tetrahedra).

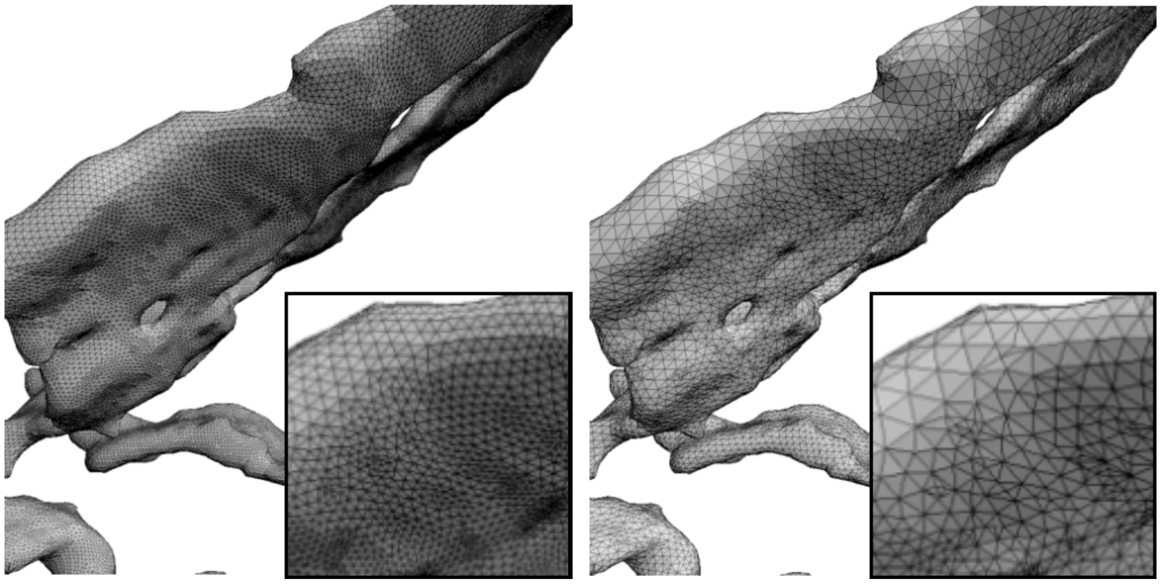


Figure 3.7: **Simplified Surface Result** Views of the mesh quality on the surface of the Rbl dataset. The original dataset is shown on the left (3,886,728 tetrahedra) and the simplified version is shown on the right (388,637 tetrahedra).

3.4.2 Performance

All timing results were generated on a 3.2 GHz Pentium 4 machine with 2.0 GB RAM. For the streaming experiments, we limit the operating system to only 64 MB RAM by using the Linux bootloader. Table 3.6 shows the results of simplifying a collection of datasets to 10% of their original size using our streaming algorithm and the same implementation optimized for in-core execution. Laying out the meshes in a stream efficient manner is a one-time operation and can be performed in-core for all the datasets we tested. Even the largest dataset (14 million tetrahedra) required only about 40 minutes to layout using our Breadth-First approach.

We were able to achieve streaming simplification with only a slight increase in time and error compared to an in-core implementation. The streaming technique has the advantage of a smaller memory footprint. With our algorithm, we were able to simplify 14 million tetrahedra while only using 20 MB RAM. Due to the large size of the SF1 dataset, certain parts of the stream were not able to be simplified accurately, resulting in a larger error. By increasing the memory slightly, the quality of the simplification is greatly improved and approaches the in-core quality. This behavior is not due to the randomization algorithm since the large buffer size always produces better quality outputs even with random seeds. Instead, the quality is improved because each set of candidates has a wider range to select their targets. Consider a set of expensive edges that are larger than the buffer size, any edge collapse will result in a large error no matter how random the target is. However, if we increase the buffer size such that the buffer is larger than the expensive edges, randomized edge collapses will take those edges that are not so expensive into account, thus improving the quality of the output.

3.4.3 Large-Scale Experiment

Although extremely large meshes exist, it is difficult to obtain unclassified access to them. To stress our algorithm on current PC hardware and to demonstrate the scalability of the technique, we performed streaming simplification on a huge fluid dynamics dataset on a Xeon 3.0GHz machine. The dataset was created from sampling slices of a 2048^3 simulation and consists of over one billion tetrahedra that use 18 GB of disk space when stored in the binary format. The tetrahedra were laid out in the order in which they were sliced, which is comparable to sorting by axis. An in-core approach to simplifying this dataset would require a machine with 64 GB RAM. We were able to simplify the data using only 829 MB RAM to 12 million tetrahedra (1.2%) in 10 hours on our test machine. Because error estimations were not possible with the entire dataset in-core, we computed the field error on subregions of the mesh separately to verify the results. In the regions that were measured, there was 10.85% maximum and 1.57% RMS field error. Figure 3.8 shows an isosurface extracted from the original and simplified fluid dynamics dataset.

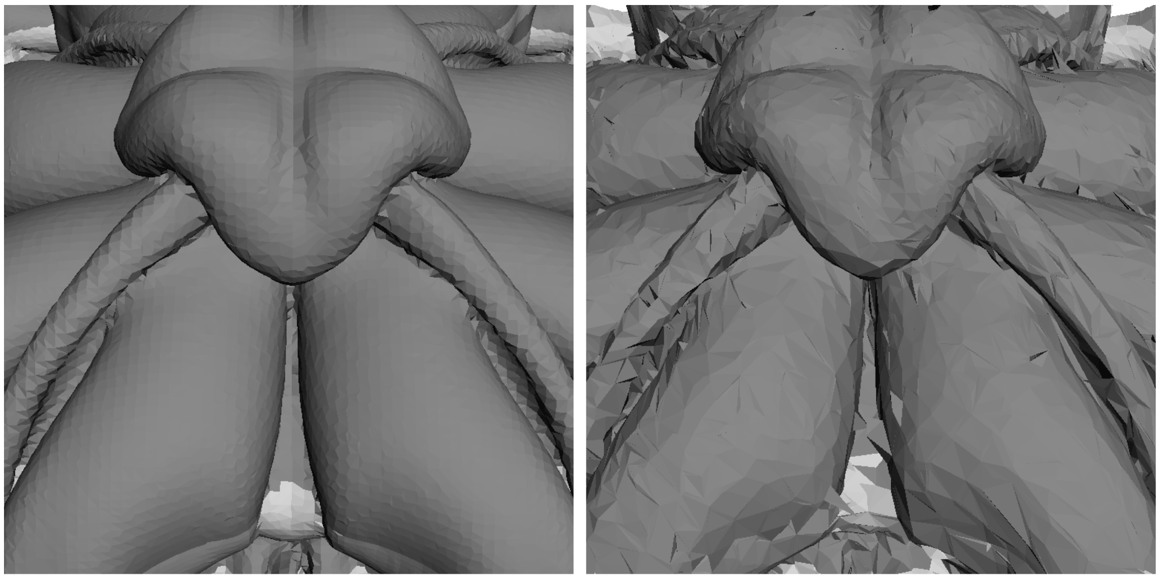


Figure 3.8: **Large-Scale Simplified Simulation Dataset** Isosurfaces of the fluid dynamics dataset. A very small portion of the isosurfaces is shown for the original dataset of over a billion tetrahedra (left) and the simplified dataset of only 12 million tetrahedra (right). The isosurfaces are shown up close using flat shading to enhance the details of the resulting surface. Our algorithm allows extensive simplification (almost 1%) with negligible numerical error (1.57% RMS) for the fluid dynamics dataset which is too large to simplify with conventional approaches.

3.5 Proposition

The use of streaming meshes for simplification reduces the memory footprint of a large mesh considerably. Nevertheless, the streaming format is general enough to work on meshes of other types, *e.g.,* hexahedra. It would be interesting to take advantage of the coherent streaming format to perform other visualization techniques fast in an out-of-core fashion, or simply improving the performance by increasing coherency with lower memory footprint.

CHAPTER 4

INTERACTIVE RENDERING OF LARGE UNSTRUCTURED GRIDS

In this chapter, we present *iRun*, a system for interactively volume rendering large unstructured grids on commodity PC clusters. This work helps us determine crucial requirements for a large-scale data-parallel visualization system. Since iRun is implemented on top VTK, many limitations of current visualization systems, such as VTK, are also revealed to us.

Interactive rendering of arbitrarily large datasets is a fundamental problem in computer graphics and scientific visualization, and a critical capability for many real applications. Interactive visualization of large datasets poses substantial challenges (see survey by Silva *et al.* [81]). Current systems for rendering large datasets employ many of the elements proposed by Clark [24] including the use of hierarchical spatial data structures, level-of-detail (LOD) management, hierarchical view-frustum and occlusion culling, and working-set management (geometry caching). Systems along the lines of the one envisioned by Clark have been used effectively in industrial applications for scenes composed primarily of polygonal geometry.

For more complex scenes, such as those composed of tetrahedral elements, the problem is not as well studied and can be more difficult for several reasons. First, rendering tetrahedra is not natively supported by current graphics hardware. Thus, efficient algorithms for handling this type of data robustly are required. Second, tetrahedra must be projected in visibility order to accurately composite transparency. This requires special care to traverse the out-of-core hierarchy in the correct order. Finally, visibility techniques such as occlusion culling are not practical because the opacity of the volume is controlled by the user. iRun addresses these issues while still maintaining interactivity on extremely large dataset.

The visualization pipeline may be broken down into four major stages: retrieval from storage, processing in main memory, rendering in the Graphics Processing Unit (GPU), and display on the screen. The performance of each of these stages is limited by several potential bottlenecks (*e.g.,* disk or network bandwidth, main memory size, GPU triangle throughput, and screen resolution). iRun uses out-of-core data management and speculative visibility prefetching to maintain a working-set of the geometry in memory. Our rendering approach uses GPU-assisted volume rendering with a dynamic set of tetrahedra and uses an out-of-core LOD traversal. Finally, our system was implemented in VTK [51] and allows distributed rendering for high-resolution displays. Using a single commodity PC, we show how our system can render datasets consisting of 36 million tetrahedra while maintaining interactive frame rates.

4.1 Interactive Out-of-Core Volume Rendering

iRun interactively renders large unstructured grids in several stages, as illustrated in Figure 4.1. First, a preprocessing step prepares the data for hierarchical traversal. Second, our algorithm interactively traverses the out-of-core data structure and keeps a working-set (geometry cache) of the geometry in memory by using visibility culling, speculative prefetching, and LOD management. Finally, the contents of the geometry cache are rendered using a hardware-assisted visibility sorting algorithm. This scales to multiple PCs for improved image quality or large display capability. The iRun design inspired by iWalk [28], however, requires a more complex solution for volume rendering on unstructured grids. Due to many similarities in the two approaches, we defer comparisons to Section 4.3.3.

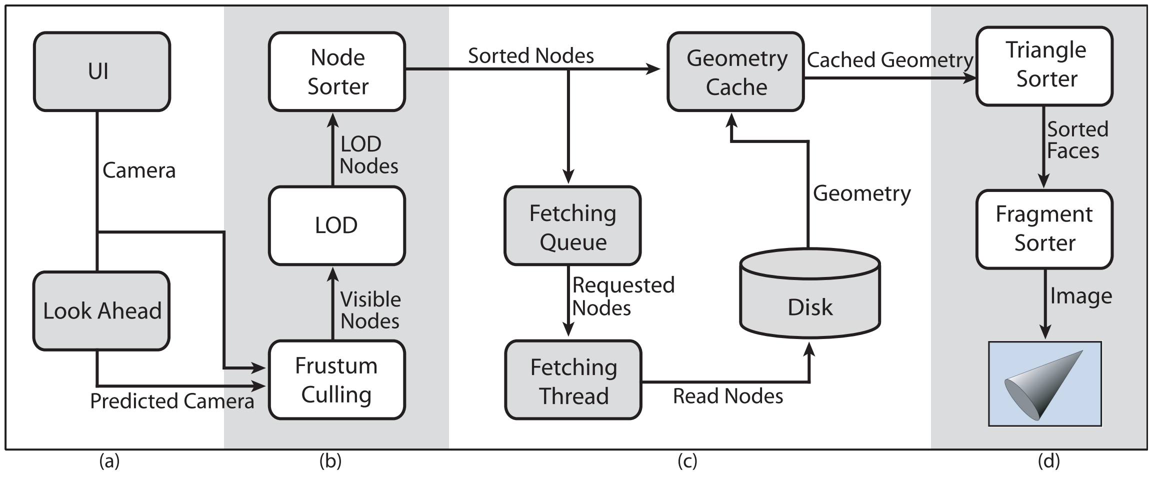


Figure 4.1: **iRun Overview**. (a) The user interacts with the UI by changing the camera from which our system predicts future camera positions. (b) Our octree traversal algorithm selects the octree nodes that are in the frustum, determines the appropriate LOD, and arranges the nodes in visibility order. (c) The geometry cache keeps the working-set of triangles while a separate thread is used to fetch additional geometry from disk. (d) Finally, the geometry is sent to the renderer for object-space sorting followed by a hardware-assisted image-space sorting and compositing which is performed using a modified version of the HAVS algorithm.

4.1.1 Preprocessing

iRun utilizes an efficient and fully automatic preprocessing algorithm that operates out-of-core for large datasets. We begin by extracting the unique triangles that compose the tetrahedral mesh. This is done out-of-core by writing the four triangle indices of each tetrahedron into a file and using an external sort to arrange the indices from first to last. The resulting file contains adjacent duplicate entries for faces that are on the inside of the mesh and unique entries for boundary triangles. A cleanup pass is performed over the triangle index file to remove duplicates and to create a similar file that contains a boundary predicate for each triangle.

iRun employs an out-of-core, hierarchical octree [77] in which each node contains an independent renderable set of vertices and triangles, similar to iWalk. Because the number of vertices in a tetrahedral mesh is generally much smaller than the connectivity information, for simplicity, we keep the vertex array in-core while creating our out-of-core hierarchy. This allows us to keep the global indexing of the triangles throughout our preprocessing which facilitates filtering in the final stages. Our octree is constructed by reading the triangle index and boundary predicate files in blocks and adding the triangles one-by-one to the out-of-core octree structure.

While adding triangles to the octree, we perform triangle-box intersection to determine one or more nodes that contain the triangle. If the triangle spans multiple nodes, we replicate and insert it into each. This can lead to the insertion of a triangle into a node where any or all of the triangle vertices lie outside the node. When a node reaches a preset capacity the node is split into octants and the triangles are redistributed. The result of this phase is a directory structure representing the octree and a *hierarchy structure* file that contains the octree structure information (see [28]).

A subsequent LOD propagation stage works by populating a parent node with a subset of the triangles that are not on the boundary from each of the children. Again, the triangles are replicated as they move up the octree to create self-contained nodes. The subset is selected based on a dynamic LOD strategy introduced by [14]. The idea is to sample the full resolution geometry to achieve a subset that minimizes the image error. We choose to select the triangles that are the largest to maximize node coverage and thereby minimize image error for that node. To ensure that there are no holes in the LOD structure, the boundary triangles are simplified to a reduced representation (*e.g.,* 5%) of the original and inserted into every intersecting node except the leaf nodes.

Finally, a cleanup pass on the octree is performed which inserts the referenced vertices into each octree node and clips the triangles based on the bounding box of the node.

4.1.2 Out-of-Core Dataset Traversal

iRun uses an out-of-core traversal algorithm that has been extensively optimized for volume rendering. For each camera received from the user interface, we apply frustum-culling on the octree to find all nodes that are visible in this view and mark them as visible nodes. For polygonal meshes, visibility algorithms such as view-frustum and occlusion culling are important for maintaining interactive frame rates (see [26] for a recent survey). For volume rendering, occlusion culling is not feasible due to transparency. Therefore, only view-frustum techniques [33, 27] are deployed here.

Depending on whether or not the user is interacting with iRun, the LOD will decide which nodes are to be rendered next. Next, everything is passed to the visibility sorter (see Figure 4.2) and only those that have been cached in the geometry cache are used for rendering while the others are put onto the fetching queue. iRun also does cameraxoxo prediction for each frame by linearly extrapolating previous camera parameters. All of the nodes selected in the predicted camera will also be put on the fetching queue.

The LOD management in iRun is a top-down approach working in a priority-driven manner. Given a priority function $\mathbf{P}(C,N)$ which assigns priority for every node N of the octree with respect to the camera C, the LOD process starts by adding the root R to a priority queue with the key of $\mathbf{P}(C,R)$. Next, iterations of replacing the highest priority node of the queue with its children are repeatedly executed until such refinement will exceed a predefined number of triangles (Figure 4.3).

In our experiments, we use two different priority functions to control the LOD of iRun. The first is a Bread-First-Search (BFS) based function that is used during user's interactions: $\mathbf{P_{BFS}}(C,N) =< l,d >$, where l is the depth of N and d is the distance of the bounding box of N to the camera C. In this case, each node's priority is primarily determined by how far it is from the root and subsequently by its distance to the camera when the nodes are on the same level. Briefly, our goal is to evenly distribute data of the octree on the screen to improve the overall visualization of the dataset. An example of this scheme is shown in Figure 4.4.

While interacting with iRun, the target frame rate can be achieved by setting a limit on the maximum number of triangles rendered in the current frame. This number is calculated based on

```
SORT (Camera C, Node R, List SortedNodes)
  if R is not culled
    if R.Selected
      SortedNodes.Push(R);
    else if R.HasChildren
      SC = R's children sorted ascendingly by distances to C
      for each node N in SC
        SORT(C, N, SortedNodes);
```

Figure 4.2: **iRun Visibility Sorter** Pseudo-code for visibility sorter based on the camera distance in iRun.

```
LOD (Camera $\mathbf{C}$, Node $\mathbf{R}$, PriorityFunction $\mathbf{P}$, int $\mathbf{MaxTri}$)
  PriorityQueue $\mathbf{Q}$;
  $\mathbf{R}.Selected = true$;
  $\mathbf{Q}.\mathrm{Push}(\mathbf{P}(\mathbf{C},\mathbf{R}),\mathbf{R})$;
  $\mathbf{Total} = 0$;
  while !($\mathbf{Q}$.Empty())
    Node $\mathbf{N} = \mathbf{Q}.\mathrm{Pop}()$;
    if $\mathbf{N}.HasChildren$
      $\mathbf{TC}$ = the total number of triangles in $\mathbf{N}$'s children
      if $(\mathbf{Total} - \mathbf{N}.NumberOfTriangles + \mathbf{TC}) < \mathbf{MaxTri}$
        $\mathbf{Total} = \mathbf{Total} - \mathbf{N}.NumberOfTriangles + \mathbf{TC}$;
        $\mathbf{N}.Selected = false$;
        for $\mathbf{i} = 0$ to 7
          if $\mathbf{N}.Children[\mathbf{i}]$ is not empty or culled
            $\mathbf{N}.Children[\mathbf{i}].Selected = true$;
            $\mathbf{Q}.\mathrm{Push}(\mathbf{P}(\mathbf{C},\mathbf{N}.Children[\mathbf{i}]),\mathbf{N}.Children[\mathbf{i}])$;
```

Figure 4.3: **iRun LOD Traversal** Pseudo-code for the octree traversal algorithm.

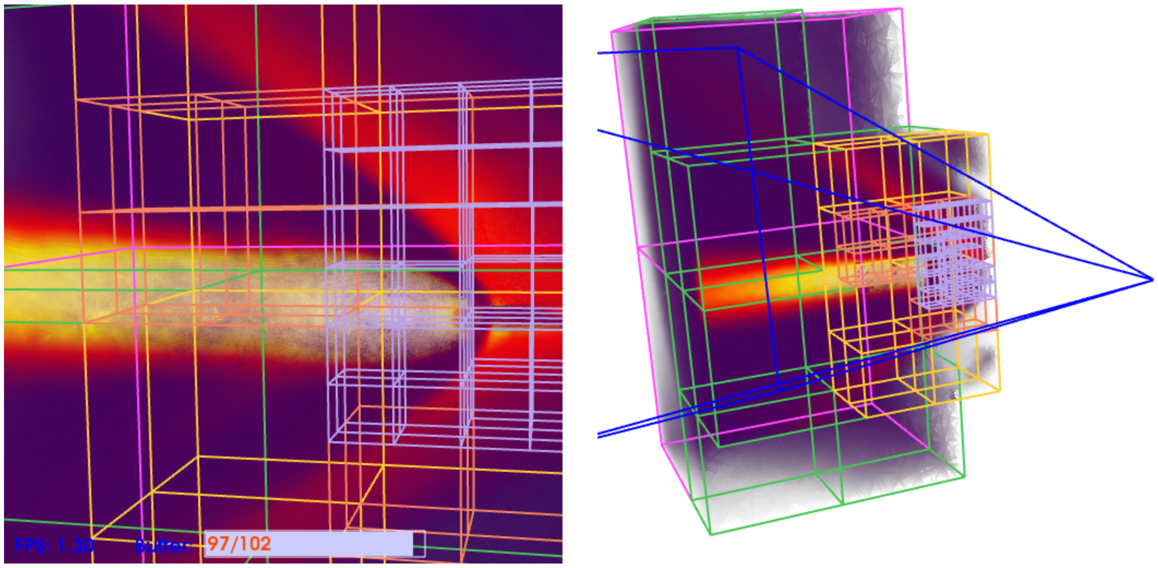


Figure 4.4: **A Snapshot of iRun Refining the LOD** The image on the left is rendered as the user would see it from the current camera position. On the right is a bird's-eye view of the same set of visible nodes. Different colors indicate different levels-of-detail. The geometry cache is limited to only 64MB of RAM in this case.

the number of triangles that were rendered, and the rendering time, for the previous frame.

For increased image quality at a given view, iRun will automatically adjust itself to increase the LOD using as much memory as possible when interaction stops. Since we want to cover as much of the screen as possible, a priority function reflecting the projected screen area is necessary for the LOD. We define $\mathbf{P_{area}}(C,N) = \mathbf{A}$, where $\mathbf{A}$ is the projected area of the bounding box of N onto the screen. However, this function can be easily replaced by any other heuristic approaches, such as those reflecting nodes scalar ranges, transfer functions, *etc.*, to achieve the best image quality.

This approach, however, could raise a problem when the user begins interaction again and the geometry cache is already full. Our next frame would be displayed incompletely since a lower LOD is not available and the higher LOD is too large to be rendered at an interactive rate. To overcome this problem, we will not flush the current data on the screen when increasing the LOD but only when the camera has changed. The trade-off in image quality is insignificant because the amount of memory used by this data is usually very small (*e.g.,* 1%) when compared to the total memory of the geometry cache.

iRun separates the fetching from the building of sets of visible nodes. If the fetching queue is empty, the fetching thread will wait until new requests arrive. Otherwise, it will read the requested node from disk and move it to the geometry cache. If the geometry cache is full, cached data will be flushed using a least-recently-used scheme.

As a result, the target frame rate of iRun is guaranteed to stay the same throughout user-interaction since the rendering process will never stall while waiting for IO. This improves interaction and does not introduce any significant degradation in image quality to the system. Because none of the visible geometry will be culled by occlusion, the amount of geometry shared between frames is very similar–every frame will have at least as much data as the previous frame. In the worst case, there will be at most one level of difference in LOD of the frame because of our BFS-based priority function.

4.1.3 Hardware-Assisted Rendering

Leveraging the performance of graphics processing units (GPUs) for direct volume rendering has received considerable attention (for a recent survey, see [82]). Shirley and Tuchman [80] introduce the Projected Tetrahedra (PT) algorithm, which splits tetrahedra into GPU renderable triangles based on the view direction. For correct compositing, the neighborhood information of the original mesh is used to determine an order dependence. More recent work by Weiler *et al.* [90] performs ray-casting on the mesh by storing the neighbor information on the GPU and marching through the tetrahedra in rendering passes. As with PT, the hardware ray caster requires the neighbor information of the tetrahedra for correct visibility ordering. A more extensible approach was introduced by Callahan *et al.* [15], which operates on the triangles that compose the mesh and requires no neighbor information. This makes immediate mode rendering, working-set management, LOD, and preprocessing much simpler than it would be by using the tetrahedra directly. Because of this, the algorithm has since been extended to perform dynamic LOD [14] as well as progressive volume rendering [13] for large datasets.

An important consideration for the interactive rendering of unstructured grids is the choice in volume rendering algorithms. Three aspects need to be considered: speed, quality, and the ability to handle dynamically changing data. By using a hardware-assisted volume rendering system, we can address both the speed and quality issues. In our system, we use the Hardware-Assisted Visibility Sorting (HAVS) algorithm of Callahan *et al.* [15] because it combines speed, quality, and most importantly, it does not require topological information and thus handles dynamic data.

HAVS operates by sorting the geometry in two phases. The first is a partial object-space sort that runs on the CPU. The second is a final image-space sort that runs on the GPU using a fixed A-buffer implemented with fragment shaders called the *k*-buffer. The HAVS algorithm considers only the triangles that make up the tetrahedral mesh. Thus it does not require the original tetrahedra or the neighbor information of the mesh. This allows us to render a subset of the octree nodes independently without merging the geometry.

Unlike rendering systems for opaque polygonal geometry, special care needs to be taken when rendering multiple octree nodes to ensure proper compositing. At each frame, our algorithm resolves the compositing issue by sorting the active set of octree nodes that are in memory in visibility order (front-to-back). When octree nodes of different sizes are in the active set, we sort by the largest common parent of the nodes. The original HAVS algorithm has also been modified to iterate over the active set of nodes in one pass and perform the object-space and image-space sort on each piece in visibility order. To ensure a smooth transition between octree nodes, the *k*-buffer is not flushed until the last node is rendered.

4.1.4 Distributed Rendering

Though our algorithm can efficiently render large unstructured grids on a single commodity PC, it was designed to be employed on a cluster of PCs in a distributed manner. Chromium [41] is a system that was introduced to perform parallel rendering on a cluster of graphics workstations.

Distributed rendering can also be used to visualize the data on larger displays. Moreland and Thompson [66] describe a parallel rendering algorithm that uses Chromium and an image-composite engine (ICE-T) built with VTK for visualizing the results on a display wall.

A major distinction between iRun and Chromium is that while Chromium *pushes* data through the pipeline to the render devices, iRun *pulls* data from a geometry-cache to the render devices. This *pulling* approach results in a conceptually simpler framework for parallel rendering where the CPU and GPU are tightly connected and data are fetched to the geometry-cache as needed before being transformed into graphics primitives.

iRun can partition rendering across a distributed network. This feature is useful for driving a display wall, where each system controls a single tile of display. This approach can further improve performance when rendering scenes with complicated geometry.

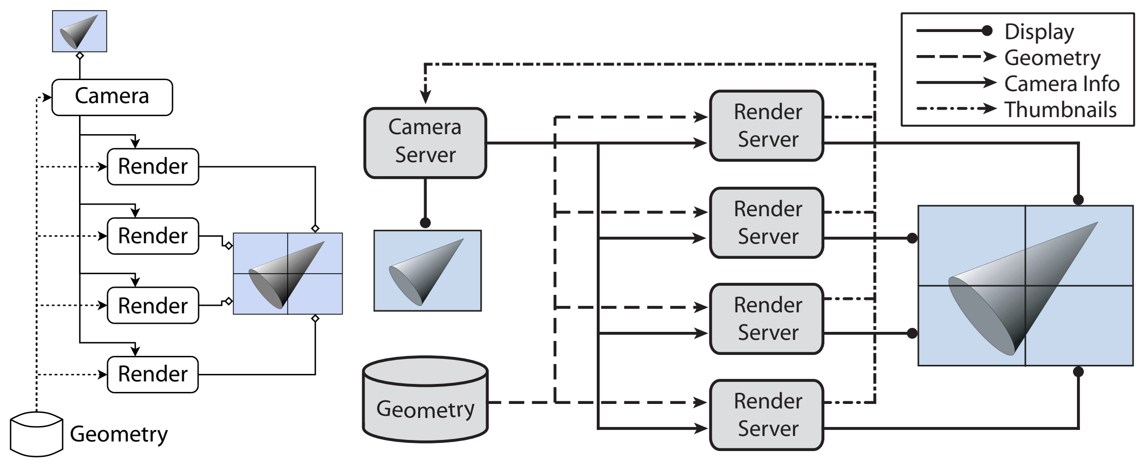
In iRun, each display on the display wall is driven by a dedicated render-server implemented in VTK. The portion of the display wall that a render-server will be responsible for is specified to the render-server on startup. A skewed view frustum is calculated based on the region of responsibility, and this frustum allows the render-server to cull the geometry to only the set visible on its display. Each render-server has access to the full geometry, but only loads the portions that it needs or anticipates needing.

The render-servers are coordinated by a single system that controls the camera. They receive camera description asynchronously to the render cycle, and wait for the render cycle to complete before updating VTK's camera. The update mechanism is implemented as a `vtkInteractorStyle` to allow for seamless integration.

The camera-server allows render-servers to establish and break TCP connections arbitrarily. Camera descriptions are sent out periodically, regardless of whether the camera has changed, to allow recently connected render-servers to quickly synchronize with the rest of the display wall.

Two clients have been written to run on the camera-server, both implemented in VTK. The first (Figure 4.5a) is nearly identical to the render-servers, except that it broadcasts camera coordinates instead of receiving them. This is accomplished by listening for the timer event which the interactor styles use for motion updates. The camera is read inside the event handler and provided to the broadcast code. This client has access to the same geometry and renders it on its own. It requires an adequately functional GPU, and relies on the automatic LOD adjustment to maintain interactive frame rates.

The second client (Figure 4.5b) is able to run on a less capable system. It also runs as a VTK implementation, but uses VTK's interaction styles with no geometry loaded. In this mode, the render-servers additionally frame capture their rendered output and transmit downsampled versions back to the camera-server. The camera server receives and assembles the snapshots asynchronously to the render cycle, and periodically requests the message loop to execute code to display the composite.



(a) Thumbnail Client (b) Full Client

Figure 4.5: **Distributed Rendering with iRun** (a) with a thumbnail client, the client receives thumbnails from the render-servers and composites them and (b) with a full client, the client accesses and renders the geometry directly.

Table 4.1: Preprocessing Results

Data Set	Input				Time (h:m:s)		Output			
	Verts	Tets	Tris	Size	Total	Clip	Tris/leaf	Verts	Tris	Size
Torso	169K	1.1M	2.2M	34MB	01:45	01:12	13K	3.2M	6.7M	158MB
Fighter	257K	1.4M	2.8M	50MB	02:27	01:28	18K	3.6M	7.8M	182MB
Rbl	730K	3.9M	7.9M	143MB	06:13	04:08	42K	7.8M	17M	410MB
Mito	972K	5.5M	11M	206MB	10:59	05:00	30K	11M	22M	538MB
Turbo Jet	1.7M	10M	20M	344MB	15:49	07:58	27K	12.	31M	697MB
Sf1	2.5M	14M	28M	516MB	37:41	25:56	58K	24M	63M	1.4GB
Bullet	6.3M	36M	62M	1.3GB	1:10:30	42:19	32K	48M	117M	2.8GB

4.2 Results

We generated all of our results on a 3.0-GHz Pentium D machine with 2.0 GB of RAM and a 500 GB SATA hard-disk with an nVIDIA 7800 GTX. Table 4.1 shows timing results and data sizes before and after preprocessing. We were able to preprocess the largest dataset, which contained 36 million tetrahedra or 63 million triangles, in just over an hour. For all models in this chapter, we target the output octree to have at most 10 thousand triangles per leaf. Due to the triangles added during simplification and clipping, the final number of triangles per leaf is slightly higher. The rest of the section will use these datasets unless otherwise stated.

In Figure 4.6, we show an example of how iRun can give users various levels-of-detail on demand. Assume a user has a machine with approximately 512MB of RAM available for rendering, and wants to use iRun to explore different features of the Turbo Jet dataset using a 512×512

viewport. The user can use iRun to load the dataset with 512MB of RAM dedicated to the geometry cache and a target frame rate of 5 (for example). Interactions at this frame rate only give the user a low LOD representation of this dataset since iRun can only render as many as 40 thousands triangles at each frame. It is almost impossible to get a coherent overview of the dataset at a resolution less than 0.1%. Thus, the user would want to stop interactions for iRun to refine its visualization. With more detail, the user can now select a specific region of interest to explore. For example, the user may wish to take a closer look at the region in the center of the dataset through a zoom. This causes the LOD to be automatically adjusted by iRun. With a full use of the geometry cache, the quality of our image is very close to the full resolution at that same view using 1.5GB of RAM.

Figure 4.7 is a plot showing the difference between the displayed and fetched nodes while performing interactions with the Turbo Jet as in the case study above. We selected the target frame rate of 2 frames per second to increase the readability of the plot while the interaction speed is fairly high (mouse-movement of over 30 pixels/frame). Our set of movements contains excessive rotations, zooms, and translates. As shown in the figure, the total number of nodes that need to be loaded for the next frame stays relatively low compared to those that are rendered.

For distributed rendering results, we set up four machines—with two Dual-Core AMD Opterons on each—as clients. Our goal was to render the bullet dataset consisting of 3GB of data (117M faces) displayed on another server. The render window on the server had a 1024$\times$1024 viewport and each client only rendered a subwindow of 512$\times$512. Figure 4.8 illustrates the differences between distributed rendering versus local rendering on a single client machine. The right image shows the results of the distributed rendering. With the target frame rate of 10, we were able to achieve a good volume-rendered image after filling up 24 nodes of the buffer. The frame rate of the distributed renderer can be reported fairly as the minimum frame rate of the four clients. For this view we achieved a frame rate of 11.62 fps. The left image shows the same dataset with the same view rendered using only one of the clients with the same window size. By filling up only 14 nodes of the buffer the frame rate is reduced to 2.77 fps on a single machine. This is because there are four times as many pixels that need to be rendered at each frame. Moreover, disk accesses also increase up to four times compared to the distributed rendering.

4.3 Discussion

4.3.1 Limitations

There are issues in our current implementation that could use improvement. First, due to clipping and boundary triangles, node size can grow larger than desired. Bounding this limit would be a useful feature. Another limitation is that our current LOD strategy may not be suitable for all

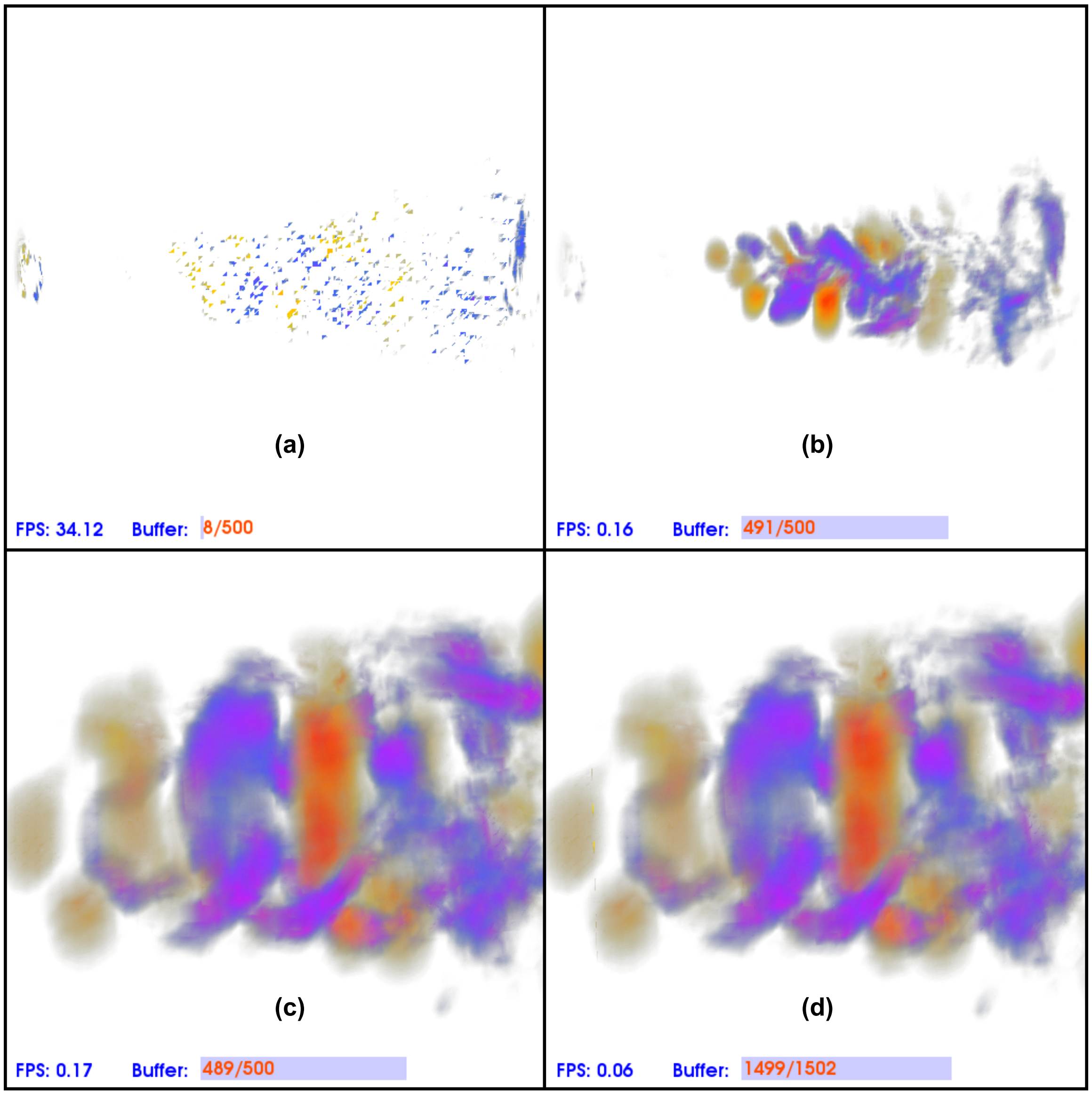

(a)
FPS: 34.12 Buffer: 8/500
(b)
FPS: 0.16 Buffer: 491/500
(c)
FPS: 0.17 Buffer: 489/500
(d)
FPS: 0.06 Buffer: 1499/1502

Figure 4.6: **Effects of Geometry Cache Size in iRun** iRun visualizes the Turbo Jet model consisting of over 10 millions tetrahedra with multiple levels-of-detail. The geometry cache size of (a), (b) and (c) is 512MB while (d) is 1.5GB. (a) In interactive mode, iRun only displays 40K triangles. (b) In still-rendering mode, most of the geometry cache is used. (c) The user zooms in to a particular region. (d) The geometry cache size is tripled.

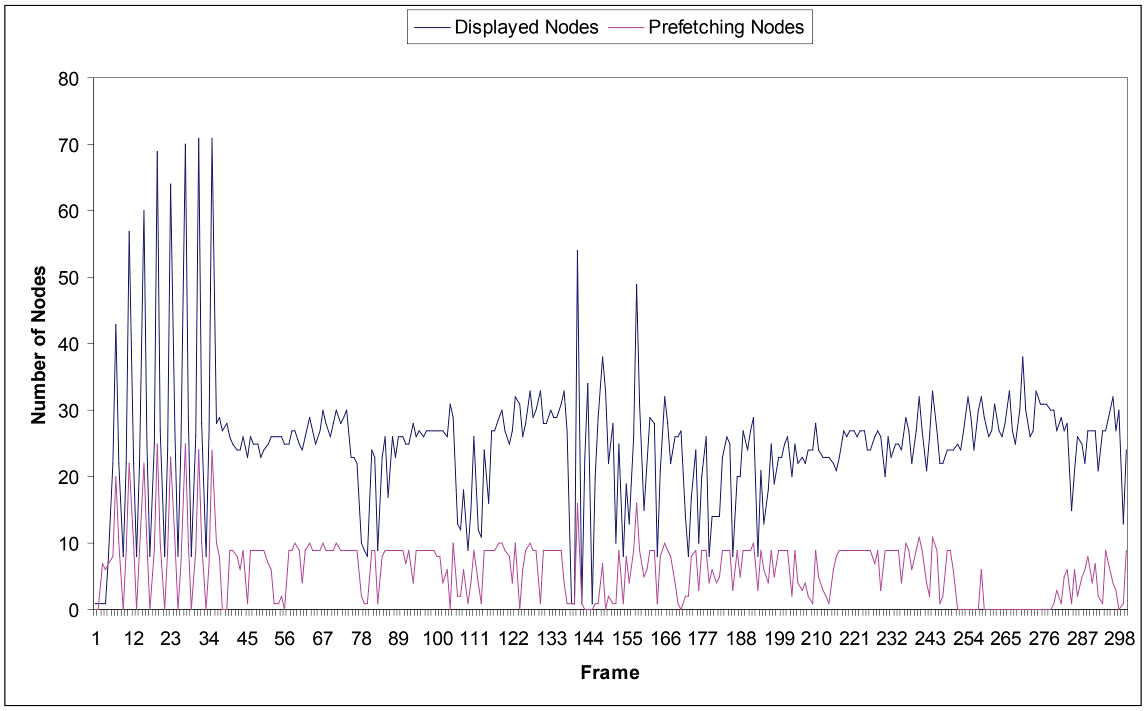


Figure 4.7: **Geometry Cache Statistics** The difference between the displayed nodes of the geometry cache and the ones being fetched during interactions of the Turbo Jet at 2 fps.

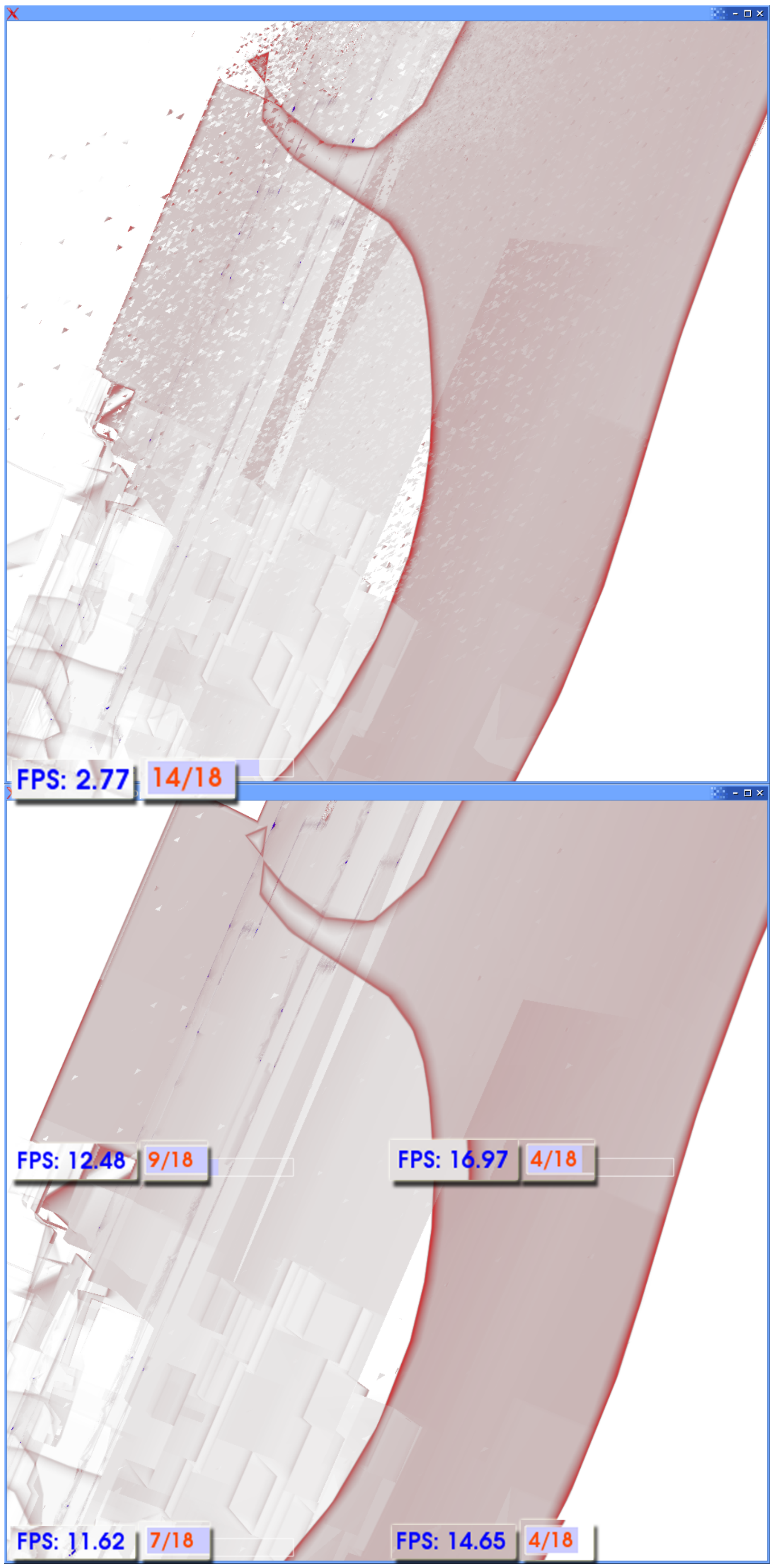


Figure 4.8: **Distributed Volume Rendering** The bullet dataset rendered with a single machine versus a distributed rendering with four clients.

datasets. Therefore a more automatic way of determining the LOD triangles would improve image quality. Finally, even though the number of vertices is generally much less than the number of tetrahedra, our current method of keeping the vertices in-core during preprocessing would not be feasible on a PC for extremely large datasets with hundreds of millions of tetrahedra.

4.3.2 VTK

In term of coding, we found VTK to be very helpful when implementing our system. By leveraging existing functionality provided by this framework, we were able to focus on algorithmic instead of engineering contributions. For example, the simplification and clipping phase of the preprocessing are all done using VTK classes. Additionally, the client-server and user interface are built on top of VTK. However, using VTK provided the following challenges:

- Because VTK is for general use, we were forced to modify existing classes to get desired functionality. For example, we added our own timers and modified the rendering pipeline order. In addition, current VTK data structures for geometry are incompatible with OpenGL, which makes using fast display structures such as triangle arrays difficult.
- VTK is not thread-safe. iRun required solving many synchronization problems among threads. For example, one condition was occurring when the visibility sorter received the camera slower than the rendering window, causing compositing artifacts in the resulting image.
- To manage the client-server architecture for parallel rendering, modifications to the VTK interactors were necessary. Specifically, the addition of an `asyncExec` method to `vtkRenderWi-ndowInteractor` and its derived classes was necessary to queue commands for later execution.

4.3.3 iRun vs. iWalk

iRun shares many ideas with iWalk [28]. However, due to the complexity of volume rendering tetrahedral data, many of the algorithms presented in iWalk were not suitable for iRun. In iRun, each node of the octree contains a self-describing `vtkUnstructuredGrid` of varying LODs that can be rendered independently. iWalk only keeps triangles in the leaf nodes and depends on occlusion culling instead of LOD to remain interactive. This also affects the traversal algorithm, which is more sophisticated in iRun because LOD as well as screen coverage need to be considered due to the transparent nature of the nodes. The fetching thread also works differently. iRun separates fetching from building visible sets of nodes because of the added complexity of visibility sorting for the nodes.

Another difference is that due to compositing problems that occur with overlapping triangles from neighboring nodes, we require a more exact triangle-box intersection that is based on the triangle, not the vertices. iRun also requires the clipping of triangles that extend beyond the node's bounding box to avoid incorrect visibility ordering across neighboring nodes. Unlike iWalk, we require boundary triangles (simplified or full resolution) at each level of the tree to avoid holes in the rendered image. Finally, we note that the implementation of both systems is completely disjoint.

4.4 Proposition

Through our experiences with iRun, we have learned that (1) interactive large-scale visualization pipelines need both *push* and *pull* updating policy for their level-of-detail and view-dependant features, respectively, and (2) current visualization systems such as VTK are not designed for concurrent executions of modules (not even thread-safe) and their pipeline constructs are limited by supporting backward-compatibility.

CHAPTER 5

PARALLEL VISUALIZATION ON LARGE CLUSTERS USING MAPREDUCE

In this chapter, we take a first step at investigating the suitability of cloud-based infrastructure for large-scale visualization. We have designed a set of MapReduce-based algorithms for memory-intensive visualization techniques, and performed an extensive experimental evaluation. Our results indicate that MapReduce offers a potential foundation for a combined storage, processing, analysis, and visualization system that is capable of keeping pace with growth in data volume (attributable to scalability and fault-tolerance) as well as growth in application diversity (attributable to extensibility and ease of use).

Because MapReduce framework can be decomposed into a pair of data-parallel jobs with a serial (shuffling) phase in between, a careful study on this topic can help us determine necessary parallel constructs that are required in a modern dataflow architecture to efficiently support large data visualization. We also found that common visualization algorithms can be naturally expressed using the MapReduce abstraction. Even simple implementations of these algorithms are highly scalable.

Cloud computing has emerged as a viable, low-cost alternative for large-scale computing and has recently motivated both industry and academia to design new general-purpose parallel programming frameworks that work well with this new paradigm [31, 71, 95]. In contrast, large-scale visualization has traditionally benefited from specialized couplings between hardware and algorithms, suggesting that migration to a general-purpose cloud platform might incur in development costs or scalability.[1]

The MapReduce framework [31] provides a simple programming model for expressing loosely-coupled parallel programs using two serial functions, *Map* and *Reduce*. The Map function processes a block of input producing a sequence of *(key, value)* pairs, while the Reduce function processes a set of values associated with a single key. The framework itself is responsible for "shuffling" the output of the Map tasks to the appropriate Reduce task using a distributed sort. The model is sufficiently expressive to capture a variety of algorithms and high-level programming models,

[1]Scalability refers to the relative performance increase by allocating additional resources.

while allowing programmers to largely ignore the challenges of distributed computing and focus instead on the semantics of their task. Additionally, as implemented in the open-source platform Hadoop [39], the MapReduce model has been shown to scale to hundreds or thousands of nodes [31]. MapReduce clusters can be constructed inexpensively from commodity computers connected in a shared-nothing configuration (*i.e.,* neither memory nor storage are shared across nodes). Such advantages have motivated cloud providers to host Hadoop and similar frameworks for processing data at scale [25, 92, 7].

These platforms have been largely unexplored by the visualization community, even though these trends make it apparent that our community must inquire into their viability for use in large-scale visualization tasks. The conventional modus operandi of "throwing datasets" through a (parallel) graphics pipeline relegates data manipulation, conditioning, and restructuring tasks to an offline system and ignores their cost. As data volumes grow, these costs — especially the cost of transferring data between a storage cluster and a visualization cluster — begin to dominate. Cloud computing platforms thus open new opportunities in that they afford both general-purpose data processing as well as large-scale visualization.

5.1 MapReduce Overview

MapReduce is a framework to process massive data on large distributed systems. It provides an abstraction inspired by functional programming languages such as Lisp, relying on two basic operations:

- Map: Given input, emit one or more *(key, value)* pairs.
- Reduce: Process all values of a given key and emit one or more *(key, value)* pairs.

A MapReduce job is comprised of three phases: map, shuffle and reduce. Each dataset to be processed is partitioned into fixed-size blocks.

In the map phase, each task processes a single block and emits zero or more *(key, value)* pairs. In the shuffle phase, the system sorts the output of the map phase in parallel, grouping all values associated with a particular key. In Hadoop, the shuffle phase occurs as the data is processed by the mapper (*i.e.,* the two phases overlap). During execution, each mapper hashes the key of each key/value pair into bins, where each bin is associated with a reducer task and each mapper writes its output to disk to ensure fault tolerance. In the reduce phase, each reducer processes all values associated with a given key and emits one or more new key/value pairs. Since Hadoop assumes that any mapper is equally likely to produce any key, each reducer may potentially receive data from any mapper. Figure 5.1 illustrates a typical MapReduce job.

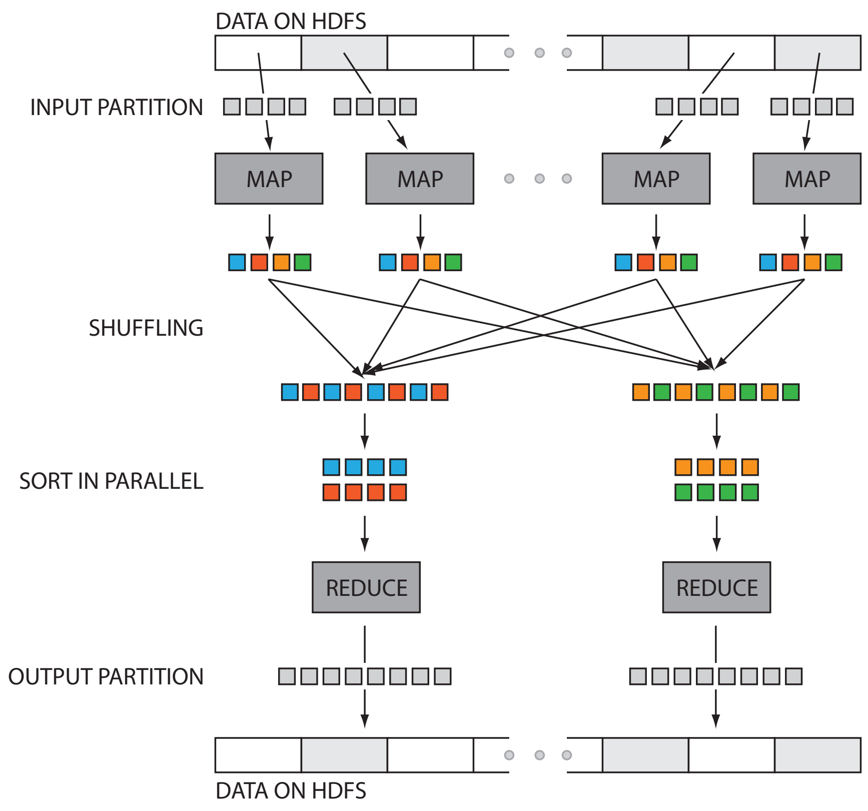


Figure 5.1: **Hadoop Architecture** Data transfer and communication of a typical MapReduce job in Hadoop. Data blocks are assigned to several Maps, which emit key/value pairs that are shuffled and sorted in parallel. The reduce step emits one or more pairs.

To further illustrate this model, consider the following pseudocode example to compute the number of occurrences of the word in a corpus of text:

```
Map(data):
    FOR word IN data
         emit(word, 1)

Reduce(key, values):
    count = 0
    WHILE(values.has_next())
         count += values.next.value()
    emit(key,count)
```

The mapper simply parses the words from the string of data and emits the word as the key and 1 as the value. The reducer will receive all *(key, value)* pairs for a given word and will sum their values. After the values are summed, all the reduce emits the word and count.

MapReduce offers an abstraction that allows developers to ignore the complications of distributed programming — data partitioning and distribution, load balancing, fault-recovery and interprocess communication. Hadoop is primarily run on a distributed file system, and the Hadoop

File System (HDFS) is the default choice for deployment. Hadoop has become a popular runtime environment for higher-level languages for expressing workflows, SQL queries, and more [71, 40]. These systems are becoming viable options for general purpose large-scale data processing, and leveraging their computational power to new fields can be a very promising prospect. For example, MapReduce systems are well-suited for *in situ visualization*, which means that data visualization happens while the simulation is running, thus avoiding costly storage and postprocessing computation. There are several issues in implementing *in situ visualization* systems as discussed in [60]. We posit that the simplicity of the implementation, inherent fault-tolerance, and scalability of MapReduce systems make it a very appealing solution.

5.2 Visualization Algorithms Using MapReduce

In this section we describe MapReduce algorithms for three widely-used and memory-intensive visualization techniques: isosurface extraction, rendering of large unstructured grids, and large model simplification.

5.2.1 Isosurface Extraction

Isosurfaces are instrumental in visual data exploration, allowing scientists to study function behavior in static or time-varying scenarios. Giving an input scalar volume, the core of extraction is the computation of the isosurface as a collection of simplicial primitives that can be rendered using common graphical methods. Our MapReduce-based algorithm for isosurface extraction is based on the Marching Cubes algorithm [59], which is the de facto standard for isosurface extraction due to its efficiency and robustness.

As Figure 5.2 illustrates, partitioning relies on the natural representation of a scalar volume as a collection of 2D slices. The Hadoop distributed file system uses this strategy to partition data into blocks for each mapper, but imposes some constraints. First, each partition must contain complete slices. Second, it allows the overlap by one slice in only one direction to account for triangles spanning across partitions. Although it may result in duplication of input data, there is no duplication of output triangles since this overlap only occurs in one dimension. In practice, the duplication of input data is small and has no significant effect on the performance of the system. Each mapper computes the triangles of several isosurfaces using the Marching Cubes algorithm as implemented in the Contour Library [16] and emits a (key,value) pair for each isovalue. The key is the isovalue and the value is the triangle data for the each cube in binary format. The reducer receives the data sorted and binned by isovalue. Thus, the reduce stage only needs to act as a pass-through, writing the isosurface as a triangle soup to file.

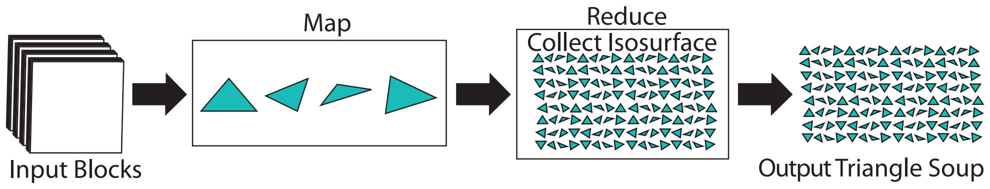


Figure 5.2: **MapReduce Isosurface Extraction** The map phase generates an isovalue as key and triangle data as value. The reduce phase simply outputs the isosurface as a triangle soup to file.

5.2.2 Rendering

Out-of-core methods have been developed to render datasets that are too large to fit in memory. These methods are based in a streaming paradigm [34], and for this purpose the rasterization technique is preferred due to its robustness, high parallelism and graphics hardware implementation. We have designed a MapReduce algorithm for a rasterization renderer for massive triangular and tetrahedral meshes. The algorithm exploits the inherent properties of the Hadoop framework and allows the rasterization of meshes several gigabytes in size and images with billions of pixels.

Hadoop partitions the input "triangle soup" among mappers with the constraint that each partition must be a multiple of 36 bytes, thus avoiding the block boundary splitting a triangle. For each triangle, the mapper computes its projection onto the image plane and its corresponding pixels. For each pixel, the mapper outputs a (key,value) pair, with the key being the pixel location in the image plane (x, y), and the value being the depth and color of the pixel. The MapReduce framework sorts pixels into the proper order (row-major) to construct the final image. Pixel colors emitted by the mapper that share the same image plane location are grouped by this sorting. The reducer emits the smallest depth value for each pixel location, therefore accomplishing the z-buffering algorithm automatically. In Figure 5.3, we give an overview of this algorithm. Mappers and reducers are viewed as geometry and multifragment shaders, respectively, in two distinct phases. Note that this parallels a graphics hardware pipeline and can be similarly extended to handle more advanced visualizations by custom "geometry shaders" and "fragment shaders." For example, in a volume renderer, each reducer just needs to sorts its fragments and composite them instead of a selection based on depth.

5.2.3 Mesh Simplification

Despite advances in out-of-core methods for rendering structured or unstructured meshes, it may still not be feasible to use the full resolution mesh. Several mesh simplification techniques have been proposed [35, 93, 79]. Memory usage is a key aspect of this problem, since techniques often require storage proportional to the size of the input or output mesh. An alternative is given by the OoCSx (improved Out-of-Core Simplification) algorithm [58], which decouples this relationship

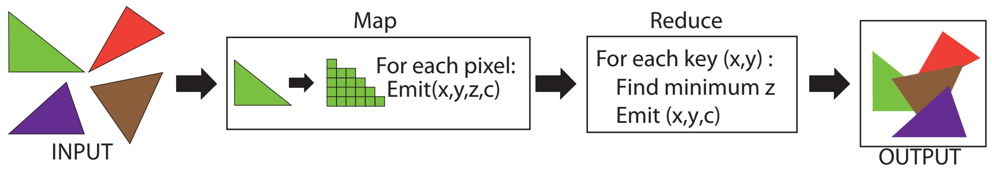


Figure 5.3: **MapReduce Rasterization** The map phase rasterize each triangle and emits the pixel coordinates as value, and its color and depth as value. The reducer emits the smallest depth value for each location.

and allows the simplification of meshes of arbitrary sizes. This is accomplished by superimposing a regular grid over the underlying mesh with associations between grid cells and vertices: every grid cell that contains a vertex of the input mesh must also contain a vertex on the output mesh, and every cell must have only one representative vertex. The problem is broken into finding all triangles that span three unique cells, and then finding an optimum representative vertex for each cell. Only a linear pass through the triangle mesh to hash each vertex is needed to find its representative bin before the output of all triangle indices.

The linear pass of OoCSx is not suitable for a parallel implementation. However, if we do not use the minimal error criteria for optimal representative vertices, we are able to implement the algorithm using only the Map phase of Hadoop. This not only avoids the expensive sorting and merging phases of MapReduce but also allows the simplification performance to scale directly with the number of mappers at our disposal. We use two MapReduce jobs to implement the full algorithm since it requires two sorting phases (see Figure 5.4). The first Map phase bins each vertex into a regular grid to ensure that all triangles contributing vertices to a particular bin arrive on the same node in the Reduce phase. It also computes the quadric measure vector associated with the contributing triangle. For each triangle, three (key, value) pairs are emitted, one for each vertex. The key is the the bin coordinate that contains the vertex, and the value is a concatenation of the quadric measure vector with the three indices of the triangle.

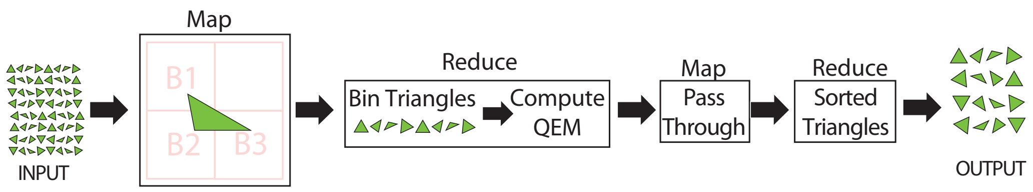


Figure 5.4: **MapReduce Mesh Simplification** The first map phase generates the bin coordinate of a vertex as key and a quadric error measure along the three triangle indices as value. The first reduce emits the representative vertex for each triangle. A second map and reduce is applied since multiple current reduce phases are not currently supported.

The first Reduce phase receives the same (key, value) pair from the Map phase, but sorted and grouped by key. It reads each unique key (bin), and uses the quadric measures of all triangles falling into that bin to compute the representative vertex. If the indices of all vertices of a triangle contributing to a representative vertex are unique, the Reduce phase emits the indexed triangle as key, and the current grid cell and vertex. Thus, across all reducers, there will be exactly three (key,value) pairs with the same key (triangle), each storing a different representative vertex and corresponding bin as its value. Since multiple Reduce phases are currently not supported, we use a second MapReduce job to complete the dereference. This second Map phase merely reads and emits the data output from the first Reduce job. Keyed on triangle index, the second Reduce phase receives the exact three bin-vertex pairs, and emit as final output the simplified mesh.

5.3 Experimental Analysis

We have performed an in-depth analysis of the algorithms presented in the previous section. We designed our experiments to evaluate, for each algorithm, its ability to scale up and down, as well as the overhead introduced by Hadoop. The first series of tests shows the cost of data transfer through a MapReduce job without any computation, followed by a detailed evaluation of each individual algorithm. By default, the number of mappers that Hadoop launches for a job is a multiple of the number of data blocks, taking care to not exceed the actual number of blocks on its HDFS (counting all replications). On the other hand, the number of reduce tasks can be specified. To simplify comparison, in our tests we maximize the number of reducers to the system capacity while keeping its ratio to the number of mappers equal to 1. The number of mappers and reducers always equal in our experiments whenever the number of input data blocks permits.

Tests were performed on two Hadoop-based systems: a local cluster and the NSF CLuE cluster managed by IBM [25]. The local cluster consists of 60 nodes, each with two quad-core Intel Xeon Nehalem 2.6GHz processors, 24GB of memory and a 320GB disk. The CLuE cluster consists of 410 nodes each with two single-core Intel Xeon 2.8GHz processors, 4GB of memory and a 400GB disk. While still a valuable resource for research, the CLuE hardware is outdated if compared to modern clusters, since it was originally built in 2004 with both low-speed processors and limited memory. Thus, we mostly utilize the performance numbers from the CLuE cluster as a way to validate and/or compare with our results on the local cluster. Since the CLuE cluster is a shared resource among multiple universities, there is currently no way to run experiments in isolation. We made sure to run all of our experiments at *dead* hours to minimize the interference from other jobs. HDFS files were stored in 64MB blocks with 3 replications.

5.3.1 MapReduce Baseline

To evaluate the cost incurred solely from streaming data through the system, several baseline tests were performed. For our scaling tests, we have evaluated our algorithms' performance only for weak-scaling. (*i.e.,* scaling the number of processors with a fixed data size per processor). This was chosen over strong scaling (*i.e.,* scaling the number of processors with a fixed total data size) since the latter would require changing a data blocksize to adjust the number of mappers appropriately. The Hadoop/HDFS is known for degraded performance for data with too large or small blocksizes depending on job complexity [84]. Therefore strong scaling is currently not a good indicator of performance in Hadoop. The weak-scaling experiments vary data size against task capacity and proportionally change the number of mappers and reducers. An algorithm or system that has proper weak scaling should maintain a constant runtime. To avoid biasing results by our optimization schemes, we use the default MapReduce job, with a trivial record reader and writer. Data are stored in binary format and split into 64-bytes record, with 16 bytes reserved for the key. Map and reduce functions pass the input directly to the output, and are the simplest possible jobs such that the performance is disk I/O and network transfer bounded.

The top table in Figure 5.5 shows the average cost for map, shuffle and reduce tasks, respectively, in the local cluster. The last two columns depict the overall disk I/O throughput and data throughput for a particular job. I/O rates were computed by dividing the total number of disk reads and writes including temporary files over the total time, while data rates represent how much input data pass through the job in a second. For map tasks, Hadoop was able to keep the runtime constant, since input files are read in sequence on each node and directed to appropriate mappers. In the reducing step, even though the amount of data is the same as to the map phase and each node writes data to its local disk, there is also a local external sorting that incurs in overhead. Nevertheless, both runtimes are still considerably constant, except for the jump from 64GB to 128GB. At this point, the number of reducers guarantees each node has to host at least two reduce tasks if distributed properly. Therefore each disk now has double the I/O and seek operations. This can be seen in the disk I/O rates, where the throughput is optimal at 64 tasks on the local cluster with 60 disks and drops while maintaining a relatively high speed for the larger number of tasks.

The shuffle phase of Hadoop is where weak scaling is not linear. This accounts for the data transfer between the map and reduce phase of a job along with sorting. In Hadoop each mapper is likely to contribute a portion of data required by each reducer and therefore is not expected to scale well. The plot in Figure 5.5 illustrates the breakdown of the three phases. The *Hadoop Overhead Time* table shows (only) the overhead of communication across the map and reduce phases. Note that each phase takes about 15 seconds to start on our local cluster. We also include weak-scaling

WEAK-SCALING OF DATASIZE VS. THE NUMBER OF TASKS (on Cluster)

Datasize	#Maps	#Reduces	Map Time	Shuffle Time	Reduce Time	Total Time	I/O Rate	Data Rate
1GB	16	1	7s	18s	27s	63s	84 MB/s	16 MB/s
2GB	32	2	8s	18s	27s	66s	161 MB/s	31 MB/s
4GB	64	4	9s	24s	30s	75s	283 MB/s	55 MB/s
8GB	128	8	10s	26s	29s	78s	545 MB/s	105 MB/s
16GB	256	16	10s	32s	29s	90s	944 MB/s	182 MB/s
32GB	512	32	12s	56s	32s	130s	1308 MB/s	252 MB/s
64GB	1024	64	11s	69s	30s	153s	2222 MB/s	428 MB/s
128GB	2048	128	13s	146s	57s	320s	2125 MB/s	410 MB/s

HADOOP OVERHEAD TIME (on Cluster)

#Maps	#Reduces	Map Only	Total
16	1	15s	30s
32	2	15s	30s
64	4	15s	30s
128	8	15s	30s
256	16	15s	30s
512	32	15s	33s
1024	64	15s	35s
2048	128	15s	36s

WEAK-SCALING (on CLuE)

Datasize	Total Time	I/O Rate	Data Rate
1GB	971s	5 MB/s	1 MB/s
2GB	946s	11 MB/s	2 MB/s
4GB	986s	22 MB/s	4 MB/s
8GB	976s	44 MB/s	8 MB/s
16GB	1059s	80 MB/s	15 MB/s

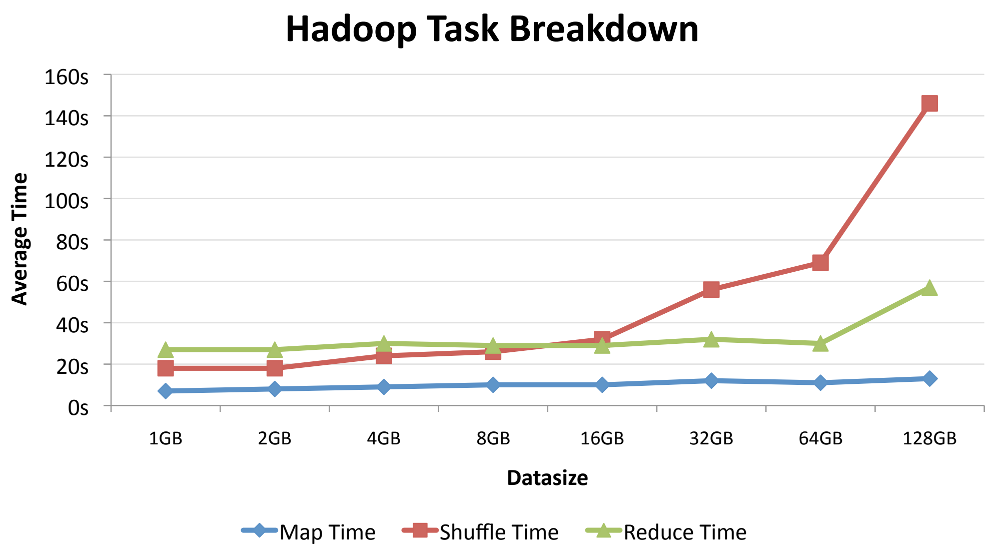


Figure 5.5: **Hadoop Baseline Results** Hadoop baseline test evaluates data transfer costs. In the local cluster we achieve rates up to 428MB/s

results for the CLuE cluster for comparison. The I/O rates are considerably lower than the local performance, and, as expected, this is likely due to age of the system and the shared usage. From the weak scaling tests we conclude that the MapReduce model can be robust when the number of nodes scales with the data size. Little cost is incurred for using more input data, and the effective transfer rates scale proportionally to the input data size. However, in order to ensure fault tolerance, disk I/O is heavily involved and could bound the overall performance.

5.3.2 Isosurface Extraction

We tested the isosurface MapReduce algorithm on two datasets: a small skull volume (256MB) and a larger ppm Richtmyer-Meshkov instability simulation volume (7.6GB). Since our baseline testing has shown that the amount data produced can effect Hadoop's performance, we performed tests that varied the number of isosurfaces generated in a single job, since this can have a drastic effect on the amount of data being produced. For few isosurfaces, we expect a small number of triangles to be produced. Conversely, for many isosurfaces, more data will be output to disk than were used for input. We keep the number of maps constant at 256, as this is the largest power-of-two we can use without pigeon holing more than 1 mapper to a node of the CLuE cluster.

We observed that in the extraction of dozens of isosurfaces (as part of parameter exploration) data output increases proportionally to runtime. Jobs are relatively fast for the standard case of fewer isosurfaces, since input and output data are partitioned evenly to mappers and reducers, thus the amount of disk I/O for input and output is relatively small (*e.g.,* approximately 32MB per Mapper). The runtime of this algorithm is mostly affected by the shuffling phase, where triangles emitted from mappers exceed the available buffer and are sorted out-of-core before being transferred to the reducers. The performance of this phase depends on the amount of temporary disk I/O used when the mapper runs out of in-core buffer space.

In Table 5.1, the *File Written* column denotes the amount of temporary data produced (not the HDFS output). For the skull data set, the algorithm runs quite fast up to 8 isosurfaces, close to Hadoop's overhead. When moving to 16 isosurfaces, the disk I/O starts to increase abruptly causing the algorithm to slow down. This increase denotes the amount of temporary disk storage needed for the Mappers to sort the data. Nevertheless, while the data increase quadratically, the run-time only triples. For the PPM dataset, we also see similar behavior in the disk usage. The run-time increases from 10 minutes to 28 minutes and 2 minutes to 5 minutes when the disk I/O increases from 104GB to 252GB for the CLuE and the local cluster, respectively. Figure 5.6 shows the isosurfaces for this dataset. The rendering for the combustion PPM dataset is achieved by chaining the map task of our isosurface extraction with the map and reduce job of our surface rendering described in Section 5.3.3.

5.3.3 Rendering

Performance of the rasterization algorithm is dependent on the output resolution, camera parameters and geometry size. The impact of geometry and resolution is proportional to the number of triangles to be rasterized and fragments generated, while the impact of camera parameters is more difficult to estimate. For instance, depending on a camera position, pixels may receive no or multiple fragments. Hence, reducers may receive no data or several fragments to compute depth

Table 5.1: Performance Results for Isosurface Extractions

PPM Richtmyer-Meshkov instability (7.6GB)

#Isovalues	CLuE time	Cluster time	File Written
	256M/256R	480M/480R	
1	2min 55s	57s	1.78MB
2	2min 00s	57s	1.81MB
4	9min 35s	2min38S	35.26GB
8	10min 26s	2min37s	103.92GB
16	28min 29s	5min27s	252.38GB

SKULL (256MB)

#Isovalues	CLuE time	Cluster time	File Written
	256M/256R	480M/480R	
1	2min 22s	54s	1.94MB
2	2min 00s	56s	15.83MB
4	2min 02s	56s	31.22MB
8	2min 11s	58s	3.35GB
16	6min 06s	1min27s	18.74GB

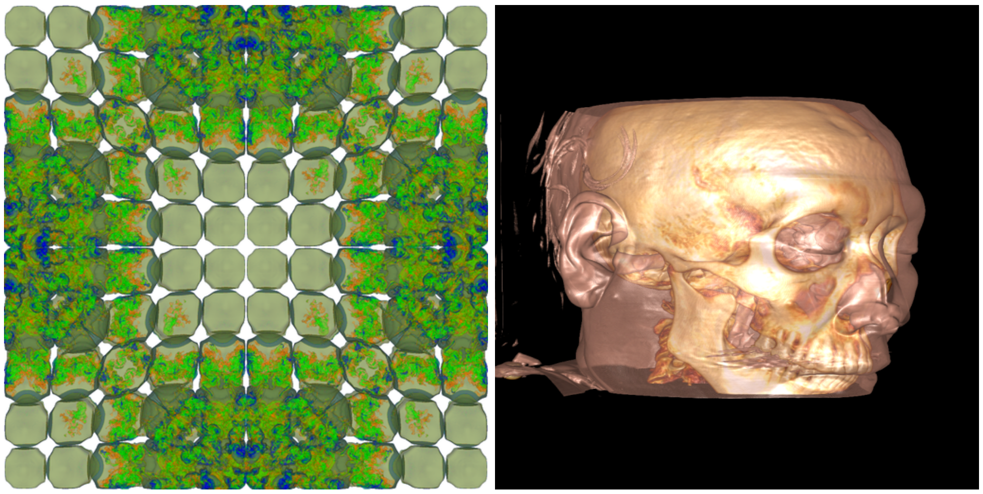


Figure 5.6: **Isosurface Extractions with MapReduce** Isosurface results using the MapReduce framework for the PPM Richtmyer-Meshkov instability and Skull datasets.

ordering. Figure 5.7 and Table 5.2 show rendering results and weak scaling tests with run times and temporary disk usage. For the CLuE cluster, the cost for rendering images of 100MP or less is insignificant compared to the Hadoop overhead. For our local cluster, this threshold is more than 1GP. For images of this size, the cluster is stretched to its limit and performance is limited by the amount of data written to disk. There is a significant increase in the data size due to the large amount of temporary caching by the system due to insufficient buffer space for the shuffling phase.

Figure 5.8 illustrates a volume rendering pipeline modified from our surface rendering pipeline. Tetrahedral meshes are broken down into a triangle soup, with each tetrahedron face split into a

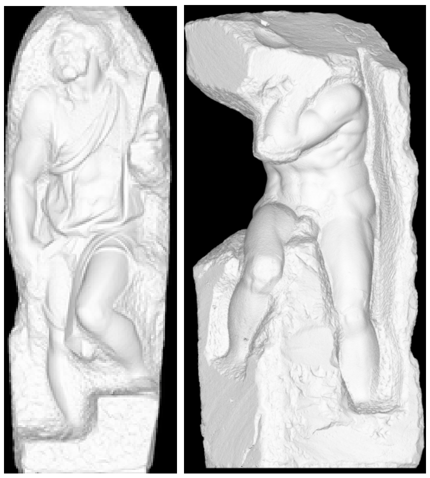


Figure 5.7: **Surface Rendering with MapReduce** Rendering results for the St. Matthew (left), Atlas (right).

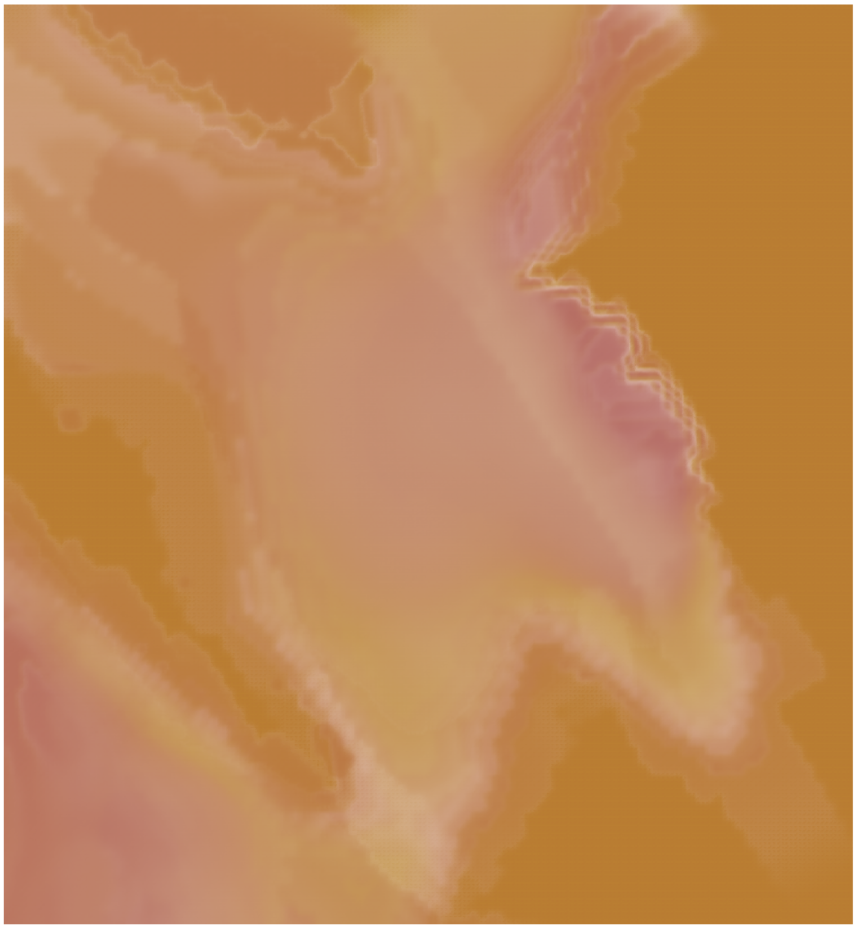
Table 5.2: Performance Results for Surface Rendering

WEAK SCALING (RESOLUTION)

	St. MATTHEW (13 GB)				ATLAS (18 GB)			
Resolution	#M/R	CLuE time	Cluster time	File Written	#M/R	CLuE time	Cluster time	File Written
1250x1250	256/256	1min 54s	46s	33MB	273/273	1min 55s	46s	41MB
2500x2500	256/256	1min 42s	46s	147MB	273/273	2min 11s	46s	104MB
5000x5000	256/256	1min 47s	46s	583MB	273/273	2min 12s	46s	412MB
10000x10000	256/256	1min 40s	46s	2.3GB	273/273	2min 12s	46s	1.6GB
20000x20000	256/256	2min 04s	46s	10.9GB	273/273	2min 27s	47s	5.5GB
40000x40000	256/256	3min 12s	1min08s	53.14GB	273/273	3min 55s	55s	37.8GB
80000x80000	256/256	9min 50s	2min55s	213GB	273/273	10min 30s	1min58s	151.8GB

WEAK SCALING (RESOLUTION AND REDUCE)

	St. MATTHEW (13 GB)				ATLAS (18 GB)			
Resolution	CLuE	256M	Cluster	480M	CLuE	256M	Cluster	480M
	#R	time	#R	time	#R	time	#R	time
1250x1250	4	1min 13s	8	46s	4	1min 18s	8	46s
2500x2500	8	1min 18s	15	46s	8	1min 19s	15	45s
5000x5000	16	1min 18s	30	46s	16	1min 51s	30	46s
10000x10000	32	2min 04s	60	47s	32	1min 52s	60	47s
20000x20000	64	2min 04s	120	49s	64	2min 34s	120	46s
40000x40000	128	4min 45s	240	1min06s	128	5min 06s	240	55s
80000x80000	256	9min 50s	480	2min14s	256	10min 30s	480	1min41s



TETRAHEDRAL MESHES VOLUME RENDERING (on Cluster)

Model	#Tetrahedra	#Triangles	Time	#Fragments	Bytes Read	Bytes Write
Spx	0.8 millions	1.6 millions	3min 29s	9.8 billions	320 GB	473 GB
Fighter	1.4 millions	2.8 millions	2min 20s	5.3 billions	172 GB	254 GB
Sf1	14 millions	28 millions	6min 53s	16.8 billions	545 GB	807 GB
Bullet	36 millions	73 millions	4min 19s	12.7 billions	412 GB	610 GB

Figure 5.8: **Volume Rendering with MapReduce** Volume rendering of the earthquake dataset (SF1) using a 100MP image. Table shows volume rendering statistics for other tetrahedral meshes. The Bullet dataset with 36 million tetrahedra is rendered in 4min and 19s.

separate triangle. The reduce phase is modified to perform color, opacity mapping and compositing of the fragments. The accompanying table shows results for a variety of additional meshes. As also shown in the table, the most time-consuming image to render at 100MP is not the largest dataset (Bullet) but the earthquake dataset (SF1). This is due to the many large (and flat) tetrahedra that define empty regions at the bottom of the model. Scalar values of these triangles rarely contribute to the final image, but generate a large number of fragments which cause a more expensive shuffle phase.

5.3.4 Mesh Simplification

To analyze the out-of-core simplification algorithm in the MapReduce model, we use two large triangle meshes as input: the Atlas statue (18GB) and the St Matthew statue (13GB) from the Digital Michelangelo Project at Stanford University. In these tests, we are interested in seeing the effects of scaling the simplifying grid size. The amount of work done in the Map phase should be very consistent, as each triangle must always compute a quadric measure vector and bin its three vertices. Smaller grid sizes force more vertices to coincide in any particular bin, thus changing the grouping and potentially reducing the parallelism in the Reduce phase.

However, the decrease in performance from this should be amortized by the decreased output of the Reduce phase, as fewer triangles will be generated. In the tables of Table 5.3 we observe that this is exactly what occurs. Since our method must be a two pass algorithm in Hadoop, we have included the run times for both jobs (Job 1 and Job 2). Rendered images of simplified models of the Atlas and the St Matthew statue are also shown in Figure 5.9 with the grid sizes varying from 8^3 to 1024^3. Decimation rates for these results are all under 5% and they were all rendered using the renderer proposed in Section 5.3.3.

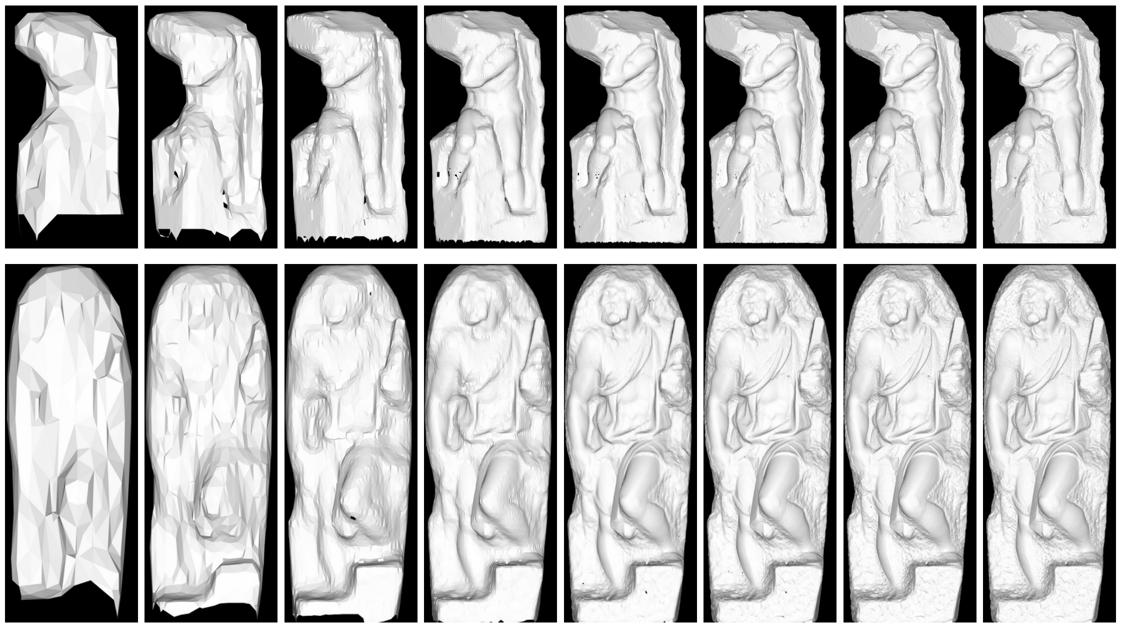
5.4 Discussion

In this section, we discuss some of the "lessons learned" from our experience with MapReduce and Hadoop. For users of visualization techniques, it is difficult to know when the results or workload will push beyond the cluster limits and severely increase runtimes. It was clear from

Table 5.3: Simplification with MapReduce

CLuE time		ATLAS (18 GB)			St MATTHEW (13 GB)		
#Gridsize	#Maps/ #Reduces	Job 1 Time	Job 2 Time	Output Size	Job 1 Time	Job 2 Time	Output Size
8 ^3	256/256	5min 45s	52s	22 KB	5min 45s	52s	23 KB
16^3	256/256	3min 54s	49s	98 KB	3min 54s	49s	105 KB
32^3	256/256	3min 51s	49s	392 KB	3min 51s	49s	450 KB
64^3	256/256	3min 40s	49s	1.6 MB	3min 40s	49s	1.9 MB
128^3	256/256	4min 12s	49s	6.4 MB	4min 12s	49s	7.5 MB
256^3	256/256	3min 50s	49s	26 MB	3min 50s	49s	30 MB

Cluster time		ATLAS (18 GB)			St MATTHEW (13 GB)		
#Gridsize	#Maps/ #Reduces	Job 1 Time	Job 2 Time	Output Size	Job 1 Time	Job 2 Time	Output Size
8 ^3	377/377	58s	56s	22 KB	54s	55s	23 KB
16^3	377/377	58s	55s	98 KB	54s	54s	105 KB
32^3	377/377	55s	54s	392 KB	51s	52s	450 KB
64^3	377/377	57s	54s	1.6 MB	55s	55s	1.9 MB
128^3	377/377	55s	58s	6.4 MB	52s	52s	7.5 MB
256^3	377/377	55s	55s	26 MB	55s	55s	30 MB
512^3	377/377	55s	55s	102 MB	55s	52s	119 MB
1024^3	377/377	55s	57s	399 MB	55s	53s	461 MB



(a) 8^3 (b) 16^3 (c) 32^3 (d) 64^3 (e) 128^3 (f) 256^3 (g) 512^3 (h) 1024^3

Figure 5.9: **Comparisons of Simplification with MapReduce** Simplified meshes using volumes from 8^3 to 1024^3 using decimation rates under 5%.

our experiments that keeping the number of maps and reduces below the maximum task capacity is the first step towards avoiding such pathological cases. While nodes can run multiple tasks, we find that increasing the number of nodes in proportion to the data size provides the most reliable and consistent scalability, suggesting that the overhead to manage additional nodes is not prohibitively expensive.

Additional tasks may improve performance in certain cases, but should not be a default assumption. The results from our exploratory implementations are encouraging and match the scalability we expected, up to a limit. When the size of the output data is unpredictable, as in the case of isosurface extraction, memory requirements can quickly exhaust available resources, leading to disk buffering and ultimately increasing runtime. Scalability, in our experience, is only achieved for data reductive tasks — tasks for which the output is smaller than the input. Most visualization tasks satisfy this property, since they typically render (or generate) data that are smaller than the input mesh or volume. It should also be pointed out that this cost is insignificant when compared to today's standard practice of transferring data to a client, and running a local serial or parallel algorithm. Indeed, the cost of transferring the data to a local server alone dwarfs the cost of any such MapReduce job.

For those interested in developing visualization algorithms for MapReduce systems, MapReduce algorithms was relatively simple. However, as with any highly-parallel system, optimization can be painstaking. In the case of MapReduce, we found that the setup and tuning of the cluster itself were just as important, if not more important, than using the right data format, compressor, or splitting scheme. To analyze the suitability of existing algorithms to the MapReduce model, attention should be paid to where and how often sorting is required. As the model only allows a single sort phase per job, multipass algorithms can incur significant overhead when translated naïvely into MapReduce. Specifically, a MapReduce implementation will rarely be competitive with state-of-the-art methods in terms of raw performance, but the simplicity and generality of the programming model is what delivers scalability and extensibility. Further, the degree of parallelism in the Reduce phase is determined by both the intended output of the algorithm, as well as the distribution of data coming from the Map phase. Careful consideration of the hashing method may or may not have a dramatic effect on the algorithm performance.

We summarize below observations and conclusions we made in our work with the Hadoop system:

- Results from our scaling tests show Hadoop alone scales well, even without introducing optimization techniques.

- Considerations about the visualization output size are very important. Visualization techniques should decrease or keep relatively constant the size of the data in the pipeline rather than increase it. MapReduce was not designed to handle enormous intermediate datasets, and performs poorly in this context.

- From a qualitative standpoint, we found the MapReduce model easy to work with and implement our solutions. Optimization, in terms of compression and data reader/writers required thought and experimentation. Configuring job parameters and cluster settings for optimal performance was very challenging. We feel that this complexity is always inherent in a large distributed environment, and therefore is acceptable. Further, this work can potentially be performed once per cluster, and the cost can therefore be amortized over many MapReduce jobs.

- The inability to chain jobs makes multijob algorithms such as the mesh simplification slightly cumbersome to execute, and more difficult to analyze. Projects such as Pig [71] and Hive [85] that offer a high-level yet extensible language on top of MapReduce are promising in this regard.

- The Hadoop community could greatly benefit from better progress reporting. Uneven distribution of data across reducers may result in display of near completion (*e.g.,* 98%) when in fact the bulk of the work remains to be completed. This is problematic if the user does not know *a priori* what a good reducer number should be, and arbitrarily chooses a high value.

- While at any particular time, job runtimes are fairly consistent, they vary as a whole from day to day. This is most likely due to the state of the HDFS and movement of replicated data. Being aware of these effects is important in order to make meaningful comparisons of performance results. On that note, all data within any one table were generated within a short time span.

5.5 Proposition

The analysis performed in this chapter has shown that the MapReduce model provides an alternative approach to support large-scale exploratory visualization for data that are online on the cloud . The fact that data transfer alone is more expensive than running such a job *in-situ* is sufficient justification, and will become more evident as datasets grow in size. The availability of a core set of visualization tools for MapReduce systems will allow faster feedback and learning from new and large datasets. Additionally, as these systems continue to evolve, it is important for the visualization community to periodically reevaluate their suitability. Here, we provide a baseline for such a comparative analysis. We have shown how three visualization techniques can be adapted to MapReduce. Clearly, many additional methods can be adapted in similar ways, in particular memory-insensitive techniques or inherently parallel techniques. What remains to be investigated is how to combine visualization primitives with conventional data management, query, and processing algorithms to construct a comprehensive scalable visual analytics platform. This requires a flexible, yet efficient, parallel data-flow architecture to support the design as well as executions of these tasks.

CHAPTER 6

INITIAL DESIGN FOR MULTICORE SYSTEMS

As Chapter 3 and Chapter 4 have already shown the importance of streaming capability and a comprehensive updating policy for large-scale visualization pipelines, in this chapter, we propose the design of a flexible dataflow scheme aimed to leverage the two features on multicore platforms. 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, the highlights of our system are the following:

- A new scheme for executing pipelines on multicore 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

A diagram outlining an overview of our system is shown in Figure 6.1. 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

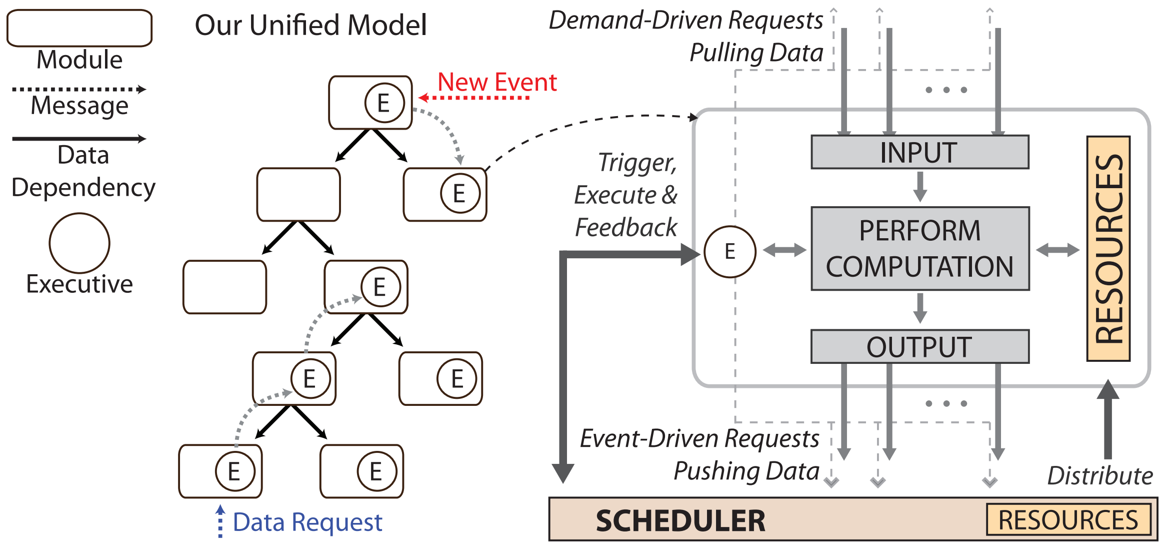


Figure 6.1: **Parallel Dataflow Architecture for Multicore Platforms** The overall architecture of our system. The execution process is triggered by the first request of any pipeline module. The scheduler will distribute resources and schedule the next execution of the whole pipeline. Each module is allocated with a specific sum of resources (of CPU and GPU). Inputs will be converted into the appropriate medium for resource before entering actual computation.

contains a resource specification indicating the number of threads its algorithm can utilize at run-time. This information is used by the scheduler.

6.1 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 6.2. Data are 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 are available (*e.g.,* viewing data as they are 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>)
```

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 6.2, 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
```

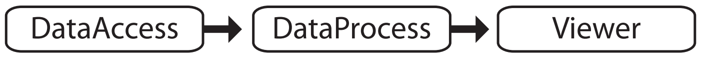


Figure 6.2: **A Simple Rendering Pipeline**

```
...
this->ReleaseInputs();
// process the copied data in local memory
...
```

6.2 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, 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 6.3, 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 regard 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

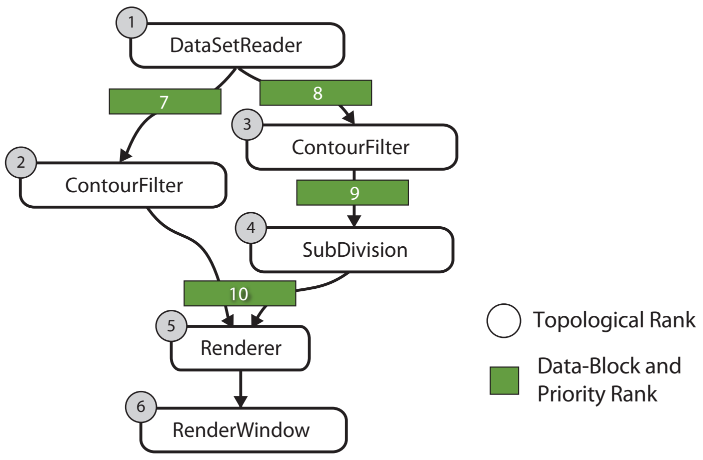


Figure 6.3: **Scheduler Assigments** Streaming priority assignments by our scheduler

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 do not 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 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. In a single branch, subpipelines 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.

6.3 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 dimensions of overlapping regions are defined by the finalization of the stream element 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, *i.e.,* it can be discarded to free memory.

6.4 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. 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 multithreaded 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.

6.5 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 multithreaded 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.

6.5.1 Multicore 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, and is only applicable for subclasses of **vtkThreadedImageAlgorithm**. 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 multithreaded 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 6.4 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 multithreaded 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 6.5. VTK first runs all multithreaded image filters, then executes the single threaded smoothing module only when the multithreaded 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 6.6 clearly shows our execution model is superior to VTK's default threaded model. Table 6.1 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 [69]. 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 [74]. In practice, we have found this method 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.

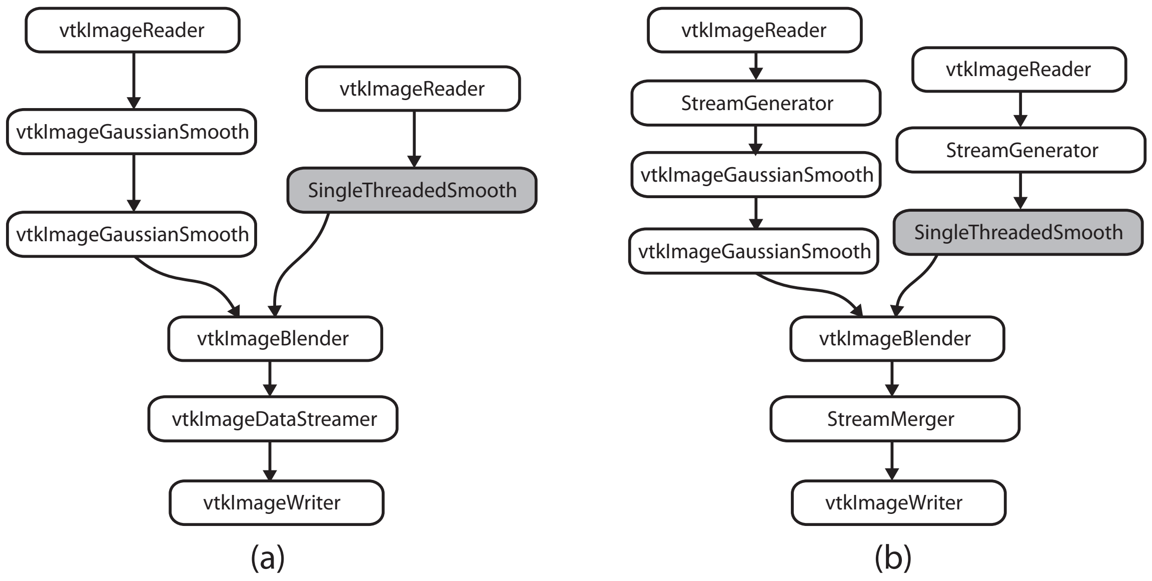


Figure 6.4: **Gaussian Image Pipeline** (a) using VTK; (b) using ours

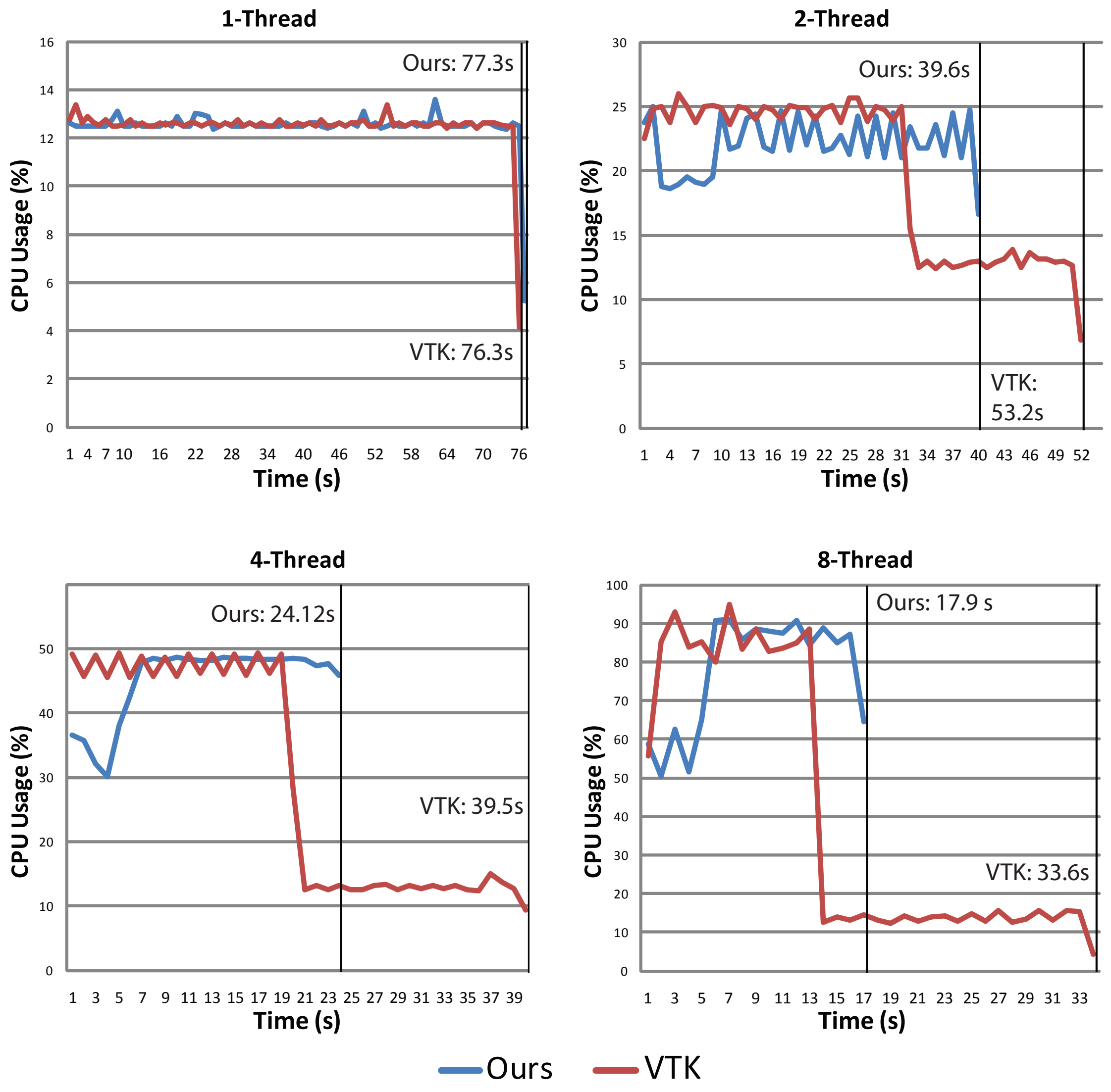
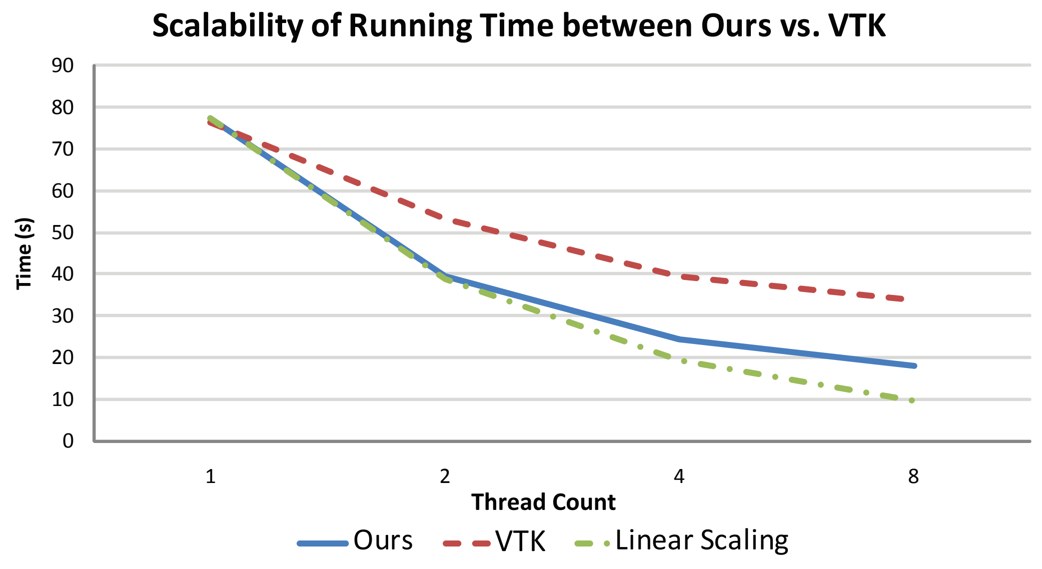


Figure 6.5: **Gaussian Image Pipeline Results** CPU usage from 1 to 8 threads



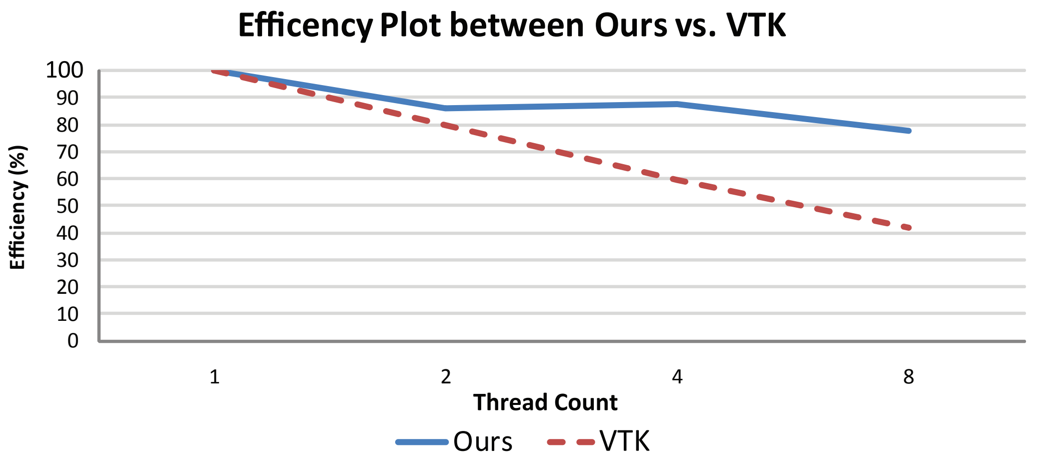


Figure 6.6: **Gaussian Image Pipeline Results** Strong scaling plot (top) and efficiency plot (down)

Table 6.1: Running Time and Efficiency Ratio of CPU Usage Between VTK and Our System for a Simple Gaussian Pipeline.

	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%

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 6.7a 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 are 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 6.7c and Table 6.2.

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 [69]. 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 are rendered in a progressive manner as it is completed. See Figure 6.7b 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. After that, the performance stays at that optimal peak. Table 6.2 illustrates these results with timing numbers. In the scalability plot in Figure 6.7c, we observe a slight super linear speedup probably due to a coherent disk cache when multiple threads access data simultaneously.

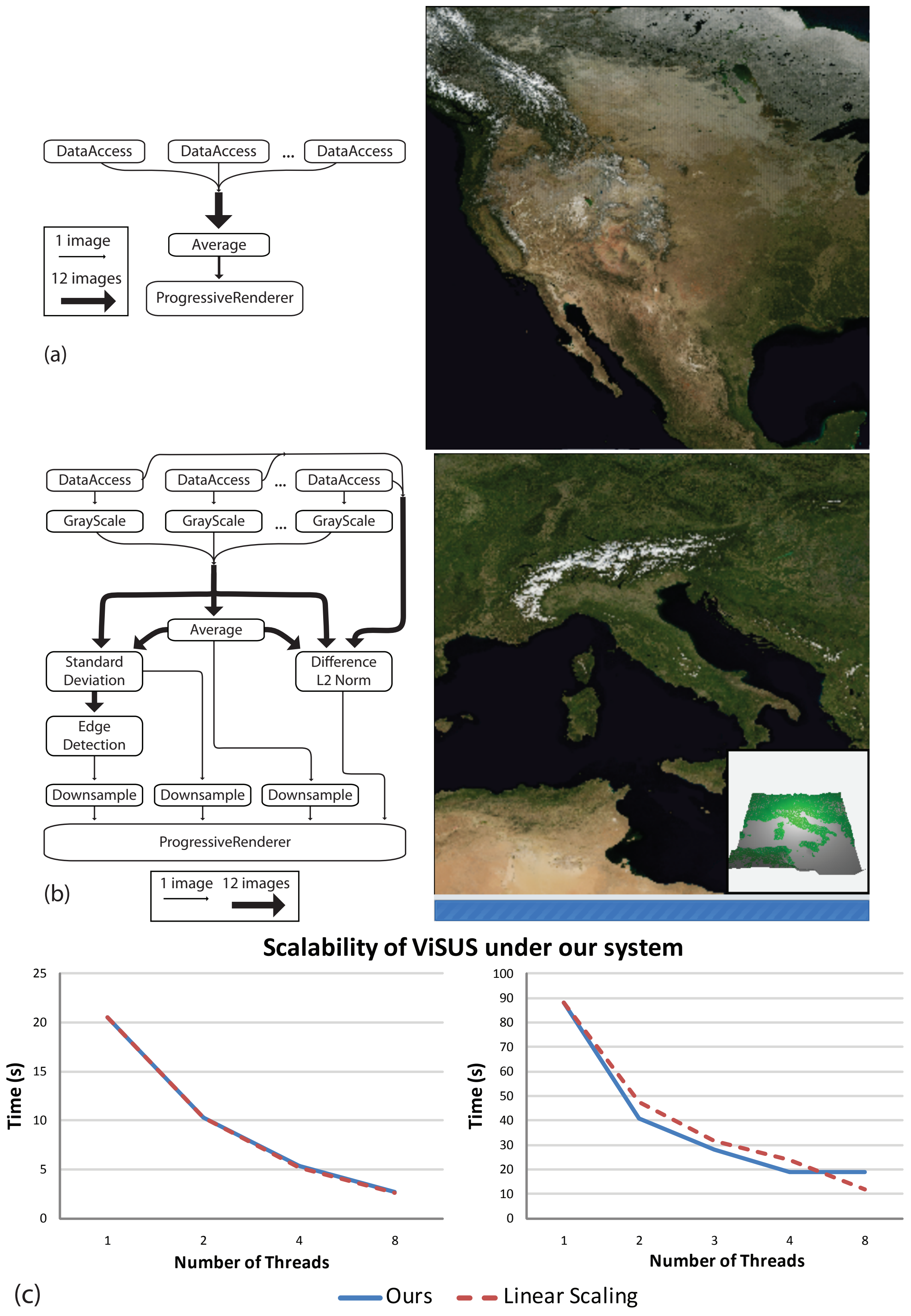


Figure 6.7: **ViSUS Performance with Our System** Two ViSUS pipelines performing: (a) the 12-Month Average, (b) Multi-View selective rendering and (c) the scalability plot.

Table 6.2: Timing for ViSUS

	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

6.5.2 Streaming Tetrahedral Mesh Simplification

To test and demonstrate the flexibility in extending our framework to include unstructured streaming capabilities, we modified the implementation of streaming tetrahedral mesh simplification technique from Chapter 3. 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.

In our system with the generalized streaming scheme extension, we are able to construct and execute the corresponding pipeline as shown in Figure 6.8. 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.

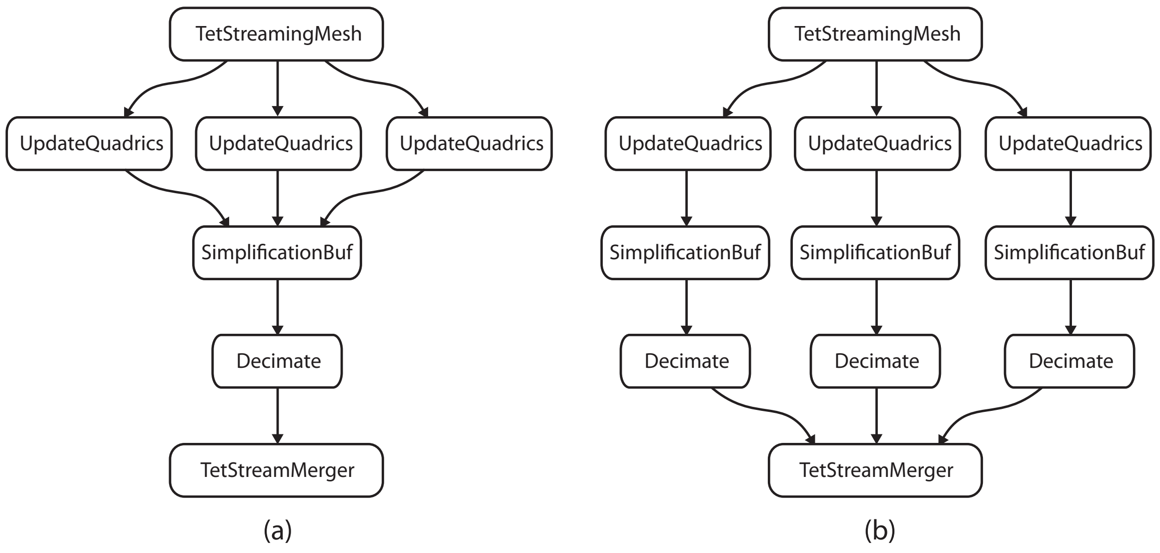


Figure 6.8: **Streaming Simplification with Our System** Streaming simplification of tetrahedral meshes under our system (a) with data-parallelism and (c) with a complete parallelism

Figure 6.8a 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 if the single module is executed in sections without any changes to the code. Due to the increase in performance inherent in 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 6.8b 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 6.3 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 6.8c, we have converted the pipeline to have complete parallelism; duplicating all three processes into three concurrent executions. As we can see in Table 6.3 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 constraints.

Even though an optimal 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.

6.6 Discussion

In this chapter, we propose new techniques for exploiting multicore 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 while also combining the

Table 6.3: Simplification Time for Achieving 10% Resolution

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

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 both multicore, 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, this should only be a lower bound on the performance increase possible. We have designed this scheme with scalability as a primary consideration. We hope to expand this scheme to utilize 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. In the next chapter, we introduce HyperFlow, a parallel dataflow architecture that can take advantage of both CPUs and GPUs to increase the system performance through various parallel constructs.

CHAPTER 7

HYPERFLOW: AN EFFICIENT DATAFLOW ARCHITECTURE FOR SYSTEMS WITH MULTIPLE CPUS AND GPUS

One limitation of the parallel dataflow architecture proposed in Chapter 6 is that resources (*e.g.,* threads) are created and destroyed every time a module is executed, thus creating a large overhead. Also, the proposal is not suitable to a multi-GPU system because modules are allocated with a fixed context to run, and thus, it is unintuitive and cumbersome for kernels to execute on a specific device inside a given module implementation. In addition, there is no coherency and synchronization across threads, with no support for favoring data blocks to stay on the same thread or GPU device. This is one of the requirements found in such pipelines mentioned in Chapter 5. Moreover, since the framework is implemented inside VTK, it also bears a major limitation of VTK (for backward compatibility), that is the over-complication in execution path and pipeline structure. In this chapter, we propose a light-weight dataflow architecture, that is designed to support streaming at different levels of granularity, with data structures that can be "packed" and "unpacked" according to the processing elements that will perform the computation on each stream. We addressed the short-comings of the architecture proposed in Chapter 6 by introducing the notion of *Virtual Processing Elements* (VPE), *Task-Oriented Modules* (TOM) and a scheduling mechanism that controls the assignment of tasks to elements.

7.1 HyperFlow Architecture

In this section we describe HyperFlow, our framework for parallel execution of dataflows on heterogeneous systems. Figure 7.1 shows a high-level description of the HyperFlow architecture. At the application developer level, our framework provides a C++ template API that dynamically allows construction and execution of scientific pipelines. By design, HyperFlow separates module *specification* (*i.e.,* its input and output ports) from *implementation*, thus allowing pipelines to be constructed without prior knowledge of the underlying system's computational resources. Thus, a single module may have more than one implementation and depending on the system it runs on, the

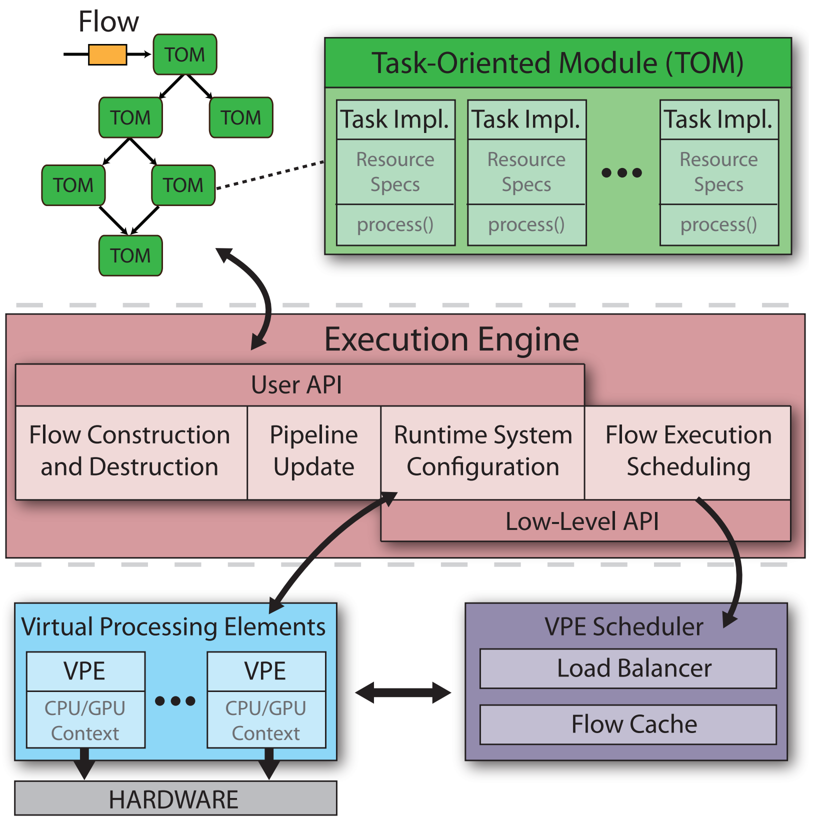


Figure 7.1: **HyperFlow Architecture Overview**

most appropriate one(s) will be executed. This is one of the main advantages of HyperFlow over other dataflow systems.

Three abstraction levels can be identified in the HyperFlow architecture. At the highest level, pipelines are defined as interconnected Task-Oriented Modules (TOMs), and executed as a set of token-based data instances called Flows. Immediately below this, the Execution Engine (EE) manages Flows and prepares them for execution. Finally, at the lowest level, a scheduler distributes tasks to Virtual Processing Elements (VPEs), which form an abstraction layer over the actual computing resources available in the system. We detail these components in the sections that follow.

7.1.1 Pipeline Construction Using Task-Oriented Modules

A pipeline in HyperFlow consists of a set of interconnected Task-Oriented Modules (TOMs). Each TOM defines and holds parameters needed for a specific computational task, such as the number of input and output ports. To allow the same pipeline to be executed across a wide range of different computational resources transparently, TOMs do not explicitly store the task implementation. Instead, they store a list of *task implementation* objects, which are dynamically scheduled to perform the actual computation on a given set of inputs. This separation of task specification and implementation is one of the main differences between HyperFlow and similar systems. The requirement for the execution of a TOM in a given system is that it should have a task implementation that matches the system resources (*e.g.,* CPUs or GPUs).

The HyperFlow API is composed of a set of C++ classes and functions that exposes various ways to construct scientific pipelines to the developer. The base class to manage all ports and implementation objects is called **TaskOrientedModule**. Developers who wish to build their own modules should instantiate or extend this class. Listing 7.1 demonstrates the basic interface of **TaskOrientedModule** and shows a simple example on how to construct a pipeline in HyperFlow. The **SourceImpl**, **FilterImpl** and **SinkImpl** classes must inherit from another important HyperFlow class, **TaskImplementation**, which is used as the base class to specify task implementation objects.

7.1.2 Pipeline Execution Using Flows

Similar to token-based hardware dataflow architectures, HyperFlow executes pipelines by sending instruction tokens to processing units for execution, and sending data tokens back as results. The fundamental data container is called a *flow*, which passes between connected TOMs in the pipeline. Each flow contains a data reference along with its meta information, such as source and destination modules, to control and track pipeline executions. Flows are ordered for execution using priority ranks that are based on their time steps or could be defined by users.

Execution is triggered when a flow is sent to a pipeline module. Flows are classified as *waiting, live, or dead* depending on their status (waiting for execution, executing, and finished, respectively). After a module completes execution, new flows might be generated for subsequent modules in the pipeline to process.

HyperFlow execution is similar to traditional dataflow architectures, with the exception that it does not restrict pipelines to be directed acyclic graphs (DAGs). While other systems make use of DAGs to guarantee the module execution order, it comes at a cost of limited pipeline parallelism and static execution. HyperFlow, on the other hand, supports dynamic execution, without DAG restriction for enabling pipelines with feedback communication. In order to determine when a module is ready to execute, HyperFlow maintains a flow cache that holds incoming flows temporarily until all of the input data arrive, and then trigger the module execution.

7.1.3 Virtual Processing Elements (VPEs)

One of the main features of HyperFlow is the ability to schedule pipeline modules for execution across heterogeneous computing resources such as GPUs and CPUs in a transparent way. In order to manage different computational resources, we introduce the fundamental concept of *Virtual*

```
class TaskOrientedModule {
public:
  TaskOrientedModule(int numberOfInputPorts,
                     int numberOfOutputPorts);
  void addImplementation(TaskImplementation* impl);
  static void connect(
      TaskOrientedModule* srcModule, int srcPort,
      TaskOrientedModule* dstModule, int dstPort
    );
  // Additional parameters if subclassed
  ...
};

// Construction of the pipeline
TaskOrientedModule Source(0, 1, "Image Reader")
Source.addImplementation(new SourceImpl());

TaskOrientedModule Filter(1, 1, "Gaussian Blur")
Filter.addImplementation(new FilterImpl());

TaskOrientedModule Sink(1, 0, "Viewer")
Sink.addImplementation(new SinkImpl());

TaskOrientedModule::connect(&Source,0,&Filter,0);
TaskOrientedModule::connect(&Filter,0,&Sink,0);
```

Listing 7.1: **Task-Oriented Module Interface** public interface of the **TaskOrientedModule** class, and example showing how to construct a simple pipeline using the HyperFlow architecture.

Processing Elements (VPEs), which are abstract layers that manage execution contexts of specific computing resources.

VPEs are designed as a service that waits for tasks to be executed when a resource becomes available. Once this happens, a VPE first verifies if all input data are properly transferred to their current context, since data may reside in different, mutually inaccessible, memory areas, such as CPU and GPU memory. In HyperFlow, we assume that each VPE, regardless of their underlying hardware, has access to main CPU memory. Therefore, a data-transfer path between potentially very different VPEs is always possible. This does not *imply* that all memory transfers must be made through host memory, since users are free to implement their own efficient data transfer routines by extending the class **Data**, which is described in more detail in section 7.1.6.

By default, HyperFlow defines two types of VPEs: CPU Threads and CUDA Devices, which map to individual CPU threads and CUDA-enabled hardware, respectively. The architecture also naturally supports customizable VPEs, since users can extend the **VirtualProcessingElement** class, allowing potentially arbitrary computing resources to be transparently integrated into HyperFlow. Among others, the two main functions that need to be implemented in VPE subclasses are **enterEx-**

ectution() and **leaveExecution()**, responsible for configuring and finalizing execution contexts, respectively. For the case of CPU Threads, for instance, this is where thread affinity is enforced.

7.1.4 Execution Engine

The main control model of HyperFlow is the *Execution Engine* (EE), which is responsible for assigning flows for execution and controlling the execution of VPEs. At runtime, the EE initializes a set of VPEs that are mapped to the corresponding computing resources available. By default, HyperFlow supports a quick configuration of VPEs using the number of CPUs and GPUs available.

The EE manages all flows passing through the system, as well as the status of all VPEs. A polling thread is the main control module of the EE, which waits in a nonblocking way for either a waiting flow to be generated in the TOMs, or a resource VPE to become available. Once a waiting flow is generated, the EE dispatches it to the VPE scheduler for execution. This polling thread also instructs the VPE scheduler when a VPE resource becomes available.

Tracking flows during execution allows the EE to maintain a proper execution order with precise memory management regardless of pipeline topology. In HyperFlow, an individual Flow can be created using the following API call:

```
Flow *createFlow(Flow *refFlow=NULL);
```

If a reference Flow is supplied to this function, the new Flow will share the same identification and data as the reference. Once a Flow is created, it can be sent to the VPE scheduler using the method **sendFlows()**. All Flows sent in the same call to this method are assigned the same identification, and thus executed with the same priority. The interface for **sendFlows()** is:

```
void sendFlows(int n, Flow *flows, RETTYPE ret, bool all);
```

The value of **ret** is used to determine when this function should return, and can be either **RET_IMMEDIATELY**, **RET_ON_DEPLOY** or **RET_ON_FREE**. **RET_IMMEDIATELY** determines that the function should just add the Flows to the scheduler queue and return immediately, while **RET_ON_DEPLOY** makes the function return only when Flows have been assigned to VPEs. Finally, **RET_ON_FREE** forces the function to wait until Flows have been completely executed and their resources freed. The final argument determines if these return conditions should be met by all Flows or just one of them. This allows developers to write streaming applications exploiting different levels of parallelism depending on the application. To illustrate this, we show below how a typical streaming application can be constructed in HyperFlow:

```
while <data is coming> do {
  Flow *flow = engine->createFlow(data);
  engine->sendFlows(1, &flow, RET_ON_DEPLOY, true);
}
```

Typically, initial modules in a pipeline have no input ports. Nevertheless, they need to be executed to generate Flows that will trigger the execution of the entire pipeline. This is done in HyperFlow by sending an empty Flow to the initial modules, which can be done conveniently using another EE method, **sendUpdate()**.

The EE also provides different types of waiting conditions that allow client-customized execution. Some examples include **waitForAllDone()** and **waitForAvailable()**, which causes the EE to stop processing until all currently live flows finish or a VPE becomes available. This allows synchronization points to be added to pipeline execution, a feature valuable for debugging purposes.

7.1.5 VPE Scheduler

The VPE scheduler is responsible for managing and scheduling waiting flows for execution on available VPEs. Internally, it maintains two priority queues of flows: a *waiting queue* which contains flows generated by the EE, and a *live queue* of flows currently executing. Flows sent from the EE are initially added to the VPE waiting queue and sorted by their identification number, which defines the flow execution order. While it is possible to order all flows statically before execution, it also requires the system to keep the pipeline topology statically as a DAG. HyperFlow only needs to look at the source and destination module of each flow to determine this order. Moreover, the static ordering will limit the parallelism in execution, in particular the pipeline parallelism, when there is more than one data piece streaming through the system.

Flow identification also allows the scheduler to cache input flows for determining when a module has all of its input ready. Since flows typically arrive at their destination modules asynchronously, the scheduler can only dispatch input flows when the last one arrives at their destination module. This cache is stored as a hash map keyed by a pair of destination modules and the flow identification.

Just before the VPE scheduler assigns a flow to a task implementation, it checks whether the VPE associated with the flow origin can be reused to execute the destination module. This is to increase memory coherency as well as avoiding expensive memory transfer between CPUs and GPUs. If the same VPE could not be used, a data transfer is needed. In this case, the scheduler will first tell the source VPE to convert the flow data to the main CPU host memory. The VPE has to be responsible for this conversion instead of the VPE scheduler because it is not always possible for the VPE scheduler to access the internal memory context of each flow due to driver regulation. After the conversion is completed, newly converted flows will be passed on to the destination VPE for further processing. The fact that HyperFlow automatically takes care of this conversion process in a thread-safe manner allows GPU code to not worry about thread migration issues. For instance, in CUDA, GPU memory that is allocated inside a thread can only be used or freed on that same thread. Thus, in order to perform computation across two GPUs, users have to normally go into the first GPU thread and copy the data to main memory. Then in the second thread, another copy has to be performed to transfer the data to the second GPU thread. Since HyperFlow always performs data transfer in this fashion, this is no longer a problem. Figure 7.2 shows the pseudo-code of the main procedure of the VPE scheduler.

7.1.6 Memory Transfer

HyperFlow supports execution on different devices (such as CPUs and GPUs, for instance) interchangeably, which requires the transfer of data objects between devices. For this purpose, we enforce that all data objects must inherit from the predefined **Data** class. This class implements

```
Algorithm: scheduleNext (List VPEs)
$W \leftarrow$ priority queue of waiting flows;
$R \leftarrow$ priority queue of live flows;
$C \leftarrow$ flow input cache;
foreach $f$ in $W$ do
    $C = C \oplus f$;
    if $\exists g \in (W \cup R) : C[f.id] \not\equiv$ all inputs of $g$ then
        remove $f$ and continue;
    end
    $I \leftarrow$ findImplementation(f.sourceVPE, f.dstModule);
    if $\exists I$ then
        $R \Leftarrow (f.\text{sourceVPE}, I, C[f.id])$;
        remove $f$ and continue;
    else
        if there is another flow to schedule then
            continue;
        end
    end
    foreach $V$ in $VPEs$ do
        $I \leftarrow$ findImplementation($V$, $f$.dstModule);
        if $\exists I$ then
            $R \Leftarrow (V, I, C[f.id])$;
            remove $f$ and continue;
        end
    end
end
Algorithm: findImplementation (VPE V, TOM M)
if $\exists I \in$ M.implementations: V can process I then
    return I
end
```

Figure 7.2: **Scheduling Logic of HyperFlow** Pseudo-code for the main strategy of the VPE scheduler.

reference-counted objects that are required to be copied on write by default. Thus, tasks are encouraged to read from but not to write to input data. This allows HyperFlow to pipe a single output data to multiple module inputs simply as references to the same **Data** object. This approach has two advantages: one is the obvious smaller memory footprint, since multiple references to the same object use much less space than copies of this object. Also, the copy-on-write approach implies that HyperFlow has no read-after-write or write-after-write hazards, typical of many concurrent programming approaches. However, for a pipeline that passes a large amount of data, data copy can be very expensive. In this case, users may override the **Data** class to not follow this approach. This means that data access policy has to be handled with great care by module implementations to avoid the above hazards.

A data object in HyperFlow must define two properties: its *medium* and *conversion procedure*. Currently there are two types of data mediums in HyperFlow, **DM_CPU_MEMORY** and **DM_GPU_MEMORY**, which specify the main memory and GPU device memory, respectively. The purpose of the conversion procedure, **createInstance(DMTYPE medium)**, is to allow the engine to request a new data object with a specific type of medium that is equivalent to the original data. This allows data to be passed between different VPEs in a customizable manner. As was discussed previously, we require that all **Data** objects implement at least a conversion to and from main CPU memory, to ensure that there will always be at least one data-transfer path between any pair of VPEs. An example demonstrating the implementation of this conversion method is shown in Listing 7.2.

Similar to data creation, data deletion in HyperFlow must necessarily run on the original context in which the **Data** object was created. Therefore, when a **Data** object reaches zero references, the EE schedules the VPE to release its data resources, instead of releasing the memory directly.

Similar to data creation, data deletion must run on the original context in which the **Data** object was created. Therefore, when a **Data** object reaches zero references, the EE schedules the VPE to release its data resources, instead of releasing memory directly.

7.1.7 Classes of Parallelism Supported

HyperFlow supports a combination of the three fundamental types of parallelism present in typical dataflow architectures. As illustrated in Figure 2.2, given a single input data block, *task parallelism* focuses on assigning independent tasks (or modules) of a pipeline to different processes (or threads). In contrast, *data parallelism* focuses on running the same tasks concurrently on different data blocks.

Both types of parallelism are natively supported in HyperFlow using the default scheduling strategy. As long as there are available VPEs, the VPE scheduler concurrently assigns flows with

```cpp
class DImageData : public Data {
public:
  DImageData(BaseImage *img, DMTYPE medium) {
    // Store image with the medium type
    // ...
  }
  virtual Data* createInstance(DMTYPE targetMedium) {
    // Convert from CPU Image to CUDA Image
    if (this->medium==DM_CPU_MEMORY &&
        targetMedium==DM_GPU_MEMORY) {
      CUDAImage *newImage = new CUDAImage();
      cudaMemcpy(newImage, this->image, cudaMemcpyHostToDevice);
      return new DImageData(newImage, DM_GPU_MEMORY);
    }
    // Convert from CUDA Image to CPU Image
    else if (this->medium==DM_GPU_MEMORY &&
             targetMedium==DM_CPU_MEMORY) {
      CPUImage *newImage = new CPUImage();
      cudaMemcpy(newImage, this->image, cudaMemcpyDeviceToHost);
      return new DImageData(newImage, DM_CPU_MEMORY);
    }
    // Use default transfer methods for CPU-CPU, etc.
    else
      return Data::createInstance(medium);
  }
};
```

Listing 7.2: **Data definition in HyperFlow** Example of how an interchangeable data structure could be defined in HyperFlow.

different ids or without any upstream/downstream relations (*independent flows*). Since independent tasks and distinct input data blocks produce independent flows, HyperFlow can interchangeably coordinate between task and data parallelism. Because flows with smaller ids have higher priority on the waiting queue, HyperFlow favors task parallelism over data parallelism. This allows HyperFlow to maintain a small memory footprint with better cache behavior, as an individual data block travels as far down the pipeline as possible before new data blocks are processed.

The third class of parallelism is *pipeline parallelism* focusing on partitioning a pipeline into subnetworks that are capable of operating on different data blocks concurrently. This is similar to data parallelism but without replicating modules. Since HyperFlow automatically instantiates modules as part of the data-parallelism paradigm to utilize all computational resources, each module may have more than one instance at any time, thus not strictly enforcing pipeline parallelism. In this case, pipeline parallelism is rather promoted to "streaming" data parallelism, where threads are only required to execute different data-blocks on a subnetwork. However, if a group of task implementations is not reentrant, *i.e.,* cannot be run with multiple instances, data will be passed through that subnetwork with true pipeline parallelism. Currently, HyperFlow allows users to specify a task implementation to be not reentrant by using an internal mutex locking mechanism.

An example of a 12-step pipeline running with various types of parallelism is shown in Figure 7.3 and Figure 7.4. A flow between modules A and B is identified by (A,B), and is colored green for the first data block and orange for the second. The execution happens as follows: (1) first (green) flow $(\emptyset,A)$ is generated and goes to the waiting queue; (2) then (orange) flow $(\emptyset,A)$ is generated and goes to the waiting queue, while green flow moves to the live queue and triggers execution of module green A in the VPE; (3) orange flow moves to the live queue and triggers orange A to execute concurrently with green A in the VPE; (4) green A terminates, and generates green,

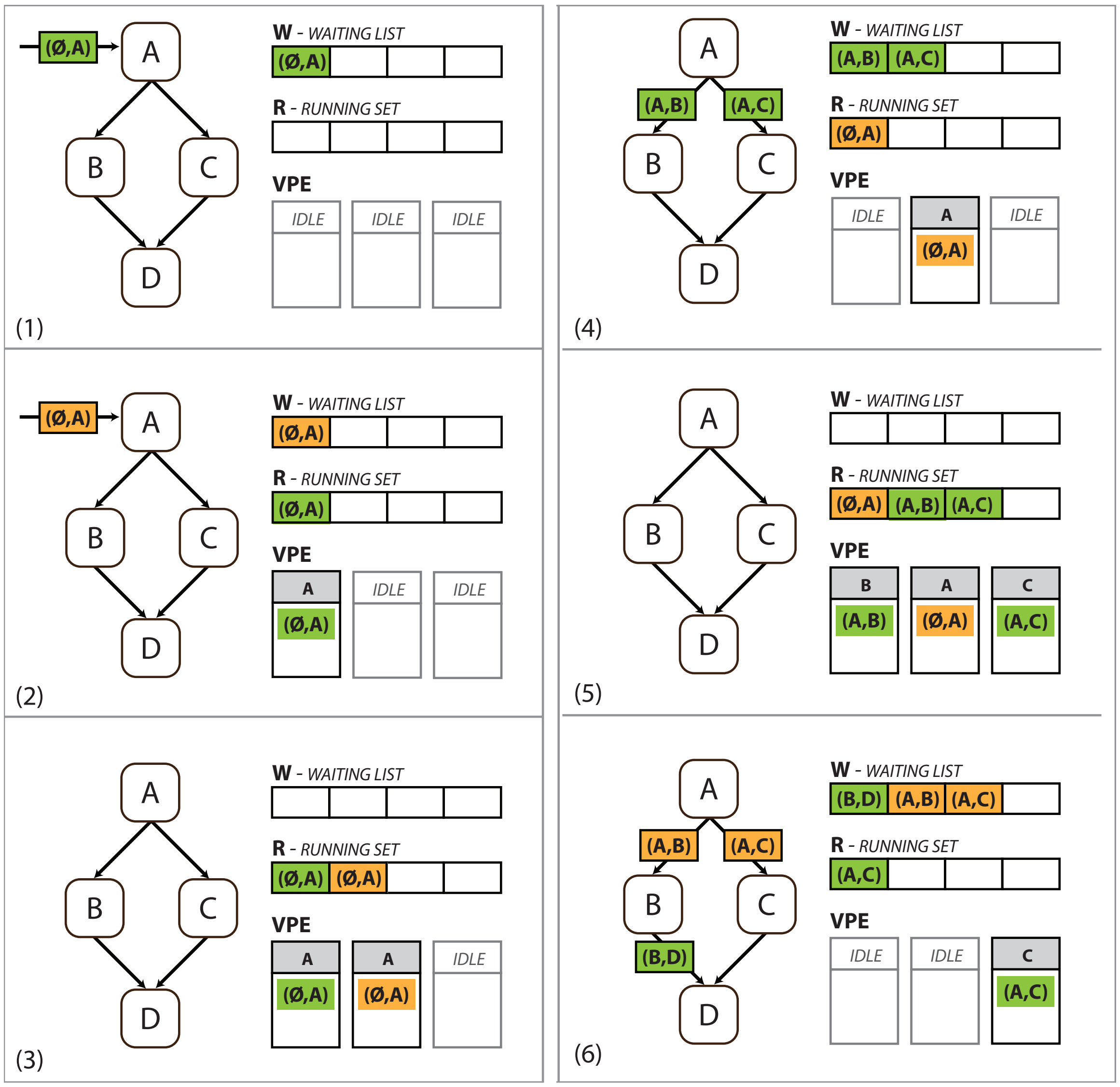


Figure 7.3: **Execution and Parallelism in HyperFlow** Processing two data blocks, part one.

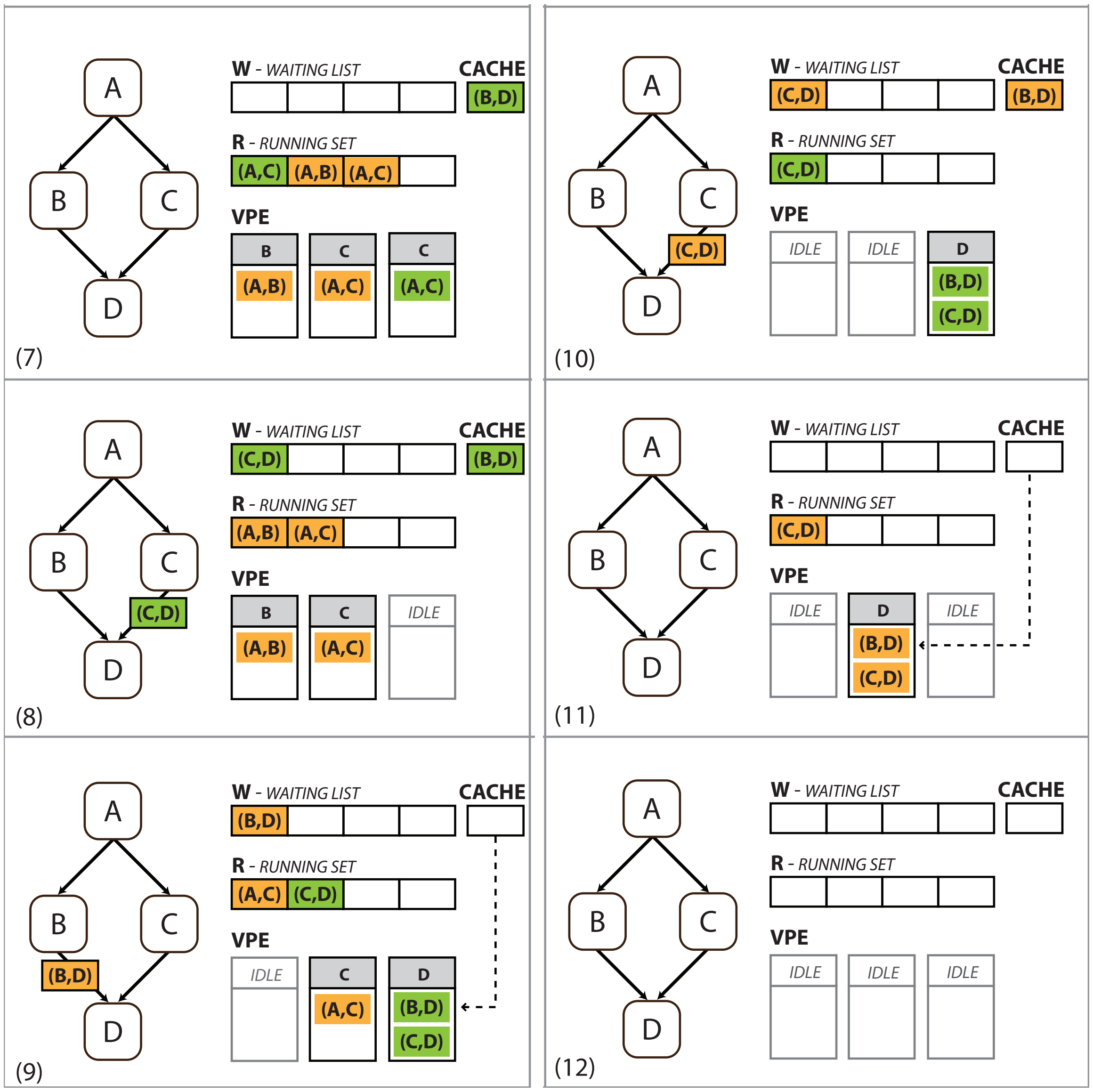


Figure 7.4: **Execution and Parallelism in HyperFlow** Processing two data blocks, part two.

(A,B) and (A,C) flows; (5) green (A,B) and (A,C) become live and trigger execution of green B and C modules; (6) orange A terminates, and generates orange (A,B) and (A,C). green B terminates and generates green (B,D); (7) orange (A,B) and (A,C) become live and trigger execution of B and C; green (B,D) moves to the cache since D can only execute when green $(C,D$ is live; (8) green C terminates and generates green (C,D); (9) orange B terminates and generates orange (B,D); green (C,D) becomes live, and along with green (B,D) in the cache trigger D to execute; (10) orange C terminates and generates orange (C,D); orange (B,D) moves to the cache; (11) orange (C,D) becomes live, and along with orange (B,D) in the cache trigger D to execute; green D terminates; (12) orange D terminates. Task parallelism and data parallelism are in effect at steps (5,7) and (3,7), respectively. Pipeline parallelism occurs at steps (5,9), and the flow cache is used in (7-11).

7.2 Synthetic Applications

We demonstrate the potential of HyperFlow through a set of synthetic and real applications. In this section we focus in a set of micro-benchmarks designed to stress test both computational performance and data bandwidth. Experiments were conducted on two systems: (A) an Intel Quad Core i7 @ 2.67 GHz (8 HT cores), 6 GB of RAM; 3 GPUs: 2x GeForce GTX 295 with 1 GB each, 1x Tesla C1060 with 4 GB; (B) 16x Intel Xeon with 6 cores @ 2.67 GHz (96 cores), 1 TB of RAM.

7.2.1 Micro-Benchmarks

Micro-benchmarks were designed to evaluate scheduling as well as data handling strategies. The idea is to implement modules that keep the computational device occupied just like a regular implementation, without the need of informing actual code. To allow different pipeline configurations, we designed a framework that generates benchmark programs from a description of basic network topology and module description.

Each module contains a list of implementations, which are parametrized by an expression describing execution time as a function of the inputs of a module, and input/output ratio for each execution. This description consists of a python script that allows to procedurally generate arbitrarily large networks. Figure 7.5 illustrates the main benchmark classes used: asymmetric, split-join, and scan.

The asymmetric benchmark is used to stress unbalanced computation, where data reach the destination much earlier through one path, causing many flows to be queued up at the final module. It is desirable that flow scheduling in the longer path is not affected by this bottleneck. This benchmark is parametrized by the number of modules on the longest path.

The split-join benchmark model pipelines that divide the input into independent sequences of computation. This benchmark is parametrized by its length and width. This benchmark has the highest degree of parallelism up until the last module, which merges the pieces back together. A linear benchmark that models a typical streaming pipeline can be seen as a subset of this benchmark. The scan benchmark model the typical parallel operation by the same name, using the number of reduction steps as parameter.

7.2.2 Performance Results

We conducted several experiments using a great variety of micro-benchmarks, and list below the main results obtained. To evaluate the performance of the scheduler, we extracted execution traces

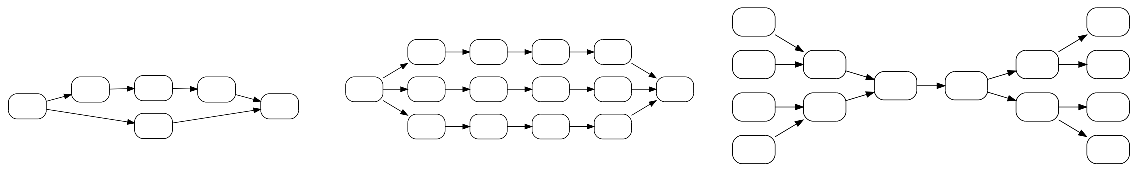


Figure 7.5: **Topology of Micro-benchmarks**

containing start and finish times for each module execution. We investigated these data using two types of visualization.

The first visualization illustrates the animation of flows passing through the micro-benchmark pipelines. Figure 7.6 shows the visualization of the scan pipeline running in HyperFlow. An animated Gantt chart is displayed at the bottom, showing when each VPE is active processing a flow.

The second visualization is shown in Figure 7.7, where a Gantt chart of the complete execution is presented, aligned with a plot of the computing resources utilization. All the benchmarks pre-

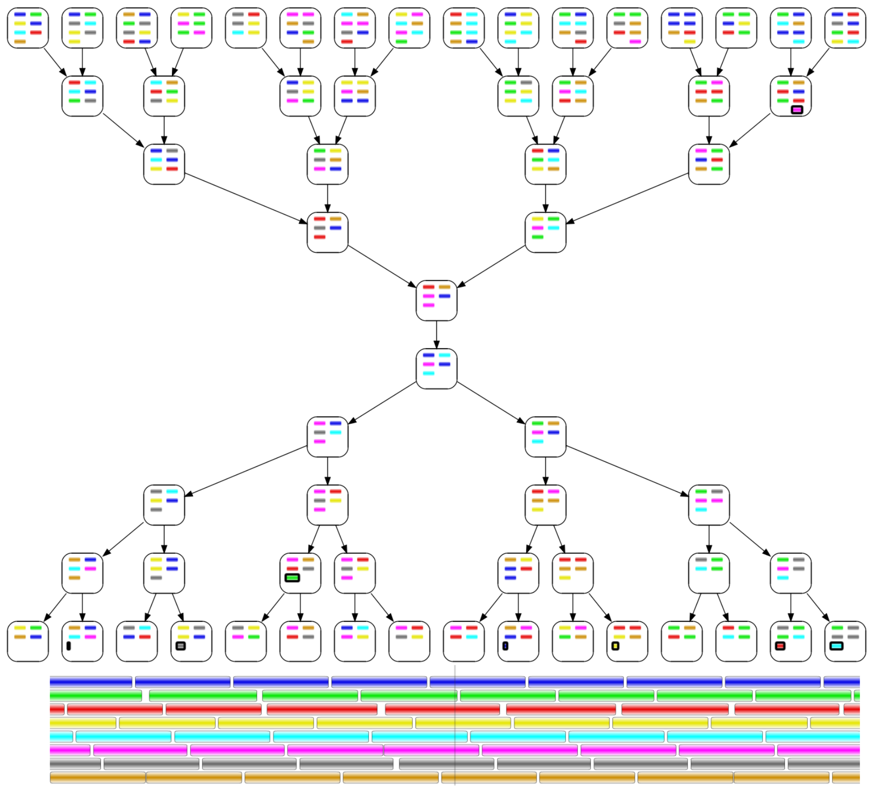
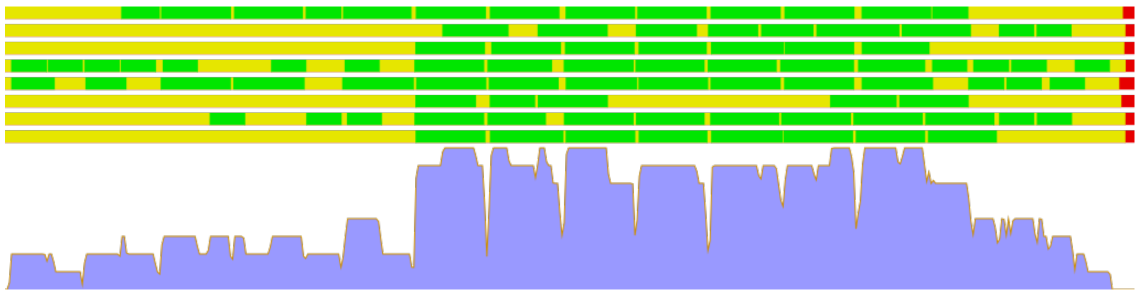
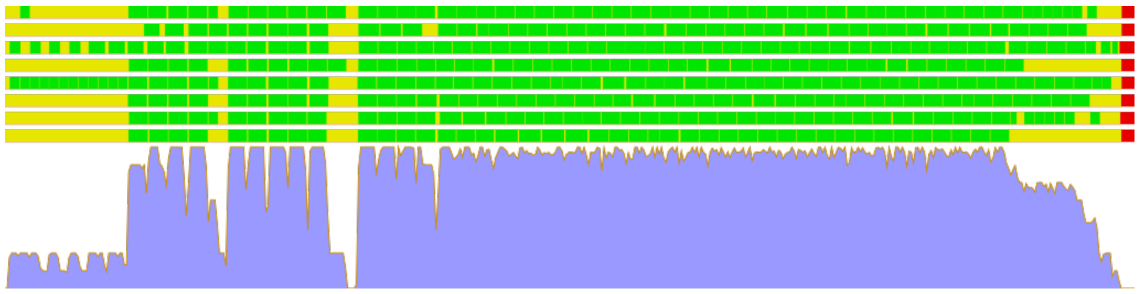


Figure 7.6: **Snapshot of the Micro-benchmark** running with eight input flows and eight VPEs, at $\frac{2}{3}$ of the total execution time. The first four inputs are completely processed at this point, as the first four flows on every module are inactive.


(a) asymmetric with $length = 10$ and 57.4% average utilization.


(b) split-join with $width = 6$, $length = 8$ and 78.9% average utilization.

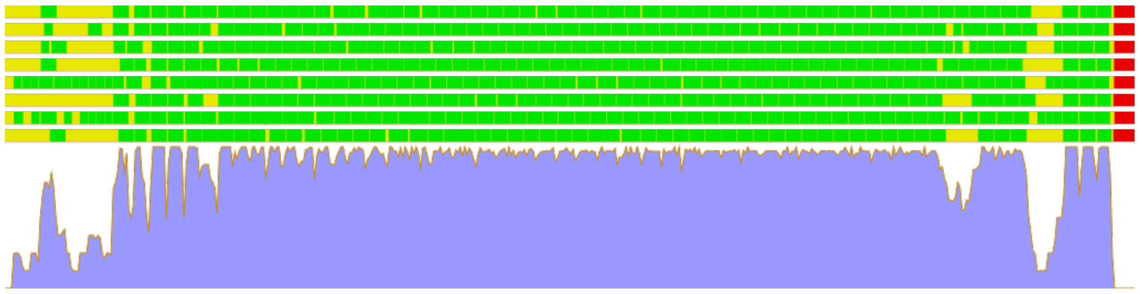

(c) scan with $levels = 4$ and 83.4% average utilization.

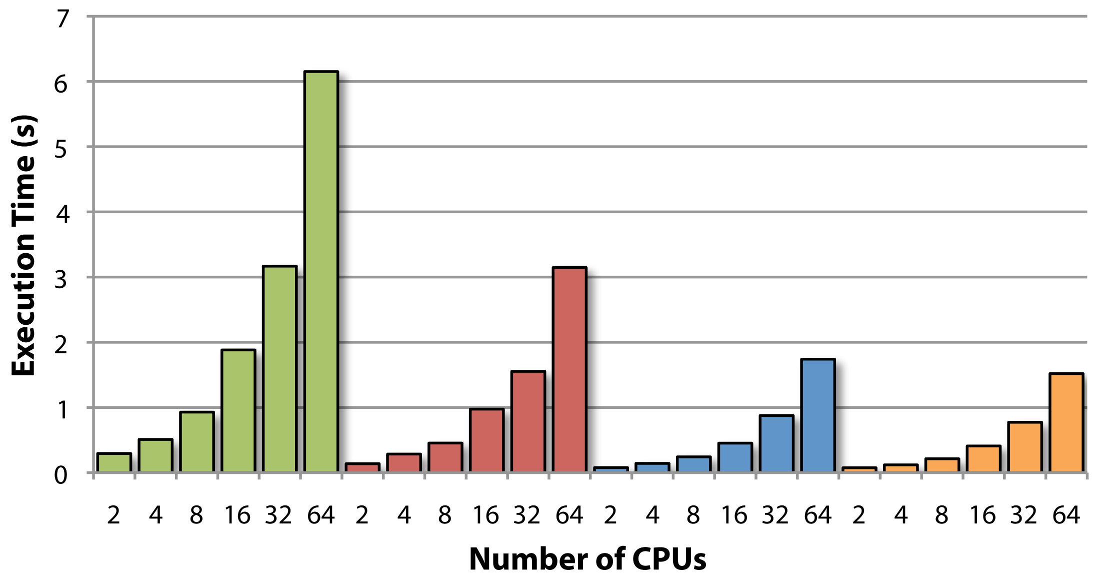
Figure 7.7: **Micro-benchmarks Gantt Charts** Gantt charts and resource utilization when processing 8 input flows. Top half shows the Gantt chart for the complete execution of each benchmark, with one horizontal bar for each VPE. A green state corresponds to the execution of a module; yellow represents an idle state; red represents the end of execution. Bottom half shows the resource utilization plot (ratio of active VPEs / VPEs available.)

sented have the same module parameters, and were executed with 8 input flows. The asymmetric benchmark shows a lower average utilization due to the simplicity of the pipeline, causing a lower workload.

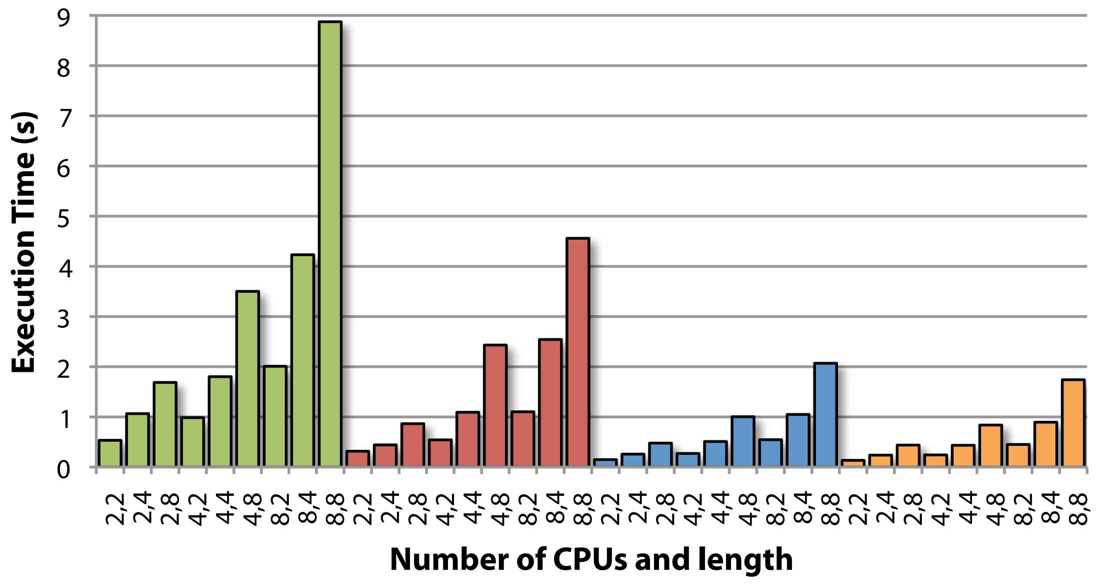
Figure 7.8 illustrates performance results for micro-benchmarks running with 1000 flows. Simulations with 10 and 100 flows produced similar graphs.

7.3 Real-Case Applications

In this section we report results in four distinct applications, which allow to evaluate several aspects of HyperFlow.



(a) asymmetric benchmarks with $length = 2, 4, 8, 16, 32, 64$



(b) split-join benchmarks with $width, length = 2, 4, 8$

Figure 7.8: **Micro-benchmark Performance for HyperFlow** Micro-benchmark simulation with 1, 2, 4 and 8 threads (green, red, blue and orange bars, respectively) and 1000 flows.

7.3.1 High Throughput Image Processing

The first application to illustrate HyperFlow is a simple edge detection framework capable of dealing with multiple images concurrently (see Figure 7.9). The pipeline starts with an image source, which loads several images on the fly and stream them to a Task Oriented Module that performs a Gaussian blur over the image. The VPE scheduler creates multiple instances of this TOM to handle incoming images, both on the CPU and GPU.

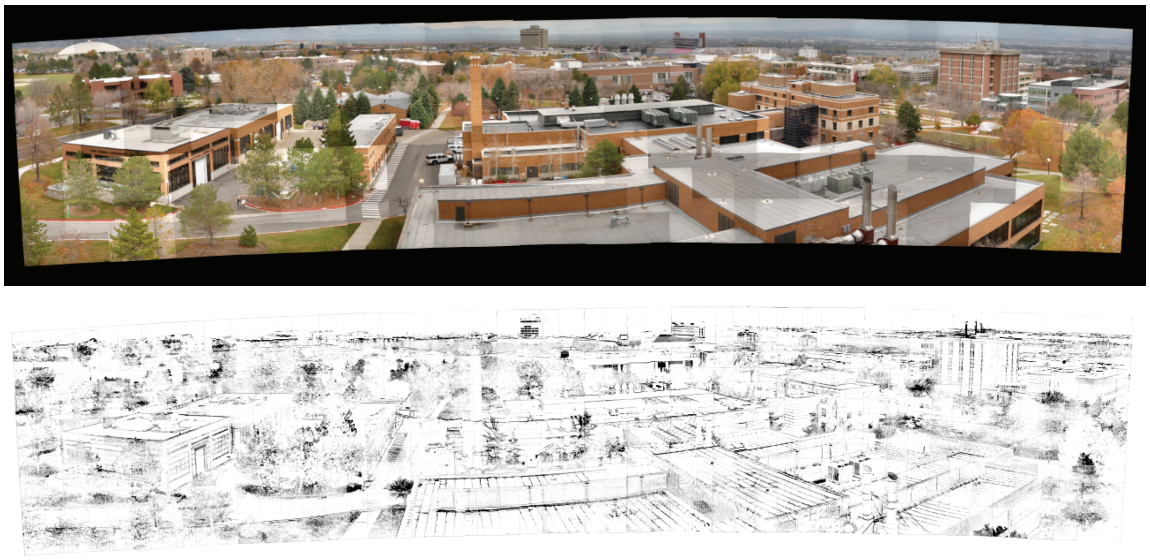
The next step in this pipeline consists of a blending operation that returns the difference between the original image and the blurred version. This module is also a TOM with implementations both on the CPU and the GPU. The combined image is streamed to a threshold operator that computes the image accumulated histogram and discards pixels whose accumulated frequency fall outside a given range (we set this range to be between 95% and 100% of the total accumulated histogram value). Finally, the pipeline sends images into an inversion TOM. The main form of parallelism in this pipeline is the streaming data-parallelism.

Figure 7.10a and Listing 7.3 illustrates how simple it is to integrate an existing code into HyperFlow. Listing 7.3 shows the function used for image inversion, as well as a small class that wraps around this function to provide a specific implementation to the image inversion TOM. Figure 7.10b and Figure 7.10c reports the execution times for different system configurations.

7.3.2 Multigrid for Gradient-Domain Operations

Gradient-based methods, though computationally expensive, have become a fundamental piece of any advanced image editing application. Given a guiding gradient field constructed from one or multiple source images, these methods attempt to find a smooth image that is closest in a least squares sense to a guiding gradient. This typically requires the solution to a 2D Poisson problem and often multigrid methods are used. As a first step in a multigrid system, the mesh is coarsened. The Poisson equation can be solved in a coarse-to-fine manner. One full iteration, from fine to coarse and back, is typically called a V-cycle. Kazhdan and Hoppe [49] have implemented this V-cycle in a streaming fashion for large panoramas. The source code for this system is available online and very well optimized for a single core machine without any trivial parallelism due to strict data dependencies. Relaxation steps, which comprise almost half of the total runtime, must be performed in a specific order in both restriction phase and prolongation phase.

We demonstrate a similar pipeline based on HyperFlow with a simple modification in the multigrid solver that allows relaxation steps to be updated in parallel automatically at runtime. Figure 7.11 shows the modified multigrid solver in our system. Instead of performing temporally blocked relaxation in a single streaming operation (as shown on the right), the Gauss-Seidel iterations are separated and chained together (as shown on the left). Hence, pipeline parallelism can be


(a) Salt Lake City Panorama

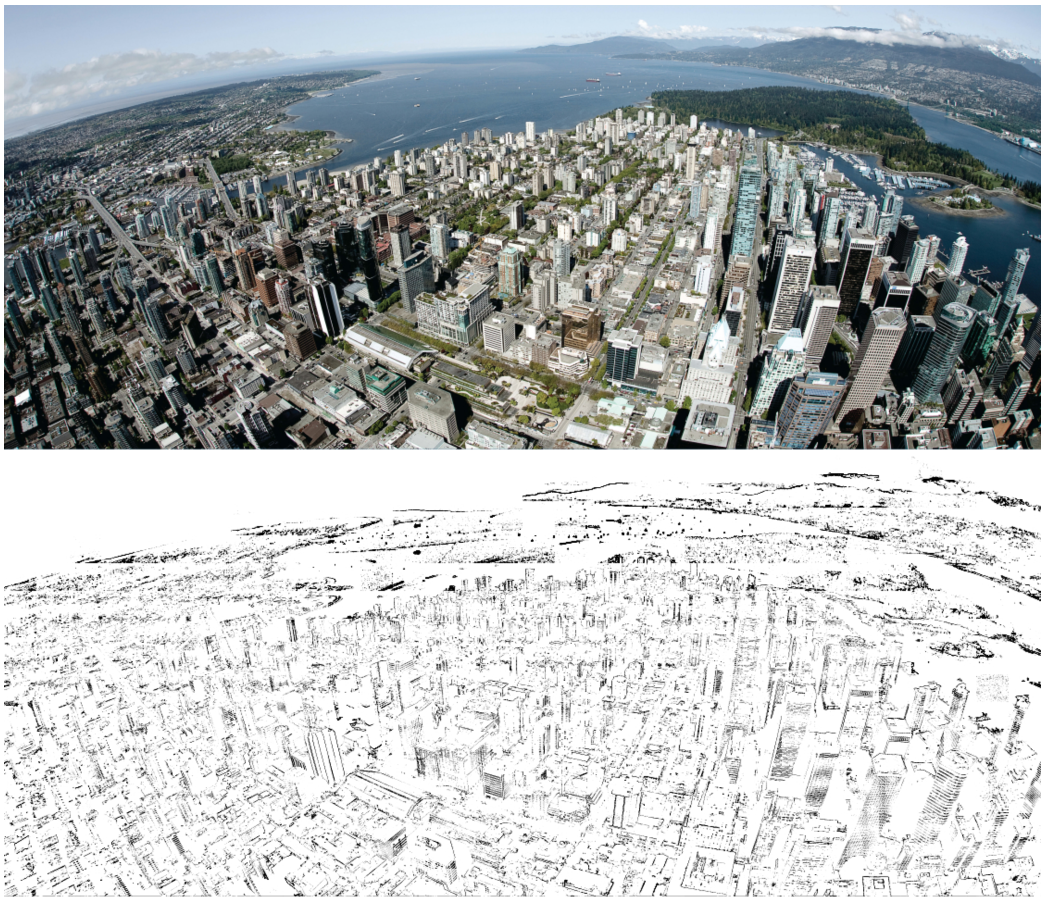
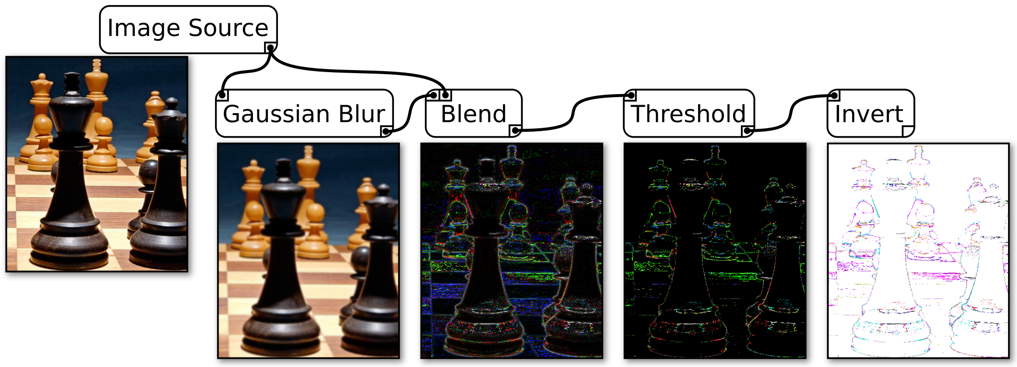
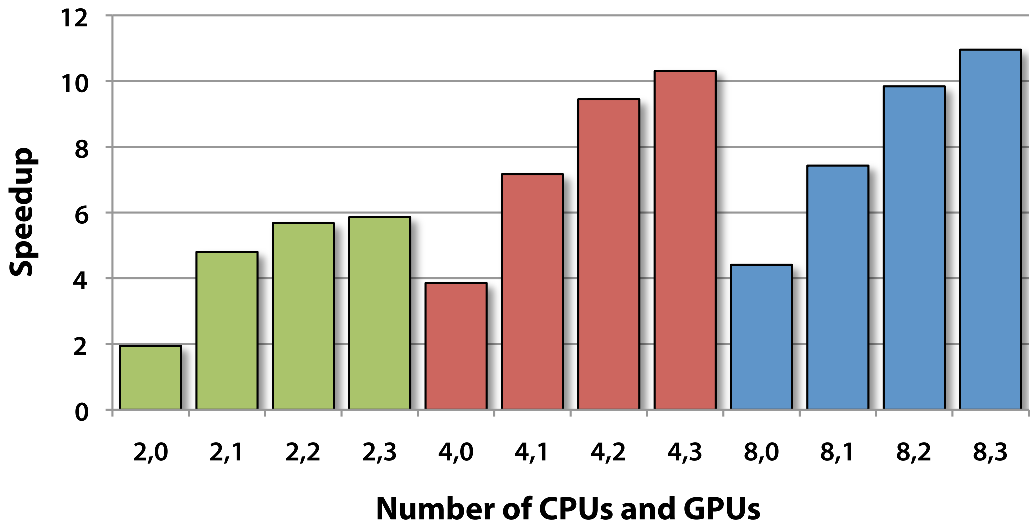

(b) Vancouver Overlook

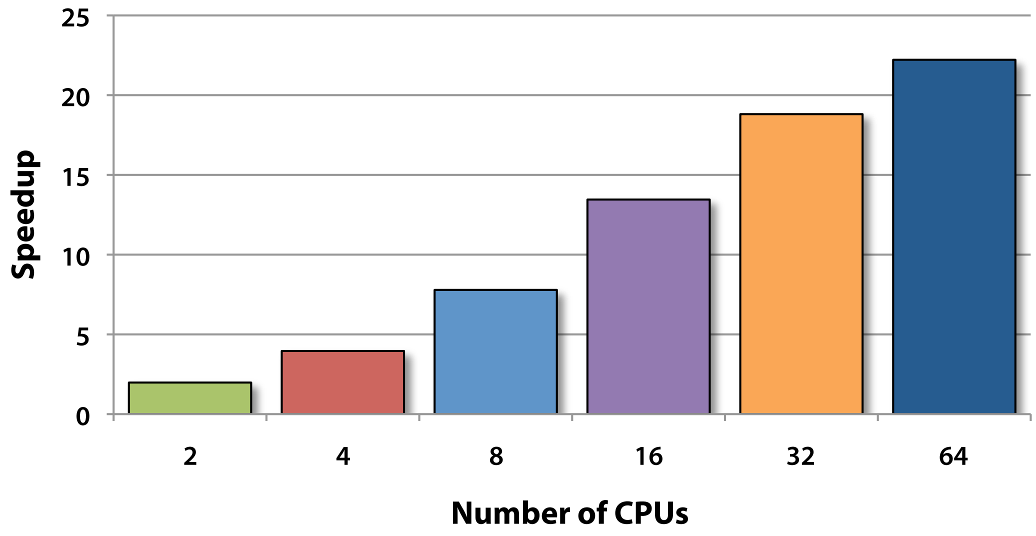
Figure 7.9: **Edge Detection Result** Top: input composed of 624 images, each approximately 6MP for (a) and 3MP for (b) in resolution. Bottom: composition of input images after edge detection pipeline.



(a) Image-based edge detection pipeline



(b) System A results



(c) System B results

Figure 7.10: **Edge Detection Pipeline** Edge detection pipeline on approximately 4GB of image data. Speedup compared to a system with 1 CPU and 0 GPUs.

```
void invert_cpu(const Image &in, Image &out)
{
  int width = in.width();
  int height = in.height();
  out = Image(in);
  for (int i = 0; i < width; ++i)
     for (int j = 0; j < height; ++j)
        out.data(i,j) = ~in.data(i,j);
}
// ...
class ICPUImageInvert : public TaskImplementation {
public:
  // ...
  virtual int process(ExecutionContext *ctx,
                      TaskData *input, TaskData *output)
  {
    Flow       *inFlow = input->flows[0];
    DImageData *inData = inFlow->data;
    DImageData *outData = new DImageData();
    invert_cpu(inData->asImage(), outData->asImage());

    Flow *flow=ctx->createFlowForOutputPort(0, inFlow);
    flow->data=outData;
    output->addFlow(flow);
    return 0;
  }
  // ...
};
```

Listing 7.3: **Wrapping functions into HyperFlow** Image inversion function and class wrapper to transform the function into an implementation that will be added to a TOM.

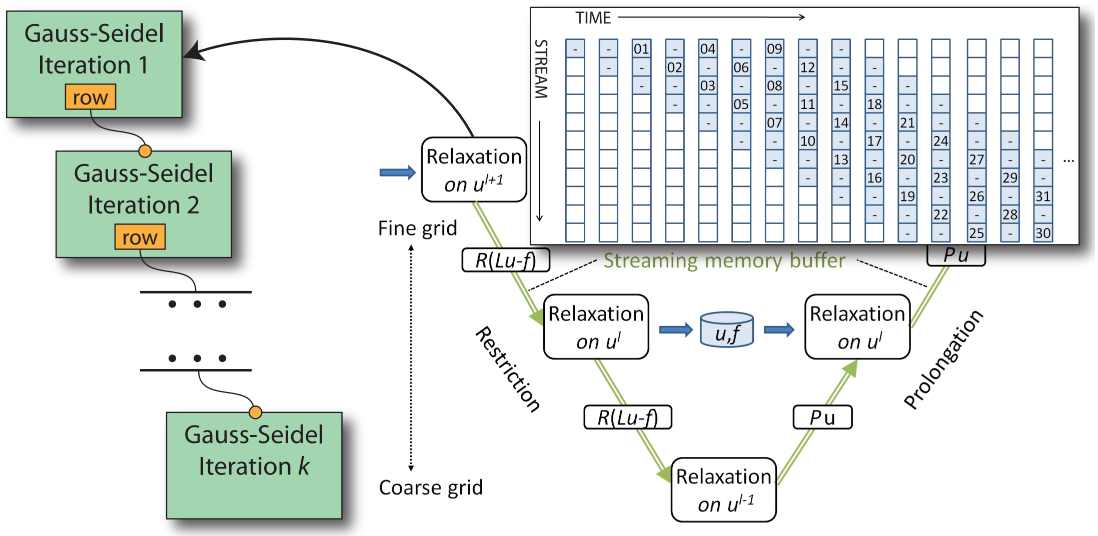


Figure 7.11: **The Modified Multigrid Solver**

triggered automatically by HyperFlow as rows of images are streamed through this solver. However, since each iteration of the update also has its own streaming buffer, rows needs to be copied among iterations. This results in a higher memory footprint and bandwidth. Thus, there is a trade-off between concurrency and memory usage in this pipeline.

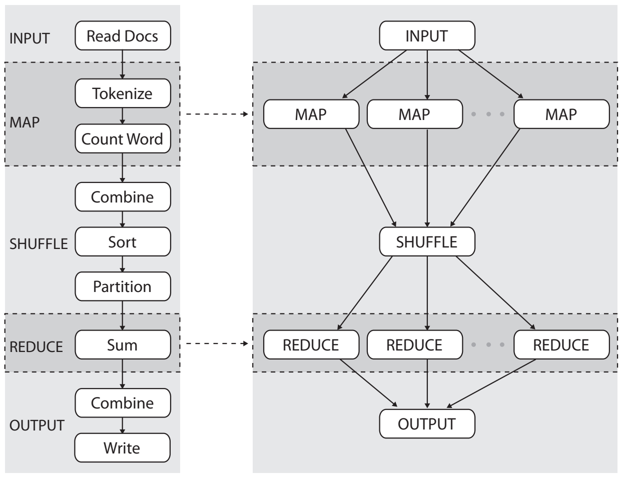
Table 7.1 shows performance timings for the modified pipeline in HyperFlow. Though the original implementation was very well optimized, it is shown that the performance can still be improved by a simple transformation into HyperFlow. However, the amount of parallelism here depends directly on the number of iterations involved. Due to the small number of iterations needed by the application, *i.e.,* 5, and the additional data copies between iterations, HyperFlow was only able to achieve the maximum of 1.4x speedup.

7.3.3 Map/Reduce Word Count

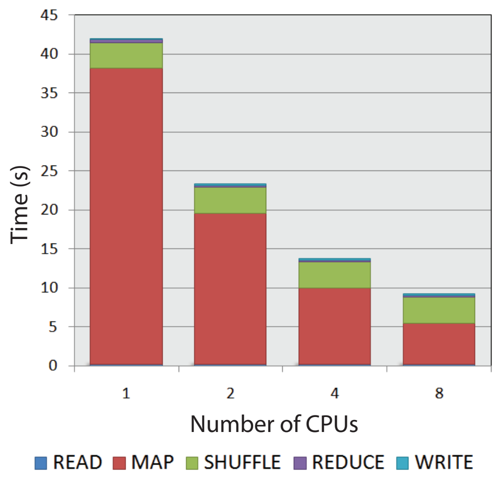
We use the word count of MapReduce example to demonstrate the flexibility of HyperFlow. Though simple, this example requires a careful design of the dataflow architecture to run efficiently. As shown in Figure 7.12a, the pipeline starts by reading in a document from disk, which is then

Table 7.1: Performance Results for Streaming Multigrid Gradient-Domain Image Editing

Image	Original	HyperFlow	Speedup
PNC3	39.78s	31.68s	1.26x
Edinburgh	73.73s	55.06s	1.34x
Red Rock	126.41s	90.06s	1.40x



(a) MR pipeline



(b) Performance

Figure 7.12: **Map-Reduce Example in HyperFlow** (a) HyperFlow pipeline definition (left) and how it should be executed with high efficiency, *i.e.,* modules are desired to run in parallel while the rest have to be sequential since it uses global shared resources. (b) Performance Results. Mapping is the most time consuming phase since it has to perform both word parser and accumulate word counts for each text block. Fortunately, it can be parallelized with very little effort in HyperFlow.

broken into as many chunks as the number of available mappers. Mappers execute independently and ideally in parallel. Each of them consists of a word tokenizer that breaks its input into words, counts them and outputs a list of tuples with the format (word, count). In the shuffling phase, all mapper outputs are combined (through concatenation) into a single list of (word, count). Following this step, we sort the list by words, and subsequently group sort lists and partition then to all reducers. Each reducer sums up all word counts and sends the final result to the writer, which writes them sequentially to the output file.

Two execution scenarios need to be addressed: (1) an upstream module, which executes sequentially, but generates output for concurrent execution downstream and (2) a downstream module which can only be executed sequentially and has to collect flows from upstream modules executed in parallel. Scenario (1) occurs when data are moved from the reader to mappers, as well as from the shuffling phase to reducers. This is necessary to enable data parallelism in the Map and Reduce phases. Scenario (2) happens when the execution comes back from the Map and Reduce to the Shuffling and Writer.

Fortunately, HyperFlow supports both scenarios with great simplicity. For scenario (1), in order to create a data parallelism from a module to another where there is only one port connection, the upstream can generate multiple flows with unique identifications. When the scheduler processes these flows, it separates them as distinct executions and triggers task implementations separately for each one of them. For scenario (2), it limits the task implementations of a module to be run in sequential (which can be done through a lock of the TOM), accumulate all data and only execute if all the above flows have arrived. Listing 7.4 shows the implementations of both scenarios through the Reader and Combine modules.

The results for executing the Map-Reduce pipeline in HyperFlow are reported in Table 7.2 and Figure 7.12. All module implementations are only done in CPU but one can extend Map and Reduce functions to use CUDA devices as needed. Input to the algorithm is a collection of 500 e-books consisting of over 83 millions words that are publicly available through the Gutenberg Project [65].

```
class IReader: public TaskImplementation {
  virtual int process(ExecutionContext *ctx,
                      TaskData *input, TaskData *output) {
    Data *data[N];
    ... // Read input file and place them into data[]
    for (int i=0; i<N; i++) {
      // We are creating flows with unique ids
      Flow *flow = ctx->createFlowForOutputPort(0);
      flow->data = data[i];
      output->addFlow(flow);
    }
    return 0;
  }
};
class MCombine: public TaskOrientedModule {
public:
  int         numberOfMappers;
  SafeCounter doneCount;
  Mutex       mutex;
  ListOfData  data;
};
class ICombine: public TaskImplementation {
  virtual int process(ExecutionContext *ctx,
                      TaskData *input, TaskData *output) {
    MCombiner *M = (MCombiner*)ctx->module;
    // Locking resource to accumulate
    M->mutex.lock();
    M->data.push_back(input->flows[0]->data);
    M->mutex.unlock();
    if (M->doneCounter->increase() < M->numberOfMappers)
      return -1;
    ... // Process data in module->data
    return 0;
  }
};
```

Listing 7.4: **Mixing Sequential and Parallel Execution** Reader module enable data-parallel execution to downstream modules, *e.g.,* Mappers, while Combine module collect upstream parallel execution to begin a sequential process.

Table 7.2: Timing Results for the Word Count Pipeline 83-Million Words

#CPU Cores	Read	Map	Shuffle	Reduce	Write
1	0.23	37.95	3.35	0.36	0.16
2	0.23	19.33	3.39	0.23	0.16
4	0.23	9.82	3.33	0.22	0.17
8	0.23	5.27	3.36	0.22	0.17

7.3.4 Broad-Phase Collision Detection

To demonstrate how HyperFlow can be easily integrated into existing software we modified the Open Dynamics Engine (ODE), an open source physics engine, to parallelize the broad-phase collision detection phase.

Collision detection is often performed in two steps: broad-phase and then narrow-phase. The broad-phase step corresponds to querying a spatial data structure (called a “space” in ODE) for pairs of objects (represented only by their bounding geometries) that are potentially intersecting. Only those pairs go through narrow-phase collision tests, which compute the exact intersections that are later used for contact constraints.

Unless some application-specific optimizations are employed, the broad-phase collision detection step tends to take up a significant part of the physics simulation time. We created a simple parallel implementation using a regular grid subdivision. Each grid cell contains one space to store objects completely inside it, and a “border space” to contain objects that are partially inside the cell. As the objects move around they are removed from one space and inserted into another. An object can only exist in one space at time, a condition easily satisfiable when completely inside a grid cell. Objects on the cell’s border are present in multiple border spaces, so we choose only one of them to hold the object. We base this choice on an arbitrary total ordering of the border spaces (such as their position in memory.)

The collision detection algorithm is as follows: for each grid cell, in parallel, we check for (a) collisions completely inside the cell, (b) collisions completely inside the cell’s border, and (c) collisions between each object in the border and cells (and their borders) that overlap with the given object. Each of those collision tests are independent, but we parallelized only the top-level for to keep the granularity of the task coarse, minimizing the synchronization required to merge all the collision results.

This algorithm was implemented using both HyperFlow and OpenMP; the later offers less guarantees than HyperFlow, thus having smaller coordination and synchronization overhead, serving as an optimal baseline for comparison. We tested both implementations by randomly placing 10^5 objects on a $20 \times 20 \times 2$ grid (Figure 7.13). The comparison speedup between HyperFlow and

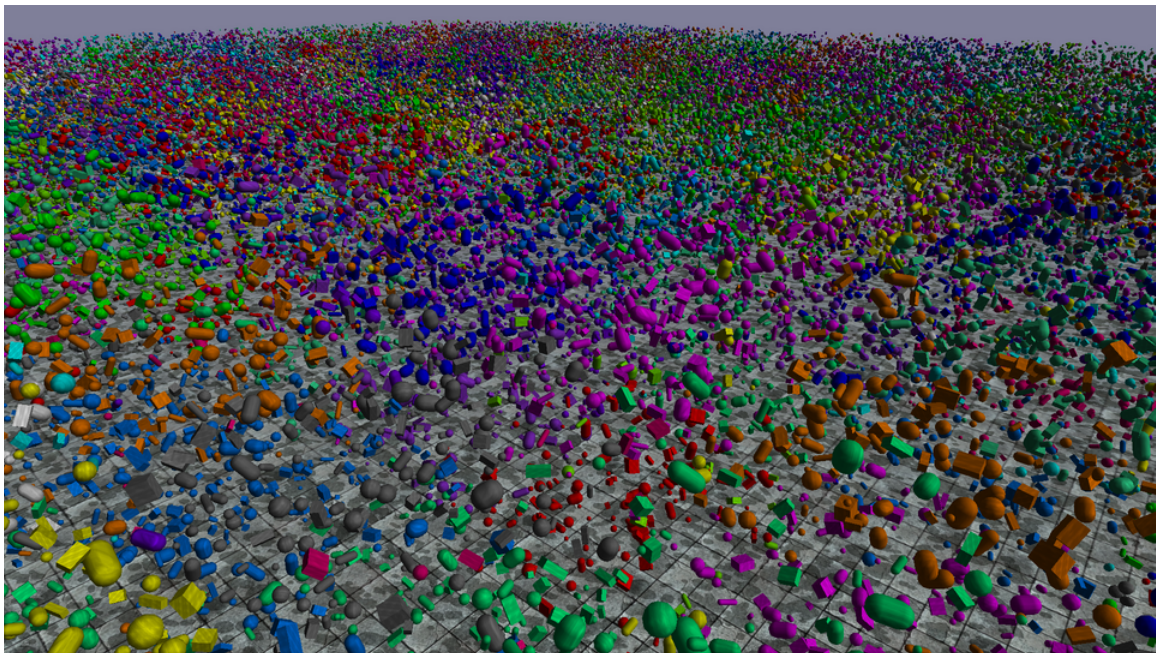


Figure 7.13: **Collision Detection Scene** 10^5 objects are contained in a $20 \times 20 \times 2$ regular grid of spaces. Each object is colored differently according to the space where it is inserted.

OpenMP is given in Figure 7.14, which shows that the performance becomes similar as the number of CPUs increases. Figure 7.15 shows a section of the Gantt charts of both implementations, zoomed into two time steps of the simulation. The overhead incurred in the HyperFlow scheduling is visible, but the performance still similar to OpenMP.

7.3.5 Isocontouring Structured Grids

We conclude our evaluation by considering a less load balanced problem: data-parallel isosurface computation. As input we use the RMI data set from LLNL [44]: a $2,048 \times 2,048 \times 1,920$ regular grid of single-byte values totaling 8 GB of data. This data set was partitioned into blocks of size $128 \times 128 \times 64$ voxels, with each block mapped to a separate flow. Because some blocks contain no isosurface while others may be quite dense, a static assignment of blocks to processors can introduce considerable load imbalance, something that HyperFlow avoids.

The contouring computation cannot process each block independently, because contours must be extracted also *between* voxels residing in adjacent blocks. Figure 7.16 illustrates (using a 2D analogy) the data communication needed between adjacent blocks, here shown in blue. The red and green thin blocks correspond to shared block “edges”; yellow blocks are shared corners. Note in particular that this duplication of block boundary voxels adds a considerable number of thin layers as additional flows processed by HyperFlow. These flows and data dependencies are further

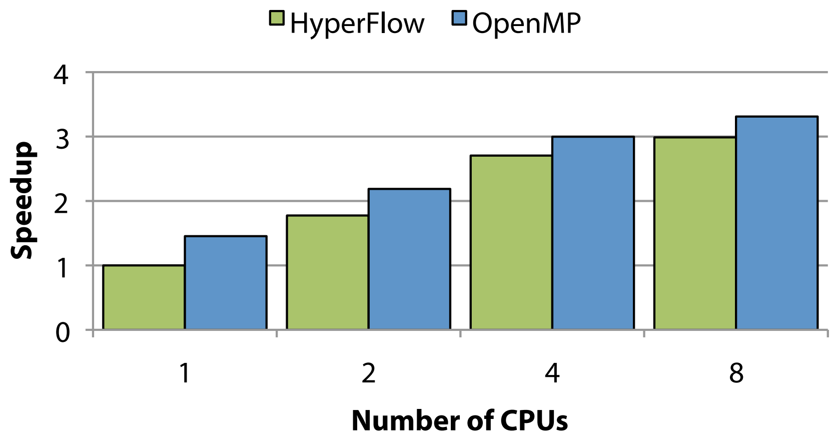
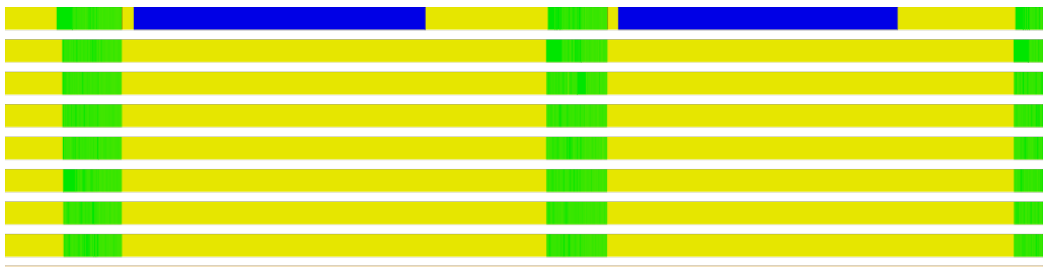
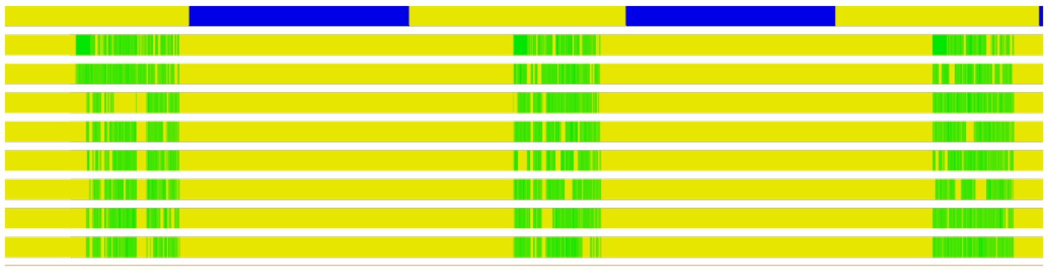


Figure 7.14: **OpenMP vs. HyperFlow** Comparison for collision detection.



(a) parallel collision detection in OpenMP with 8 threads.



(b) parallel collision detection in HyperFlow with 8 VPEs

Figure 7.15: **Gantt Charts for the Parallel Broad-phase Collision Detection** The green states correspond to the execution of the broad-phase algorithm, blue is the constraint solving and integration phase, and yellow represents idle or unrelated computations.

illustrated by the pipeline in Figure 7.17.

We compare the performance of HyperFlow with Isenburg et al.'s [44] implementation of their parallel streaming isocontouring algorithm—here referred to as "Ghost"—which pads each block with a surrounding layer of *ghost zones* (voxels from adjacent blocks) corresponding to the red, green, and yellow blocks in Figure 7.16. The Ghost algorithm was designed for distributed computation, and therefore uses MPI for (in-memory) interprocess communication. Table 7.3 and

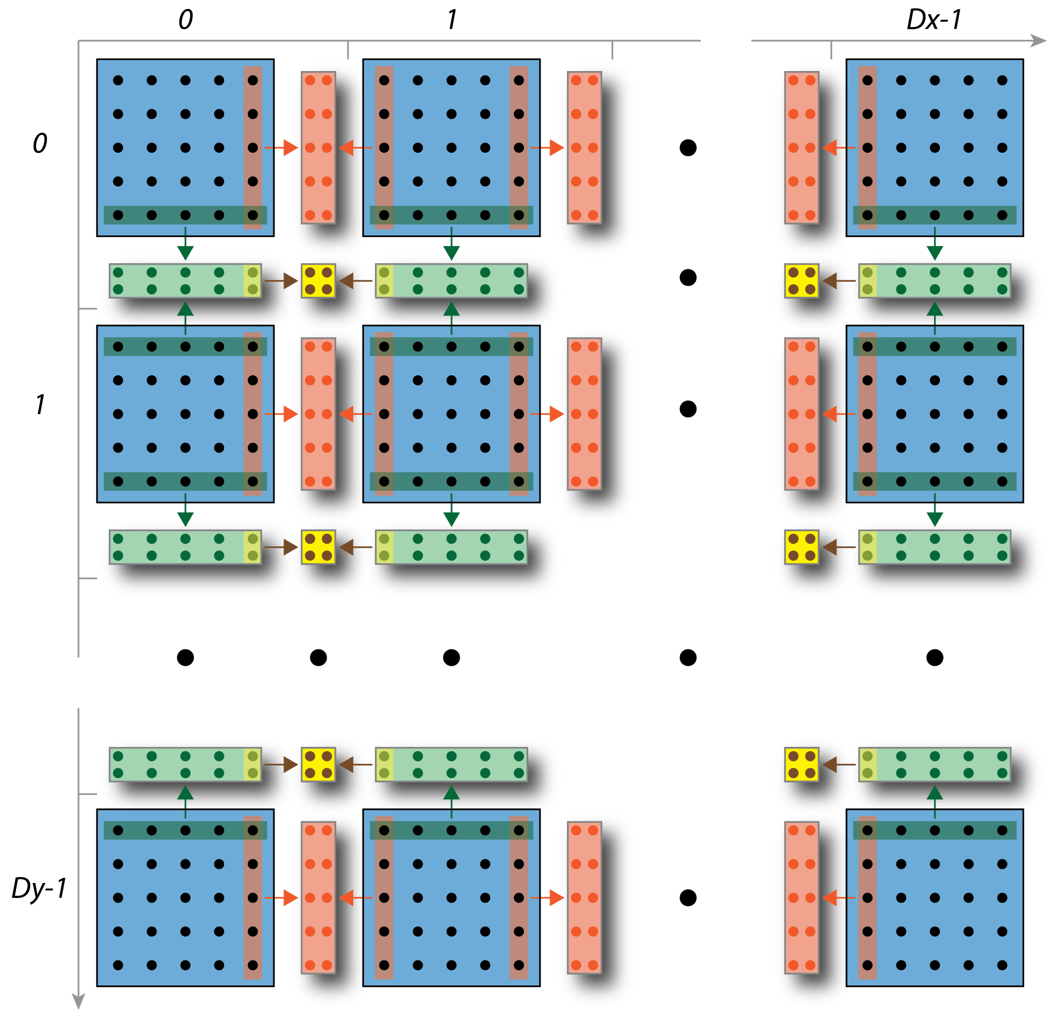


Figure 7.16: **Data Communication Diagram in HyperFlow** for the Isocontouring Pipeline

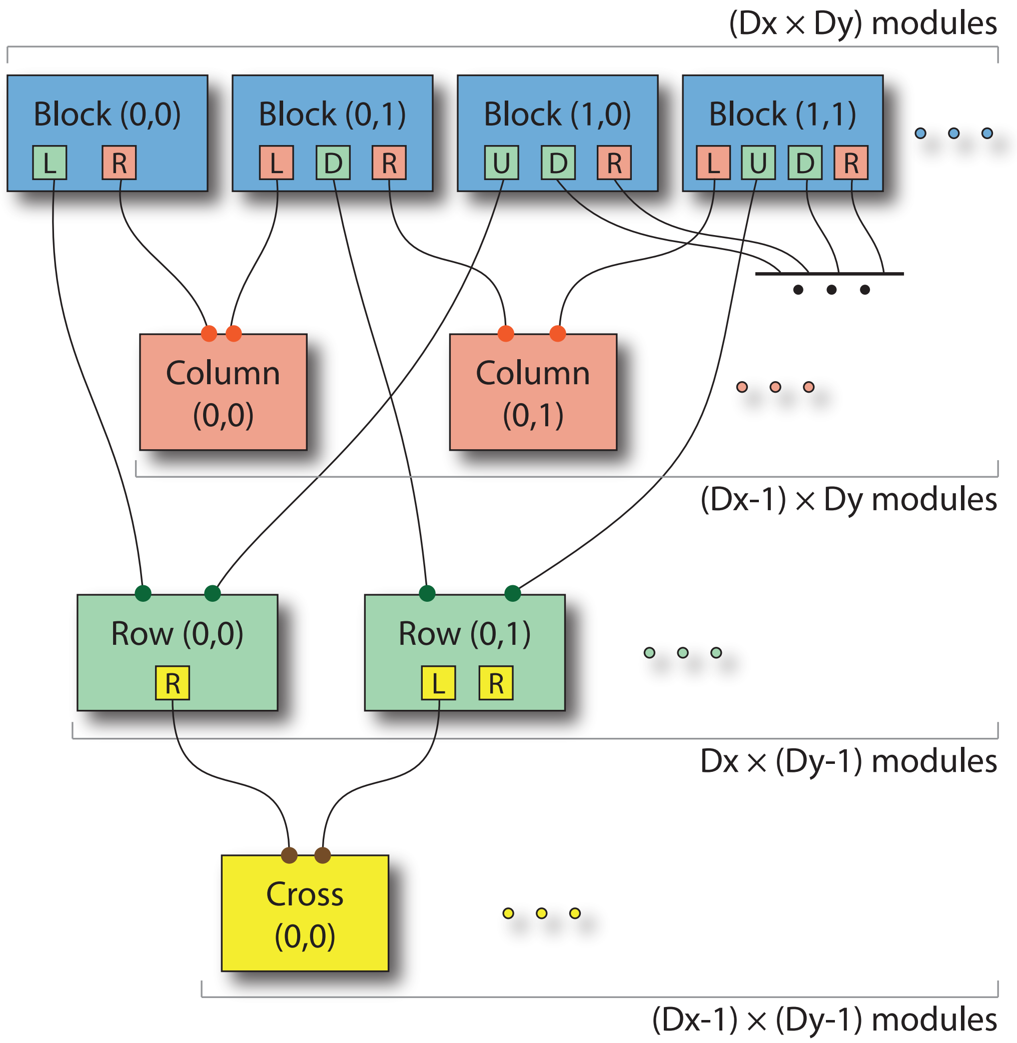


Figure 7.17: **Pipeline Structure in HyperFlow** for the Isocontouring Pipeline

Table 7.3: Large Iso-Surface Extraction Using Ghost [44] and HyperFlow

# Procs	GHOST		HyperFlow
	Process Time	Total Time	Total Time
1	177.88s	179.51s	157.71s
2	90.66s	92.24s	78.26s
4	45.61s	47.19s	39.23s
8	23.12s	24.77s	19.88s
16	11.64s	13.27s	10.06s
32	5.89s	7.50s	4.93s
64	3.07s	**4.85s**	**2.79s**
128	**1.70s**	5.83s	3.23s
256	3.37s	8.12s	4.99s
22			7.63s
44			3.73s
66			2.76s
88			**2.45s**
132			3.23s
154			3.45s
48	4.46s	6.23s	
96	2.41s	5.67s	
160	1.90s	6.29s	

Figure 7.18 summarize the execution time for both methods. As is evident, both methods achieve near-linear speedup up to 64 processors, with HyperFlow consistently outperforming Ghost. We conjecture that this result is due in part to better load balancing in HyperFlow, which contrary to Ghost performs a dynamic rather than static allocation of tasks, as well as MPI overhead in Ghost.

7.3.6 Saturation of Memory Bandwidth

As Table 7.3 shows that the performance of the implementations using Ghost [44] and HyperFlow did not scale for more than 64 processors (or 88 in HyperFlow), we believe that it was caused by the saturation of memory bandwidth of the system. In order to verify our explanation, we have performed three additional experiments.

First, we take computation out of the implementation in HyperFlow, *e.g.,* no triangles were generated, only traversing through the dataset. This resulted in a similar speedup curve like the full implementation with HyperFlow. Thus, we can eliminate load balancing as a cause of the unscalability here.

Second, we take memory accesses out of the implementation in HyperFlow, *e.g.,* no traversal of the grid, though there were still noop computations. This is shown as *No Mem HF* in Figure 7.18.

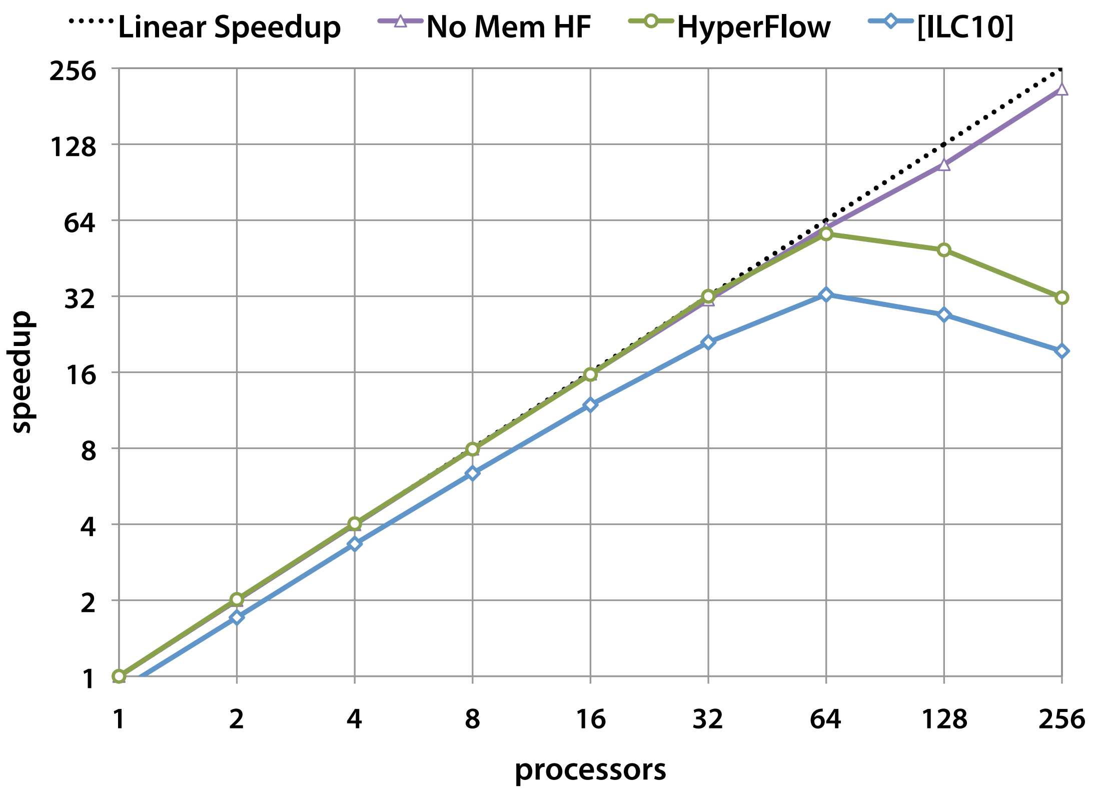


Figure 7.18: **HyperFlow vs. Ghost [44]** for parallel computation of an isosurface. The speedup of both methods is measured relative to the wall clock time of HyperFlow running on a single processor. *No Mem HF* refers to the same HyperFlow implementation without memory accesses.

As expected, the implementation was able to achieve almost a linear speedup up to 256 cores. This indicates that memory accesses were the bottleneck in this application.

Finally, we measure the memory bandwidth of our application to see if it matches the system maximum bandwidth. The sustained bandwidth of our system was measured using Stream [63] and reaching about 500GB/s at its peak. For our application, the memory bandwidth was measured by counting the number of memory transactions (64-bit each) during its execution and dividing the total throughput by the execution time. The measurement was done through PAPI [67]. As expected, the result shows that our implementation in HyperFlow reaches almost the system peak bandwidth using 64 to 90 cores.

We conclude that, in this application, HyperFlow has achieved a linear speedup up, only being limited by the saturation of the memory bandwidth due to the memory access pattern of the application.

7.4 Discussion

Pipeline developers often spend a considerable amount of time tweaking workflows to make sure they optimally utilize available resources, which requires manually writing modules to run either on the CPU or the GPU. This approach has a serious portability drawback, since the same pipeline, when executed on a different system, might not run with the same efficiency, or even not run at all (if the target machine lacks the necessary computational resources). We addressed this problem with the concept of a TOM, in which tasks may have more than one implementation. Each implementation specifies its required computational resources, which allowed EE to distribute the pipeline computation to maximize efficiency and minimize resource idle time.

The HyperFlow architecture is based on a set of abstractions designed to enforce a clean separation between module *specification* and *implementation*. This allows developers to construct efficient, high-level pipelines without prior knowledge of the computing resources available at execution time. For example, in HyperFlow the exact same pipeline can be executed on machines with different numbers of CPU cores and GPU devices, with the VPE scheduler automatically distributing computation to minimize resource idle time. This abstraction is straightforward in HyperFlow by building/compiling individual tasks into Callable Objects, such as CPU function pointers and GPU kernels. The VPE scheduler places these into appropriate execution contexts, such as CPU masks or individual GPU devices. It is important to observe that to support Pipeline Parallelism, Callable Objects need to be reentrant. In our current implementation, Callable Objects are simply compiled as function pointers. To allow HyperFlow to run across a distributed system, these Objects need to be compiled into stand-alone executable files that can be sent over the network, in such a way that execution can be performed on each cluster node.

Although the examples discussed in this chapter include VPEs defined only for CPU threads and GPU devices, HyperFlow is designed to allow arbitrary VPEs, with no inherent limitation on the kind of computing resource that can be integrated into the architecture. Therefore, it is possible to define VPEs that exploit many other computational resources, such as Web Services or Cloud Computing servers, among others.

CHAPTER 8

FUTURE WORKS

In this chapter, we discuss potential extensions to HyperFlow to support distributed environment as future work. We also would like to experiment our system with higher computation complexity.

8.1 Remote Computing Resources and VPE

The current implementation of HyperFlow proposed in Chapter 7 only supports two types of computing resources: local threads and GPU devices. In order to run distributedly, new resources have to be defined to map to remote processing elements. A possible solution is to take current thread and GPU device resources and make them pointing to remote threads and remote GPU devices. This will create a uniformity in resource specification. Both local and remote processing elements would be specified with the same interface, if from a pipeline developer's perspective. However, customized jobs may require a task to run on a specific node and such configuration would not be specified in this manner. Moreover, allowing local nodes to specify remote threads and GPU devices will also complicate the scheduler's logic. A more sophisticated solution would be adding a new VPE, a "node" VPE, to HyperFlow that represents remote machines. An advantage of this approach is that each remote VPE is allowed to have its own scheduling strategy. In fact, this is necessary for HyperFlow to efficiently utilize computing power on heterogeneous platforms. For example, in a cluster, a node with Non-Uniform Memory Access (NUMA) may want to schedule executions differently than a node without NUMA to gain the optimal performance. Another advantage of having a "node" VPE is that the centralized scheduler does not have to deal with remote resources, instead, it only needs to handshake to remote schedulers, thus simplifying much of the scheduler's logic.

Figure 8.1 shows a potential implementation of the *Node VPE* that can handle remote computing resources as mentioned above. At runtime, when being launched from a client, this VPE will start an instance of HyperFlow on a remote node. The remote HyperFlow will instantiate a set of VPEs mapping to remote computing resources, and a slave node VPE, that will listen and receive tasks from the local node VPE. Communications across machines can be very costly. Thus, it is very undesirable for remote schedulers to report to the local scheduler each time a flow is generated for

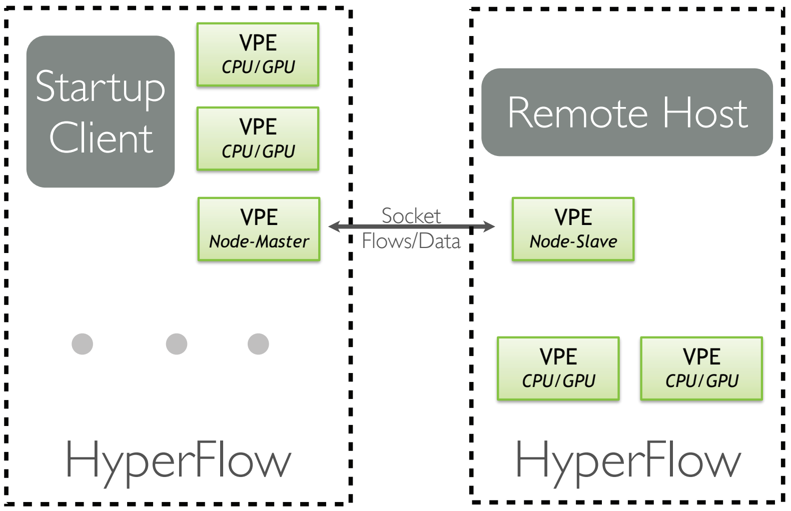


Figure 8.1: **Node VPE** Computing resources on remote machines are utilized with tasks assigned from Node VPE of local nodes.

a new scheduling. Therefore, a feasible scheduling strategy is that a node only sends its flow back to the centralized scheduler when it cannot process a flow with its available computing resources. However, this approach may create a problem of poor load-balancing across node. Thus, a reload-balancing periodically may need to be implemented.

8.2 Distributed Scheduling

Since each node schedules execution independently from the others and a module may receive data from different machines asynchronously, each module has to check whether all of its inputs have arrived (from other machines) before execution. This can be done distributedly by extending the flow cache of HyperFlow to check for machine ids in addition to the flow id. Unfortunately, this flow cache has to live globally in the centralized scheduler though data transfers across machines only need to be performed to the processing node once when all inputs are available. Figure 8.2 illustrates how the flow cache would help with a greedy scheduling strategy: (a) is a pipeline in the middle of its execution with two flows **f1** and **f2** generated from the **Fetch** module of a local node, assuming that **f1** and **f2** that have been scheduled to flowed into **Filter A** and **Filter B** on two different machines, A and B, respectively. In (b), **Filter A** is done first and its result flows into the **Viewer** module. However, since not all of its inputs are ready, the flow get saved to the flow cache. It should be noted that the data still live on machine A at this point. In (c), both flows are done, and the flow cache contains both results from **Filter A** and **FilterB**. (d) indicates that one of the machine, *e.g.,* B, are scheduled to run the **Viewer** module, data from machine A will now be transferred to B for execution.

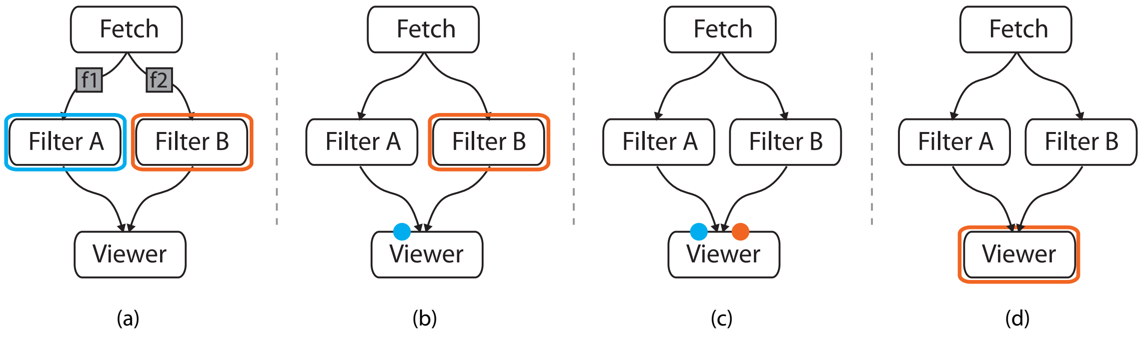


Figure 8.2: **Distributed Scheduling Without Persistent Module**

8.3 Persistent Module

Split and merge of data are very common in large scale visualization pipelines, such as the MapReduce framework described in Chapter 5. Though data splitting can be achieved through the automatic construct of data-parallel in HyperFlow, there is no similar construct for merging data. For shared memory architecture, a task implementation may use mutex and lock to simulate this effect. However, this does not apply to distributed environment. In order to address this, we propose to add a notion of *Persistent Module* to HyperFlow. For a persistent TOM, only a single instance of its task implementation will be executed across all available VPEs in HyperFlow, including remote ones. Figure 8.3 illustrates how persistent modules could be used to construct split and merge in a MapReduce pipeline. Splitting is achieved with automatic data parallelism in Figure 8.3a, where output flows from **Read** will be processed in parallel based on the available resource. The persistent module **Shuffle** would catch all map's output and run them sequentially on a single VPE context. Figure 8.3b shows an explicit parallelism where all data splitting and merging are specified directly on the pipeline. The main difference between the two pipelines is that data merging with persistent modules are triggered many times sequentially while the same thing is achieved as a single execution without persistent module.

Figure 8.4 shows how a persistent module would work in HyperFlow using a greedy distributed scheduling. (a) shows a simple pipeline with a persistent module and two input flows and (b) shows the underneath expansion of the pipeline. Both flows are run through the **Fetch** modules on two different machines. In (c), they now proceed to the two **Filter**. Given that **f1** is done before **f2**, **Merger** will be called with **f1** first as shown in (d). Then, the same instance of **Merger** will be called again with **f2** only when **f1** has been processed.

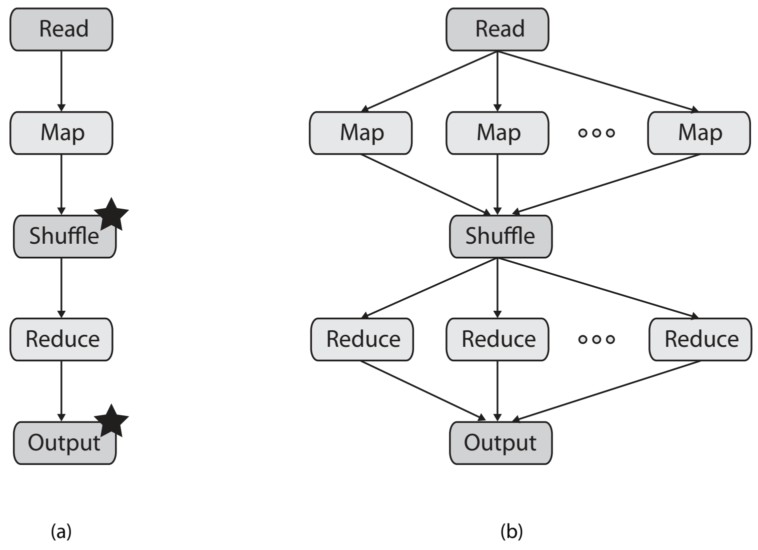


Figure 8.3: **Implicit and Explicit Data Parallelism** Two types of data-parallelism that can be supported in HyperFlow: (a) implicit and (b) explicit.

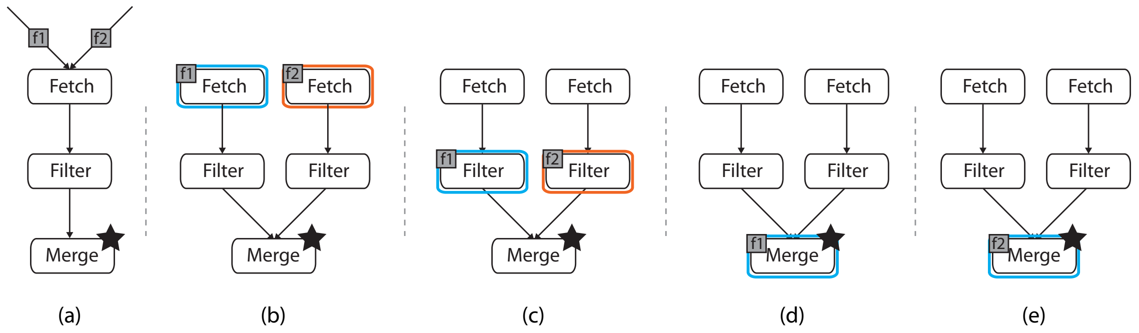


Figure 8.4: **Distributed Scheduling with Persistent Module** The stars specify persistent modules.

CHAPTER 9

CONCLUSIONS

This dissertation has presented a number of experiments in large-scale visualization, through which to determine necessary requirements of a parallel dataflow architecture suited for those applications. It also proposes a design that makes full use of data streaming and efficiently runs on heterogeneous platforms.

In Chapter 3, we have presented a streaming technique for simplifying tetrahedral meshes of arbitrary size. We describe several methods for laying out a tetrahedral mesh on disk in a coherent, I/O-efficient format. We also show an analysis of these layouts and the effects they have on the final simplified mesh. We provide a technique for simplifying small portions of the mesh in memory while obtaining a smooth simplification over the entire mesh. The simplification occurs in one pass, preserves mesh topology and scalar information, requires little memory, and runs quickly. We also provide optimizations to traditional simplification data structures that improve speed and efficiency. We present a linear solver that improves stability and speed of quadric-based simplification. Finally, we provide stability and error analysis of our algorithm with results for specific examples that show the time and memory required for processing.

Chapter 4 has introduced the iRun system, which is capable of volume rendering tens of millions of tetrahedra at interactive rates on a commodity PC. The preprocessing required by our algorithm occurs completely out-of-core and is light, fully automatic, and does not result in a large increase in data size. This enables our system to start up and render the geometry immediately. We have shown how hierarchical level-of-detail, parallel prefetching, and hardware-assisted volume rendering can be combined to maintain interactivity in an environment where occlusion culling is not suitable. Finally, we have shown how out-of-core volume rendering can be applied in a distributed manner to improve mesh quality and increase display size.

In Chapter 5, we take a first step in evaluating the suitability of the MapReduce framework to implement large-scale visualization techniques. Specifically, we implement and evaluate a representative suite of visualization tasks (isosurface extraction, mesh simplification, and polygon rasterization) as MapReduce programs, and report quantitative performance results applying these algorithms to realistic datasets. Our results indicate that the parallel scalability, ease of use, ease of access to computing resources, and fault-tolerance of MapReduce offer a promising foundation for a combined data manipulation and data visualization system deployed in a public cloud or a local commodity cluster.

From the experiences found in Chapter 3, 4, and 5, we propose new techniques for exploiting multicore architectures in the context of visualization dataflow systems in Chapter 6. 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 while also combining 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. Our new parallel execution strategy offers significant benefits over a both multicore, serially-executed visualization pipelines and pipelines that are computed in streaming modes.

Next, in Chapter 7, we have presented HyperFlow, an improved architecture that allows pipelines to take full advantage of modern, heterogeneous computational systems. HyperFlow is comprised of several components carefully designed to create an efficient abstraction layer that allows developers to design pipelines without knowing in which types of processors their modules get executed. We introduced the concept of a Task Oriented Module, which elegantly encapsulates the difference between task specification and implementation. Therefore, pipeline modules can have multiple heterogeneous implementations, individually designed to take advantage of a particular type of processor. HyperFlow automatically instantiates TOMs to minimize processor idle time and maximize execution efficiency. Pipeline execution relies on a token-based mechanism in which Flows are sent to processing units. We also introduced the notion of a Virtual Processing Element, an abstraction that allows processing elements to be offered without explicitly knowing if they will be actually implemented as a CPU thread or a GPU device.

In Chapter 8, we discuss potential extensions to HyperFlow to support cluster systems with heterogeneous configurations. We have introduced the notion of Node VPE and persistent modules to support distributed execution with different degree of parallelism.

As future work, we plan to implement and validate ideas in Chapter 8 to HyperFlow and test with applications that have high computational complexity.

REFERENCES

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

[2] 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.

[3] M. Aldinucci, M. Torquati, and M. Meneghin. Fastflow: Efficient parallel streaming applications on multi-core. *CoRR*, abs/0909.1187, 2009.

[4] J. Allard and B. Raffin. A shader-based parallel rendering framework. In *IEEE Visualization '05*, pages 17–25, 2005.

[5] AMD. Stream sdk. http://developer.amd.com/gpu/atistreamsdk/.

[6] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 Best Papers Issue*, 2010. Accepted for publication, to appear.

[7] Amazon web services - elastic mapreduce. `//aws.amazon.com/elasticmapreduce/`.

[8] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. T. Vo. VisTrails: Enabling interactive, multiple-view visualizations. In *IEEE Visualization '05*, pages 135–142, 2005.

[9] 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, November/December 2007.

[10] F. Bodin and S. Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Sci. Program.*, 17(4):325–336, 2009.

[11] C. P. Botha and F. H. Post. Hybrid scheduling in the devide dataflow visualisation environment. In *SimVis*, pages 309–322, 2008.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[13] S. P. Callahan, L. Bavoil, V. Pascucci, and C. T. Silva. Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1307–1314, 2006.

[14] S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva. Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05*, pages 199–206, 2005.

[15] S. P. Callahan, M. Ikits, J. L.D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[16] E. Camahort, 1999. The Contour Library http://www.ticam.utexas.edu/ccv/software/libcontour/.

[17] CAPS. Hmpp workbench. http://www.caps-entreprise.com/hmpp.html.

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

[19] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. M., B. J Whitlock, and N. Max. A contract-based system for large data visualization. In *IEEE Visualization '05*, pages 190–198, 2005.

[20] P. Chopra and J. Meyer. TetFusion: An algorithm for rapid tetrahedral mesh simplification. In *IEEE Visualization '02*, pages 133–140, 2002.

[21] P. Cignoni, D. Constanza, C. Montani, C. Rocchini, and R. Scopigno. Simplification of tetrahedral meshes with accurate error evaluation. In *IEEE Visualization '00*, pages 85–92, 2000.

[22] P. Cignoni, C. Montani, C. Rocchini, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[23] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[24] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[25] Nsf cluster exploratory (nsf08560). `//www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm`.

[26] D. Cohen-Or, Y. Chrysanthou, C. T. Silva, and F. Durand. A survey of visibility for walk-through applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[27] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *IEEE Symp. on Parallel and Large-Data Visualization and Graphics*, pages 1–8, 2003.

[28] W. T. Corrêa, J. T. Klosowski, and Cláudio T. Silva. iWalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[29] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Visualization '97*, pages 235–244, 1997.

[30] B. Cutler, J. Dorsey, and L. McMillan. Simplification and improvement of tetrahedral models for simulation. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 93–102, 2004.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, 2004.

[32] R. Dolbeau, S. Bihan, and F. Bodin. Hmpp: A hybrid multi-core parallel programming environment. *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[33] J. El-Sana, N. Sokolovsky, and C. T. Silva. Integrating occlusion culling with view-dependent rendering. In *IEEE Visualization '01*, pages 371–378, October 2001.

[34] R. Farias. Out-of-core rendering of large, unstructured grids. *IEEE Comput. Graph. Appl.*, 21(4):42–50, 2001.

[35] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[36] M. Garland and Y. Zhou. Quadric-based simplification in any dimension. *ACM Transactions on Graphics*, 24(2), April 2005.

[37] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

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

[39] Hadoop. `//hadoop.apache.org/`.

[40] Hive. `//hadoop.apache.org/hive/`. Accessed March 7, 2010.

[41] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH*, pages 693–702, 2002.

[42] Intel. Threading building blocks. http://www.threadingbuildingblocks.org.

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

[44] M. Isenburg, P. Lindstrom, and H. Childs. Parallel and streaming generation of ghost data for structured grids. *Computer Graphics and Applications, IEEE*, 30(3):32–44, 2010.

[45] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large mesh simplification using processing sequences. In *IEEE Visualization '03*, pages 465–472, 2003.

[46] V. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. *High Performance Embedded Architectures and Compilers*, pages 19–33, 2010.

[47] C. Johnson and C. Hansen. *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004.

[48] T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of hermite data. *ACM Transactions on Graphics*, 21(3):339–346, July 2002.

[49] M. Kazhdan and H. Hoppe. Streaming multigrid for gradient-domain operations on large images. *ACM Transactions on Graphics*, 27:21:1–21:10, August 2008.

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

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

[52] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H. Seidel. Feature-sensitive surface extraction from volume data. In *ACM SIGGRAPH '01*, pages 57–66, 2001.

[53] KronosGroup. OpenCL. `http://www.khronos.org/opencl/`.

[54] 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, October 1999.

[55] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):99, 2006.

[56] P. Lindstrom. Out-of-core simplification of large polygonal models. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 259–262, 2000.

[57] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *ACM Symposium on Interactive 3D Graphics*, pages 93–102, 2003.

[58] P. Lindstrom and C. T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization '01*, pages 121–126, 2001.

[59] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[60] K. Ma. In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE*, 29(6):14 –19, nov.-dec. 2009.

[61] K. Ma and S. Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Comput. Graph. Appl.*, 21:72–83, July 2001.

[62] 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.

[63] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[64] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth. Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. *VIS '04: Proceedings of the Conference on Visualization '04*, pages 171–178, 2004.

[65] P. Miceli. The Gutenberg Project. `http://www.gutenberg.org/`.

[66] K. Moreland and D. Thompson. From cluster to wall with VTK. In *IEEE Symp. on Parallel and Large-Data Visualization and Graphics*, pages 25–32, 2003.

[67] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[68] C. Muller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A compute unified system architecture for graphics clusters incorporating data locality. *IEEE Transactions on Visualization and Computer Graphics*, 15:605–617, 2009.

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

[70] NVIDIA. CUDA programming guide. `http://developer.nvidia.com/object/cuda.html`.

[71] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD*, pages 1099–1110, 2008.

[72] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[73] S. G. Parker and C. R. Johnson. SCIRun: A scientific programming environment for computational steering. In *SuperComputing*, page 52, 1995.

[74] V. Pascucci, D. E. Laney, J. R. 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.

[75] 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.

[76] J. Rossignac. 3D compression made simple: Edgebreaker with zip&wrap on a corner-table. In *Proceedings of the International Conference on Shape Modeling & Applications*, page 278, 2001.

[77] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[78] 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`.

[79] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–70, New York, NY, USA, 1992. ACM.

[80] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *San Diego Workshop on Volume Visualization*, 24(5):63–70, 1990.

[81] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization '02*, 2002. Course Notes for Tutorial 4.

[82] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon. A survey of GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing*, 12(2):9–29, 2005.

[83] 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.

[84] Impetus Technologies. Hadoop performance tuning - white paper.

[85] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[86] Craig Upson et al. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, 1989.

[87] 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*, 13:145–155, January 2007.

[88] H. T. Vo, S. P. Callahan, N. Smith, C. T. Silva, W. Martin, D. Owen, and D. Weinstein. iRun: Interactive rendering of large unstructured grids. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, 2007.

[89] H. T. Vo, D. K. Osmari, B. Summa, J. L. D. Comba, V. Pascucci, and C. T. Silva. Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum*, 29(3), 2010.

[90] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *IEEE Visualization '03*, pages 333–340, 2003.

[91] J. Wu and L. Kobbelt. A stream algorithm for the decimation of massive meshes. In *Graphics Interface '03*, pages 185–192, 2003.

[92] Yahoo! expands its m45 cloud computing initiative, adding top universities to supercomputing research cluster. `//research.yahoo.com/news/3374`.

[93] J. Yan, P. Shi, and D. Zhang. Mesh simplification with hierarchical shape analysis and iterative edge contraction. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):142–151, 2004.

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

[95] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Kumar Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating Systems Design and Implementation*, 2008.