DISTRIBUTED INTERACTIVE RAY-TRACING FOR LARGE VOLUME VISUALIZATION

by

David E. DeMarle

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

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2003

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

David E. DeMarle

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Steven G. Parker
	Charles D. Hansen
	Peter S. Shirley

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of David E. DeMarle in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Steven G. Parker
Chair: Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

There is a growing need for tools that aid in the understanding of high-resolution data. Examples include medical data produced by magnetic resonance imaging scanners, and data produced by computational fluid dynamics simulations. Visualization techniques for large data include offline rendering of still images, subsampling to reduce the size of the data, and interactive rendering using supercomputers. Offline rendering makes it difficult to examine three-dimensional structure in the data. In an interactive renderer, the user can move the viewpoint and examine occlusion. Subsampling can make the data small enough for interactive rendering, but it destroys information content. The Scientific Computing Institute's interactive ray-tracer (*-Ray) can solve both of these problems by exploiting the performance of large shared memory supercomputers. Unfortunately these machines are expensive and of limited availability.

A lower cost cluster of PCs, with standard high speed network interfaces, can also interactively render large volumes. To prove this we have created a new ray-tracer, derived from *-Ray, which runs on clusters. The program is an image parallel ray tracer with a supervisor/workers structure. The supervisor and workers are multithreaded applications running on separate nodes. Between nodes, applications communicate over a new network library called Legion. Legion adds a single layer of abstraction to the standard network interface, enabling asynchronous handling of in-bound messages. The network library also includes a software distributed shared memory that makes the aggregate memory space of the cluster usable. With small data, the program replicates the entire scene on each node. For multi-Gigabyte volumetric data, workers obtain data bricks on demand from the distributed shared memory. We explain the implementation of the distributed interactive renderer and analyze its performance.

To Patti

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Ray Tracing is a simple, accurate, and flexible method for creating realistic images. For every pixel, a ray extends from some viewpoint into a scene. Every ray is tested for intersection with every object in the scene. A pixel gets the color of the first object hit by the corresponding ray. Phong lighting makes opaque surfaces look more realistic. Reflective and refractive objects include the color contribution of secondary rays. The secondary ray reflects off or transmits through the object, and takes the color of the next hit object. Shadows require one more ray, which extends from the intersection point toward the light source. If the shadow ray is blocked, the pixel intensity is reduced. Figure 1.1 illustrates Whitted style ray tracing, and Figure 1.2 shows a ray traced image.

The main advantage of ray tracing is that it produces accurate, realistic images. However, Whitted ray tracing produces imperfect images. Aliasing, hard shadows

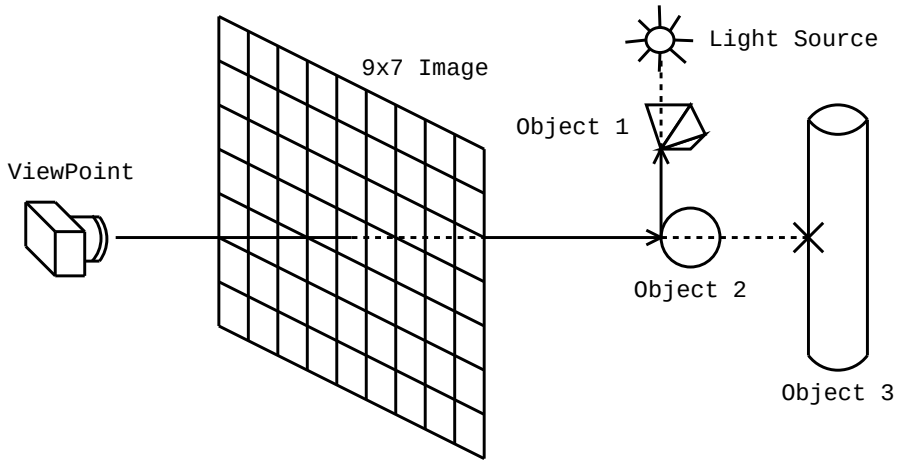

Figure 1.1. Ray Tracing One Pixel of a 9x7 Image of Three Objects.

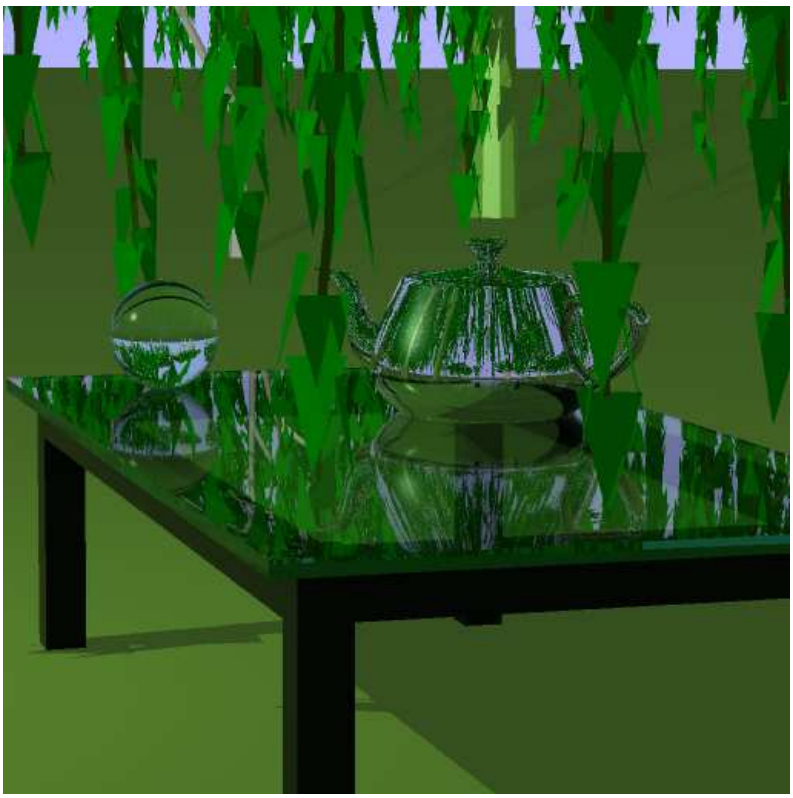

Figure 1.2. A Ray Traced Image.

and too "perfect" looking reflective and refractive objects result from point sampling space with infinitely thin rays. Cook [7] showed that all that was needed to fix these artifacts was to trace several randomly distributed rays for each pixel. Figure 1.3 is a ray traced image that demonstrates Cook's depth of field and motion blur effects. Still, Whitted style ray tracing neglects global illumination. The basic algorithm cannot generate color bleeding between objects, or caustics caused by focused light. This too can be fixed, simply by tracing more rays. Instead of using Phong lighting, all colors are computed by casting secondary rays from the point of intersection into the scene. The color of a point on an object is a function of the object's innate color and the color of the light reflected on to it.

Because ray tracing is straightforward, it is easily extended to any degree of realism. But realism has a price - a huge number of intersection calculations. A 512x512 image requires more than a quarter of a million rays, every one of which

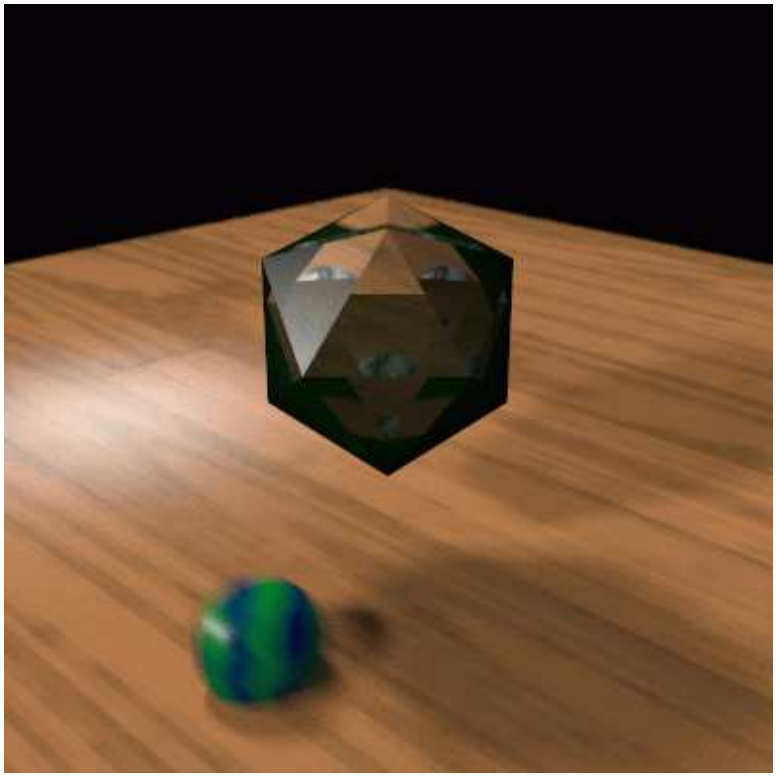

Figure 1.3. A Ray Traced Image with Cook Effects.

has to be tested against every object in the scene. Not surprisingly, there has been a great deal of research into accelerated ray tracing.

Acceleration structures reduce the number of intersection tests by sorting the objects in space during preprocessing. With an acceleration structure, a ray can ignore most of the objects that it cannot possibly hit. By 1980, Rubin and Whitted [31] were using hierarchical trees of bounding volumes. In a bounding volume hierarchy, a simple object acts as a surrogate for a complex object. The complex object is not tested unless the ray hits the surrogate. Furthermore, the surrogates are grouped into a tree of higher level surrogates. A ray that misses a high level surrogate eliminates many objects with one test.

Spatial subdivision techniques divide space instead of classifying objects. The simplest example divides space into an axis aligned grid [10]. Rays step though the grid cell by cell, considering only objects that at least partially occupy each

cell. Objects tend to clump together in small regions, leaving large regions of space empty. In a regular grid, the over full cells degenerate into a linear search and the empty cells cause needless traversal time and memory overhead. Adaptive subdivision structures use larger divisions in empty space and smaller ones where objects are densely packed. Examples include octrees [13], and binary space partitioning [16] trees. Figure 1.4 illustrates the use of a bounding volume hierarchy and a regular grid. These sorting techniques make the ray tracing hidden surface removal algorithm an $O(\log N)$ sorted search problem; the scan line zbuffer algorithm remains an $O(N)$ problem.

Acceleration structures reduce the number of intersection calculations per ray, but it still takes an enormous number of rays to generate an image. Every pixel requires a ray, and realistic images require many rays per pixel. Fortunately, all pixels are independent, which makes parallel processing highly efficient. In an image parallel renderer, several machines split the image, and ray trace pixels concurrently. Figure 1.5 shows how processors can divide the image plane and work in parallel. In this image sixty-two machines add unique gray levels to the pixels that they ray trace. In theory, N machines can compute the same image in $1/N$'th the time required by one machine. In practice, significant challenges arise in the pursuit of this goal. The machines have to cooperate to divide the image and to put their pixels back together. Interactive frame rates require efficient use of a parallel machine's architecture.

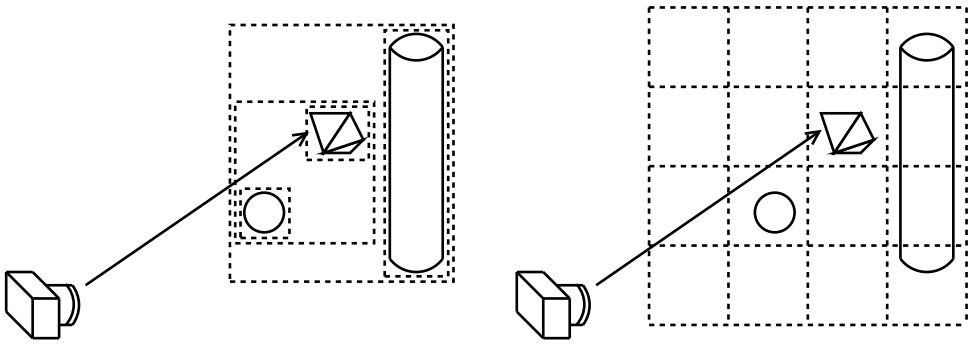

Figure 1.4. Bounding Volume Hierarchy and Grid Acceleration Structures.

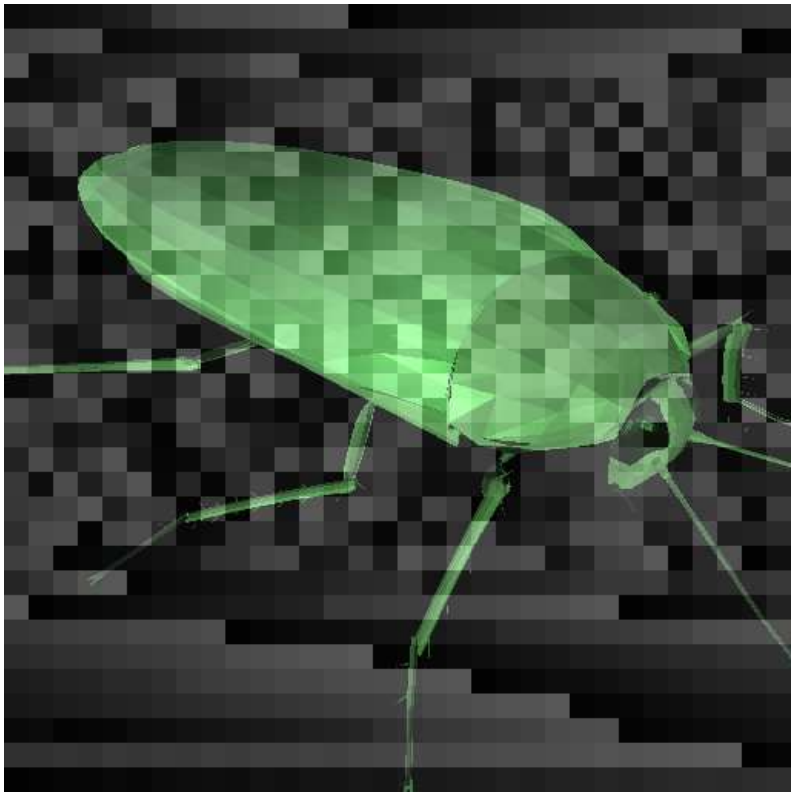

Figure 1.5. Work Distribution in the Image Plane.

In fact, the challenge is great enough that interactive ray tracing is a relatively new phenomenon. *-Ray [26, 27] was one of the first interactive ray tracers. It uses parallelism and acceleration structures to attain interactivity. The program is tuned to take advantage of high performance, shared memory parallel supercomputers like Silicon Graphics' (SGI) Origin [32]. For example, acceleration structures exactly align on cache line and memory page boundaries. Also, the program synchronizes with a fast, machine level barrier. *-Ray is an image parallel renderer, where the independent ray tracing engines are threads running on separate CPUs. A supervisor thread breaks the image into tiles, and the worker threads retrieve tiles whenever they are idle. The supercomputer's shared memory gives all threads fast access to the same memory space, so communication between threads is automatic.

*-Ray is a general purpose renderer, capable of rendering any type of object, including volumes defined by arrays of scalar values. Some applications that generate

this type of data include computerized tomography (CT) and magnetic resonance imaging (MRI) medical scans, and computational fluid dynamics (CFD) simulations. *-Ray makes it possible to interactively render large volumetric datasets [27] at full resolution. In medical applications, accuracy and resolution are especially important. The two rendering modes considered here are isosurface and maximum intensity projections (MIP). An isosurface rendering is a drawing of the locus of points within the volume that share a particular value. A MIP is a projection of the maximum values within the volume. The SGI Origin is extremely well suited to large volume visualization because its fast interconnection network gives every 64-bit processor access to a huge memory space.

The trend in high performance computing is moving away from large monolithic supercomputers and toward groups or clusters of lower cost PC class machines. The primary motivation behind this trend is price and the related consideration of availability. Tremendous competition in the PC market has driven vendors to create reliable, powerful and affordable machines. The processors within today's PCs are nearly as powerful as those used in supercomputers. With powerful affordable PCs, researchers are turning to clusters of PCs to solve computationally intensive problems such as the modeling of the earth's mantle or the study of the complex chemical reactions in fires. Although clusters are extraordinarily well suited to computational intensive problems that do not require frequent communication among processing elements, it is less clear that they are suitable for problems that require constant communication. This is because the network infrastructure that links PC's into a cluster still lags the dedicated hardware within supercomputers.

My thesis is that cluster-based interactive ray tracing of multi-Gigabyte volumetric datasets, which are larger than the memory space of any one node in the cluster, is possible. We have developed a cluster-based interactive ray tracer, derived from *-Ray, that demonstrates the verity of the thesis. Like its predecessor, the new program is designed to efficiently use the capabilities of the machine on which it runs. The distributed ray tracer is able overcome the slower interconnection

between processing elements, and even to use the network to surpass the memory limit of the processors that it runs on.

Chapter 2 summarizes related work in cluster communication technology, interactive ray tracing, and visualization of large datasets. Chapter 3 explains the networking library that connects distinct executables, running on separate computers, into a cohesive program. The library, called Legion, adds a minimum amount of overhead to the socket layer, provides thread safe transport, allows asynchronous handling of inbound messages, and includes a distributed shared memory facility. Chapter 4 discusses how the distributed interactive ray tracer works when the scene database is small enough to be copied into each node's memory. We analyze the performance of the renderer and compare the program's performance with that of *-Ray. Chapter 5 explains how the program uses Legion's distributed shared memory to render large volumes. We analyze the factors that influence the extended program's interactivity and again compare the program with *-Ray. In Chapter 6 we summarize the results of our investigations and discuss possible future research.

CHAPTER 2

RELATED WORK

Libraries like PVM [12] and MPI [15] abstract the details of network programming. These libraries add facilities like logical subgroups, heterogeneity support, debugging support, additional reliability, distributed process control, high level synchronization operations and so on, to the underlying socket layer. PVM was one of the first network abstraction libraries for cluster computing. Significant features of PVM are support for heterogeneity and console driven distributed process control. A free version of PVM is available online [24].

In 1994 a drive to standardize the facilities provided in libraries like PVM resulted in MPI, the standard interface for message passing libraries. As a standard, MPI gives parallel programs the important feature of machine and vendor independence. A program written on a cluster of PCs using MPI should also work when it is compiled and run on a supercomputer that supports MPI, regardless of the network infrastructure. MPICH [14] is a free implementation of the MPI standard. LAM-MPI [5] is a more highly tuned implementation of the standard. MPICH and LAM-MPI are also available online [2, 1].

On a cluster, the facilities that PVM and MPI add to sockets can increase run time overhead. When a program only needs fast transfer between machines, it is sometimes better to forgo the additional features and use sockets directly. This is especially true when using dedicated clusters, for which heterogeneity and reliability are not of great concern. With this in mind, Legion was designed to add a minimal abstraction layer on top of sockets.

A criticism of the message passing environment is that the programmer has too much responsibility for the sharing of data among parallel processes. The programmer must explicitly control every exchange of data. In a shared memory

environment, the same data space is accessible everywhere, and the programmer needs only ensure proper synchronization.

Software distributed shared memory (SDSM) has been an active area of research since the late nineteen eighties. The first SDSM was IVY [18], a sequentially consistent page-based distributed shared memory. It used the operating system to trap accesses to pages of shared data, and on each access ensured that the data was consistent everywhere. The major criticisms of IVY was that it was inefficient, requiring too frequent communication, and that it was subject to false sharing problems, caused when unrelated data items happen to share the same page. Numerous systems have followed that attempt to improve performance. Judge et al. have written an excellent survey [33] that succinctly explains the issues in SDSM design and provides representative example systems.

A major direction taken by researchers to improve performance was to weaken the consistency model. In a weakly consistent memory, the memory is only guaranteed to be consistent in relation to synchronization operations, and in general processing can continue while consistency maintenance is taking place in the background. The Midway [4] SDSM uses a kind of weak consistency called entry consistency. In an entry consistent memory each shared object is made consistent with acquire and release operations. When one shared object is made consistent, all others are unaffected.

A more recent concept in SDSM design has been investigation of extensible shared memory models. A library called CVM [17] exemplifies this design. An extensible shared memory allows variation in the shared memory model, so that the application designer can tune the model to best match the library to the application that uses it. In CVM, a base class implements a rudimentary shared memory and separate derived classes extend the base in different ways. CVM provides two lazy release consistent models and one sequentially consistent model. Legion's distributed shared memory borrows the concept of managing data objects individually from Midway and the idea of an open memory model from CVM.

Ingo Wald et al. [35, 36, 34] at the University of Saarbrücken have done important recent work on interactive ray tracing with personal computer class machines. They demonstrated the low complexity of ray tracing by showing that for very large scene databases ray tracing on a single PC can exceed the frame rate of zbuffer rendering on high end dedicated graphics processors. To maximize performance, they restrict their renderer to surfaces composed of triangles in a binary space partitioning acceleration structure. This leaves generality behind in favor of fully optimizing ubiquitous triangle-based models. They only allow binary space partitioning trees and triangle intersections in order to trace multiple rays in parallel with vector parallel instructions. This exploits parallelism and improves memory coherence. Data needed for one pixel is likely to be needed for the pixel's nearest neighbors. For highly complex scenes, they render on a cluster. In their renderer, a large central server holds the entire database. Render clients replicate the top levels of a Binary Space Partitioning (BSP) tree and fetch the bottom levels on demand from the server. The clients maintain a small cache to reduce the number of network transfers. To hide data transfer delay, stalled rays go into a wait queue, and ready rays continue. The program described in this thesis differs in that it remains a general-purpose renderer, even if it is not as highly optimized for triangular data. It can ray trace scenes that contain implicit or polygonal shapes as well as volumetric data, and can use several acceleration structures.

Using programmable graphics hardware for ray tracing is a recent development. Ray tracing on custom hardware has been investigated before but ray tracing specific hardware has never found wide acceptance. Today, graphics co-processing cards like those from NVIDIA [23], ATI [3] and others are widely available and are becoming very powerful. Tim Purcell et al., demonstrated that it is feasible to ray trace with off the shelf programmable graphics hardware [30]. They implement a ray tracer in the fragment processing stage of a GPU simulator that emulates leading edge graphics card functionality. The algorithm encodes triangles and a regular grid acceleration structure into texture memory. A fragment program traverses the grid, and computes ray triangle intersections. They compare a multipass algorithm

to a single pass branching algorithm that requires a more significant change to graphics card architecture. Like the Saarbrücken renderer, this system significantly accelerates the ray tracing process, but lacks generality.

Our approach to large volume visualization is similar to the work of Corrie and Mackerras [8]. They implemented volume rendering on a Fujitsu AP1000, a distributed memory, message passing parallel computer. They demonstrated that caching makes feasible the volume rendering of data sets that are too large for the memory of any one computing element. Our approach implements a similar algorithm on modern commodity hardware. Our algorithm allows hybrid parallel rendering, where each node runs multiple render threads. To achieve greater interactivity, we present techniques for reducing the number of shared data accesses, improving the hit rate, and decreasing the access time.

Palmer et al. [25] previously rendered large data on a hybrid distributed shared memory machine by optimizing memory access throughout the memory hierarchy. Our system shares the technique of data bricking for memory coherence with their work. Their system is a data parallel renderer where each node is responsible for a subset of the volume. Each node renders its own data, and then shares partially completed image tiles for final depth composition. For better load balancing, their system will migrate blocks among nodes and replicate some blocks onto several nodes. A key difference between our work and theirs is that theirs uses a central server to manage the data distribution. Although the central server is a potential bottleneck, it is able to optimize the distribution of blocks for better performance.

Michael Cox and David Ellsworth [9] have also done prior work on rendering large data sets. They work in a uniprocessor environment for the visualization of computational fluid dynamics data. Because they restrict their work to single workstations, they limit their investigations to loading pages on demand from the file system using the mmap() system call. We have implemented a memory-mapped server that works similarly. We find that in a distributed environment, it is faster to get data over the network than it is to get it from the local hard disk.

Antonio Garcia and Han-Wei Shen [11] render large volumes on PC clusters with a combination of image-space and object-space partitioning. Their "interleave factor" allows the user to control the ratio between the two. They divide the cluster into $N/2^i$ groups where N is the number of nodes in the cluster and i is the interleave factor. Each group has a different subvolume. The nodes within a group conserve memory by down sampling their common subvolume by taking interleaved filtered slices. Each member of the group renders a different portion of the screen. All nodes that have the same rank within their respective group have the same pixels and composite pixel results using Ma et al.'s binary swap algorithm [20]. They report that interleaved sampling with filtering removes most, but not all, of the down sampling artifacts. Our work differs in that we work with the full resolution data at all times.

David Porter and a team from the University of Minnesota [29] do direct volume rendering of large data sets on PC clusters. They create animated sequences by recording an interactive preview session with down sampled data. Later, they render the recorded session off line at full resolution data. They report full resolution render times in the tens of seconds per frame. Although the rendering method is different, the contribution of this thesis is the ability to interactively render, i.e., at multiple frames per second, the data set at full resolution.

CHAPTER 3

LEGION

"My name is Legion," he replied, "for we are many."

Mark 5:10

3.1 Communication

3.1.1 Connection and Transfer of Data

Legion is a networking library that adds very little functionality to the underlying Transmission Control Protocol (TCP) socket layer in the pursuit of eliminating networking overhead. The application programmer's interface (API) to Legion is given in Appendix A. The basis of Legion is a C++ class called communicator that establishes a fully connected graph of TCP sockets between the nodes in a cluster. The application program can control the sockets directly, or the communicator can handle network traffic for the program in its own thread.

To start a distributed application, there must be some way to start a program on each of the nodes in the cluster. A program with a communicator can call communicator::rspawn() to start programs on the other nodes in the cluster. "rspawn" uses a secure shell to log in to each node and start a named program. Each instance of the program receives mandatory command line arguments, including working directory, node rank, and a TCP port number for the distributed application. The spawned instances use communicator::parse_args() to retrieve the arguments.

At this point, every node has a running program, but none of the programs are connected. communicator::init_web() establishes the connections by creating socket links between the programs. "init_web" establish the links by causing each

node to initiate connections to lower ranked nodes, while accepting connections from higher ranked nodes. The communicator stores the resulting connected socket descriptors in a connection list. When every node is connected to every other node, the application is free to use the sockets directly.

The communicator can direct network traffic to free computational threads from IO duty. If the communicator has a control thread to run on, it will asynchronously handle inbound messages. To use this capability, the application must register message handling functions by calling communicator:register_msgs(). Registration maps a message ID to a class method or static function that can receive and process the named message. To support this functionality, every transfer between nodes has to begin with an ID. communicator::send_msgs() prepends the header to the outgoing data stream, sending the header and the data with one system call.

The communicator also provides receive methods, which can be called within message handling functions to pull in the data stream. The advantage of using the communicator's send and receive methods is that they are thread safe. When multiple threads share a communicator, the threads cannot send simultaneously, or the outgoing data stream will become garbled. The communicator protects each send with a pthread mutex. For efficiency, the communicator internally uses sendmsg() and recvmsg() system calls to transfer multiple buffers of data with a single socket operation.

The communicator thread works by looping around a select() system call. The select puts the thread to sleep and watches all of the socket descriptors in the connection list. When the select sees activity on one of its sockets, it wakes the communicator, which promptly reads the message header and calls the appropriate handler. The handler takes the socket, the rank of the sending node, and the message ID, and reads the rest of the message directly into application memory. The handler can process the data, or let a computational thread take over. The later is preferable because the communicator cannot return to handling IO until the handler returns. Figure 3.1 shows a high level view of a distributed program

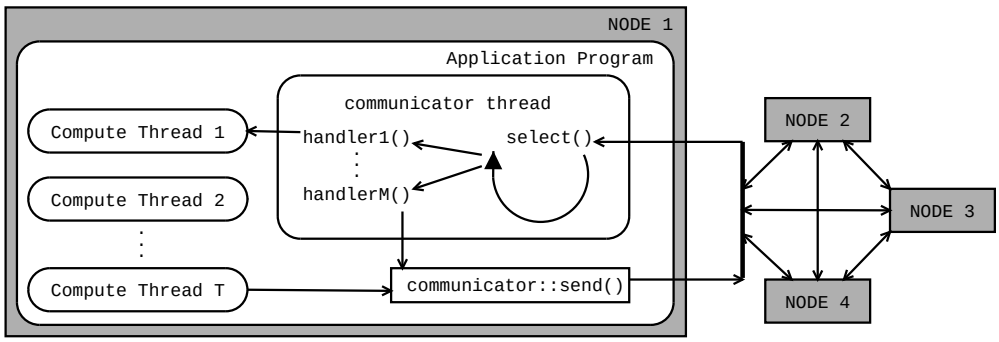

Figure 3.1. Asynchronous Message Handling.

running on four nodes, where the first node has $T + 1$ threads and M registered message handling functions.

3.1.2 Logs, Statistics and Distributed Barriers

In addition to basic communication, a few extra facilities are included in the library. These include per node and per thread message logging, statistics gathering, distributed barriers and distributed shared memory. Each of these are optional, and designed so that they can be omitted for better efficiency. Section 3.2 explains distributed shared memory; the rest of Legion's optional facilities are explained below.

The file logger.h has preprocessor directives that control Legion's logging and statistics gathering facilities. When enabled, logging and statistics add significant overhead to the program because they increase the size of the Legion's data structures, cause frequent file IO, spend time calling gettimeofday(), and require thread synchronization. Because of this, they are implemented with preprocessor directives that can be completely elided from the program.

The "log" macros control error, Legion specific, and application specific, message priority levels. There are two versions of each macro, one directs output to a node specific file, and the other directs output to a node and thread specific file. The log macros operate on C++ IO streams.

The DOSTATS macro enables communication and distributed shared memory statistics gathering. Communication statistics include a count of the number of select(), send() and recv() socket operations, a count of the number of bytes sent and received, and a measurement of the total time spent communicating. DSM statistics include counts of the number of accesses, the number of hits, the number of remote accesses, the number of associative hits and a measurement of the total time spent on each access. Statistics are written to the node specific log files when the program terminates.

Synchronization is essential in most parallel programs, so Legion includes a distributed barrier primitive. A program can use any number of barrier objects, each of which may contain any subset of the nodes in the cluster. Internally a barrier object contains a list of nodes, an arrival marker for each node in the list, and a message handling method. communicator::register_barrier(), takes in a list of nodes and returns a newly created barrier over those nodes. The application calls legion_barrier::arrive() to use the barrier. This sends a short message to all nodes in the node list, and then checks the array to count the nodes that have arrived. If all nodes are present, the arrive call returns immediately. If not, the calling thread sleeps until the barrier's message handler sees the last node's arrival message.

3.2 Distributed Shared Memory

3.2.1 Common Principles

In a tightly coupled parallel machine, shared memory allows threads in a parallel program to operate directly on the same globally visible memory. In a clustered environment, the memory of each node is private, only accessible to other nodes through program directed transfers. Legion's DataServer class provides a mechanism whereby every node has access to the aggregate memory in the cluster. The memory is object-based, and does not rely on the operating system to manage pages of shared memory. This reduces the problem of false sharing and allows 32-bit programs to access more than 4 GB of memory.

The application obtains a DataServer object by calling communicator::register_dataserver(). The memory model is extensible, and five different classes implement the DataServer interface in slightly different manners. Four of the five implementations only allow read access. The ray tracing application that Legion was designed to support does not write to the shared memory during run time, so it is efficient to omit write consistency maintenance entirely. The four read only classes are: single threaded, direct mapped, set associative and secondary storage memory mapped. The writable class is a sequentially consistent shared memory with fixed distributed ownership and a write invalidate protocol.

The DataServer organizes memory into blocks, and identifies each block by an integer key. The application specifies the block size and the total number of blocks when it creates the DataServer. The blocks can be aligned in memory for better translation look aside buffer and cache efficiency. Each node allocates 1 GB of local memory to hold blocks, but the DIRT_MAXDSMMEM environment variable can override the default. With N nodes, each node assumes ownership of $1/N$ of the blocks, and uses the rest of the reserved space to cache blocks from other nodes. The DataServer keeps its own data in an area called the resident set. The cached data goes into the local cache. The rank of the node that owns key k is given by $k \bmod N$. On the owning node, the key will be found at entry k/N in the resident set. On remote nodes, the same key will be found within A slots of entry $(k * A) \bmod LC$, where A is the associativity and LC is the size of the local cache.

To use the memory, an application thread calls DataServer::get_data(k) to get a pointer to the beginning of block k. If the block is not in local memory, the DataServer will suspend the caller while it obtains the specified memory block over the network. When the application thread finishes using a block, it must call DataServer::release_data(k) to free the block's space. In between the two calls, other threads cannot replace the block. The structure of a three node program, running two application threads per node, and using the DataServer is illustrated in Figure 3.2.

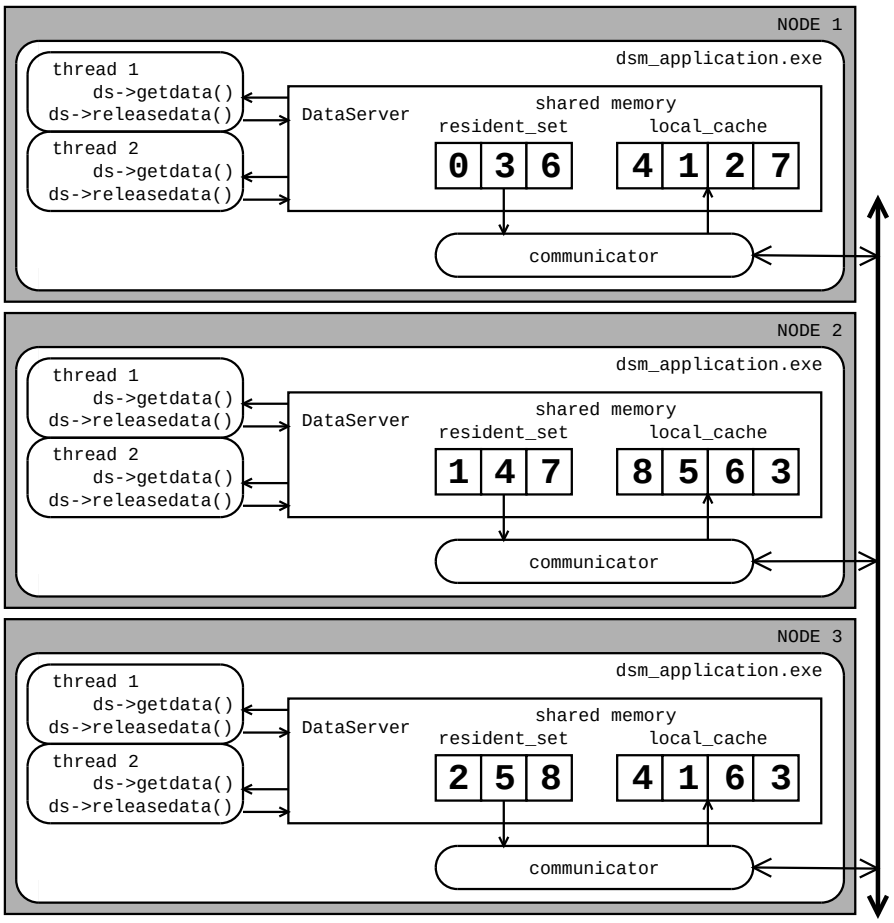

Figure 3.2. Distributed Shared Memory Architecture.

Acquiring a block involves the following actions. The acquiring thread queries the DataServer to access the block. If the block is in local memory, the DataServer marks the block in use and returns its address. Otherwise, the thread reserves a space for the block, sends a request to the node that owns the block, and goes to sleep. When the block's owner receives the request, its DataServer enforces a single writer multiple reader policy. Next, the owner marks the block in use and sends a copy of the block to the requesting node. When the acquiring node's DataServer receives the block, it receives that block into the reserved space and then wakes the acquiring thread.

3.2.2 Read Only Memories

A read only memory can eliminate write consistency maintenance delay entirely. The simplest of the read only memories is the single threaded DataServer_single class. It assumes a single application thread to omit consistency checks among compute threads in a node.

For the multithreaded memories, a counting semaphore protects each location in the local cache. The legion_semaphore class puts a wrapper around a posix semaphore to use it as a counting semaphore. The semaphore counts the number of reading threads for the block in a given location. Before a thread can write to a location, for instance when it reserves a block to fill in over the network, it must atomically decrement the count to zero.

The DataServer_direct class adds thread safety to the DataServer_single class. In these two classes, each key can only reside in location *key* mod *local_cache_size*. The DataServer_assoc class is an A way set associative cache with a least recently used replacement policy. Associativity has the potential to reduce network traffic, but it adds overhead. Each lookup has to check A locations, and all accesses have to call gettimeofday() to update the time stamp. The DataServer_mmap class is not actually a distributed memory. Instead of fetching blocks over the network, this class invokes the mmap() system call to fetch blocks from the file system. The memory map class does not have a resident set, using all of the reserved space for the local cache instead.

3.2.3 Writable Memory

The DataServer_rw class has separate acquire and release operations for read only and writable access. The writable operations are DataServer::get_data_for_write() and DataServer::release_written_data(). The read and write acquire operations enforce a single writer, multiple reader policy. No thread can write to a block until every other thread on the cluster releases it. Threads can read the same block simultaneously, but no thread can read a block while another thread is writing to it.

Before obtaining data, write acquires must first negotiate for sole access to the target block. The process begins with a write request message sent to the owner. When the owner sees a write request for a block, its DataServer starts queuing later requests for that block. Next the owner's DataServer waits for all local threads to release the block. When no local threads have access, the DataServer broadcasts an invalidate message to the rest of the nodes. When the other nodes see the invalidate message, they wait for sole access among their own threads, invalidate locally cached copies, and then send an acknowledgment back to the owner. Because no other thread can acquire the block at this point, there can be no data inconsistencies. When the owner receives a full set of invalidate acknowledgments, its DataServer sends the requested block to the acquiring thread. The owner continues to queue requests for the block until the requester releases the block. The requester releases the block by sending the new contents back to the owner. With an up to date copy, the owner is once again free to grant access to the block.

In the read only memories the communicator thread was able to service remote acquire requests by executing the DataServer's message handler. The handler simply sent the block to the requester and this did not seriously delay the sender's communicator thread. With write access, the owner often has to delay the reply. A simple sleep will not work here, because the communicator has to be ready for the next incoming message. To break the deadlock, the writable DataServer has its own thread, and a queue of pending block requests. The communicator thread uses the DataServer_rw's message handler to enqueue block requests, and the DataServer_rw's thread consumes them at its leisure.

3.3 Analysis

In this section, we do a series of tests to examine the performance of the communicator and the DataServers. All tests use a cluster that consists of 32 dual-CPU Linux PC's and an Extreme Networks 6816 Black Diamond Gigabit Ethernet switch. The configuration of the cluster nodes is described in Table 3.1. In the tests, elapsed time is measured by calling gettimeofday().

Table 3.1. Cluster Components.

Component	Type
Motherboard	Supermicro P4DC6+ (Intel 860 chip set)
CPU	Dual Intel Xeon 1.70GHz (256KB cache size)
Memory	2x512M Corsair ECC RDRAM
Network card	Intel Pro1000/XT Driver version 4.4.12-k1
GPU	NVIDIA GeForce3 (64MB) Driver version 1.0.2960
Hard drive	18GB Seagate Cheetah U160 (15000 RPM)
Kernel	Linux 2.4.20 (Nov 28, 2002)
Compiler	GCC 3.0.4 -O3
Cluster File system	PVFS version 1.5

3.3.1 Communication Performance

We use a ping-pong test [22] to compare the basic data transfer rate of TCP sockets, Legion, LAM-MPI version 6.5.6 and MPICH version 1.2.3. The MPI transfer operations use the MPI_BYTE data type to omit type conversion in transfer. In the test, a central node cycles a buffer around a ring of nodes, sending to and receiving from each node in turn. The central node records the elapsed time for 1000 trips around the ring. The size of the message and the number of nodes are varied.

The elapsed time measurements for all tests are given in Table 3.2. The average and extreme values of latency and bandwidth for the tests are given in Table 3.3. Overall, MPICH is slowest, TCP is fastest, and Legion and LAM-MPI are roughly equal.

Legion under-performs TCP because it adds headers to all messages, it adds a communication thread, and it requires three system calls per reception. As the message length increases, the select() and recv() calls that retrieve the header become less important, and the data reception time dominates. LAM-MPI handles small messages faster than Legion does because it obtains a header and some data

Table 3.2. Complete Network Library Comparison Results.

Nodes	Bytes	TCP[s]	Legion[s]	LAM-MPI[s]	MPICH[s]
2	1	0.08623	0.09416	0.1025	0.1014
	16	0.08638	0.09750	0.1026	0.1086
	64	0.08794	0.1001	0.1060	0.1086
	256	0.09398	0.1104	0.1172	0.1219
	1024	0.1617	0.1746	0.1866	0.1883
	4096	0.2420	0.2519	0.2691	0.2760
	16384	0.4793	0.4907	0.5417	0.5693
4	1	0.2054	0.2586	0.2651	0.2853
	16	0.2080	0.2591	0.2653	0.2843
	64	0.2153	0.2689	0.2746	0.2938
	256	0.2440	0.2967	0.3063	0.3308
	1024	0.3896	0.4355	0.4536	0.4691
	4096	0.6064	0.6396	0.6835	0.7178
	16384	1.312	1.376	1.506	1.570
8	1	0.4458	0.6146	0.6019	0.6634
	16	0.4544	0.6218	0.6008	0.6744
	64	0.4641	0.6434	0.6248	0.6870
	256	0.5375	0.7063	0.6951	0.7562
	1024	0.8379	0.9907	0.9974	1.060
	4096	1.344	1.448	1.518	1.593
	16384	2.993	3.181	3.438	3.551
16	1	1.141	1.575	1.437	1.679
	16	1.149	1.569	1.443	1.680
	64	1.186	1.691	1.484	1.726
	256	1.344	1.845	1.655	1.902
	1024	2.148	2.708	2.483	2.738
	4096	3.309	3.768	3.690	3.954
	16384	6.852	7.640	7.786	8.205
32	1	2.521	4.322	3.090	
	16	2.545	4.345	3.091	did
	64	2.616	4.515	3.253	
	256	2.967	4.894	3.623	not
	1024	4.799	6.746	5.522	
	4096	7.266	8.739	7.983	finish
	16384	14.58	16.68	15.95	

Table 3.3. Summary of Network Library Comparison Tests.

	TCP	Legion	LAM-MPI	MPICH
Min Transfer[μs]	31.84	43.10	42.91	47.38
Max Bandwidth[Mb/s]	584.8	550.1	509.0	492.8
Avg Transfer[μs]	80.77	95.38	94.16	98.81
Avg Bandwidth[Mb/s]	144.7	130.6	127.1	122.1

with one recv() call. As the message length increases, Legion begins to outperform LAM-MPI. Note that LAM-MPI scales better, and the performance crossover point increases with the number of nodes. MPICH is the slowest on all tests, and this version of the library is unstable at high node counts. It would not complete the test at all with 32 nodes.

3.3.2 Distributed Shared Memory Comparison

We use a random read test to analyze Legion's distributed shared memory. In the test, client threads acquire and release randomly chosen keys. Each thread measures the elapsed time from the first acquire to the last release. Unless otherwise noted, each test uses 32 nodes, each of which runs a single client thread; the total data size is 1 GB; each block is 16 KB; each node has a 256 MB local cache; and each client acquires and releases 10000 random keys. These tests emphasize the miss penalty because the number of keys accessed is small compared with the total number of keys.

The first test compares the DataServer implementations. The results are given in Table 3.4. The elapsed time shows that the single threaded memory throughput is highest, followed by direct, associative, writable reads, memory mapped and writable writes. The order corresponds to the complexity of key lookup and synchronization checks in each class, which are reflected in hit and miss times. We test the associative memory with a one element set to examine the overhead added to the direct memory. In the write test, no accesses hit, because all accesses require network traffic to maintain sequential consistency.

Table 3.4. Comparing the DataServers.

Memory Type	Total Time [s]	Hit Rate [%]	Hit Time [μs]	Miss Time [μs]
Single Threaded	5.461	10.17	2.333	616.0
Direct Mapped	5.461	10.02	2.331	619.9
1 Way Associative	5.477	10.18	2.461	618.3
Memory Mapped	17.45	6.813	2.624	579.9
Writable Reads	5.549	10.16	2.872	625.1
Writable Writes	46.00	0	0	5232.

The associative cache is sensitive to the access pattern. Block reuse is the exception with a random access pattern. Concentrated accesses to a few entries favor associativity. To demonstrate this point, we test the associative and the direct mapped cache, and change the associativity and the fraction of selectable blocks. In each run, the program chooses random blocks from the set of blocks that align to a portion of the local cache. The smaller the set, the more often blocks that use the same slot will be accessed. The results are given in Figure 3.3. The more the selection is constrained, the more associativity helps. In this test the total memory size is four times the local cache size. No more than four blocks can share a slot, and the performance improvement stops at the four-way cache.

In Table 3.4, memory mapped reads are 3.2 times slower than network reads. The hit rate is 3% lower than for the other read tests because the memory mapped DataServer lacks a resident set, but this is hardly enough to account for the difference. The time spent when the application accesses the data after the acquire does account for the difference. Only when the application touches the blocks does the operating system actually load the page. The time spent paging is sensitive to the size of the shared memory and to the kind of file system. When the shared memory is smaller than the physical memory, the memory mapped server can outperform the other memories; after the first load, pages do not need to swap. When pages swap, it is faster to use a file on the local hard drive than it is to use a Network File System (NFS) mounted file for obvious reasons.

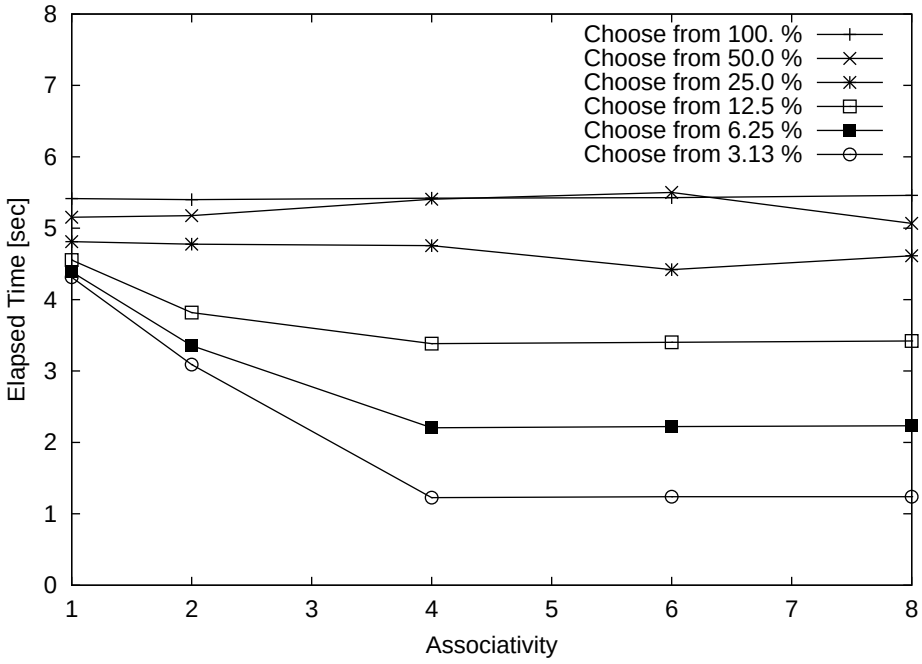

Figure 3.3. Associative Memory and Competition for Cache Slots.

With large files, replication on local disks is wasteful. Replication also limits the data size to that which will fit on a single hard disk. The Parallel Virtual File System (PVFS) [19] is a compromise, it requires network access, but no one machine holds all of the data. This makes the network bottleneck wider and the maximum file size larger.

Figure 3.4 plots the results of a test that varies the size of the data set, the location of the data file, and compares the direct and memory mapped DataServers. The NFS test was stopped at 512 MB because it takes an excessive amount of bandwidth away from the other users of the file system.

In Table 3.4, the write test was 8.2 times slower than the read test, because of synchronization overhead and a lack of parallelism in the implementation. The implementation broadcasts invalidation messages and waits for a complete set of acknowledgments on every write. The implementation lacks parallelism because it uses a simple fixed ownership scheme that is incapable of caching writable data. A single node that repeatedly accesses the same block cannot cache it because the

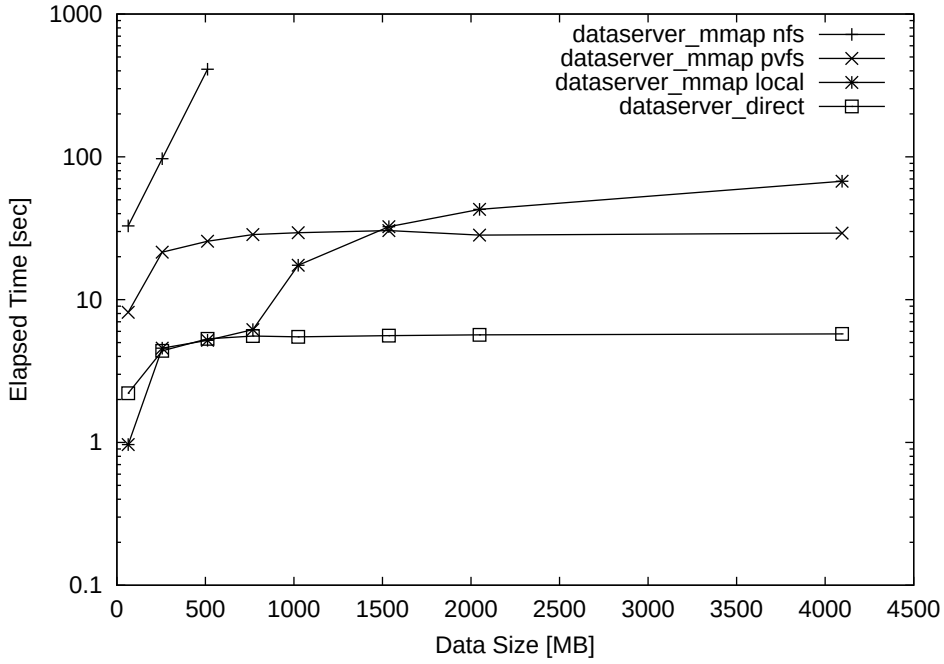

Figure 3.4. Comparison of Network and Demand Paged Memory.

node has to obtain it from and return it to the owner on every access. Additionally the single writer policy prevents multiple nodes or threads from writing to a block concurrently, even when the writes do not conflict. The expected performance in a real application depends upon the ratio of the number of reads and writes. Until a new DataServer extension overcomes the write shortcomings, it will behoove the application designer to reduce the number of writes.

3.3.3 Distributed Shared Memory Scaling

In a parallel system, scaling is the prime measure of efficiency. In Legion's message passing facility, each additional node adds the cost of one socket check to every select call. When messages are infrequent, this cost is negligable.

To analyze how well the DataServer performs with more nodes, we test the DataServer_direct class, varying the number of nodes and the number of threads per node. The local cache size remains constant at 256 MB, or one quarter of the 1 GB data set. We record the elapsed time for the test and divide by the number

of threads, nodes, and accesses to obtain the average block access time. This is the inverse of the total number of blocks touched per second. Figure 3.5 shows the results. Adding clients, with more nodes or threads per node, reduces the average time per block but not linearly. Ideally, increasing from two to four clients would decrease the time from 120 μs to 60 μs. In this test adding two nodes brings the access time to 92.0 μs. Adding two threads instead yields 81.4 μs. Three threads are not better than two because there are only two CPUs per node. In both node and thread dimensions, the amount of scaling is limited by the fact that each node has only one DataServer. The single DataServer becomes more of a bottleneck when the number of clients grows.

3.3.4 Characteristics of the Memory Space

The granularity, total size, and amount of memory cached also influence run time. Changing the block size changes the miss penalty and the hit rate. A large block covers more of the total data space. Therefore, a fetch of a large block is

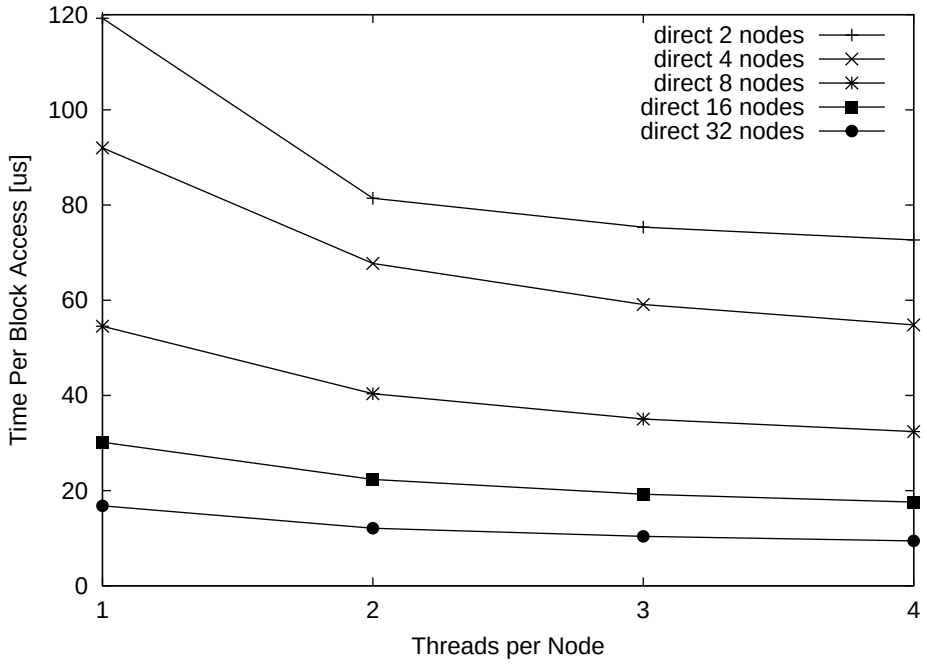

Figure 3.5. DataServer Performance with Increased Node and Thread Counts.

more likely to service additional hits. The penalty for large blocks is that the transmit time is longer. We analyze the effect by testing the DataServer_direct with varied block sizes. We enable Legion's diagnostics to measure the hit rate and miss penalty. We hold the total data size to 1 GB and the local cache to 256 MB but vary the number of blocks to obtain different block sizes. This test models data coherence because 10000 random accesses more completely cover a smaller number of keys made from larger blocks.

Table 3.5 shows the results of the test along with a predicted miss time. There is a strong correlation between miss penalty and total test time in this test because misses dominate. The miss penalty is approximately a startup cost of 220 μs plus the block size divided by an average transfer bandwidth of 40 Mb/s. The bandwidth is lower than that measured in subsection 3.3.1 because in this test all of the nodes are communicating at the same time, often competing for access to the same block serving nodes. Likewise the startup time is higher than the measured latency because of the time required for two messages and the intervening service delay. To summarize the results, larger blocks yield a longer run time because the increase in hit percentage does not cover the extra cost of transmitting bits.

The direct mapped memory graph in Figure 3.4 makes it clear that throughput and memory size are not strongly correlated. In that test misses dominate and the run time was determined by the number of accesses and the miss time. Because

Table 3.5. Shared Memory Granularity.

Size [Bytes]	Keys	Total Time [s]	Hit Rate [%]	T Miss [μs]	Predicted [μs]
256	4194304	2.254	3.345	225.16	226.10
512	2097152	2.336	3.402	233.76	232.20
1024	1048576	2.545	3.720	256.39	244.41
2048	524288	2.903	4.193	293.46	268.82
4096	262144	3.186	5.118	326.08	317.65
8192	131072	3.989	6.921	417.25	415.31
16384	65536	5.567	10.16	603.18	610.62

the hit time is much smaller than the miss penalty, we expect that increasing the hit rate will greatly increase performance.

The hit rate can be modeled by assuming completely random access, over a long run on N nodes. On average $1/N$ accesses will hit in the resident set and $(N-1)/N$ accesses will use the local cache. Of the non-resident accesses, LC/TS will hit, where LC is the size of the local cache and TS is the total size of the data. The time spent accessing the distributed shared memory is then related to equation 3.1.

$$hit\_time\left(\frac{1}{N}+\frac{N-1}{N}\frac{LC}{TS}\right)+miss\_time\left(\frac{N-1}{N}\frac{TS-LC}{TS}\right) \tag{3.1}$$

If the data size is fixed, we can increase the number of nodes to increase throughput. Adding nodes will divide the data more finely, making the resident sets smaller; freeing up space for the local cache. In the limit, the local cache will occupy all of the local memory space, and the nodes will be able to cache as much of the working set as the hardware allows.

The assumption of completely random access motivates this discussion. In a real application the data access patterns will often be more coherent. The more coherent, the more the local cache will be able contain the data that is needed, and the closer the total time will come to the number of accesses multiplied by the hit time.

CHAPTER 4

REPLICATED DATA RAY TRACING

4.1 Implementation

4.1.1 Program Structure

The new distributed renderer inherits a supervisor/workers structure and image space parallelism from *-Ray. The supervisor interacts with the user and assigns pixel tasks. The workers update the image with rendered tiles. On the cluster, the supervisor and workers are complete programs running on separate PC's. The supervisor program consists of one or more display threads and a communicator thread. The worker programs consist of one or more render threads and a communicator. Figure 4.1 shows the structure of the new program. In the diagram, rounded boxes represent threads of execution and square boxes represent classes without threads. The arrows represent the flow of information during rendering.

All nodes need to share the rendering state. For instance, when the user moves the viewpoint, all workers have to render from the new position. In *-Ray, all threads have access to state information in shared memory. In the distributed program, each node has a local copy of the state. To keep the state consistent, only the supervisor is allowed to make updates. The exception is pixel task results, which the workers generate. The state is divided into three groups based upon the how often the variables change. Subsection 4.1.2 describes in detail both why and how the shared state was divided.

In Figure 4.1, the supervisor executable runs on node zero. The supervisor starts each frame by broadcasting the state and assigning initial tasks. At the end of each frame, the display updates the state with user input and posts the rendered image

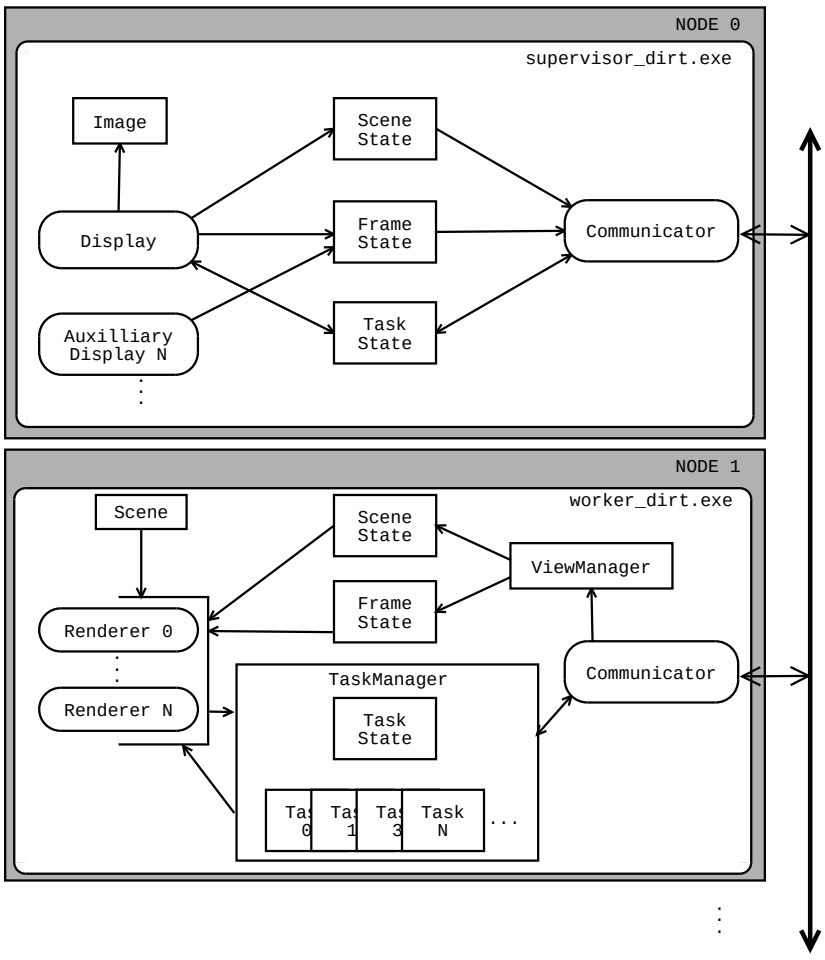

Figure 4.1. Distributed Ray Tracing Program Structure.

buffer. In the meantime, the communicator thread receives results and assigns new tasks. The communicator does this by calling a registered display class method.

Auxiliary displays are optional. Objects in the scene that allow additional user control create their own displays. For instance, isosurface volumes create volume displays that allow the user to change the isovalue. The auxiliary displays work by updating optional data within the frame state.

In Figure 4.1, a worker executable runs on node one. The communicator receives frame state updates and task assignments. The ViewManager class places scene and frame state updates into local shared memory. The TaskManager processes task assignments. It tries to keep a backlog of work in a task queue so that the

render threads will not have to wait for assignments. The queue keeps the render threads busy while tasks are being transferred. Render threads wait on the queue and atomically dequeue tasks from it. The render threads draw each tile according to the render state and the contents of the local copy of the scene. Render threads implicitly request additional work with every returned task. Subsection 4.1.3 discusses the task assignment strategy in detail.

4.1.2 Communication

Communication needs among functional units in *-Ray guided the design of the distributed ray tracer. Inspection exposed several types of shared data. In most cases, the shared data was referenced through a class member in one functional unit that contained a pointer to another functional unit. The new program uses a different mechanism because the nodes cannot directly reference each other's address space.

In *-Ray, the shared data consists of the following:

- Scene - This contains all of the shapes in the virtual world. The workers read the scene, and the scene is rarely written to.

- Image - This consists of two buffers of pixels, one is displayed while the other is filled. The workers write to the image and the display swaps the two buffers.

- Assignment Queue - This contains the pixel task assignments. The display fills the queue at the beginning of every frame, and the workers drain tasks from it during the frame.

- Rendering Options - Some examples are the shadow algorithm, the ray depth limit, and a time stamp for animated shapes. These are kept throughout the program; many are found in the scene and the display objects. When the user interacts with the program the display updates these. The workers read them and render accordingly.

- Camera - This represents the viewpoint from which the render threads draw the scene.

- Object Rendering Options - These variables control the appearance of a few objects in the scene. The scene creates them, the auxiliary displays hold and modify them and the workers read them.

- Ray Statistics - As the workers ray trace pixels, they keep statistics such as the number of rays traced and the number and kind of objects tested. The workers write and the display reads these.

- Processor Statistics - Whenever a worker finishes a task, it records the time. The display can read and display these time stamps to indicate the amount of load imbalance.

In the distributed program, state objects encapsulate most of the shared information. Every node keeps a copy of the state objects in local shared memory where all render threads can access them. The supervisor updates its master copy of the variables, and broadcasts them at the top of each frame. The shared state is divided into three groups based on the rate at which the data change.

The SceneState contains all of the variables that remain constant after initialization. To conserve network resources during rendering, the SceneState is only transmitted during initialization. Other variables change every frame or sporadically, typically in response to the user, but only change in the time between frames. The FrameState holds all of these variables. The TaskState represents the task assignment queue, which changes every time a worker receives and completes a pixel task. The TaskState is a static class with methods that transmit and receive individual tasks. Subsection 4.1.3 discusses pixel task transport more fully. Figure 4.2 presents an overview of sharing in the distributed program.

During initialization, the supervisor and the workers must agree upon the initial SceneState and FrameState. The supervisor has command line arguments, and the worker has arguments to and information from the scene file. Before the first frame,

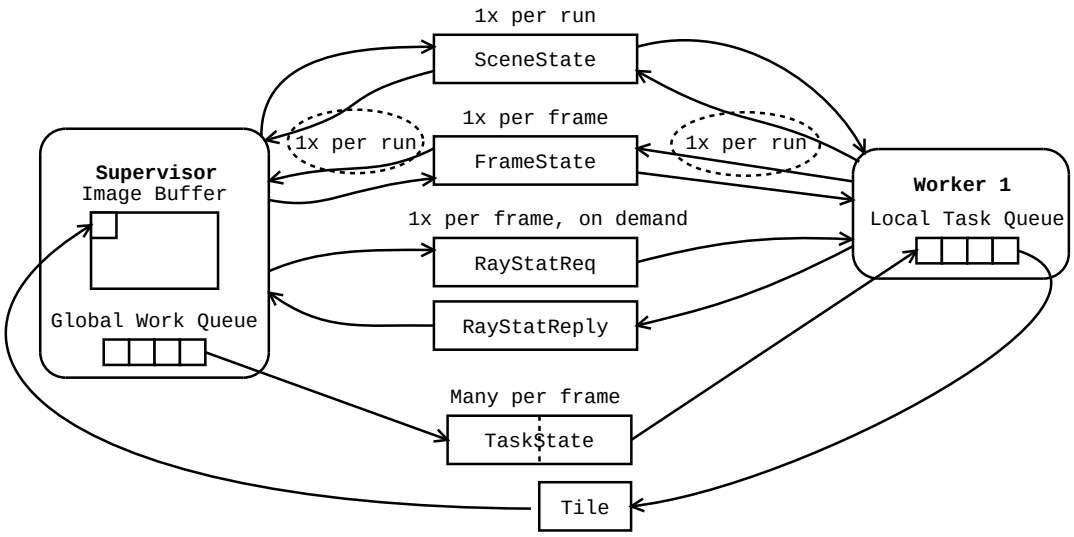

Figure 4.2. Information Sharing Among Nodes.

the supervisor requests the state from the first worker. When the worker replies, the supervisor updates its own copy with the information from the scene. The supervisor applies user specified global command line arguments and broadcasts the updated scene and frame state to every worker. At this point, the workers learn what kind of acceleration structure to use and preprocess the scene accordingly before drawing the first frame.

The frame state contains some items that change infrequently. The supervisor transmits them every frame for two reasons. First, the expensive part of the transfer is initiation. The cost of sending a few additional bytes is inconsequential compared with the startup time. Second, keeping the transfer size constant allows the sender and receiver to use a single system call and avoid conditional execution. That being said, the data is aligned to be more compact. The FrameState transfer is 152 bytes without auxiliary data. When the scene requires auxiliary displays, the FrameState structure grows to accommodate the extra data. During a single run, its size does not change.

The SceneState and FrameState classes are not identical in the supervisor and worker programs. Some of the structures that the state classes touch are not linked

to both programs. For instance, the supervisor executable needs a volume display class to create auxiliary windows for volumes. The worker does not strictly need the class, or the window libraries that it depends upon, so the worker executable is not linked to it. The code to handle state updates has to be different because of this. The worker and the supervisor define the interface to the state objects in a shared header file, but use different source files to implement the interface. The SSceneState.cc and SFrameState.cc files implement the state in the supervisor and the WSceneState.cc and WFrameState.cc files do so the worker. Appendix B lists the contents of the shared state structures.

The shared state structures do not contain all of the shared data. For instance, it is more efficient to replicate the scene database because it is large and unchanging. The replication happens when the supervisor program spawns the worker programs. Trailing command line arguments give each worker a scene file name and scene file specific parameters. Each worker then creates its own copy of the scene, only sharing it internally among render threads. To handle animated scenes, the FrameState contains a time stamp. The ViewManager gives the time stamp to the scene, which animates the moving objects within.

In contrast to the duplicated scene, is more efficient to have one copy of the image and of the work queue. In *-Ray, each worker inserts computed pixels into the image back buffer and atomically retrieves new assignments from the work queue. In the distributed ray tracer, the supervisor keeps both structures and updates them for the remote workers. Subsection 4.1.3 explains how the supervisor and workers cooperate to issue tasks from the supervisor's work queue and to place completed pixel results into the image.

In *-Ray, the workers directly update globally visible statistics. In the cluster, this requires explicit communication. To reduce the network load, ray statistics are transmitted only when activated by the user, and processor statistics are not transmitted at all. Ray statistics track the number of rays traced and the number of objects tested. When enabled, the supervisor broadcasts a ray statistic request at the top of each frame. Each renderer records its own ray statistics while it renders

pixels. The workers only transmit their counts to the supervisor in response to the ray statistic request.

Processor statistics measure load imbalance, by recording the completion time of every task. In *-Ray, the workers add their time stamps to a global structure. In the distributed program, the supervisor takes over this responsibility, by making a time stamp whenever it receives a result. The supervisor needs to know both the rank of the sending node and the rank of the rendering thread on the node to make the timestamp. The node rank is implicit in the socket descriptor, but the thread rank has to be included with the task results. To keep the message size constant, the result message contains the thread rank even when the statistics are turned off. The supervisor will not call gettimeofday() unless processor statistics are on.

4.1.3 Task Assignment

Task assignment is a crucial determinant of performance; it must be fast and it should evenly distribute the workload. Assignment begins with the supervisor. At the top of each frame, the supervisor fills its own work queue and then sends an initial task to each renderer. As each renderer finishes a task, it sends the pixels to the supervisor, which receives the results directly into the image buffer. The supervisor pops the next task from its work queue and assigns it to the worker. The supervisor transitions to the next frame when it receives the last task.

In a dynamic assignment strategy like this one, the task allocation changes to give every processor the same amount of work. More specifically, it is a demand driven strategy in which the workers request additional work to balance the load. This implies that tasks must be both assigned and returned. Static assignment does not change the task distribution from frame to frame. The workers know what tiles they have to trace and so the supervisor does not need to transmit assignment messages. This will improve the frame rate unless one processor happens to be stuck with too much work.

The assignment granularity is orthogonal to the assignment strategy. Large assignments reduce communication overhead by amortizing control and latency

overhead per pixel. Large tasks can consist of large tiles of pixels or of multiple tiles sent in a single transfer. Unfortunately, large assignments are susceptible to load imbalance. With small assignments, more workers cooperate on difficult portions of the image.

The distributed ray tracer has a flexible assignment scheme that the user can control. It defaults to *-Ray's demand driven strategy. For better load balancing, we use a heuristic that assigns large tasks first, gradually reducing the size until the end of the frame. The new program adds three capabilities. First, the user can dynamically change the size of the individual tiles that make up the tasks. Second, the user can dynamically change the sliding granularity parameter, which controls the number of tiles per task. At one extreme, the supervisor divides the screen into equal portions and gives each worker one task. At the other extreme, each task can contain a single tile. Third, the user can change between dynamic and static assignment. In static assignment, the workers constantly recycle the previous frame's assignments.

For the dynamic scheme, after transmitting the FrameState, the supervisor gives one assignment to each worker. The granularity parameter determines the number of tiles in the assignment tasks. Larger values produce smaller tasks. The initial tasks always contain at least one tile for each render thread. This ensures that two or more render threads will not synchronize on access to singly assigned tasks. In the worker, the TaskManager receives the task, and then splits it into subtasks, one subtask per renderer. If the initial task contained more than one tile per renderer, some of the subtasks will also contain multiple tiles. The TaskManager atomically enqueues subtasks onto the worker's local work queue. Meanwhile, the render threads wait to atomically dequeue subtasks. We use two semaphores to protect access to the queue, one for availability, and the other for access. Figure 4.3 illustrates task assignment to a node with two render threads.

Internally, each render thread has a buffer for task results. The buffers have space for a few tiles with their associated headers. The renderer computes and then packs three byte RGB pixels into the buffer. When the local buffer fills up, or when

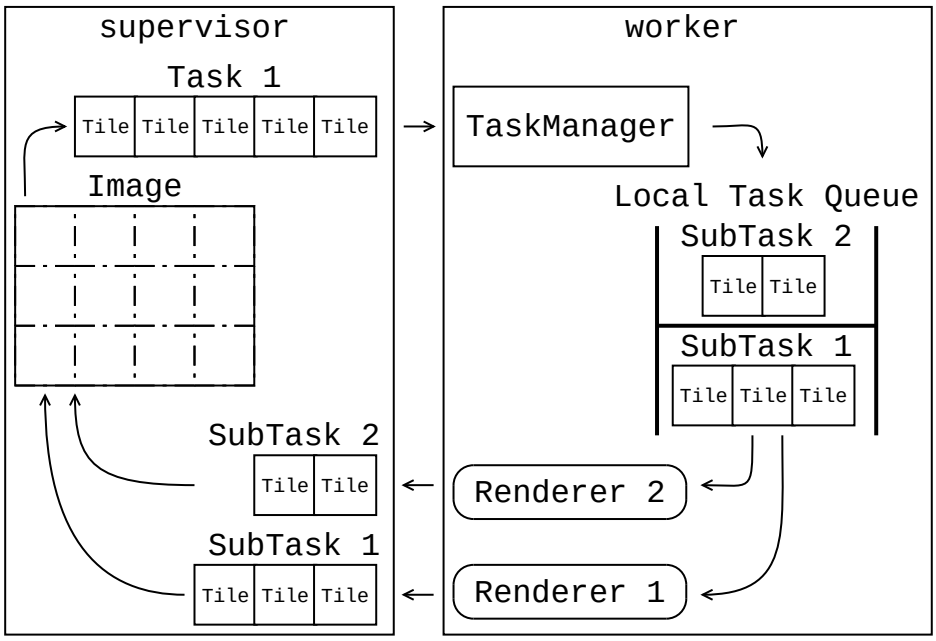

Figure 4.3. Dynamic Task Assignment.

the thread finishes all of the tiles in the task, the renderer sends the buffer back to the supervisor. The send uses a sendmsg() system call that packs multiple tiles, and control headers, into a single network operation.

The supervisor receives each tile in the task in turn. The supervisor uses a recvmsg() system call to receive scan lines directly into the image buffer. The supervisor treats each task as a request for another assignment. This eliminates the need for explicit request messages from the workers. The supervisor sends the next assignment after reading the first task in the tile, with the goal of having work available for the renderer by the time it finishes sending the rest of the tiles.

The static/dynamic selection, tile size and granularity controls are part of the FrameState. The FrameState update happens at the top of each frame, which ensures that the workers and the supervisor always agree on the size of the task messages.

For static assignment, the workers continuously recycle the local task queue. Whenever a renderer dequeues a task, the task queue copies the task onto a backup queue. In dynamic mode, the workers delete their backup queues at the top of each

frame. In static mode, they swap the full backup queue with the empty assignment queue instead. This eliminates assignment messages, yet still allows some load balancing. The assignment starts in a balanced state because it is the result of a previous demand driven frame, but over time as the viewpoint and scene change the load will become unbalanced. When it does, two or more render threads within each node can cooperate to complete the static set of tasks assigned to the node.

When the tiles are small and the bottleneck is network latency, static assignments can nearly double the frame rate. Static assignment can also improve memory coherence. Because a node only processes $1/N$'th of the image, it needs less of the scene data over a sequence of frames and can better use the memory hierarchy. Large tasks send many tiles with one transmission and reduce communication delay. Large tiles are analogous - it takes fewer large tiles to complete an image. Each parameter represents a tradeoff between lower communication and higher load imbalance. The user can control these throughout a session and, with processor statistics, find the best settings interactively. An algorithm that does this automatically is left for future research.

4.2 Analysis

We analyze the factors that influence the frame rate of the distributed ray tracer by performing a series of benchmarks. All tests were conducted on the same cluster that was described in section 3.3. Each test has one supervisor and uses at most 31 workers. The tests discard 10 initial frames and record the elapsed time for the next 100 frames. Discarding the first ten frames ignores startup cost to more accurately demonstrate the performance during an interactive session. We repeat each test three times and take the average.

4.2.1 Network Performance

The speed of the interconnection fabric on a parallel machine is a critical determinant of performance. Network speed has two aspects. Latency is a measure

of the minimum time that it takes to start a transfer. Bandwidth is a measure of the amount of data per unit time that the channel can carry.

To measure the bandwidth, we use a single worker and let the tile size equal the image resolution. To eliminate computation time from the result, we restrict the worker to return blank results immediately instead of tracing rays. With these conditions, each frame requires three data transfers: a FrameState update, a task assignment, and a pixel result reply. The FrameState, task assignment, and result header together take 196 bytes, and the pixel results take three bytes per pixel. The first two transfers are short so as the resolution increases the frame time becomes dominated by bandwidth. We determine the bandwidth by dividing the number of bytes per frame by the measured time per frame. Table 4.1 shows that, in the limit, it takes 12 ns to transfer each byte, which is 62% of the ideal for Gigabit Ethernet.

Network latency adds transmission delay to every task assignment and result. In section 3.3.1, the round trip transfer time on the cluster using TCP sockets was measured in a ping-pong test to be 32 μs. The distributed ray tracer does better than that by using Legion to decouple sends from receives. When the

Table 4.1. Network Bandwidth Measurement.

Resolution [pix]	Frame Rate [f/s]	Bytes	1/Bandwidth [ns/B]	Efficiency [% Gbit]
64x64	3107.	12484	25.78	28.90
128x128	1343.	49348	15.09	49.38
192x192	601.2	110788	15.01	49.63
200x200	626.0	120196	13.29	56.06
256x256	386.6	196804	13.14	56.68
384x384	168.6	442564	13.40	55.61
400x400	161.6	480196	12.89	57.81
512x512	101.6	786628	12.51	59.57
600x600	75.83	1080196	12.21	61.03
700x700	56.00	1470196	12.15	61.35
768x768	46.91	1769668	12.05	61.85
800x800	42.51	1920196	12.25	60.82
1000x1000	27.33	3000196	12.20	61.08
1024x1024	25.73	3145924	12.35	60.31

ping-pong test is rearranged so that 10 or more sends are issued, followed by the same number of receives, the average transfer time drops to 19 μs. This order is faster because the test does not wait on the round trip transfer of each piece of information before sending the next piece. In this situation the network card can buffer several messages, which makes the network card and the application thread more independent. Because of buffering the transmission time and the application execution time can overlap somewhat

To measure the latency per tile, we modify the previous test. We use 62 rendering threads and many small tasks. As the tasks become smaller and more numerous, the transfer time becomes dominated by latency instead of bandwidth. The results given in Table 4.2 show that, in the limit, each tile takes at least 19 μs to transfer.

The network prevents the distributed ray tracer from scaling perfectly or indefinitely. The ray tracing computation time is trivially parallel and the extra overhead added with every additional node is minimal. Each frame the supervisor needs to send one extra FrameState update, which takes less than 32 μs. However, no matter how many workers are ray tracing, a frame will finish no faster than the

Table 4.2. Network Latency Measurement.

Tile Size [pix]	Tiles [Tiles/f]	Frame Rate [f/s]	Tile Time [μs/tile]
32x32	256	73.56	53.11
28x28	361	67.91	40.79
24x24	484	55.89	36.97
20x20	676	50.19	29.47
16x16	1024	34.41	28.38
12x12	1849	20.08	26.93
8x8	4096	9.421	25.91
4x4	16384	2.638	23.14
2x2	65536	0.7224	21.12
1x1	262144	0.2013	18.95

supervisor can assign and receive all of the tasks. This time can be estimated using equation 4.1:

$$frame\_time \geq 19\mu s \left(\left\lceil \frac{res}{tile} \right\rceil \right)^2 + 12ns * 3 * res^2 \tag{4.1}$$

To demonstrate, we benchmark an empty scene. We graph the measured frame rate and the maximum frame rate predicted by equation 4.1. Figure 4.4 shows the results. Note that the predicted maximums are consistently low, possibly indicating that the bandwidth measurement was too low. The analysis assumes dynamic assignment with fine granularity where every task contains a single tile.

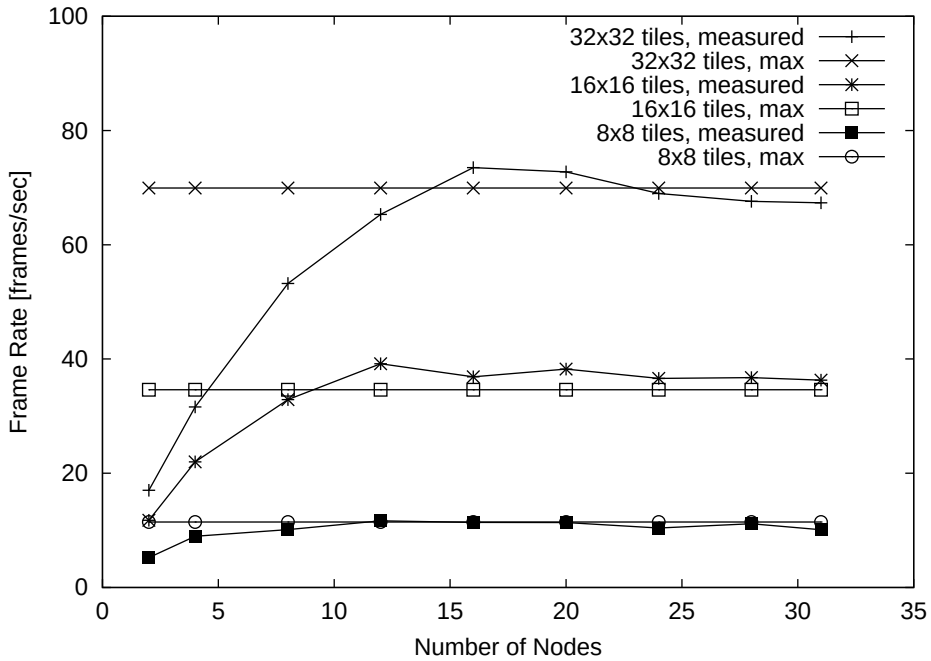

Figure 4.4. Network Limitation on Maximum Frame Rate.

4.2.2 Compute Power

Often the frame rate is compute bound rather than network bound. The compute time depends generally upon rendering options, image complexity, and scene complexity. The scene complexity is the primary determinant, and it is a function of the number of primitives in the scene.

We use a forest scene to analyze the performance of the ray tracer versus the number of primitives. Figure 4.5 shows the scene with one million polygons from the three tested vantage points. To isolate the primary ray intersection calculation, all surfaces use Phong lit color without shadows. The test was executed on 62 render threads with 16x16 tiles in a 512x512 image.

Figure 4.6 shows the frame rate for three views of the forest scene as the number of polygons is varied. The graph shows that the distributed renderer on this cluster will ray trace scenes with two million polygons at approximately two frames per second. With fewer primitives, the frame rate approaches the network limit of 35 frames per second. Above two million primitives, the frame rate drops precipitously as the data and acceleration structure exceed the installed physical memory and begin to swap from disk.

To first approximation, the network time is indivisible and the computation time is completely divisible. Amdahl's law explains the performance of the ray tracer well. In equation 4.2, the frame time is the sum of the network time and the ray tracing time. The ray tracing time is the time that a single render thread takes to ray trace the scene, divided by the number of render threads. To demonstrate, we test the tree scene again from the top view, with 250 thousand polygons, varying the number of workers. Figure 4.7 plots the frame rate as the number of processors is increased, the ideal frame rate under perfect scaling, and the frame rate predicted by equation 4.2:

$$frame\_time \approx 19\mu s \left(\left\lceil \frac{res}{tile} \right\rceil \right)^2 + 12ns * 3 * res^2 + \frac{one\_node\_time}{\#nodes} \qquad (4.2)$$

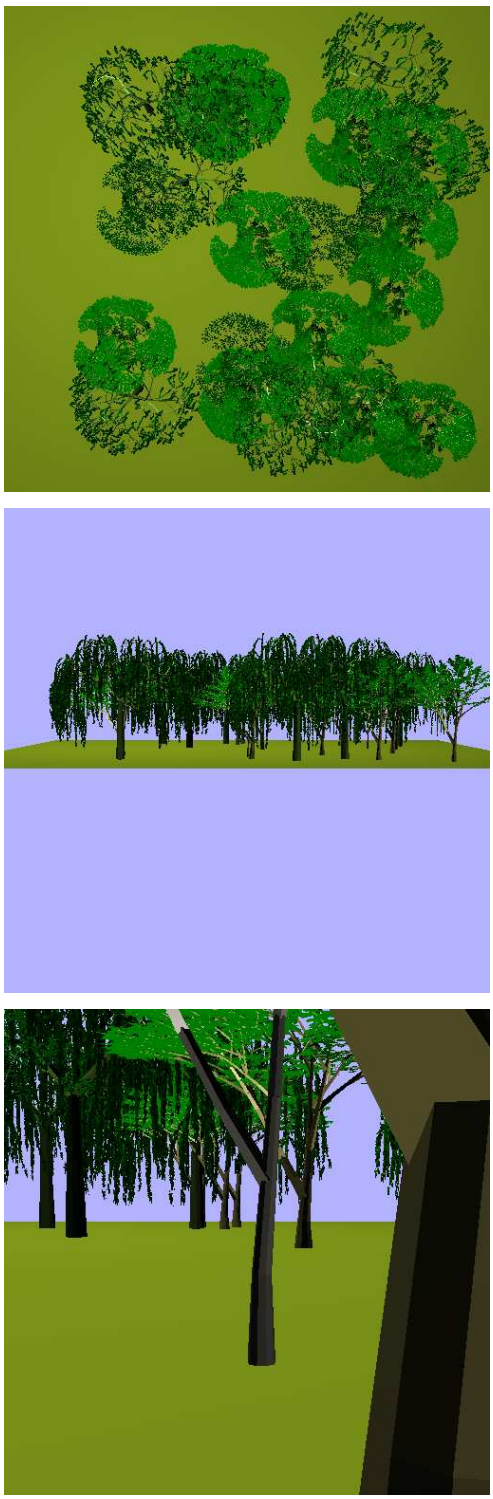

Figure 4.5. 1 Million-Polygon Forest - Top, Edge and Inner Views.

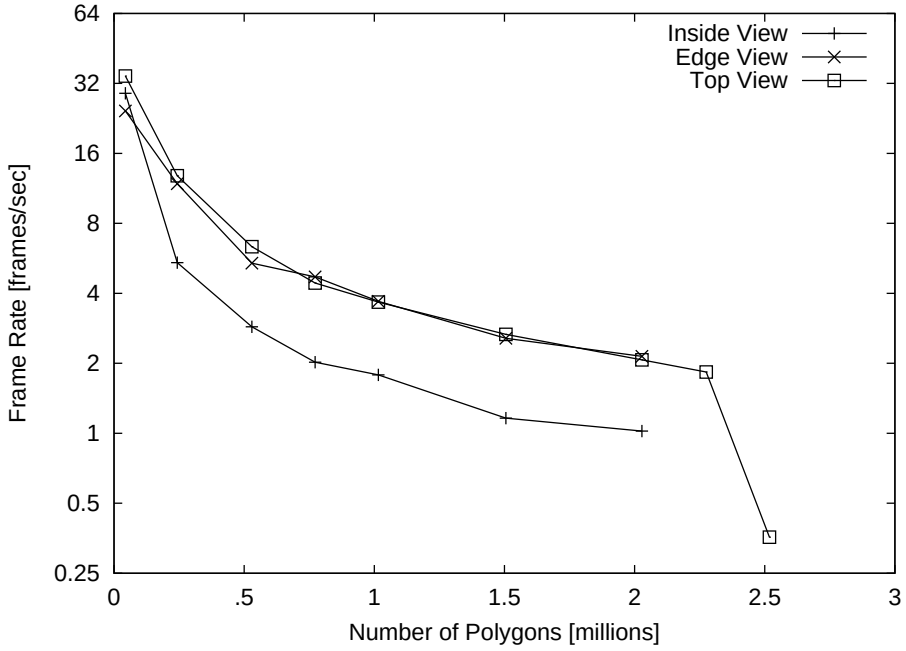

Figure 4.6. Frame Rate verse Polygon Count.

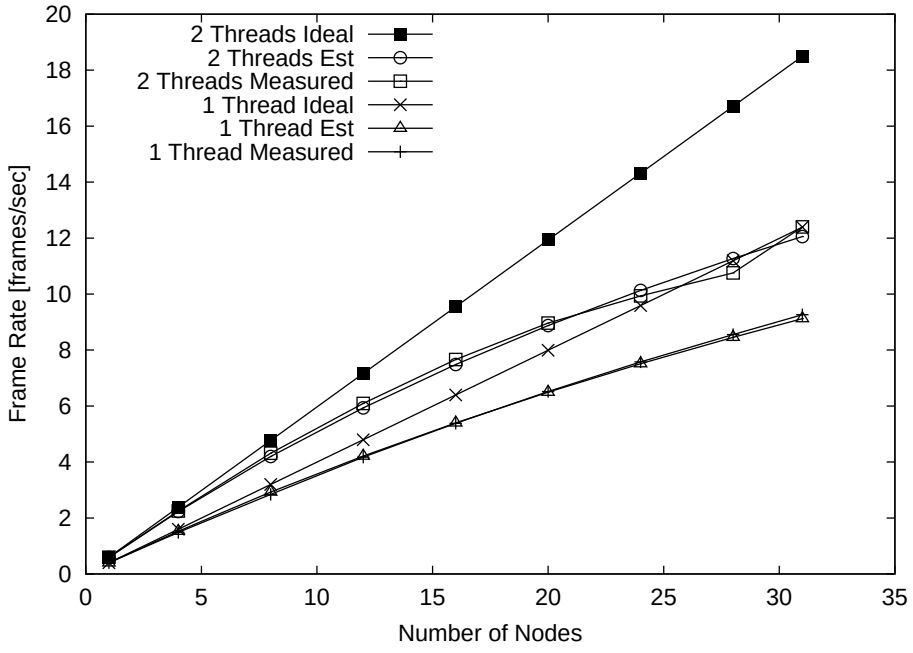

Figure 4.7. Scaling when Computationally Bound.

4.2.3 Communication and Computation Comparison

To bring the discussion of computation and communication into focus, we measure the time spent by the program performing computation and communication duties. We use two versions of a sphereflake test scene to show how the scene complexity influences the compute/communicate ratio. The first version has 98 primitives in the scene, the second has 7388. Figure 4.8 shows the more complex sphereflake scene. Both tests use a bounding volume hierarchy, render with hard shadows, and produce a 512x512 image. Sixty-two render threads render 16x16 pixel tiles.

With 98 primitives each frame takes 56.05 ms to render, yielding nearly 18 frames per second. Each render thread spends 20.26 ms per frame waiting for assignments, and then takes 37.78 ms to ray trace those assignments. With this scene, control overhead takes more than one-third of the total rendering time. Of

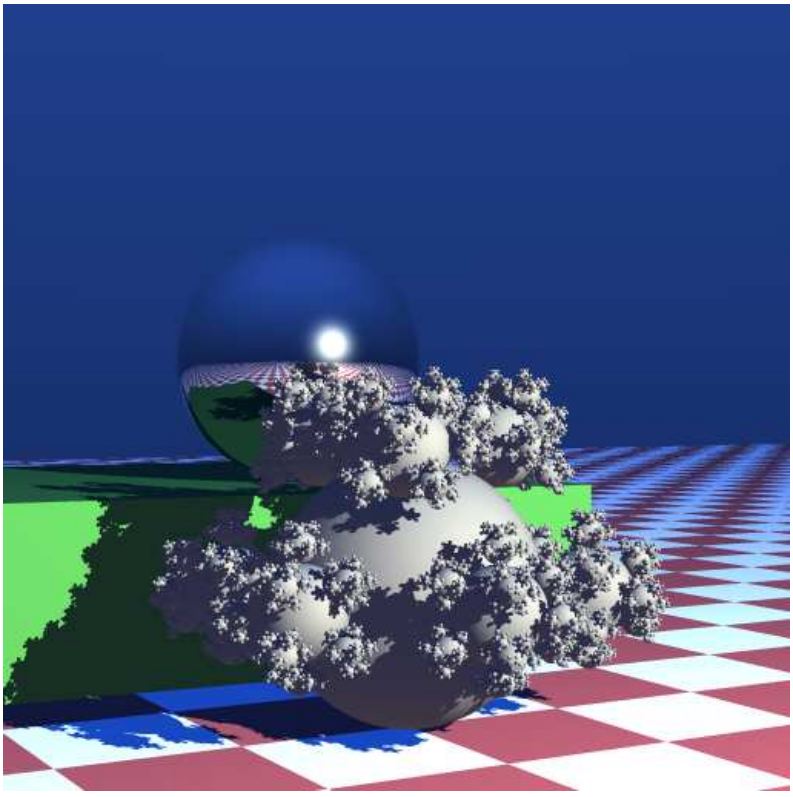

Figure 4.8. Sphere Flake Test Scene with Hard Shadows and 7388 primitives.

the control overhead time, only 285.4 μs is spent in the socket calls. The rest of the wait time is spent waiting for the supervisor to refill the local task queue. Meanwhile, the supervisor spends 32.5 ms transferring tasks and task results, and spends 23.6 ms waiting for results With a trivial scene then, communication and control overhead consume 36.15% or more of the total render time.

With 7388 primitives, each frame takes 130.3 ms to render, yielding nearly 8 frames per second. With this more complex scene, the render thread control and communication time is nearly the same, at 26.58 ms per frame, but the ray tracing computation time is much larger at 113.0 ms per frame. The supervisor still spends 32.5 ms transfering tasks and task results, but now has to wait for 103.1 ms waiting for results. With this mildly complex scene computation consumes 86.72% of the render time.

Although the control overhead is a possible target for further optimization, the ray tracing computation time is the more important bottleneck for non trivial scenes. The control overhead may be reduced with more advanced scheduling algorithms that do a better job of keeping the workers busy, or by widening the single supervisor bottleneck. Computation time may be reduced with hyperthreading and vector parallel instructions, and with the use of more highly optimized acceleration structures.

4.2.4 Architecture Comparison

A major goal of this work was to evaluate whether a cluster could be competitive with a shared memory supercomputer for interactive rendering. SGI's Non-Uniform Memory Architecture (NUMA) architecture is a high performance interconnection architecture that is 12 to 53 times faster than switched Gigabit Ethernet. The SGI uses a hypercube interconnection network with a link bandwidth of 1.6 GByte per second and 600 ns worst-case latency between compute and memory elements [32]. To evaluate how much this disparity effects the ray tracing process, we do side by side comparisons of the distributed interactive ray tracer and the original shared memory ray tracer.

*-Ray runs on a 32 processor SGI Origin 3000. Each processor is a 400 MHz R12K, and the entire machine has 16 GB of memory. The program was compiled as a 64-bit binary using the native CC compiler with -O3 compiler optimization. Both cases render identical 512x512 images of the sphere flake scene shown in Figure 4.8. Each node in the cluster uses a single render thread for this test. We test with 8x8 and 32x32 tiles to examine the network efficiency.

The results given in Figure 4.9 demonstrate that the supercomputer outperforms the cluster, but not by a large amount. *-Ray is largely insensitive to changes in the tile size. The SGI's low latency makes the additional small tile assignments essentially free. This allows the supercomputer implementation to scale almost indefinitely. *-Ray has been proven to scale up to 1024 processors. This should be weighted by the fact that both programs are interactive, and by the fact that in this test the cluster uses one thread per node.

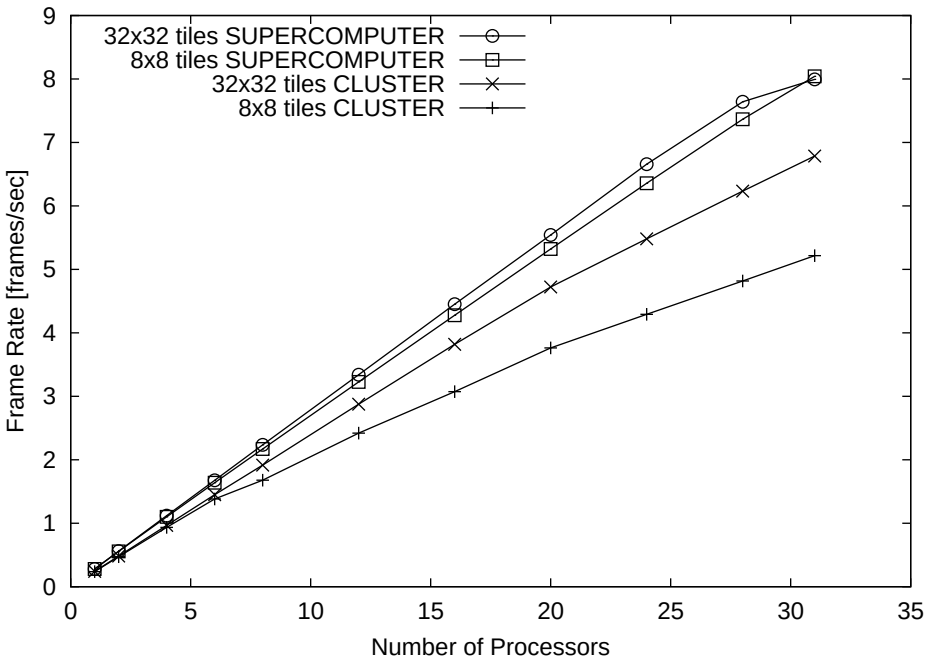

Figure 4.9. Sphere Flake Machine Comparison Results.

4.2.5 Memory

One domain in which the supercomputer has a clear advantage is rendering large data sets. The 32-bit x86 PC architecture limits the size of replicated scene databases to 4 GB. In our cluster each node has 1 GB of physical memory, and 1 GB of swap space. The program will not be able to create scenes that are larger than 2 GB, and scenes larger than 1 GB will render slowly because the operating system has to swap virtual memory pages between physical memory and the slow hard disk.

Static task assignment and instancing can extend the domain of interactivity. For the reasons explained in subsection 4.1.3, static assignment reduces the working set for each node. Each worker still has to create the whole data set, so the maximum data size is not increased, but the disk swap threshold is delayed.

Instancing creates copies of objects in space by applying geometric transformations to intersecting rays. The transformations can move, scale, rotate, and warp each of the copies. All of the copies refer to a single master copy of the object in memory. The acceleration structure still grows and consumes memory, but the leaf nodes take up only the size of a matrix and a pointer in memory.

To test these techniques, we test the tree scene again. We test each of the four possibilities obtained with two object types and two assignment strategies, varying the number of polygons for each. Figure 4.10 shows the recorded frame rate, applying both techniques to the tree scene. Neither of the regular object methods support more than two million polygons, but static assignment has a higher frame rate near the limit. Instances allow the renderer to draw four million polygons at 1.2 frames per second. Adding static assignment to the instances does not radically change the frame rate.

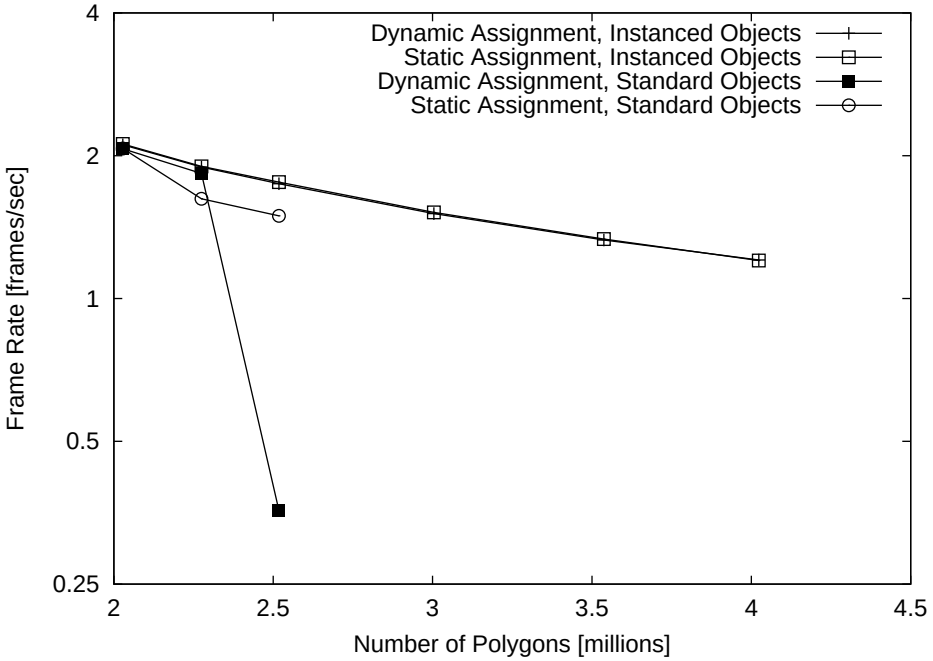

Figure 4.10. High Memory Strategies.

CHAPTER 5

DISTRIBUTED DATA VOLUME VISUALIZATION

5.1 Implementation

5.1.1 Distributed Volumes

With volumetric data, the distributed ray tracer can exceed the local memory space of the nodes in the cluster by using Legion's distributed shared memory. Each node permanently stores $1/(\#ofnodes)$ of the data in memory, and obtains remotely owned data over the network on demand. To keep the number of network transfers down, each node keeps a cache of remote data. Figure 5.1 depicts the structure of a worker using distributed data. In the diagram, heavy lines point out the new classes added to the distributed ray tracing program. The DISOVolume class produces isosurface renderings, and the DMIPVolume class produces MIP renderings. Internally both classes obtain data from the DBrickArray3 class, which wraps a 3D array interface around the DataServer.

When a render thread requires a brick of volume data, it must first acquire that brick from the DataServer. On an acquire, the DataServer checks if the brick is present locally, obtains it over the network from the responsible node if not, and then returns the address of the brick in local memory to the requesting thread. Once the thread finishes using the brick, it must release the brick so that the DataServer can reuse that cache space.

The ray tracer uses three level bricking (three-dimensional tiling) and a hierarchical grid acceleration structure for volumes. Bricking improves memory coherence and efficiently utilizes the memory architecture. The first level bricks the data onto 64 byte cache lines, the second bricks the data onto 4 KB virtual memory pages. The

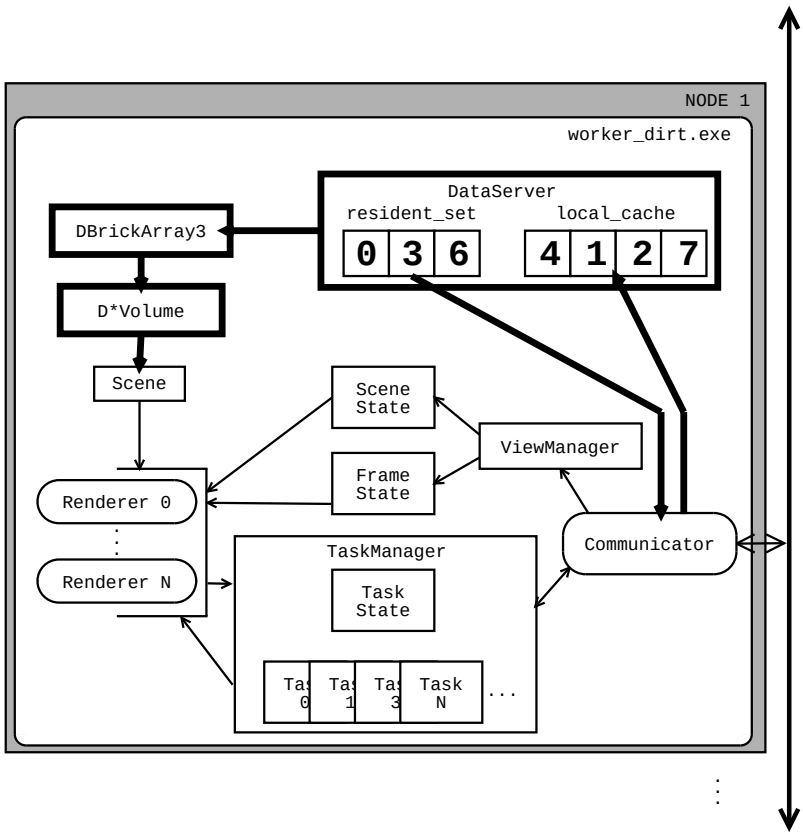

Figure 5.1. Worker Structure with Shared Data.

third level increases the data granularity for better bandwidth utilization. These largest bricks are the unit of sharing with distributed data, and one DataServer block corresponds to one-third level brick. In subsection 5.2.4, we analyze the effect of changing the granularity.

To use the DataServer, the ray tracing engine has to map data indices to brick numbers and offsets within bricks. Equation 5.1 gives the address of array element x,y,z in contiguous memory with three level bricking. Here N_y and N_z are the data y and z dimensions, n is the third root of the number of data elements per level one brick, and m and o are the same measures for level two and level three bricks respectively. As in *-Ray, we decompose the brick equation into expressions of x, y and z and tabulate to move the mod and divide operations into a preprocess step. This reduces the need for expensive divide mod operations during run time.

$$
\begin{aligned}
&address = \\
&(((x \div n) \div m) \div o)n^3m^3o^3(((N_z \div n) \div m) \div o)(((N_y \div n) \div m) \div o)+ \\
&(((y \div n) \div m) \div o)n^3m^3o^3(((N_z \div n) \div m) \div o)+ \\
&(((z \div n) \div m) \div o)n^3m^3o^3+ \\
&(((x \div n) \div m) \bmod o)n^3m^3o^2+ \\
&(((y \div n) \div m) \bmod o)n^3m^3o+ \\
&(((z \div n) \div m) \bmod o)n^3m^3+ \\
&((x \div n) \bmod m)n^3m^2+ \\
&((y \div n) \bmod m)n^3m+ \\
&((z \div n) \bmod m)n^3+ \\
&(x \bmod n)n^2+ \\
&(y \bmod n)n+ \\
&(z \bmod n)
\end{aligned}
\tag{5.1}
$$

The DBrickArray3 class contains two sets of tables to map each point in data space to both a block number and an offset. The block and offset tables are generated from equations 5.2 and 5.3. The block number comes from the first three terms of equation 5.1 and the offset comes from the rest. The DataServer manages the global starting address for each block, so we omit the $n^3m^3o^3$ terms.

$$
\begin{aligned}
&block = \\
&(((x \div n) \div m) \div o)(((N_z \div n) \div m) \div o)(((N_y \div n) \div m) \div o) + \\
&(((y \div n) \div m) \div o)(((N_z \div n) \div m) \div o) \\
&(((z \div n) \div m) \div o)
\end{aligned}
\tag{5.2}
$$

$$
\begin{aligned}
&\mathit{offset} = \\
&(((x \div n) \div m) \bmod o)n^3m^3o^2 + \\
&((x \div n) \bmod m)n^3m^2 + \\
&(x \bmod n)n^2 + \\
&(((y \div n) \div m) \bmod o)n^3m^3o + \\
&((y \div n) \bmod m)n^3m + \\
&(y \bmod n)n + \\
&(((z \div n) \div m) \bmod o)n^3m^3 + \\
&((z \div n) \bmod m)n^3 + \\
&(z \bmod n)
\end{aligned}
\tag{5.3}
$$

The Volume classes use a hierarchical grid acceleration structure to eliminate unnecessary data references. Figure 5.2 illustrates the acceleration structure. Each level of the hierarchy is a grid of "macrocells" that list the minimum and maximum data values from the cells contained in the lower levels of the hierarchy. At the bottom level, each macrocell represents approximately 8^3 voxels. At the top level, a single macrocell contains the minimum and maximum values for the entire volume.

Rays traverse the hierarchy first, using a recursive, incremental traversal algorithm. The ray examines the macrocells to find large sections of the volume to skip, reducing the number of accesses to slow distributed shared memory. For isosurface renderings, the ray finds data cells which contain the isosurface and then solves a cubic equation to test for intersection of the ray with the isosurface within. For MIP renderings, a priority queue of potential maximum regions organizes the best candidates. The ray only performs intersection tests on those cells that have the potential to contain the maximum value.

The program accesses the macrocell hierarchy constantly. Fortunately, the macrocell hierarchy grows logarithmically with the size of the data. For a 4 GB

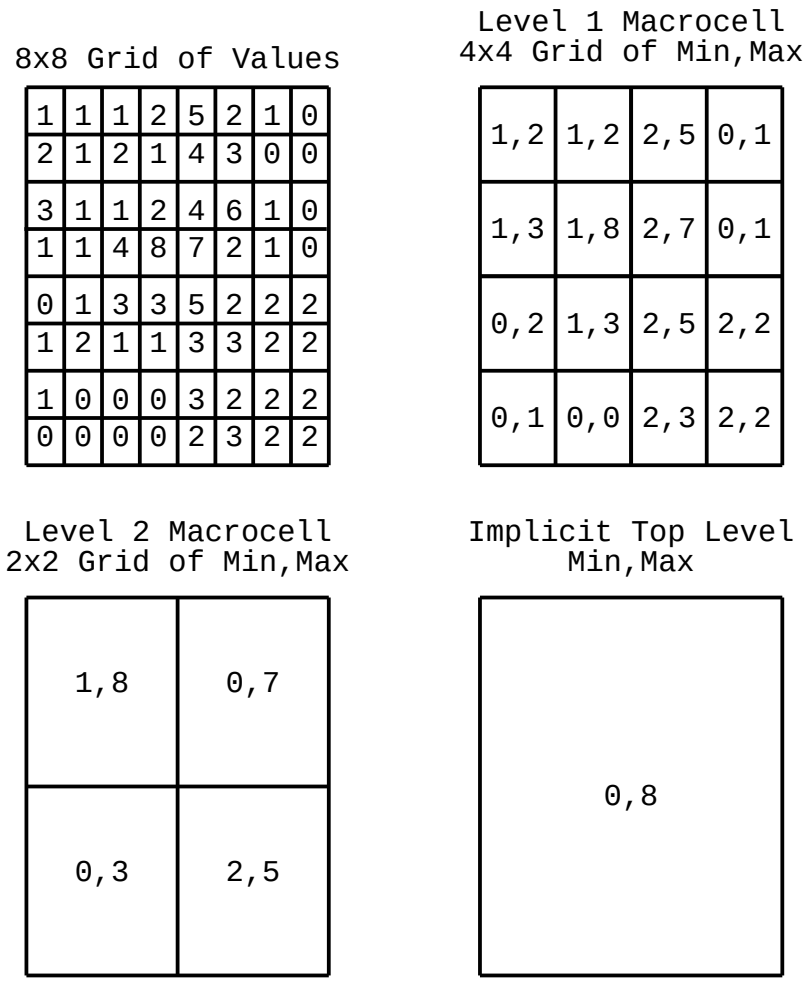

Figure 5.2. 2D Example with 2 Level Acceleration Structure.

data set, a three level hierarchy is 4.7 MB, and a four level hierarchy is 39 MB. Because it is small, we can replicate the structure and keep it in local memory.

Because access to the distributed shared, memory is slow compared with local memory access, we need to reduce the number of accesses further to achieve interactivity. To do so, we consolidate accesses to data at the bottom level of the acceleration structure. When a ray enters a bottom level macrocell, it performs a pretraversal to make a list of required bricks. The ray then acquires each touched brick in turn, copying out all intersected voxels before moving on to the next brick. Figure 5.3 illustrates a 2D example. The ray first determines that it needs the data

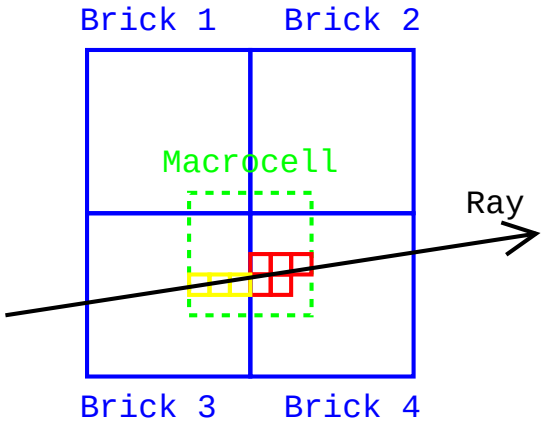

Figure 5.3. Consolidating Access to Shared Memory.

within the green macrocell. Inside the macrocell, it performs one acquire on brick three to get all of the yellow voxels, and then performs one acquire on brick four to get all of the red voxels. The naive approach would acquire 32 times, getting each voxel corner in turn, and take nearly 32 times as long.

Figure 5.4 shows a diagnostic image of a 2 GB synthetic shell volume. Differences in brightness show the tile assignments to the renderer threads in the cluster. Sixty-two shades show the tiles drawn by sixty-two render threads. Differences in hue show the data assignment within the volume. Thirty-one colors show the data bricks owned by thirty-one workers.

5.1.2 Off-Line Preprocessing

Volumes have to be preprocessed before rendering. We use an offline program to brick the volume, to create the acceleration structure, and to create a histogram for the data. The program produces three auxiliary data files from a raw data file. The file with the filename extension .brick_X, contains the volume data bricked with X^3 level two bricks per level three brick. The file with the filename extension .mcell_Y, contains the Y level deep macrocell hierarchy. The file with the filename extension .hist_Z, contains a histogram of the data with Z bins.

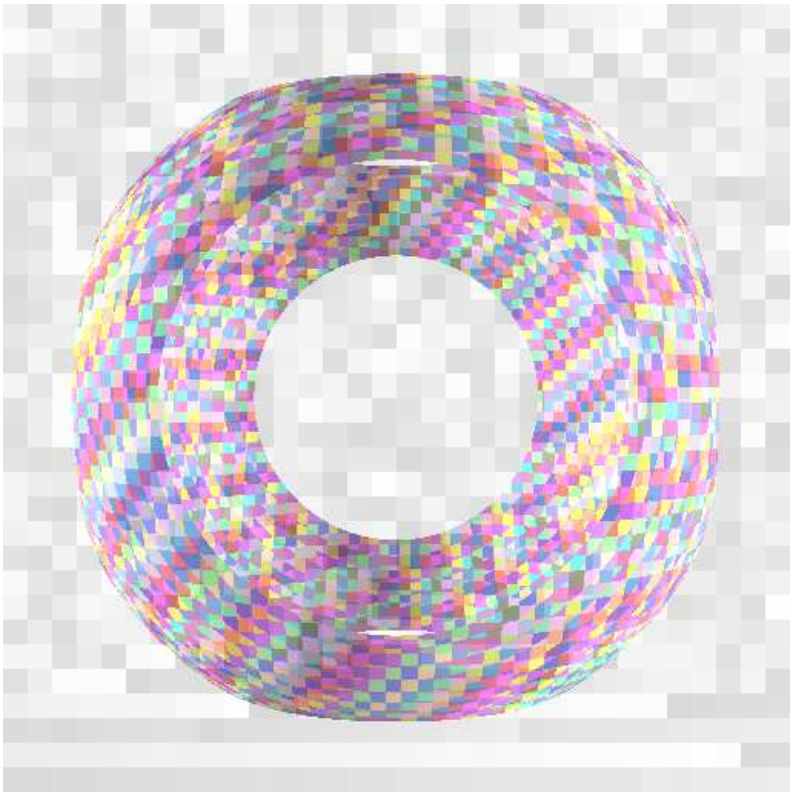

Figure 5.4. Diagnostic View Showing Data and Image Distribution.

The preprocessor is also a distributed parallel program with a supervisor worker organization. It has six stages, each separated by a distributed barrier. Synchronization between stages prevents quick workers from obtaining incomplete data from slower workers.

- Stage 1 Startup - The supervisor spawns workers and gives them the raw data file name, the data type, the depth of the macrocell hierarchy and the level three brick size.

- Stage 2 Brick and Histogram - The workers create an empty distributed memory space and read their own data values into it. Each file read takes in a 2D slice of data. The workers find the bricked location for each value, but only copy their own values into their own resident sets. The first worker also bins the data values into the histogram file in this stage.

- Stage 3 Write Brick File - The first worker obtains each brick over shared memory in turn to create the brick file.

- Stage 4 Create Macrocells - Each worker processes a different set of top-level macrocells. They recursively obtain minimum and maximum values from the lower levels of the hierarchy to complete each macrocell. At the bottom level, a method in the DBrickArray3 class finds the minimum and maximum values in an arbitrarily sized cube of data. Like the ray consolidation process, the min/max finder consolidates shared memory accesses for faster processing.

- Stage 5 Compile Macrocell Hierarchy - The workers send their completed macrocells to the first worker. The first worker receives macrocells directly into its own partial macrocell hierarchy.

- Stage 6 Write Macrocell File - The first worker saves the completed macrocell hierarchy to disk.

Table 5.1 gives the measured time for 31 workers to preprocess a synthetic shell data set. In the table, the brick column contains the elapsed time for stages 1 through 3 and macrocell column contains the elapsed time for stages 4 through 6. The data type is unsigned short, with a four level hierarchy and 27 KB bricks. The first two columns show the size of the volume and the macrocell file.

Table 5.1. Off-Line Volume Preprocessing.

Dimension [voxels]	Data Size [GB]	Accel Size [MB]	Bricks [sec]	MCell [sec]
512^3	.25	7.51	22.5	16.3
646^3	.50	13.0	48.0	26.8
813^3	1.0	12.9	118	37.0
1024^3	2.0	22.4	359	76.4
1291^3	4.0	38.6	559	124

Preprocessing could be accelerated further by reading larger slabs of raw data, by using multiple threads during macrocell creation, and by parallelizing the histogram operation.

Because the distributed program loads the bricked data in parallel, it can load large volumes faster than *-Ray does. Each worker only needs to load a fraction of the bricks, and the parallel file system is less of a bottleneck. The distributed program loads an 8 GB volume in one minute, and *-Ray loads the same volume in nine minutes.

5.2 Analysis

Several factors influence the performance of the renderer with distributed data. In this section, we investigate the following factors:

- consolidating accesses to the DataServer,
- changing the amount of memory allocated for shared data on each node,
- using static task assignment to increase data coherence,
- changing the granularity of data sharing,
- using application controlled demand paging instead of distributed shared memory, and
- increasing associativity in the local cache.

It is also important to examine how the renderer performs over time, what the extra communication costs, and how well the program scales. The first point involves looking at the dynamic behavior of the renderer as viewpoint and isovalue changes cause variation in the volume of data traffic. The second point involves using detailed diagnostics to measure how much time the program spends communicating and computing respectively. The third point involves examining how the system performs as more processors are applied.

To examine these issues, we do a series of tests. All tests were conducted on the same cluster that was described in section 3.3. Unless otherwise noted, the given results are for isosurface rendering, and 31 worker nodes, each having two render threads, are used. Shading is single sided without shadows. The image resolution is 512x512 pixels, and the images are composed from 16x16 pixel tiles, which are dynamically assigned.

We test with four synthetic volumes, and seven time steps from a scientific data set. Figure 5.5 shows the synthetic volumes. In the shell volume, every point contains the distance to the center of the volume. The isosurfaces are all concentric shells. The multishell volume consists of 200 randomly sized and placed shells, with perturbed values that make irregular surfaces. The Perlin volume is a volume defined by the Perlin noise [28] function with 10 harmonics. The white noise volume consists of completely random values. The white noise volume looks like a solid

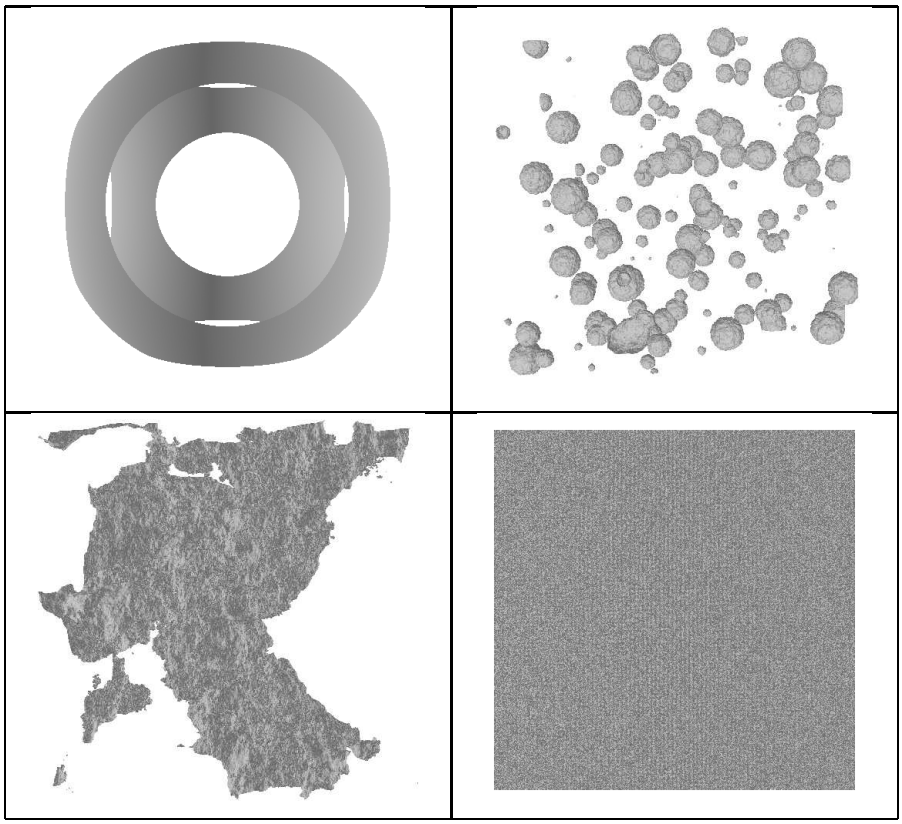

Figure 5.5. Synthetic Volumes: Shell, Multi Shells, Perlin Noise, and White Noise.

cube from a distance because it contains a huge number of samples. The synthetic volumes are composed of 16-bit unsigned integers and are 1024^3 voxels, or 2 GB, unless otherwise noted. Tests on the synthetic volumes measure the time for the first 100 frames of a static session.

Figure 5.6 shows time steps generated in a Richtmyer-Meshkov instability simulation. This Computational Fluid Dynamics (CFD) data set was produced at Lawrence Livermore National Labs [21]. The volume is composed of unsigned 8-bit integers, and each time step is 1920x2048x2048 voxels or 7.5 GB. The tests on the CFD data benchmark a recorded interactive session with frequent isovalue and viewpoint changes.

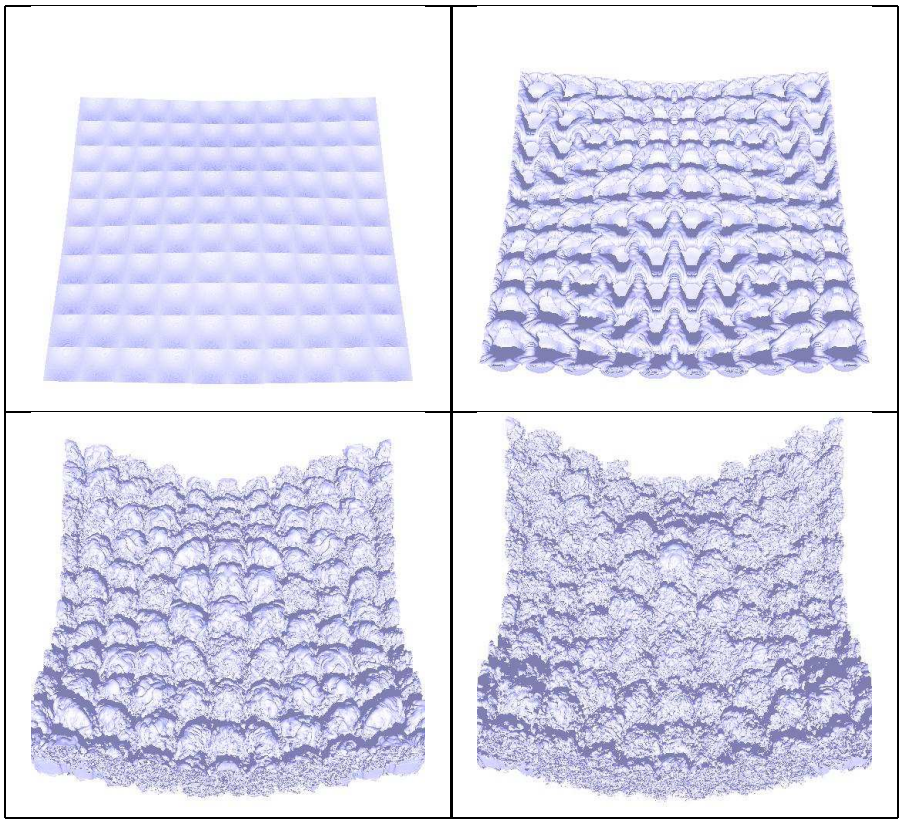

Figure 5.6. Richtmyer-Meshkov Time Steps 0,45,180,270.

5.2.1 Consolidating Access

Because distributed memory access is slow compared local memory access it is important to reduce the total number of memory accesses. We have experimented with three cell intersection routines for MIP and isosurface rendering. The first routine acquires and releases the eight corners of each cell independently. The second routine sorts the eight corners so that it does not need to acquire the same block more than once in one cell. If a cell straddles a brick corner, it will need eight acquires anyway and the sorting adds overhead. Cells on brick sides and edges are analogous. The vast majority of cells are interior to the bricks, so total access time is close to one eighth that of independent access. The third routine sorts all of the accesses within the first level macrocells. Table 5.2 shows the average number of accesses per worker per frame and the average frame rate using three access patterns for the CFD data set. Table 5.3 lists the measured frame rate using the three approaches for MIP and isosurfaces renderings of the 2 GB synthetic Perlin volume. In each test, the frame rate increase is inversely proportional to the decrease in the number of accesses, minus the overhead of the brick sorting algorithm. Consolidation does not reduce the number of misses, so it is most effective when hits dominate the render time.

5.2.2 Local Memory Allotment

During initialization, the DataServers allocate local memory to use for shared data. The DIRT_MAXDSMMEM environment variable controls the amount of local memory. With N nodes, each node must allocate at least $volumesize/N$ bytes. The

Table 5.2. Consolidated Access to Shared Memory on the CFD Data Set.

Pattern	Accesses	Frame Rate [f/s]
Access 1	3279000	.1149
Access 8	453400	.7090
Access Many	53290	1.686

Table 5.3. Consolidated Access to Shared Memory on the Perlin Volume.

Render Method	Pattern	Accesses	Frame Rate [f/s]
MIP	Access 1	1989000	.741
ISO		452700	2.30
MIP	Access 8	238500	2.15
ISO		64370	5.98
MIP	Access many	89430	2.72
ISO		15320	6.77

nodes can allocate no more than the sum of the installed physical memory and the configured swap space, minus the footprint of all active processes. In our cluster, each node has 1 GB of physical memory and 1 GB of swap space, and in one trial 1800 MB was the upper limit.

With too little memory, the local cache cannot hold the working set and the cache will thrash. Bricks compete for space in the cache and cause frequent misses. To find out how much of a volume needs to be cached, we vary the DIRT_MAXDSMMEM parameter and measure the frame rate for the four synthetic volumes. Table 5.4 contains detailed diagnostics for the multishell volume. Figure 5.7 graphs the results of all four synthetic volumes.

The interactivity is poor with restricted memory because the miss penalty dominates the frame time. Initially, the frame rate increases as the allocated memory increases. With memory at roughly 12% of the volume size, hits dominate

Table 5.4. Local Memory Allotment Detail for the MultiShell Volume.

Allocation [MB]	Local Cache [MB]	Hit Rate [%]	Frame Rate [f/s]
70.50	4.435	86.89	.9635
74.00	7.935	96.02	2.710
112.0	45.94	97.65	4.137
512.0	445.9	99.21	7.658
1536	1470	99.43	8.446

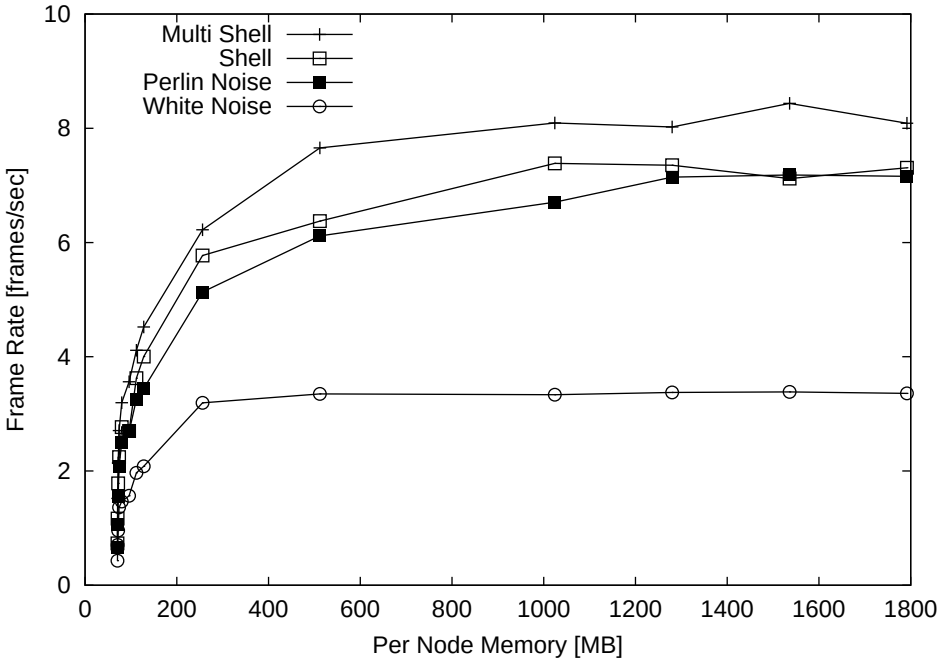

Figure 5.7. Local Memory Allotment and Frame Rate.

the frame time, and the frame rate increase levels out. Note that even above the installed physical memory of 1 GB the interactivity still improves slightly. Here, virtual memory allows each node to keep a larger number of blocks locally, and only the active ones need take up space in physical memory.

5.2.3 Static Assignment

With dynamic scheduling, the tile assignments change frequently and each worker must render most of the image over time. With static scheduling, each worker renders the same set of tiles repeatedly. Data coherence is better with static assignment because each worker constantly accesses the same small portion of the data. The draw back is that only render threads on the same node are able to cooperate to render difficult portions of the image.

Table 5.5 compares dynamic and static task assignment for seven time steps from the Richtmyer-Meshkov data set. For the first time steps, the data coherence

Table 5.5. Dynamic verse Static Assignment for CFD Time Steps.

		Dynamic		Static	
Time Step	Accesses	Hit Rate [%]	Frame Rate [f/s]	Hit Rate [%]	Frame Rate [f/s]
0	22200	99.94	6.691	99.95	4.877
45	31270	99.83	4.224	99.92	3.683
90	38160	99.77	3.352	99.92	3.161
135	41940	99.68	2.759	99.90	2.659
180	51760	99.61	2.095	99.89	2.244
225	56090	99.56	1.784	99.88	2.101
270	53580	99.48	1.686	99.84	2.088

gains do not outweigh the load imbalance penalties. For the complex time steps, the extra hits compensate for the uneven workload.

5.2.4 Data Granularity

Changing the data granularity demonstrates another tradeoff. Small bricks take less time to transmit, waste less space by better conforming to a surface, and decrease competition for local cache space. On the other hand, large bricks better amortize the transfer initiation latency and tend to improve the hit rate by servicing multiple accesses with a single transfer. The best granularity balances the competing factors, and is likely to be data dependent. To test this we brick a 1 GB shell data set and the 7.5 GB Richtmyer-Meshkov data set and vary the level three brick selection to change the network transfer data granularity.

Table 5.6 shows the results. With both datasets, the hit rate and the miss penalty climb with increased granularity. In these tests, eight page bricks to one top-level brick is fastest.

5.2.5 Demand Paged Memory

We can use a DataServer_mmap class to render by demand paging data from disk rather than from the network. The mmap() system call requires page-aligned addresses. To use it, the distributed volume classes can round the level three brick

Table 5.6. Frame Rate and Data Granularity.

Level 3 Setting	Block Size [KB]	Hit Rate [%]	Miss Penalty [μs]	Frame Rate [f/s]
1 GB x 16-bit Shell				
1	3.38	98.71	390.7	11.69
2	27.0	99.60	1077	14.57
3	91.1	99.80	2905	13.37
4	216	99.88	6922	11.11
7.5 GB x 8-bit Richtmyer-Meshkov				
1	4	98.72	587.0	1.607
2	32	99.48	1411	1.686
3	108	99.64	3972	1.309

size up and pad to the nearest 4 KB[1]. Padding bloats the brick files and increases block transfer time so it is not normally used.

We rebrick the synthetic shell volume and compare the DataServer_direct class with the DataServer_mmap class. Table 5.7 shows that memory mapping is faster for small volumes but slower for large volumes. This agrees with the result of the random access DataServer comparison in subsection 3.3.2 and the explanation is the same; once the operating system is forced to page data to disk, network misses are faster. Memory mapping may be faster for small volumes, but replication is faster still.

[1]Brick starting addresses are always aligned, but bricks are not normally padded.

Table 5.7. Network and Demand Paged Memory Comparison.

Data Size [GB]	Network Frame Rate [f/s]	Memory Map Frame Rate [f/s]
.25	19.22	22.30
.5	17.13	21.94
1	14.59	20.28
2	12.18	5.030
4	9.729	1.780

5.2.6 Associativity

The direct mapped DataServer has the advantage of low overhead in comparison with the associative DataServer, but it has the potential to thrash when a worker needs several bricks that happen to align to the same slot. Associativity increases the hit rate by providing alternate placements for each fetched brick. It also increases the hit time because it needs to search several locations and it has to keep time stamps. Table 5.8 shows the measured hit rate, hit time, and frame rate for direct mapped and 4, 8, 12, and 16 way associative caches on the CFD data set. We find that even this large volume has a working set that fits within the cache without thrashing.

When the volume grows to the point where the working set is too large for the local memory on the nodes, thrashing becomes more of a problem. Associativity can increase the frame rate here by increasing the number of hits. To examine this possibility we test the multishell volume and vary the allocated memory per node. Table 5.9 lists the measured frame rate for five different associative set sizes and five locally allocated memory sizes.

The results show that with limited memory, associativity improves interactivity. With 512 MB per node, and a six way associative cache, only 4% of the hits come from the extra slots. The associative slots are used more heavily when the local memory space is small. With 70.5 MB per node 81% of the hits come from the extra slots. In this case, fewer misses pay for the longer hit time.

Table 5.8. Associativity Test on the CFD Data.

DataServer	Hit Rate [%]	T Hit [μs]	Frame Rate [f/s]
Direct	99.48	6.857	1.686
Assoc, 4x	99.60	11.55	1.521
Assoc, 8x	99.63	13.17	1.443
Assoc, 12x	99.63	14.01	1.361
Assoc, 16x	99.65	15.48	1.286

Table 5.9. Frame Rate vs. Associativity and Allocated Memory.

DataServer	Allocated Memory[MB]				
	70.5	74	112	512	1536
Direct	.9677	2.764	4.216	7.656	8.492
Assoc, 1x	.9476	2.701	4.087	7.203	7.958
Assoc, 2x	1.488	3.232	4.248	7.982	8.161
Assoc, 4x	1.562	3.294	4.146	8.062	8.050
Assoc, 6x	1.598	3.223	4.047	8.147	8.173

5.2.7 Dynamic Behavior

The frame rate of the renderer is inversely related to the amount of remote data fetched by the render threads. Measuring the frame rate and the data traffic makes the relationship clearer. Figure 5.8 shows the recorded data traffic and frame rate over an interactive session on the CFD data set using one render thread per node.

The bottom panel shows the rendered image at five frames during the session. The next to bottom panel reproduces the isovalue and viewpoint changes that occur over the session. The entire volume is in view for the first half of the test, and in the last half of the test the view is zoomed in on a small portion of the volume. The second panel from the top shows the amount of data that each node acquires over the network at each frame. With the entire volume in view, every change in viewpoint or isosurface causes the data traffic to spike and then decay. When only a small portion is in view, the data transfer is minimal.

The top panel contains two graphs, one showing the measured frame rate, and the other showing the frame rate that would occur if no time was spent transferring data. Note that in the measured frame rate plot, for the first half of the test, every spike in data traffic causes a sharp drop in the frame rate, followed by a gradual increase. Here, each change requires the DataServer to cache a different portion of the data, and obtaining the data over the network reduces the frame rate. Over several frames, the workers obtain the entire working set, and the frame rate returns to normal. When the view is zoomed in, the workers can easily cache the visible

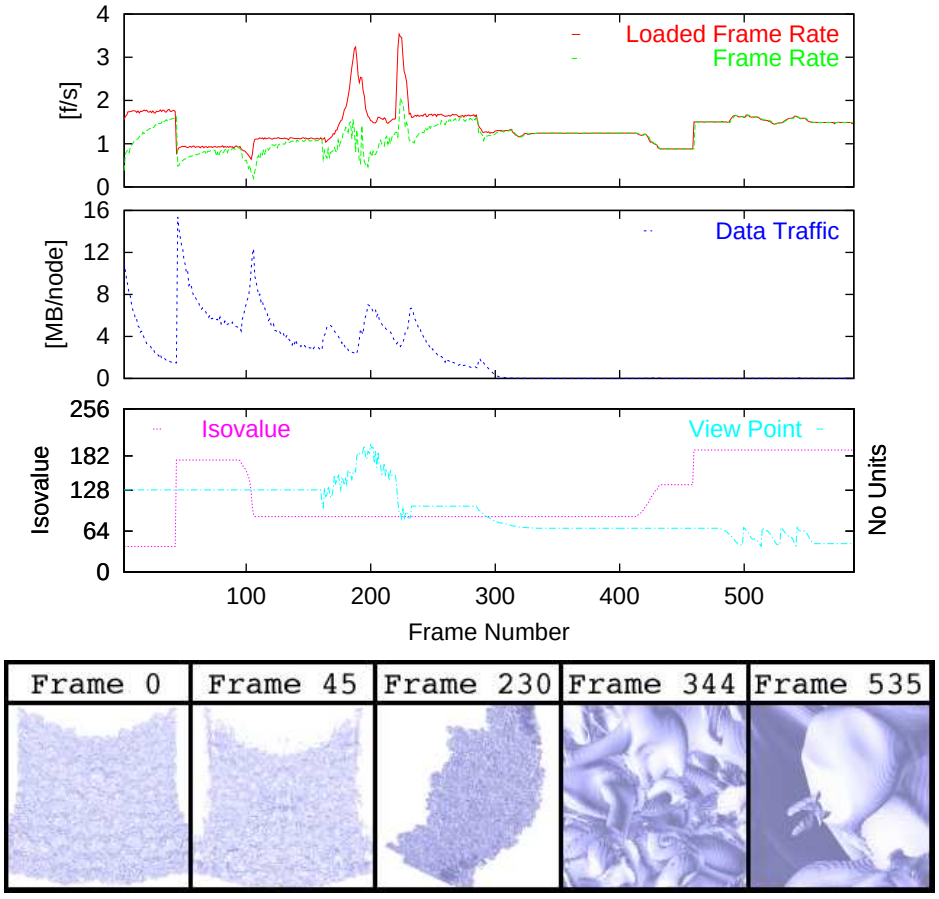

Figure 5.8. Data Traffic and Frame Rate.

data, and the speed of the rendering engine, including hit time to the DataServer, determines the frame rate, rather than the cache behavior.

The loaded frame rate graph was recorded by repeating each frame twenty times, and then plotting every twentieth frame. This plot demonstrates the rendering speed assuming a perfect cache. In this test the average frame rate is 1.392 frames per second with filled caches and 1.241 frames per second without. The dynamic behavior then depends upon the degree to which the caches are filled, which in turn depends heavily upon the rate and pattern of user interaction. In this test time spent obtaining data over the network slowed the renderer by 10.85%.

5.2.8 Scaling

When local memory contains the working set, network performance is not an issue, and the renderer exhibits the same scaling that it does with replicated data. Network latency and the number of pixel tasks determine the shape of the scaling curve. Hit times to distributed shared memory are slower than normal memory access so distributing the data slows the renderer down. On small volumes that can be replicated, we have measured that using the distributed shared memory slows the renderer down by as much as 60%. Therefore, it takes more processors to reach the frame rate limit. Figure 5.9 graphs the ray tracer's performance on the 2 GB multishell volume as the number of processors is increased.

When the working set is too large for local memory, adding workers can sometimes yield superlinear scaling. For instance, with an 8 GB data set, it takes at least eight cluster nodes to render the data, unless we take the performance hit of memory mapping from disk. Adding nodes divides the data into smaller pieces, which leaves more space for the local cache. Larger caches hold more of the working

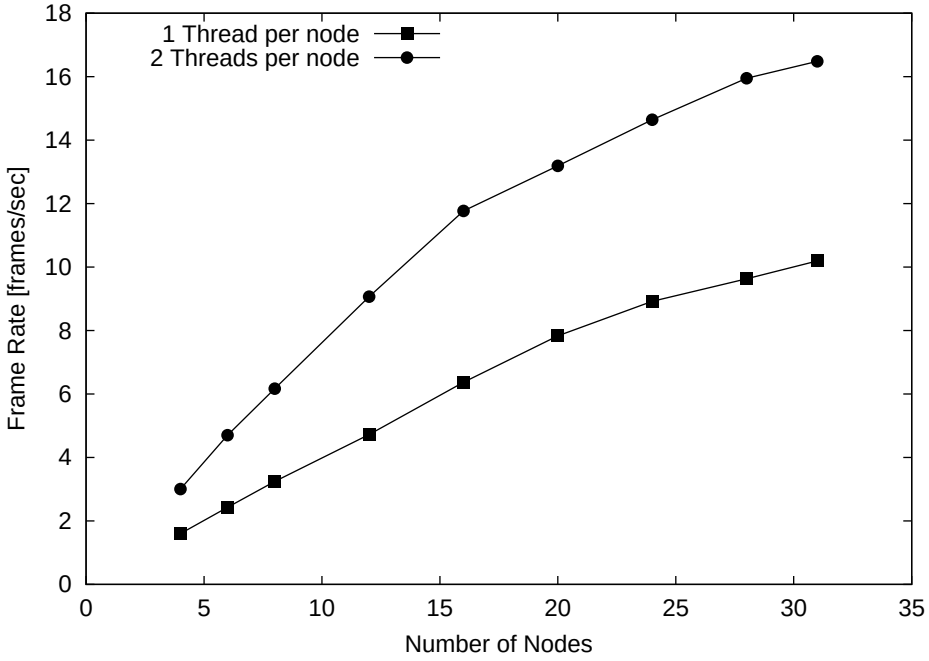

Figure 5.9. Scaling with Distributed Volume Data.

set, quickly increasing the hit rate and the frame rate. Once the local cache holds the working set, scaling returns to sublinearity.

In contrast to the beneficial effect of increasing the number of nodes, increasing the number of render threads per node increases contention for the data. With limited local memory, this can cause thrashing. Two threads per node can render a 2 GB shell volume using 70.5 MB of allocated memory at 0.88 frames per second. Because a single thread reduces contention, one thread per node will render the same scene at 1.4 frames per second. Contention is less of an issue with more memory, so the second thread normally increases the frame rate.

5.2.9 Communication and Computation Comparison

For a more concrete understanding of the impact of transmitting data over the network, we measure the time spent by the program performing computation and communication duties with a distributed volume. For this test we use time step 225 of the Richtmyer-Meshkov data. In this particular test each frame takes just under one second to render. The render threads spend on average 69.15 ms waiting for assignments from the supervisor, 920.6 ms ray tracing, and 68.21 ms communicating control data with the supervisor. In other words, 87% of the time is spent rendering, 6.5% of the time is spent communicating with the supervisor and roughly the same amount of time is spent waiting for work.

The ray tracing time is much higher than that spent in the sphereflake test of subsection 4.2.3 partially because of the extra time spent managing the distributed shared memory. In fact 46% of the rendering time is spent accessing the data, half of which is time spent in cache hits and half in cache misses. On average each thread spends 216.6 ms accessing cached data, and 210.6 ms fetching remote data. The miss time is dependent upon the network bandwidth, the round trip time and the service time on the remote block serving nodes. The hit time makes a particularly attractive choice for optimization because it is entirely independent of the network, and might be improved with small program modifications.

5.2.10 Architecture Comparison

Finally, we compare the distributed renderer to *-Ray on time step 270 of the Richtmyer-Meshkov data set. On the cluster, we select the parameters that were found to yield the highest interactivity for this volume: macrocell consolidated access to blocks, direct mapped caching, and 32 KB blocks. Although static assignment is faster on this data set, we use dynamic tile assignment on both machines and hold all additional render settings constant. Figure 5.10 shows the recorded frame rates on the cluster with one and two threads per node and on the SGI. The average frame rates over the session are 1.08, 1.69, and 4.69 frames per second for the single thread, dual thread and supercomputer tests, respectively. With equal number of processors then the cluster is roughly one quarter the speed of the SGI, and with two processors per node, the cluster is roughly one-third the speed.

More exactly, the relative performance varies with the degree to which the cluster can keep its caches filled. In the first half of the test, with the entire volume

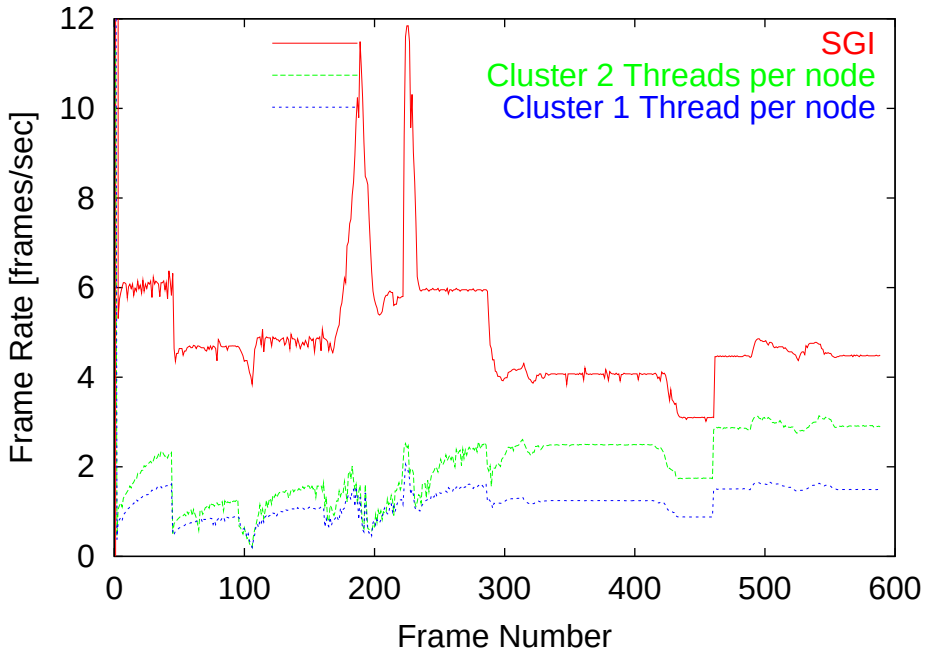

Figure 5.10. Cluster versus Supercomputer.

in view, the distributed renderer struggles to keep the cache loaded, and consistently pays the miss penalty. The effect is particularly deleterious around frames 200 and 250 where the viewpoint is quickly rotating around the volume. At these points the isosurface is seen on an edge, and most of the image is empty. The distributed caches are not able to fill quickly enough to take advantage of this situation. The SGI does not suffer from this problem. With the entire volume in view, the SGI averages 5.71 frames per second, and the cluster averages 1.46 frames per second with two threads per node. In this portion of the test, the cluster is closest to the SGI at frame 158, where it lags by 3 frames per second. It lags the most at frame 188, where it is 10 frames per second slower. When the entire volume is in view, cache loading makes the distributed renderer run at between one tenth and one third the speed of the SGI.

When the viewpoint is zoomed in on a small portion of the data, the frame rate is compute bound and the cluster program closely follows *-Ray. In the second half of the test, the SGI averages 4.17 frames per second, and the cluster averages 2.58 frames per second with two threads per node. The relative performance is more consistent with a focused view. Here the cluster stays between 1.89 and 1.27 frames per second slower than the SGI. When the caches are filled, the cluster runs consistently at 60% of the SGI's frame rate.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

We have shown that it is possible to visualize multi-Gigabyte volumetric datasets interactively using clusters. We use the ray tracing algorithm, which is well known for producing accurate, high quality images. The algorithm is inherently parallel, which makes it a natural choice for large parallel computer systems. The advantage gained in using a cluster is primarily an economic one. A cluster is significantly less expensive than an equivalent tightly coupled parallel machine, and so the new program should make interactive visualization of large datasets more available.

When programming for a cluster environment the immediate problem is to find a way to join the separate processes that make up the parallel program. Our program uses Legion, a minimalist communications library built on top of TCP. The library's capabilities are modular yet efficient; each feature is optional, and can be omitted to reduce overhead. In head-to-head tests Legion is somewhat slower than raw socket calls, faster than MPICH, and roughly equal to LAM-MPI. Legion supports multithreaded cluster applications that are parallel both within and between nodes. Asynchronous message handling in Legion helps to free computational threads from communication delays.

The ray tracing program was derived from *-Ray, which runs on shared memory parallel machines. To create the new program, we identified all of the dependencies between functional units in *-Ray, and used Legion to transmit them over the network. The distributed program is capable of interactively rendering scenes containing two million polygons at two frames per second by replicating the scene database in each node's memory. Tests show that it is not uncommon for 80% of the rendering time to be spent in computation. When the program is compute-bound

the frame rate is roughly equal to that of *-Ray. For less demanding scenes the network performance becomes the bottleneck, and the eventual frame rate limitation depends upon the number of tiles in the image. A shared memory supercomputer does not suffer from the same limitation, but with tens of processing elements instead of hundreds, both programs produce similar frame rates. We have observed sustained frame rates of up to 60 frames per second, on 512x512 images, but more typically obtain between 1 and 20 frames per second.

Scientific data sets are growing with the ability of computers to generate them. Cluster-based visualization of large volumes is problematic for three reasons. First, 32-bit PC's cannot access more than 4 Gigabytes of memory, so some large scientific data sets can not be replicated onto local memory. Second, processors have no automatic means to access the memory within remote nodes. Third, the network connection between nodes is significantly slower than memory access within a node. We use Legion's distributed shared memory to make the aggregate memory space available to every rendering thread. This is an object-based DSM, which makes the 32-bit address limit irrelevant. The largest database that we have tested is an 8 Gigabyte data set which renders at between 0.5 and 8 frames per second. Caching is the key to overcoming the slower network access time. By eliminating unnecessary and redundant accesses to the data, we obtain hit rates of greater than 95% most of the time. In the final comparison, the cluster renders multi-Gigabyte data sets at roughly one third the rate of a supercomputer that costs ten times as much.

The cluster-based renderer does have important disadvantages. The network determined frame rate limitations have been discussed in detail. Another disadvantage is that the supercomputer's shared memory hardware makes it easier to modify the program. For instance, in the new program auxiliary displays, which let the user control particular objects within the scene, need to have their state changes encoded into the FrameState structure. In the original program the user interface classes and the object classes could simply refer to each through a pointer. The same is true of any changes to the user interface of the program as a whole.

The user directly interacts with the supervisor node and the mechanism by which the supervisor interacts with the workers has to be programmed. Another example is that our object-based DSM requires that shared data items are indexed, and that accesses to the data have to be guarded by acquire and release operations. These access semantics make it nontrivial for an application programmer to use the distributed shared memory. This is in contrast to the SGI's hardware shared memory system, which lets the programmer treat the shared objects exactly like the rest of the contents of the memory. The consequence is that rendered shapes have to be made DSM-aware to be shared, and so far only volumetric data has been modified to work in this system.

The trends that motivated this research are likely to continue. Computers are becoming cheaper and faster, and the connections between computers are getting quicker. However, processor speeds are improving faster than network rates are. Loosely coupled parallel machines, including clusters, the grid, and related technologies, are becoming more prevalent as a result of these trends. We have shown that network speed is already sufficient for interactive rendering. Subsections 4.2.3 and 5.2.9 show that network access time is only a small fraction of our program's running time. Because the network is not the primary bottleneck, the faster growth of computing technology will not make our techniques irrelevant for some years to come. On the contrary, we believe that it will become even more important to find ways to use loosely knit systems efficiently. In the same way that modern memory hierarchies bridge the gap between processor speed and memory access time, techniques like our use of asynchronous message handling and local data caching will become more important in the future.

Faster networks, faster processors, and larger local memory spaces will allow more complex scenes to be rendered more realistically at higher rates in the near future. It may be possible to increase the performance of the program without waiting for faster hardware, through various optimizations. A faster rendering engine is desirable not only because it can be used to improve image quality, but

also because it will take fewer nodes to attain the same frame rates, lowering the cost of interactive rendering.

This thesis has presented an analysis of the program, obtained from performing a series of benchmarks. Additional tests can better determine exactly what parts need to be improved. For instance, our program uses TCP instead of the User Datagram Protocol (UDP) because TCP ensures reliable transport. An analysis of the number of dropped packets on the network would show to what extent the reliability enforcement overhead is necessary. Also, the render threads spend a significant amount of time waiting for tasks and data. A more exact accounting of the program should point out how best to minimize the wait time. Lastly, CPU statistics could be obtained to determine how efficiently the program is accessing the memory hierarchy, which might lead to the development of faster acceleration structures.

Because the network layer is fundamental to the functioning of the ray tracer, it makes sense to optimize it even though it is not the primary bottleneck. Socket operations involve context switch overhead when the network system calls trap to system space. A user space network layer like VIA [6] might yield some improvement. Compression could improve bandwidth utilization, especially for large shared data bricks. A fast compression routine with a high compression ratio, could significantly reduce the brick miss time, as long as the decompression time is less than the time saved transmitting a smaller stream. One approach would be to make the DataServer keep compressed copies of the resident blocks, and allow requesting DataServers to decompress blocks on the fly.

As 64-bit PCs become available it may make sense to replace the DataServer with a page-based distributed shared memory. A 64-bit page-based shared memory will not be limited to 4 Gigabytes, and brick hit time may be reduced by using the virtual memory hardware to detect shared memory access. More than half of our brick access time is spent in detecting these hits, so this has a great potential for increasing the overall frame rate. A page-based shared memory will also make the application programmers job easier, because the acquire and release semantics

will be removed, and so distributed shapes will not have to be made DSM-aware in order to work.

As we increase the realism of our images by tracing more rays, we will certainly need additional optimization. Computation time is our primary bottleneck, and additional levels of parallelism may yield better performance. By exploiting the simultaneous multithreading and instruction-level parallelism provided by the Intel Xeon processors of our cluster, we hope to accelerate the computational phase of ray tracing. There is a good chance that we might also improve the frame rate by using a more highly specialized compiler. Initial indications on whether the native Intel compiler will produce faster code than the gnu compiler are inconclusive. In another direction, we could try to exploit the graphics cards in our cluster, starting with the algorithm proposed by Purcell et al. [30]. The difficult part will be efficiently controlling the flow of data throughout the cluster between nodes, processors, memory, and the graphics hardware.

We found that the render threads spend a significant portion of the rendering time waiting. It will be productive to investigate techniques to keep the ray tracing threads busy as often as possible. In addition to trying a more advanced local task queue, we plan to experiment with preferred bias task allocation strategies that combine the benefits of static and dynamic task assignment strategies. Data distribution causes additional wait time. Ray reordering may significantly reduce the miss penalty. Prefetching techniques, which aim to request missing data before it is needed, could be combined with ray traversal reordering, which makes the ray process cached data first. Both of these could be combined with the ability to place stalled rays on hold in favor of rays that are ready to be processed.

The need for additional features and better rendering quality is tantamount to the need for faster rendering. As the program stands it is a traditional Whitted style ray tracer. Fast global illumination and Cook effects may be added. Additionally, realistic scenes require larger scene data bases, and at the moment only MIP and isosurface volume rendering can utilize the distributed shared memory. In the short term we plan to extend the ray tracer to handle time varying volume data sets. We

anticipate a minor extension to the software to make it use several DataServers in a single program. Each time step will be given its own DataServer, and the program will switch between time steps every few frames. We also plan to bring in direct volume rendering modules from *-Ray. This will increase both the amount of processing on and the number of accesses to the data. Finally, we plan to extend the program to distribute large polygonal data sets.

In conclusion, we have found that lower cost clusters of PCs are sufficient for interactively ray tracing multi-Gigabyte datasets. The degree to which a more expensive shared memory supercomputing platform performs better on the same task has been studied. The results of this study should be useful to anyone attempting to fully exploit the performance of either class of parallel machine.

APPENDIX A

LEGION APPLICATION INTERFACE

File logger.h controls Legion's statistics gathering and log file creation utilities.

```
/* Programs that use Legion have to call this to create log files. */
void start_logger(char *cwd, int rank, int numthreads=L_MAX_THREADS);
void close_logger();

/* alog is for normal operational messages of the application. */
/* Use it with C++ stream stile -> "alog(<<"Hello World"<<endl);". */
/* Changing "#if 0" to "#if 1" turns on all of the alog calls. */
#if 0
  #define alog(PIPE) \
  {\
  pthread_mutex_lock(&iolock);\
  *mymsgs PIPE;\
  pthread_mutex_unlock(&iolock);\
  }
#else
  #define alog(PIPE) {}
#endif
/* alog_t is for normal, messages from the application's threads. */
#define alog_t(x, PIPE)...

/* elog is for abnormal error messages. */
/* This should not be turned off. */
```

```
#define elog(PIPE)...
#define elog_t(x, PIPE)...

/* llog is for normal operational messages inside the Legion.
/* This should usually be turned off. */
#define llog(PIPE)...
#define llog_t(x, PIPE)...

/* Uncommenting "#define DOSTATS" makes the communicator and the */
/* dataserver record detailed statistics during the run, and */
/* print them out in the log files when the program terminates. */
// #define DOSTATS
#ifdef DOSTATS
  /* These are macros which can be used to collect stats. */

  /* Counts and increment. */
  #define countit(a) {a++;}
  #define countem(a,d) {a+=d;}

  /* Start a timestamp. */
  #define timestamps(t0)...
  /* Finish a timestamp. */
  #define timestampe(t0, te)...
#else
  #define countit(a) {}
  #define countem(a, d) {}
  #define timestamps(t0) {}
  #define timestampe(t0, te) {}
#endif
```

File communicator.h defines Legion's central communicator class, which controls all network transmissions between nodes.

```
/* Classes that contain a message handling method subclass this. */
class callee {
 public:
  virtual int handlemessage(int sock, int othernode, int MSGID) = 0;
};

/* Message handling methods and function take a socket id, a */
/* sending node id, and a message type id, then process the data */
/* stream, and return a success indication. */
typedef \
 int (callee::* callee_method) (int sock, int othernode, int MSGID);
typedef \
 int (callee_function) (int sock, int othernode, int MSGID);

/* The communicator class directs network traffic. */
class communicator : public Runnable {
 private:

#ifdef DOSTATS
  double selects;
  double sends;
  double recvs;
  double bytes_sent;
  double bytes_recvd;
  double comm_time;
  double el_time;
#endif
```

```
public:
 /* The constructor takes: list of nodes, rank of this node, */
 /* number of nodes to use, and TCP port to connect to. */
 communicator(char *nodelistfile,
              int myrank,
              int numnodes,
              int port=PORT);
 /* The destructor prints stats to the log file and cleans up. */
 virtual ~communicator();

 /* Call this to connect this communicators before calling run. */
 int initweb();
 /* Call this to begin asynchronous handling of inbound data. */
 virtual void run();

 /* Spawn a command that can create an application with a */
 /* communicator on the rest of the nodes in the cluster. */
 int rspawn(char *cwd,
            char *command,
            char *args,
            char *nodelist=0,
            int port=-1,
            int numnodes=-1,
            int numthreads=0);
 /* Parse the rspawn supplied command line args and return a */
 /* pointer to executable's args. */
 static int parse_args(int *argc, char ***argv,
                       char **_cwd,
                       char **_nodelist, int *_port,
```

```
                        int *_numnodes, int *_myrank,
                        int *_numthreads);

  /* Classes call this to register their handlers. */
  int register_msgs(int MSGID, callee *&c, callee_method m);
  int register_msgs(int MSGID, callee_function *f);

  /* Thread safe sends and receives of one or more data buffers. */
  int send_msg (const int MSGID, const int otherrank,
                const size_t size1=0, void *message1=0,
                const size_t size2=0, void *message2=0);
  int recv_buff (const int sock, void *buffer, const size_t size);
  int send_buffs(int sock, struct iovec *vect, int count);
  int recv_buffs(int sock, struct iovec *vect, int count);

  /* Accessors. */
  inline int Myrank() const {return myrank;}
  inline int Numnodes() const {return numnodes;}
  int sock(const int i) const; /* Get socket for node i. */

  /* Create distributed barriers. */
  legion_barrier* register_barrier(int barrier_id,
                                   int numnodes, int *nodeset);
  /* Create distributed shared memory space. */
  dataserver* register_dataserver(int blksize,
                                  unsigned int rs_size,
                                  unsigned int lc_size,
                                  int assoc=1, int dtype=1,
                                  int align=1);
};
```

File dataserver.h defines the interface to the distributed shared memory classes.

```
class dataserver : public callee {
 protected:
  /* Control of the shared blocks. */
  datum **cache;
  int blksize;
  int rs_size;
  int lc_size;

#ifdef DOSTATS
  double accesses;
  double hits;
  double used_assoc;
  double fill_assoc;
  double misses;
  double retries;
  double me;
  double other;
  double my_time;
  double have_time;
  double miss_time;
#endif

 public:
  /* communicator::register_dataserver() calls this. */
  dataserver(communicator *comm,
     int blksize, int rs_size, int lc_size,
     int assoc=1, /* Associativity. */
     int align=0); /* Enable starting address alignment. */
```

```
  /* The destructor prints stats to the log file and cleans up. */
  ~dataserver();

  /* Register this with the communicator. */
  virtual int handlemessage(int sock, int othernode, int MSGID)=0;

  /* Adds data to the resident set without synchronization. */
  int add_data(void *address, int key);

  /* Acquire and release read only data. */
  virtual void* get_data(int /*key*/, int &/*areturn*/){};
  virtual void release_data(int /*key*/, int /*areturn*/){};

  /* Acquire and release writable data. */
  virtual void* get_data_to_write(int key, int &areturn){
    return get_data(key,areturn);
  };
  virtual void release_written_data(int key, int areturn){
    release_data(key,areturn);
  };

  /* How many non local blocks can I cache? */
  inline int LC_size() const {return lc_size;};
  /* How many local blocks do I have? */
  inline int RS_size() const {return rs_size;};
  /* How many blocks are in the entire shared memory? */
  inline int Blocks() const { return rs_size * numnodes; };
};
```

APPENDIX B

SHARED RENDERING STATE

```
class SceneState {
 public:
  int use_bv;                //type of accel. structure
  int gridcellsize;          //a parameter for grid acceleration
  double light_radius;       //diameter of area light sources
  double bvscale;            //soft shadow scale factor
  bool no_aa;                //accumulation buffer antialiasing
  bool bench;                //benchmark the scene
  bool logframes;            //record session
  int nworkers;              //number of nodes
  int nthreads;              //number of render threads per node
  double jitter_vals[1000];  //offsets for antialiasing
  double jitter_valsb[1000];
};

class FrameState {
 public:
  Point eye;                 //camera
  Point lookat;
  Vector up;
  double fov;
  double xoffset;            //for alpha buffer antialiasing
  double yoffset;
```

```
  double animtime;           //time stamp for animated objects
  int xres;                  //screen resolution
  int yres;
  int xtilesize;             //pixel tile dimensions
  int ytilesize;
  int shadow_mode;           //the selected shadow algorithm
  int maxdepth;              //the maximum ray depth
  int numPlaneDpys;          //number of aux. plane displays
  int numVolumeDpys;         //number of aux. volume displays
  float base_threshold;      //a refraction coefficient
  bool do_jitter;            //enable jittered antialiasing
  bool stereo;               //enable stereo rendering
  bool animate;              //enable animated objects
  bool hotspots;             //enable hotspot diagnostic view
  bool ambient_hack;         //enable a lighting mode
  bool reuse_tasks;          //enable static task assignment
  bool show_tasks;           //enable show tasks
  bool show_owner;           //enable show shared vol. data

  Array1<PlaneDpyVals*> pDpys; //space for aux. display data
  Array1<VolumeDpyVals*> vDpys;
};
```

REFERENCES

[1] (2003, Jan.) LAM/MPI Parallel Computing Downloads. [Online]. Available: http://www.lam-mpi.org/download

[2] (2003, Jan.) MPICH - A Portable Implementation of MPI. Argonne National Laboratory. [Online]. Available: http://www-unix.mcs.anl.gov/mpi/mpich/index.html

[3] *ATI Radeon 9800 Series*, ATI Corporation, 2003. [Online]. Available: http://www.ati.com

[4] B. N. Bershad and M. J. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," Carnegie Mellon University, Tech. Rep. CMU-CS-91-170, 1991.

[5] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," Ohio Supercomputing Center, 1994. [Online]. Available: http://www.lam-mpi.org/download

[6] *Virtual Interface Architecture Specification*, Compaq, Intel, Microsoft, Dec. 1997. [Online]. Available: http://www.vidf.org/info/04standards.html

[7] R. L. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," in *ACM Special Interest Group on Computer Graphics and Interactive Techniques*. The Association for Computing Machinery, Inc., 1984, pp. 137–145.

[8] B. Corrie and P. Mackerras, "Parallel Volume Rendering and Data Coherence," in *1993 Symposium on Parallel Rendering*. The Association for Computing Machinery, Inc., 1993, pp. 23–26.

[9] M. Cox and D. Ellsworth, "Application-Controlled Demand Paging for Out-of-Core Visualization," in *IEEE Visualization*, 1997, pp. 235–244.

[10] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray Tracing System," *IEEE Computer Graphics and Applications*, no. 6, pp. 16–26, Apr. 1986.

[11] A. Garcia and H.-W. Shen, "An Interleaved Parallel Volume Renderer with PC-clusters," in *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization*, Sept. 2002.

[12] A. Geist *et al.*, *PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[13] A. S. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, no. 4, pp. 15–22, Oct. 1984.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sept. 1996.

[15] R. Hempel, "The MPI Standard for Message Passing," in *Proceedings of the High-Performance Computing and Networking, International Conference and Exhibition*, 1994, pp. 247–252.

[16] M. R. Kaplan, "Space Tracing: A Constant Time Ray Tracer," in *Course Notes of the ACM Special Interest Group on Computer Graphics and Interactive Techniques.* The Association for Computing Machinery, Inc., 1985.

[17] P. Keleher. (1996, Nov.) CVM: The Coherent Virtual Machine. University of Maryland. [Online]. Available: http://www.cs.umd.edu/projects/cvm/

[18] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," in *Proceedings of the International Conference on Parallel Computing*, 1988, pp. 94–101.

[19] W. Ligon and R. Ross, "An Overview of the Parallel Virtual File System," in *Extreme Linux Workshop*, 1999.

[20] K. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering," in *Proceedings of Parallel Rendering Symposium*, 1993, pp. 15–22.

[21] A. A. Mirin *et al.*, "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System," in *High Performance Networking and Computing Conference*, 1999.

[22] N. Nevin, "The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster," Ohio Supercomputer Center, Tech. Rep. OSC-TR-1996-4, Mar. 1996.

[23] *GeForce FX Product Overview*, NVIDIA Corporation, 2003. [Online]. Available: http://www.nvidia.com

[24] (2002, Nov.) PVM Parallel Virtual Machine Web Page. Oak Ridge National Laboratory. [Online]. Available: http://www.csm.ornl.gov/pvm/

[25] M. E. Palmer, S. Taylor, and B. Totty, "Exploiting Deep Parallel Memory Hierarchies for Ray Casting Volume Rendering," in *IEEE Parallel Rendering Symposium*, 1997, pp. 15–22.

[26] S. Parker, W. Martin, P. Sloan, P. Shirley, B. Smits, and C. Hansen, "Interactive Ray Tracing," *Interactive 3D Graphics (I3D)*, pp. 119–126, Apr. 1999.

[27] S. Parker, M. Parker, Y. Livnat, P. Sloan, C. Hansen, and P. Shirley, "Interactive Ray Tracing for Volume Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 3, pp. 238–250, 1999.

[28] K. Perlin and E. M. Hoffert, "Hypertexture," in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, 1989, pp. 253–262.

[29] D. H. Porter. (2002, June) Volume Visualization of High Resolution Data Using PC-Clusters. University of Minnesota. [Online]. Available: http://www.lcse.umn.edu/hvr/pcvolrend14june2002.pdf

[30] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray Tracing on Programmable Graphics Hardware," in *ACM Special Interest Group on Computer Graphics and Interactive Techniques.* The Association for Computing Machinery, Inc., 2002, pp. 703–712.

[31] S. Rubin and T. Whitted, "A Three Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics*, vol. 14, no. 3, pp. 110–116, July 1980.

[32] *SGI Origin 3000*, Silicon Graphics Inc., 2002. [Online]. Available: http://www.sgi.com/origin/3000/datasheet.htm

[33] B. Tangney *et al.*, "Overview of distributed shared memory," Department of Computer Science, Trinity College, Tech. Rep. TCD-CS-1998-24, Oct. 1998.

[34] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek, "Interactive Global Illumination using Fast Ray Tracing," in *Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*, 2002.

[35] I. Wald and P. Slusallek, "State-of-the-art in Interactive Ray-Tracing," *State of the Art Reports, EUROGRAPHICS 2001*, pp. 21–42, Sept. 2001.

[36] I. Wald, P. Slusallek, and C. Benthin, "Interactive Distributed Ray Tracing of Highly Complex Models," in *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, June 2001, pp. 277–288.