# Distributed Shared Memory for Roaming Large Volumes

Laurent Castanié, Christophe Mion, Xavier Cavin, Bruno Lévy

▶ **To cite this version:**

Laurent Castanié, Christophe Mion, Xavier Cavin, Bruno Lévy. Distributed Shared Memory for Roaming Large Volumes. IEEE Transactions on Visualization and Computer Graphics, 2006, 12 (5). inria-00104989

**HAL Id: inria-00104989**
**https://inria.hal.science/inria-00104989**

Submitted on 9 Oct 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Distributed Shared Memory for Roaming Large Volumes

Laurent Castanié, Christophe Mion, Xavier Cavin and Bruno Lévy

**Abstract**—
We present a cluster-based volume rendering system for roaming very large volumes. This system allows to move a gigabyte-sized probe inside a total volume of several tens or hundreds of gigabytes in real-time. While the size of the probe is limited by the total amount of texture memory on the cluster, the size of the total data set has no theoretical limit. The cluster is used as a *distributed graphics processing unit* that both aggregates graphics power and graphics memory. A hardware-accelerated volume renderer runs in parallel on the cluster nodes and the final image compositing is implemented using a pipelined sort-last rendering algorithm. Meanwhile, volume bricking and volume paging allow efficient data caching. On each rendering node, a *distributed hierarchical cache system* implements a global software-based *distributed shared memory* on the cluster. In case of a cache miss, this system first checks page residency on the other cluster nodes instead of directly accessing local disks. Using two Gigabit Ethernet network interfaces per node, we accelerate data fetching by a factor of 4 compared to directly accessing local disks. The system also implements asynchronous disk access and texture loading, which makes it possible to overlap data loading, volume slicing and rendering for optimal volume roaming.

**Index Terms**— Large volumes, volume roaming, out-of-core, hierarchical caching, distributed shared memory, hardware-accelerated volume visualization, graphics hardware, parallel rendering, graphics cluster.

## 1 Introduction

Advances in the precision of data acquisition technologies (seismic captors, CT scanners, MRI, ...) and large-scale numerical simulations (FEM, CFD, ...) in all scientific domains are responsible for a constant growing size of scientific data sets. In this context, the visualization of extremely large data sets is becoming even more strategic. The large gigabyte volumes of yesterday are now commonly replaced with volumes of several tens or even hundreds of gigabytes. In scientific applications, such as in oil and gas [29, 8], volume roaming is a common visualization technique that makes it possible to focus on a dynamic sub-volume of the entire data set, i.e. a probe. Out-of-core technologies make it possible to visualize this probe interactively [4, 29, 8] up to a limited size. However, the minimum relevant size of the probe increases in the same proportions as the size of the data set. With volumes of tens or hundreds of gigabytes, gigabyte-sized probes are mandatory. Over the past ten years, the visualization community has investigated several solutions to keep up with this constantly increasing demand and PC clusters are a particularly promising one [19]. They are a cost-effective yet powerful alternative to shared memory supercomputers.

We propose a hardware-accelerated parallel volume rendering system for roaming very large volumes. It is based on the out-of-core technology presented in [8] for rendering on each node and uses a sort-last decomposition [26]. In sort-last parallel volume rendering, the probe is decomposed spatially and the sub-parts are rendered separately by the cluster nodes at full image resolution. The images are blended together using a parallel image compositing algorithm. We use the pipelined sort-last rendering algorithm presented in [9]. This fully overlapped implementation of sort-last parallel rendering allows to render a gigabyte-sized probe in real-time, without being limited by the network communications. However, simply combining these two components does not allow interactive volume roaming inside several tens or hundreds of gigabytes since many cache misses and disk accesses occur on each rendering node. To reduce disk accesses, we have implemented a software-based shared memory for clusters, similar in spirit to those in [18, 2, 13]. When roaming the probe, this *distributed shared memory* (DSM) allows data sharing between the rendering nodes through fast interconnection networks and message passing, which avoids accessing local or remote mass storage. A network connection with two Gigabit Ethernet interfaces has a bandwidth of 220 MB/s while a standard SATA 150 local disk has a bandwidth of 53 MB/s (see Section 3). Our implementation of DSM is based on a *distributed hierarchical cache system* (DHCS). DHCS is composed of two levels: one level in graphics memory caches data from memory and the other level in memory caches data from a series of devices (network, disk).

To our knowledge, in previous implementations of DSM for parallel visualization such as the one in [13], each node stores a sub-part of the entire data set into a static *resident set* and caches the other nodes resident set into a single-level network cache. Such systems neither write to graphics memory nor access the disk. One consequence of the latter is that the entire data set must fit into the total memory (RAM) of the cluster.

In contrast, in our DHCS, the memory state on each rendering node is completely dynamic. We use a single memory buffer that varies over time depending on the data needed to display the probe, as the user interacts with it. The memory pages have no pre-determined position on the cluster. As a consequence, we have no theoretical limitation in the total size of the data set that we are roaming (except the available hard drive space). Only the total size of the probe is limited: it should fit into the total graphics memory of the cluster.

### 1.1 Related Work

The system presented in this paper for interactively roaming very large volumes is based on several common concepts in visualization: parallel volume rendering for visualizing gigabyte-sized probes, out-of-core visualization for roaming in even larger volumes and distributed shared memory for fast data sharing between rendering nodes across fast interconnection networks.

**Real-Time and Parallel Volume Rendering:** There are several aspects from which we can classify the different parallel volume rendering systems: the volume rendering engine and the parallel decomposition.

Real-time volume rendering techniques [30] are either based on ray

- *Laurent Castanié is with Earth Decision and in the ALICE group at INRIA Lorraine (Nancy, France), E-mail: castanie@earthdecision.com.*
- *Christophe Mion is in the ALICE group at INRIA Lorraine (Nancy, France), E-mail: christophe.mion@inria.fr.*
- *Xavier Cavin is in the ALICE group at INRIA Lorraine (Nancy, France), E-mail: xavier.cavin@inria.fr.*
- *Bruno Lévy is in the ALICE group at INRIA Lorraine (Nancy, France), E-mail: bruno.levy@inria.fr.*

*Manuscript received 31 March 2006; accepted 1 August 2006; posted online 6 November 2006.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.*

casting [21] or volume slicing [20]. Our approach uses hardware-accelerated trilinear interpolation with 3D texture mapping to display back-to-front semi-transparent sampling slices as in Cabral *et al.* [5]. It is based on advanced optical models for better image quality [25, 15, 23].

Parallel decompositions are classified as either sort-first or sort-last [26] depending on what is being decomposed. In sort-first parallel volume rendering, the screen is decomposed into lower resolution tiles rendered separately while sort-last parallel volume rendering decomposes the database. Parallel sort-first decompositions have been implemented for volume rendering, either on shared memory supercomputers [28] or on clusters [13]. In sort-last decompositions, each part of the database is rendered separately at full resolution and a parallel image compositing algorithm blends them together in a last step. The binary swap [24] and SLIC [33] are two popular software compositing algorithms. We use the pipelined sort-last implementation of binary swap presented in [9]. This implementation fully overlaps the computations, rendering and network communications at all steps of the parallel process.

**Out-of-Core Visualization:** Scientific data sets commonly have multi-gigabyte sizes; however the interpretation process often focuses on small parts of the entire data. This makes it possible to implement out-of-core and demand paging techniques to account for the limited amount of memory on the system [11]. Volume roaming coupled with out-of-core visualization is widely used in the oil and gas industry [29, 8] and other scientific domains [4].

With out-of-core techniques, the overhead resulting from accessing local or remote mass storage is critical. It can be reduced with asynchronous loading using multi-threading [14, 36]. Another approach, as in the Visapult system [3] for remote and distributed visualization, is to use a network data cache such as the *distributed parallel storage system* (DPSS) [35].

In [17], Gao *et al.* propose a smart *distributed data management system* (DDMS) to handle large time-varying volumes. It is based on efficient data structures for fast selection of the portions of data to be fetched remotely and rendered.

In this paper, we build on top of our previous cache system [8] to add fully asynchronous loading in memory and graphics memory, which makes it possible to overlap data loading, volume slicing and rendering for optimal volume roaming.

**Distributed Shared Memory:** The goal of *distributed shared memory* (DSM) as introduced by Li [22] is to port the concept of shared memory used in multiprocessor supercomputers to the context of PC clusters. One challenging issue in DSM is to ensure cache coherence when allowing write access to memory, as with the ccNUMA interconnection layer of the SGI Origin for instance. Intensive research has been done to ensure cache coherence in DSM while keeping efficient write access to memory. Systems like Munin [6], Midway [37], Quarks [7], TreadMarks [1], are examples of such DSM implementations with efficient cache coherence. In our context of visualization however, data are accessed in read-only mode. As a result, expensive coherence maintenance algorithms are not necessary.

An early implementation of DSM for sort-first parallel rendering has been proposed by Green and Paddon [18]. They use ray tracing for polygon rendering. This implementation widely inspired the following ones [2, 13, 12]. In [13], DeMarle *et al.* implement a sort-first parallel ray casting system for volume rendering. Sort-first parallel ray casting suffers from high data sharing between the rendering nodes. The local memory space on each rendering node is divided into a resident set and a cache. The data set is decomposed and the sub-parts are statically distributed to the resident sets. At rendering time, a node that needs data which is not in its resident set gets it through the network from the owner node and stores it in its cache for future use. This framework avoids accesses to a local or remote mass storage, which is mandatory for interactivity. However, one important limitation of their system is that the entire data set must fit into the total memory of the cluster.

In our hardware-accelerated parallel volume rendering system we rather use a sort-last decomposition, since each sub-part of the data set is rendered independently with little data sharing. However, in the context of volume roaming, we target volumes much larger than the total memory of the cluster and moving the probe may result in lots of accesses to the mass storage. The goal of a DSM in this context is to allow faster probe roaming by removing the overhead due to disk accesses. For this purpose, our implementation of DSM is quite different from the previous ones dedicated to sort-first parallel rendering. Each node has a hierarchical cache system that writes to texture memory and to a fully dynamic local memory. The latter implies an efficient mechanism to maintain on each node an up-to-date cluster memory state for fast data fetching over the network. Finally, we have no limitation on the total size of the data set we are roaming (in the limits of the hard drive space available) and only the probe is limited by the amount of available graphics memory.

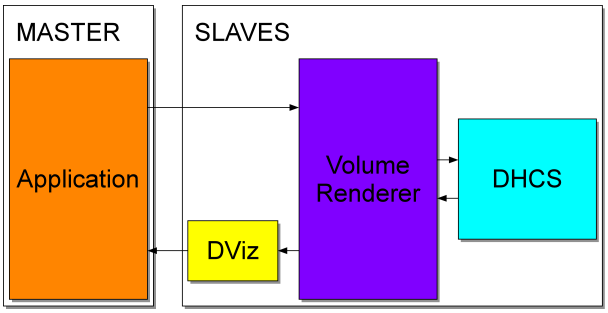


Fig. 1. Main components of the system and their interaction.

### 1.2 Contributions and Overview

In this paper, we present a system for interactively roaming a gigabyte-sized probe in a total volume of several tens or hundreds of gigabytes. This system is based on an original DSM implementation for hardware-accelerated sort-last parallel volume rendering. To our knowledge, previous implementations of DSM for parallel visualization were dedicated to sort-first parallel ray tracing/casting and were not suitable for roaming volumes that do not fit into the total memory of the cluster (Section 1.1). We have focused on both tailor-made algorithms for supporting a fully dynamic memory state on the cluster (Section 2) and on a fine tuning of the parameters to get an overall throughput near the theoretical bandwidth of the hardware components (Section 3).

Our main contributions to the domain are:

- A multi-threaded fully overlapped out-of-core volume roaming system.
- A DSM based on a distributed hierarchical cache system (DHCS) with four levels of data access: graphics memory, local memory on the node, memory on the other nodes through the network and disk.
- A DSM with fully dynamic local memory states (i.e. memory pages have no pre-determined position on the cluster) and the associated mechanism to keep the cluster memory state up-to-date on each node (efficient all-to-all broadcast).

The design of the system is given in Section 2. Section 2.1 gives a brief overview of the main components and their interactions while Section 2.2 focuses on our DHCS. Our overlapped implementation of data loading, volume slicing and rendering is exposed in Section 2.3. Several hardware dependent and general software optimizations are presented in Section 3. Finally, we give some experimentations in Section 4.

## 2 System Design

In this section, we present the design of the system. We first present the main components of the overall system and their interactions. Then, we focus on DHCS, which is dedicated to data management. Finally,

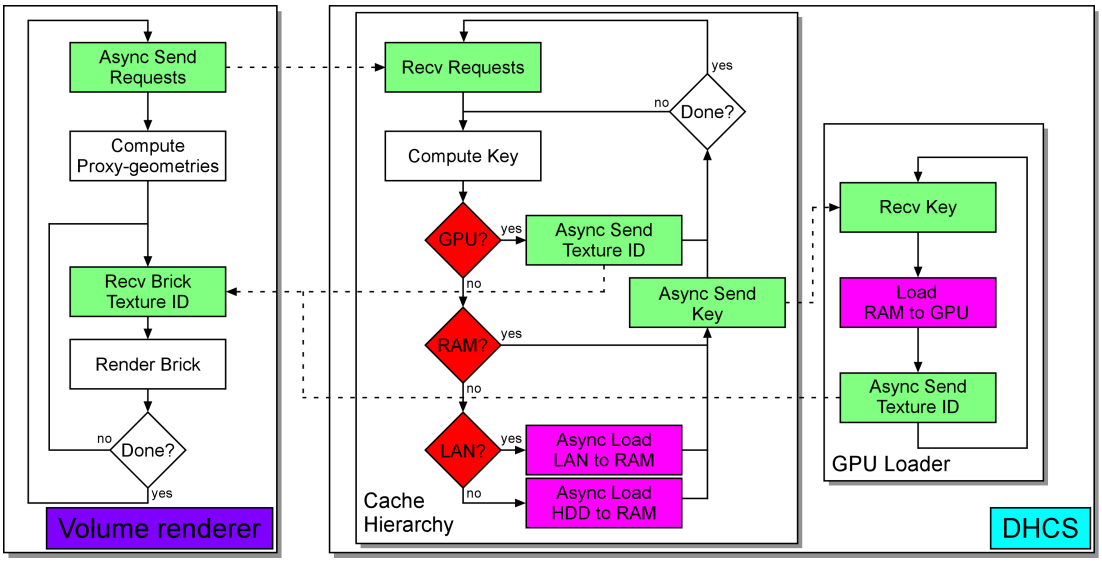


Fig. 2. Flow chart of the rendering on each cluster node with the volume renderer for actual rendering and DHCS for data access. DHCS is decomposed into two main components: a *Cache Hierarchy* and a *GPU Loader*.

we show how the data loading, volume slicing and rendering are fully overlapped.

### 2.1 Overview

The overall system is dedicated to volume roaming inside large volumes with hardware-accelerated sort-last parallel volume rendering on a cluster. A gigabyte-sized probe is decomposed into sub-parts rendered in parallel at full image resolution on the cluster nodes. The images are blended together using DViz[1], presented in [9].

As shown in Figure 1, the system is composed of four independent components: the application, a hardware-accelerated volume renderer, DHCS and DViz. The application runs on the master node while the volume renderer, DHCS and DViz run in parallel on the slave nodes. As the user is changing the point-of-view and roaming the volume, the application broadcasts the current camera position as well as a description of their sub-part of the probe to each slave's volume renderer. Each hardware-accelerated volume renderer requests its DHCS to load the necessary data in graphics memory. After each slave has rendered its image at full resolution, DViz blends them together in parallel using a fully overlapped pipelined implementation of binary swap. The result is sent to the application on the master node that draws the final image to the screen. The remainder of this section will focus on the volume renderer and DHCS components. For a detailed description of DViz, the reader is referred to [9].

### 2.2 Distributed Hierarchical Cache System

DHCS is the data loading component of the system. As shown in Figure 2, it is composed of two main components: a *Cache Hierarchy* and a *GPU Loader*. Following this flow chart, we will explain the mechanism of DHCS. Based on a memory and a graphics memory buffer, it implements out-of-core access to a virtual graphics memory for the volume renderer component of the overall system. The unit-sized data element in any out-of-core system is called a *page* [11]. Since linear or slice decompositions do not exploit data locality properly in volume visualization, our page decomposition of the volume is based on unit-sized bricks [8].

**Memory Management Unit:** DHCS is based on a hierarchy of cache buffers: one buffer in graphics memory caches data from memory and a second buffer in memory caches data from a series of devices (network, disk). Efficient caching and data access depends on the unitary size of data elements transferred between the different buffers. As shown in Section 3, the optimal size to transfer data from disk/network to memory is larger than from memory to graphics memory. For this reason, the unit-sized data element we manipulate in memory is larger than the one in graphics memory. We introduce the notion of *cluster* in memory, which is a group of bricks. Ultimately, the clusters are grouped into a series of files on disk. As a result, a volume brick in DHCS is uniquely identified by its file, the position of its cluster in the file and its own position in the cluster. As shown in Figure 3, this is encoded into a 32-bit key. This unique identifier is used as an entry into hash tables that store each cache buffer state. Note that the *File ID* may be used for instance with file systems that do not support files larger than 4 GB. In this case, our identifier supports data sets up to 1 TB. Using only the remaining 24 bits allows to store around 16 millions of bricks per file. This represents 2 TB of data with 128 KB bricks, which is far more than the tens or hundreds of gigabytes we are targeting.

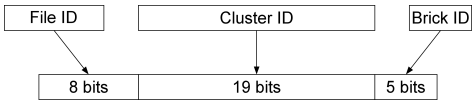


Fig. 3. DHCS uses a 32-bit key as a unique brick identifier.

The volume renderer identifies a brick by a triplet *(u,v,w)* that describes its position in the volume. When DHCS receives a request from the volume renderer, the first step consists in translating this triplet into its internal identifier. In the flow chart in Figure 2, this is done in the *Compute Key* operation. This is similar in spirit to the *memory management unit* (MMU) in operating systems virtual memory that translates a *virtual address* into a *physical address* [31]. Note that when working in the memory cache, the page granularity is larger since we manipulate clusters instead of bricks. In this case, we do not take the brick bits into account.

**Cache Hierarchy:** The core component of DHCS is a hierarchical cache between texture memory, main memory, network and disk. In Internet caching systems, hierarchical caching [10] defines the vertical *parent-child* and horizontal *sibling* relationships. In a parent-child relationship, a cache miss in the child is resolved by its parent, upper in the hierarchy, while a sibling horizontal relationship makes it possible to resolve a cache miss with a cache at the same level in the hierarchy. While a parent propagates a cache miss on behalf of its child, a sibling

[1] http://www.loria.fr/∼cavin/dviz

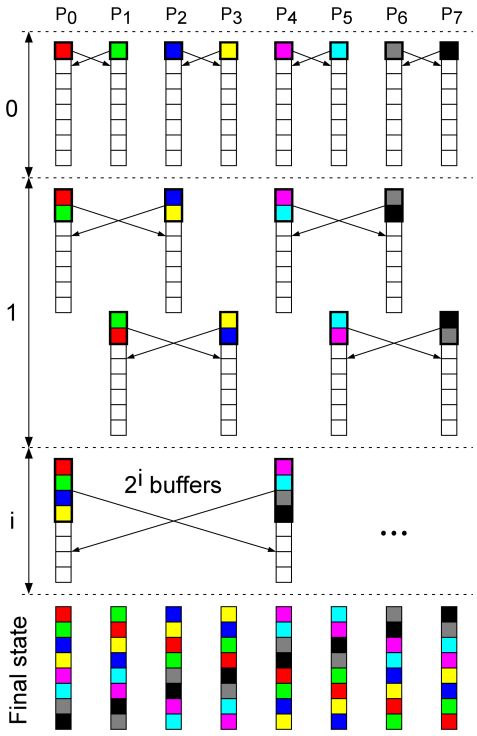


Fig. 4. *Binary all-to-all* communication in the case of 8 nodes $P_i$.

does not. Siblings are used for faster access through data redundancy among caches at the same level.

In our previous implementation of out-of-core visualization [8], we used a single workstation with a basic hierarchical cache between graphics memory, memory and disk. In sort-last parallel volume rendering on a cluster however, each node renders a sub-part of the dynamic probe. As a result, volume roaming benefits from faster access to siblings at the same level in the hierarchy through the network. Our caching mechanism is illustrated in Figure 2 with the *GPU?*, *RAM?* and *LAN?* conditions that respectively refer to a cache hit in graphics memory, memory or through the network. There is a parent-child relationship between the graphics memory and memory caches. Now, if we consider that each rendering node is running its own DHCS in parallel, the *LAN?* condition refers to a query of all the memory caches on the cluster and illustrates a sibling relationship. DHCS can actually be considered as a distributed hierarchical cache system. On each rendering node, a cache miss in graphics memory is propagated to the memory cache. Then, instead of directly accessing the disk, a cache miss in memory is propagated to the memory caches on the cluster (i.e. siblings). Ultimately, a miss in all the siblings causes an access to the filesystem.

**Network Cache Query:** In the context of interactive visualization, we cannot afford a network propagation to the siblings as in Internet caching. Instead, each node has an image of the entire memory state of the cluster. In previous implementations of DSM for visualization [18, 2, 13, 12], this image is implicitly known by the nodes since memory pages have a pre-determined position on the cluster, which yields several limitations as exposed in Section 1.1. In contrast, our memory buffer is fully dynamic and we implement an optimized all-to-all communication that maintains the network memory state up-to-date on each rendering node. This all-to-all communication is inspired by the binary swap algorithm [24]. As shown in Figure 4 for the case of 8 nodes, an array initially contains the keys stored in the local buffer. Then, each pair of nodes exchange a growing buffer at each step. Ultimately, each node has an array that contains all the keys stored in the memory of the cluster. This communication scheme is similar in spirit to the *recursive doubling* allgather operation used in many implementations of MPI [34]. The only difference in our case is that we do not reorder the buffers on each node, which removes substantial `memcpy` operations.

The theoretical lower bound for the execution time of an all-to-all communication is:

$$T_{all-to-all} = \frac{(P-1)size}{bandwidth} \quad (1)$$

where $size$ is the size of the buffer to transfer and $P$ is the number of nodes. In our case, we transfer the array of keys stored in local memory. This is the theoretical time with no latency, which is defined in networking as the minimum time delay it takes a packet to travel from source to destination independently on its size. In practice, the more the number of communication steps the more the algorithm is sensitive to communication latency. The classical approach is to implement a *logical ring* where the nodes exchange their buffer on a ring. Such strategy has a $P-1$ complexity (i.e. $P-1$ communication steps) and the resulting execution time is:

$$T_{logical\_ring} = \frac{(P-1)size}{bandwidth} + (P-1)latency \quad (2)$$

In our *binary all-to-all* communication, the size of the buffer transferred at each communication step increases, which decreases the complexity to $\log(P)$ and yields the following execution time:

$$T_{binary} = \frac{(P-1)size}{bandwidth} + \log(P)latency \quad (3)$$

where log is the logarithm to base 10. The main advantage of this binary all-to-all broadcast over the classical logical ring is its scalability thanks to the $\log(P)$ complexity.

Note that, as shown in the next section, this all-to-all communication is fully overlapped with the rendering.

### 2.3 Overlapped Implementation

In this section, we focus on the rendering components of the overall system presented in Figure 1: the volume renderer and DHCS. As shown in Figure 2, we hide the data transfers overhead by overlapping them with volume slicing and rendering on each cluster node. We adopted two complementary approaches for overlapping: multithreading and asynchronous loading.

Each component of the system, the volume renderer and DHCS, is implemented in a separate thread. When the volume renderer requests its DHCS to load a series of bricks in graphics memory, all the cache levels are scanned and the necessary data loads are launched asynchronously. *Async Load HDD to RAM* refers to the loading from disk to memory, which can be implemented on Linux using the `libaio` library for asynchronous I/O. *Async Load LAN to RAM* refers to the loading through the network. It is implemented in a separate thread using a client-server protocol with a minimal layer on top of TCP. Finally, texture loading in the GPU is handled in another thread that runs the GPU loader component. We are based on OpenGL and this thread cannot load textures directly in the rendering context of the main application thread. We rather create a new context with resource sharing.

While DHCS is scanning the cache hierarchy and launching asynchronous data loading, the volume renderer computes the proxy-geometries for rendering the bricks. In our case of hardware-accelerated volume rendering, we compute view-aligned back-to-front slices for each brick. Then, the bricks are rendered in back-to-front order as soon as they are loaded in graphics memory. The first bricks can be rendered even if all the requests are not yet satisfied.

Note that once DHCS has launched all the asynchronous data loads, the local memory state at the end of the current frame (i.e. when data loads will be achieved) is known. This allows us to launch our binary all-to-all communication overlapped with the rendering of the volume renderer.

A sequential implementation of this system with synchronous (i.e. blocking) requests would dramatically decrease the performance as data loading at each level of the cache, cluster memory state update on each node, volume slicing and rendering would be executed serially.

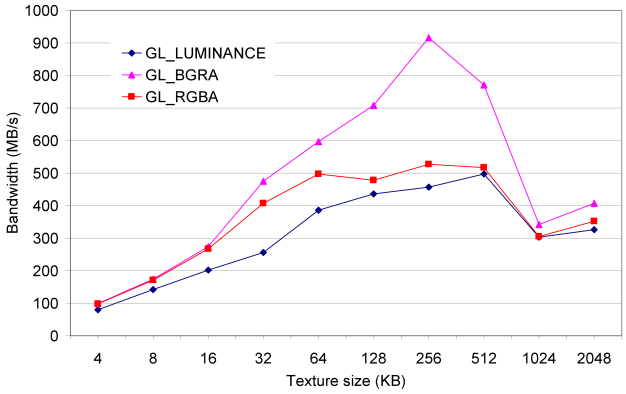


Fig. 5. Memory to graphics memory bandwidth against texture size for GL_LUMINANCE, GL_BGRA and GL_RGBA texture formats. The internal format is GL_INTENSITY for GL_LUMINANCE, and GL_RGBA8 for GL_BGRA and GL_RGBA (NVIDIA GeForce 6800 Ultra - PCI Express).

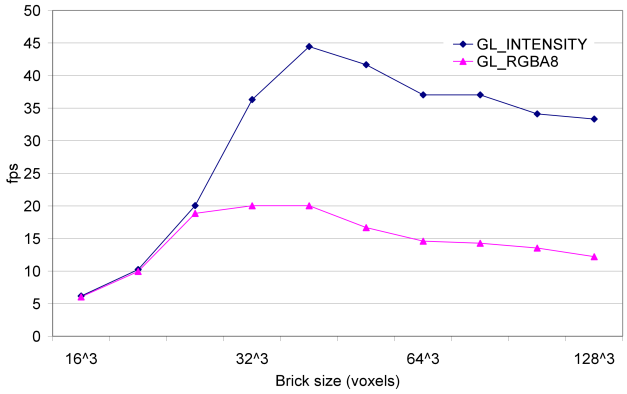


Fig. 7. Frame rate against the brick size in voxels for GL_INTENSITY and GL_RGBA8 internal formats when rendering a 50 MB test volume (NVIDIA GeForce 6800 Ultra - PCI Express).

## 3 System Optimization

In this section, we study some key parameters of the system for optimally consuming the hardware bandwidth.

**Texture Loading:** Graphics memory is the upper level of our hierarchical cache system. For this reason, this is the one that must support the larger number of write accesses. Therefore, the bandwidth sustained between memory and graphics memory has a large impact on the performance. This bandwidth is dependent on both the OpenGL texture format specified when transferring data to graphics memory [27], and on the size of data transferred (i.e. texture size). In Figure 5, we show the bandwidths we sustain using different texture formats and sizes when loading 3D textures. Note that these results highly depend on the hardware. We use a NVIDIA GeForce 6800 Ultra graphics card and a PCI Express bus. The highest bandwidth is achieved using a GL_BGRA texture format and a GL_RGBA8 internal format. The reason is that, for 8-bit textures, NVIDIA graphics cards match the Microsoft GDI pixel layout (i.e. pixel internally stored in the BGRA layout) [27]. As a result, when transferring data which is not in the BGRA layout in host memory to graphics memory, the driver has to swizzle the incoming pixels. The internal layout is independent on the GL_RGBA8 internal format that only impacts the number of bits per pixel. The second important point in Figure 5 is that for this optimal format, the highest bandwidth on our NVIDIA GeForce 6800 Ultra is achieved when transferring textures of 256 KB.

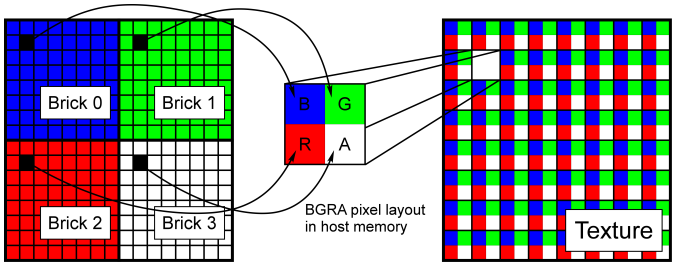


Fig. 6. Interleaving four bricks intensities into one single GL_BGRA texture (stored in GL_RGBA8 internal format).

**Texture Access:** While texture loading has a great impact on the overall performance of our system, once stored in graphics memory data access is even more important. This is highly dependent on the size of the Level 1 cache (i.e. local texture cache) [30] on the graphics chipset. We study the impact of changing the brick size on the frame rate for GL_INTENSITY and GL_RGBA8 internal formats when rendering a small 50 MB test volume. Note that we use post-classification, i.e. we convert texture intensities to colors with dependent lookups in a 1D texture in a fragment program. In case of GL_RGBA8 internal format, as shown in Figure 6, we store four bricks into one single texture, each one into a texture channel (R, G, B and A). Instead of using branching into one fragment program, we use four different programs to read in either channel so that the number of instructions is the same as in the simple case of GL_INTENSITY internal format. In the worst case, the overhead would be a fragment program switch after each brick, which as we tested has no impact on performance. The results are shown in Figure 7. With GL_INTENSITY textures, the best frame rate on our NVIDIA GeForce 6800 Ultra is obtained with 64x32x32 bricks, which corresponds to 64 KB textures. With smaller bricks, the rendering time is CPU bound (volume slicing), and bigger ones do not fit into the texture cache anymore. Now, in case of GL_RGBA8 textures, the optimal frame rate, obtained with 32x32x32 bricks, is twice slower than with GL_INTENSITY textures. The rendering time is CPU bound up to 32x32x16 bricks, which already corresponds to 64 KB textures as we store four bricks per texture in this case. Bigger bricks result in textures larger than the cache, which has a large impact on performance due to our interleaving approach. Alternatively, bricks could be stored side-by-side in GL_RGBA8 textures, which would account for the cache limitation but also require an additional overhead for correct border handling.

As a conclusion, we use the GL_LUMINANCE format and GL_INTENSITY internal format, with 64 KB pages (64x32x32 bricks). Downloading data to the GPU is faster with GL_BGRA format and GL_RGBA8 internal format (Figure 5), however GL_INTENSITY internal format is twice faster for rendering (Figure 7). The gain in rendering speed makes GL_LUMINANCE format and GL_INTENSITY internal format the best choice, even in roaming-type access.

**Network Versus Disk Access:** The discussion on texture loading and texture access exclusively concerns the top level of our DHCS, i.e. the cache in graphics memory. The following deals with network and disk access and focuses on the second level which is the cache in main memory.

As shown in Figure 8, the network bandwidth we achieve is dependent on the size of the packets sent over the network. The smaller the packet size, the more the bandwidth is sensitive to the communication latency. In our DHCS, using the same page size in main memory as in graphics memory (i.e. 64 KB) would dramatically decrease the performance, as we would send non optimal 64 KB packets over the network. For this reason, we introduce the notion of cluster in memory, which is a group of bricks. Clusters are the unit-sized data elements we manipulate in main memory (Section 2.2). For optimal DHCS be-

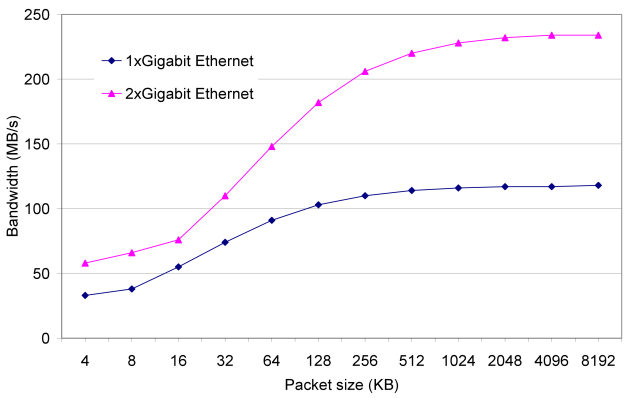


Fig. 8. Gigabit Ethernet bandwidth against the packet size for one and two interfaces.

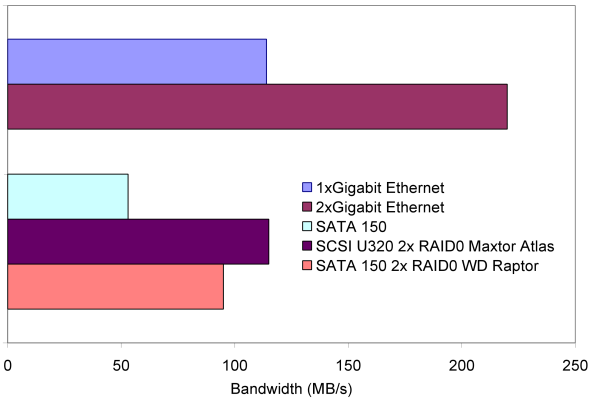


Fig. 9. Gigabit Ethernet bandwidth (512 KB pages) compared to disk bandwidth. SATA 150 is our local disk measured with SiSoftware Sandra 2005 [32] and the 2x RAID0 solutions are standard RAID0 configurations provided by Sandra for comparison.

haviour, we use 512 KB clusters (i.e. 2x2x2 bricks), which results in a 220 MB/s bandwidth using two Gigabit Ethernet interfaces. Larger clusters would unnecessarily increase the volume of communications.

| | Gigabit Ethernet | Infiniband 4x | SATA 150 |
|---|---|---|---|
| Latency | 60 $\mu$s | 5 $\mu$s | 10 ms |

Table 1. Latencies for Gigabit Ethernet and Infiniband 4x networks (obtained from [16]) compared with our SATA 150 local disk measured with SiSoftware Sandra 2005 [32].

We compare the network bandwidth using 512 KB pages with our SATA 150 local disk bandwidth in Figure 9. Using two Gigabit Ethernet interfaces, the network is four times faster than our local disk. It is also interesting to see that Gigabit Ethernet with two network interfaces outperforms standard RAID0 solutions. As shown in Figure 8, another important aspect is the latency. Table 1 reports a 10 ms measured latency on our SATA 150 local disk using SiSoftware Sandra 2005 [32]. In case of disks, latency is the time it takes to position the read/write head. Compared with the standard 60 $\mu$s Gigabit Ethernet latency [16], we have a factor of 150x. This is even more significant with Infiniband 4x interconnects, where the 5 $\mu$s latency yields a factor of 2000x. Either taking into account bandwidth or latency, mass storage accesses play a critical role in the overall performance, and must be avoided as much as possible.

## 4 Experimentation

We experiment our system on a 16-node bi dual-core AMD Opteron 275 cluster with 2 GB RAM and a NVIDIA GeForce 6800 Ultra graphics card on each node, and four Gigabit Ethernet interconnections. As shown in Figure 10, using this system, we achieve interactive roaming of a gigabyte-sized probe (1024x1024x1024, 8 bits per voxel) in a 107 GB data set (5580x5400x3840, 8 bits per voxel). The volume has been obtained by putting together 30 copies of the Visible Human [38]. The video in the supplemental material of this paper demonstrates the system in action at 12 frames per second (fps) on average when roaming in the cache and using a sampling rate of 1 (sampling distance equal to the voxel size). We use two network interfaces for our DViz parallel compositor and two others for DHCS. The screen resolution is 1024x768. Transparent bricks are discarded using the *value histogram* technique introduced in [17], which is based on bit-wise AND operations between encoded transfer functions and encoded brick histograms.

The probe is moved in each direction across the 30 visible men. Previous manipulations have filled the network memory cache. When roaming in this cache in the downward and backward movements, respectively along the blue and green volume axes, the average frame rate is 12 fps. In this case, most cache misses in memory are satisfied through the network thanks to the DHCS, which avoids file system accesses. However, the leftward movement along the red axis produces regular freezes that correspond to the disk accesses as we reach the network cache frontier. In this particular case, the system would dramatically benefit from pre-fetching strategies. Note also that the spatial decomposition of the probe plays a critical role in the overall performance of the roaming. We use a symmetric decomposition that yields a constant behavior, independently on the direction of the movement.

Finally, the last part in the video demonstrates the brute-force rendering performance resulting from the combination of our parallel compositor DViz and our volume rendering engine when rendering a static probe. For rendering a gigabyte-sized probe, we achieve 12-13 fps on average, which decreases to 8-9 fps when zooming-in. Note that either at 13 or 8 fps, we are not limited by the parallel compositing (the upper bound of DViz at this resolution is 45 fps) but rather by the rasterization on the GPU.

## 5 Conclusion and Future Works

We have presented a system for interactively roaming very large volumes with hardware-accelerated sort-last parallel volume rendering on clusters. This system is based on a fully dynamic implementation of a read-only DSM using a distributed hierarchical cache system (DHCS) with four levels of data access: graphics memory, local memory on the node, memory on the other nodes through the network and disk. The cluster memory state is maintained on each node using an efficient binary all-to-all broadcast. Compared to previous implementations of DSM for visualization, we are not limited by the total amount of memory on the cluster. We exposed the algorithmic aspects of DHCS as well as some hardware oriented optimizations to get an overall throughput near the components theoretical bandwidth.

In the future, our system may benefit from better load balancing when transferring data between the nodes. Thanks to the binary all-to-all communication, each node in the current implementation knows what the other nodes can provide. We have a local load balancing to distribute the requests over the cluster. A global load balancing needs that each node not only knows what the other nodes can provide but also what they request. This implies an additional all-to-all communication that cannot be overlapped with rendering and data loading as the other one, which introduces an overhead. A short-term solution in case of huge network load would be to access the disk. Another important aspect to be implemented is pre-fetching. Indeed, in our experimentation, the leftward movement along the red axis when no buffer yet exists would benefit from pre-fetching to hide the file system accesses.

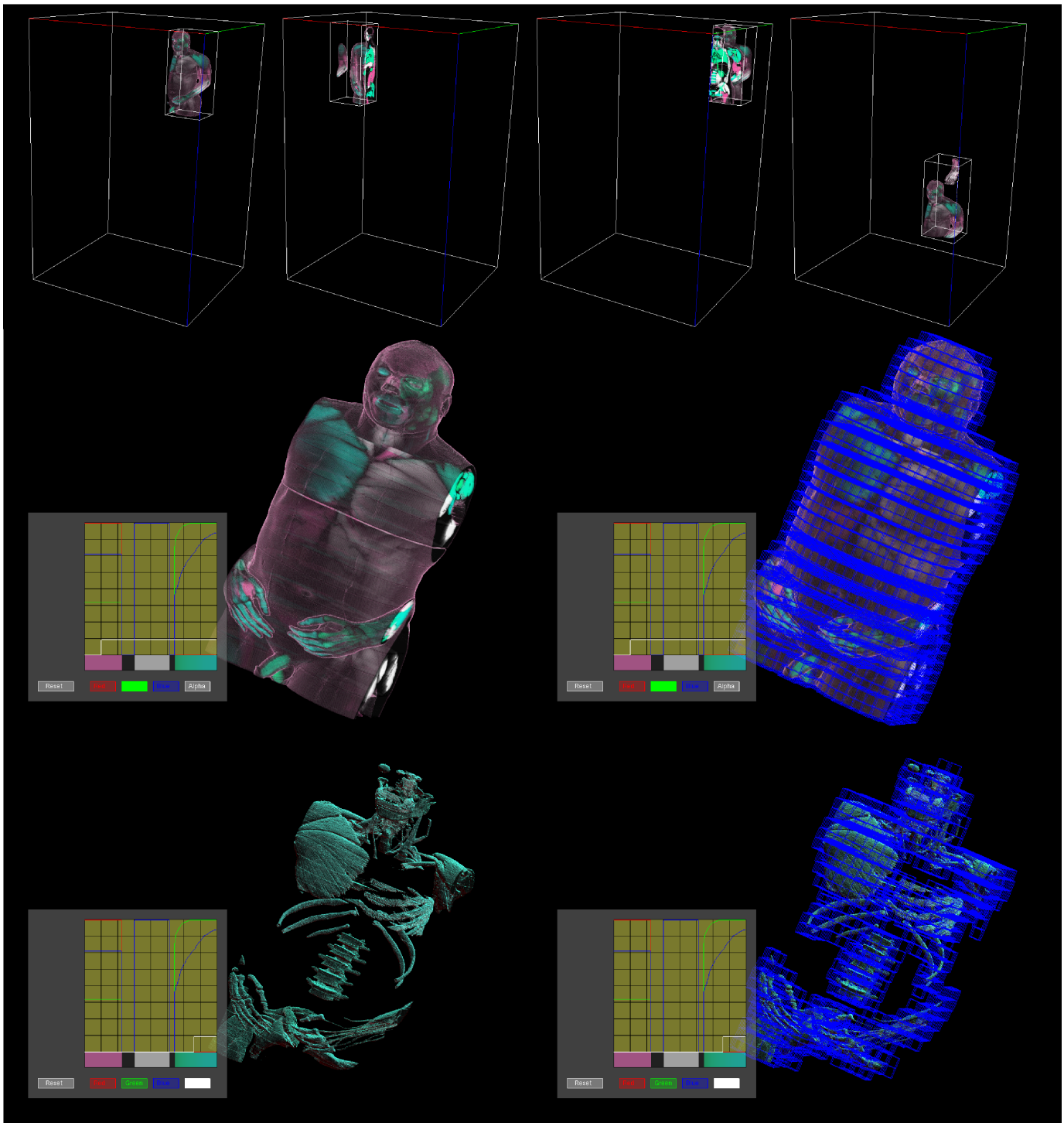


Fig. 10. Roaming a gigabyte-sized probe (1024x1024x1024, 8 bits per voxel) inside a 107 GB data set (30 copies of the Visible Human: 5580x5400x3840, 8 bits per voxel) on a 16-node PC cluster. Interactive roaming along the main axes of the volume at 12 fps on average (top). Zooming in high-quality pre-integrated volume rendering (middle) enhanced with accurate lighting on the vertebrae (bottom). The rendered bricks are shown in blue (middle and bottom, left). Transparent bricks are discarded using bit-wise AND operations between encoded transfer functions and encoded brick histograms.

## ACKNOWLEDGEMENTS

We would like to thank Alain Filbois from INRIA Lorraine-CRVHP for his precious help to generate the video. This work was partially supported by Earth Decision and grants from the INRIA and the Region Lorraine (Pôle de Recherche Scientifique et Technologique "Intelligence Logicielle"/CRVHP).

## REFERENCES

[1] AMZA C., COX A.L., DWARKADAS S., KELEHER P., LU H., RAJAMONY R., YU W., and ZWAENEPOEL W. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29:18–28, February 1996.

[2] BADOUEL D., BOUATOUCH K., and PRIOL T. Distributing Data and Control for Ray Tracing in Parallel. *IEEE Computer Graphics and Applications*, 14(4):69–77, 1994.

[3] BETHEL W., TIERNEY B., LEE J., and GUNTER D. adn LAU S. Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *Proceedings of Supercomputing Conference*, 2000.

[4] BHANIRAMKA P. and DEMANGE Y. OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 45–54, 2002.

[5] CABRAL B., CAM N., and FORAN J. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 91–98, 1994.

[6] CARTER J.B., BENNETT J.K., and ZWAENEPOEL W. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

[7] CARTER J.B., KHANDEKAR D., and KAMB L. Distributed Shared Memory: Where We Are and Where We Should Be Headed. In *Workshop on Hot Topics in Operating Systems*, pages 119–122, 1995.

[8] CASTANI L., LVY B., and BOSQUET F. VolumeExplorer: Roaming Large Volumes to Couple Visualization and Data Processing for Oil and Gas Exploration. In *Proceedings of IEEE Visualization Conference*, 2005.

[9] CAVIN X., MION C., and FILBOIS A. COTS Cluster-Based Sort-Last Rendering: Performance Evaluation and Pipelined Implementation. In *Proceedings of IEEE Visualization Conference*, 2005.

[10] CHANKHUNTHOD A., DANZIG P.B., NEERDAELS C., SCHWARTZ M.F., and WORRELL K.J. A Hierarchical Internet Object Cache. In *Proceedings of the USENIX Annual Technical Conference*, pages 153–164, 1996.

[11] COX M. and ELLSWORTH D. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proceedings of IEEE Visualization Conference*, pages 235–244, 1997.

[12] DEMARLE D., GRIBBLE C., BOULOS S., and PARKER S. Memory Sharing for Interactive Ray Tracing on Clusters. *Parallel Computing*, 31(2):221–242, February 2005.

[13] DEMARLE D., PARKER S., HARTNER M., GRIBBLE C., and HANSEN C. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 87–94, 2003.

[14] ELLSWORTH D. Accelerating Demand Paging for Local and Remote Out-of-Core Visualization. Technical Report NASA Ames Research Center, 2001.

[15] ENGEL K., KRAUS M., and ERTL T. High-Quality Pre-Integrated Volume Rendering Using Hardware Accelerated Pixel Shading. In *Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 9–16, 2001.

[16] FORCE10 NETWORKS, INC. The High Performance Data Center: The Role of Ethernet in Consolidation and Virtualization. http://www.force10networks.com/products/pdf/wp_datacenter_convirt.pdf, 2005.

[17] GAO J., HUANG J., JOHNSON R., ATCHLEY S., and KOHL J.A. Distributed Data Management for Large Volume Visualization. In *Proceedings of IEEE Visualization Conference*, pages 183–189, 2005.

[18] GREEN S. and PADDON D. Exploiting Coherence for Multiprocessor Ray Tracing. *IEEE Computer Graphics and Applications*, 9(6):12–26, November 1989.

[19] HOUSTON M. Designing Graphics Clusters. Parallel Rendering Workshop - IEEE Visualization Conference, 2004.

[20] LACROUTE P. and LEVOY M. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. In *Proceedings of ACM SIGGRAPH Conference*, volume 28, pages 451–457, 1994.

[21] LEVOY M. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8:29–37, 1988.

[22] LI K. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, 1986.

[23] LUM E.B., WILSON B., and MA K.L. High-Quality Lighting and Efficient Pre-Integration for Volume Rendering. In *Proceedings of Eurographics/IEEE Symposium on Visualization*, pages 25–34, 2004.

[24] MA K.L., PAINTER J.S., HANSEN C.D., and KROGH M.F. Parallel Volume Rendering Using Binary-Swap Image Composition. *IEEE Computer Graphics and Applications*, 14(4):59–68, July 1994.

[25] MAX N. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[26] MOLNAR S., COX M., ELLSWORTH D., and FUCHS H. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[27] NVIDIA CORPORATION. Fast Texture Downloads and Readbacks Using Pixel Buffer Objects in OpenGL. http://developer.nvidia.com/object/fast_texture_transfers.html, August 2005.

[28] PARKER S., PARKER M., LIVNAT Y., SLOAN P.P., HANSEN C., and SHIRLEY P. Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[29] PLATE J., TIRTASANA M., CARMONA R., and FRHLICH B. Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes. In *Proceedings of Eurographics/IEEE Symposium on Visualization*, 2002.

[30] REZK-SALAMA C., ENGEL K., HADWIGER M., KNISS J., LEFHON A., and WEISKOPF D. Real-Time Volume Graphics. Course 28, ACM SIGGRAPH, 2004.

[31] SILBERSCHATZ A., GAGNE G., and GALVIN P.B. *Operating System Concepts, Seventh Edition*. John Wiley & Sons, Inc., Hoboken, 2005. 921 p.

[32] SISOFTWARE SANDRA 2005. http://www.sisoftware.co.uk/.

[33] STOMPEL A., MA K.L., LUM E.B., AHRENS J., and PATCHETT J. SLIC: Scheduled Linear Image Composition for Parallel Volume Rendering. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 33–40, 2003.

[34] THAKUR R. and GROPP W. Improving the Performance of Collective Operations in MPICH. In *Proceedings of the 10th European PVM/MPI Users' Group Conference (Euro PVM/MPI 2003)*, pages 257–267, 2003.

[35] TIERNEY B., LEE J., CROWLEY B., HOLDING M., HYLTON J., and DRAKE F. A Network-Aware Distributed Storage Cache for Data Intensive Environments. In *Proceedings of IEEE High Performance Distributed Computing Conference*, 1999.

[36] WALD I., DIETRICH A., and SLUSALLEK P. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Proceedings of Eurographics Symposium on Rendering*, pages 81–92, 2004.

[37] ZEKAUSKAS M.J., SAWDON W.A., and BERSHAD B.N. Software Write Detection for Distributed Shared Memory. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 87–100, 1994.

[38] N. I. H. U.S. National Library of Medecine. The Visible Human Project. http://www.nlm.nih.gov/research/visible/visible_human.html.