COTS Cluster-based Sort-last Rendering: Performance Evaluation and Pipelined Implementation

Xavier Cavin* Christophe Mion Alain Filbois
Inria Lorraine

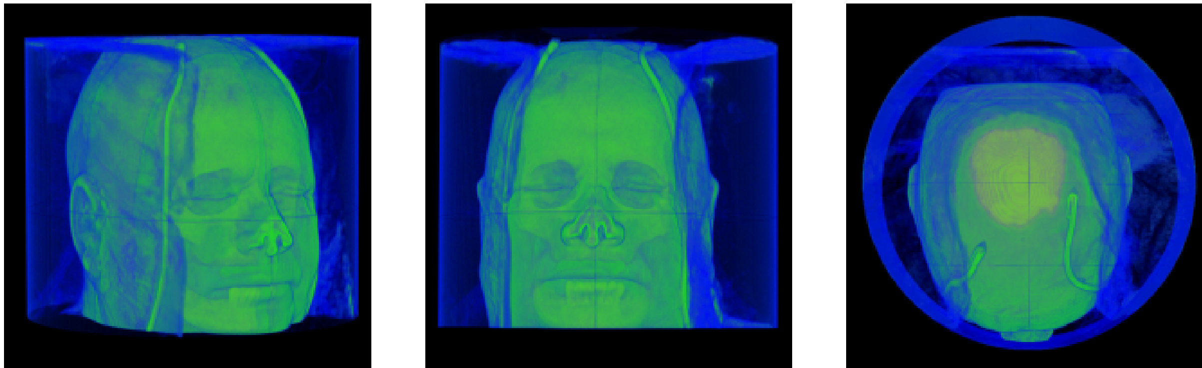

Figure 1: Views of the head section (512x512x209) of the visible female CT data with 16 nodes (a space has been left between the subvolumes to highlight their boundaries). Using a 3 years old 32-node COTS cluster, a volume dataset can be rendered at constant 13 frames per second on a 1024×768 rendering area using 5 nodes. On a 1.5 years old, fully optimized, 5-node COTS cluster, the frame rate obtained for the same rendering area reaches constant 31 frames per second. We truly expect our future work, including further algorithm optimizations and hardware tuning on a modern PC cluster, to provide higher frame rates for bigger datasets (using more nodes) on larger rendering areas.

ABSTRACT

Sort-last parallel rendering is an efficient technique to visualize huge datasets on COTS clusters. The dataset is subdivided and distributed across the cluster nodes. For every frame, each node renders a full resolution image of its data using its local GPU, and the images are composited together using a parallel image compositing algorithm. In this paper, we present a performance evaluation of standard sort-last parallel rendering methods and of the different improvements proposed in the literature. This evaluation is based on a detailed analysis of the different hardware and software components. We present a new implementation of sort-last rendering that fully overlaps CPU(s), GPU and network usage all along the algorithm. We present experiments on a 3 years old 32-node PC cluster and on a 1.5 years old 5-node PC cluster, both with Gigabit interconnect, showing volume rendering at respectively 13 and 31 frames per second and polygon rendering at respectively 8 and 17 frames per second on a 1024×768 render area, and we show that our implementation outperforms or equals many other implementations and specialized visualization clusters.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications; C.2.5 [Computer-Communication Networks]: Local and Wide-Area Networks—Ethernet

Keywords: cluster-based visualization, sort-last rendering, parallel image compositing

*e-mail: cavin@loria.fr

1 INTRODUCTION

1.1 Motivation

As PC clusters become widely available as a huge and cheap computing and storage resource, datasets obtained from 3D acquisition (3D scanners, CT scanners, MRI, ...) or resulting from large-scale numerical simulations (FEM, CFD, ...) become bigger and bigger (datasets of several Gigabytes are now a common place). Visualizing such datasets requires a similar amount of computing and graphics resources. As shown in [4], using a PC cluster for this goal slowly appears as an efficient and viable solution, compared to high-end High Performance Computing (HPC) systems.

Sort-last parallel rendering (as described in [13]) is an efficient technique to visualize huge datasets on PC clusters. As illustrated on Figure 2, the dataset is subdivided and distributed across the cluster nodes. For every frame, each node renders a full resolution image of its data using its local GPU and the images are blended together using a parallel image compositing algorithm.

Several software and hardware parallel image compositing methods are available, either in the literature, or commercially. In this paper, we focus on the use of Commodity Off-The-Shelf (COTS) PC cluster to perform this task. By COTS cluster, we mean here standard PCs with high-end graphics card, connected by a high speed (Gigabit) network, excluding specialized networks (such as Myrinet and Infiniband) and hardware image compositors. We demonstrate in this paper why and how a COTS cluster can be a viable competitor compared to more expensive solutions.

1.2 Related work

1.2.1 Software parallel image compositing

Many software parallel image compositing algorithms have been proposed in the literature, and can be applied either to volume

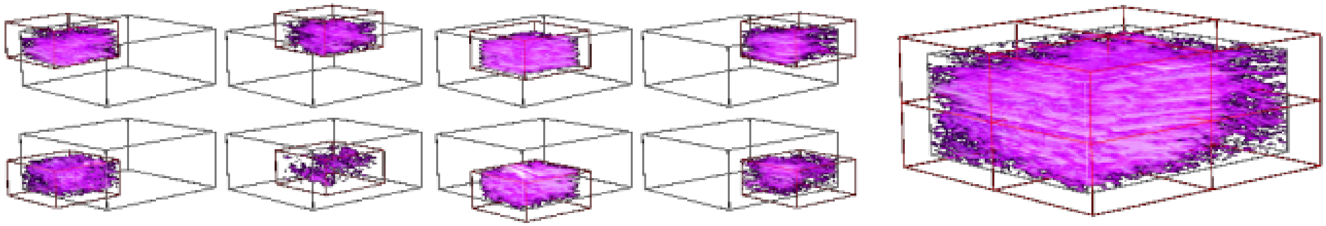

Figure 2: Sort-last parallel rendering: the dataset is subdivided and distributed across the cluster nodes, each node renders a full resolution image (left), a parallel image compositing algorithm is applied to compose the final image (right).

or polygon rendering. The most important ones include: direct send [5, 18], binary tree, binary swap [11] and parallel pipeline [9]. Although these algorithms were not designed with PC clusters in mind, we will show in Section 2 that these methods are equivalent (see Section 2.3) and efficient on a moderately sized COTS cluster. A similar study has been done in [22] for shared memory architectures.

Many optimizations, taking advantage of the sparsity of the images locally generated on each node, have been proposed for the software methods, including parallel pipeline [9], binary tree [1], direct send [28] and binary swap [31, 30, 24]. In this paper, we have not implemented any of those methods to concentrate on the worst case situation, but we will show in Section 2.6 that any compression could be advantageously integrated into sort-last rendering methods, including ours, as long as Equation 21 is respected.

Recent works using software parallel image compositing on graphics PC cluster include optimized direct send [28], optimized binary swap [30], binary tree over the Photonic Computing Engine [8], visualization of compressed volume datasets using direct send [29] and binary swap with Chromium [6, 4].

Stompel et al. [28] obtain several (1 or 2) frames per second for a $600 \times 800 \times 129$ volume dataset on a 1024^2 viewport, using a 64-node (1 GHz CPU) PC cluster with 100BaseT interconnect.

Takeuchi et al. [30] report 45 frames per second (taking only image compositing into account) for a $512 \times 512 \times 730$ volume dataset on a 512^2 viewport, using a 64-node (dual-Pentium III 1 GHz) PC cluster with Myrinet-2000 interconnect.

Kirihata et al. [8] obtain 14 frames per second for a 256^3 volume dataset on a 512^2 viewport, using a 16-node (dual-Xeons 1.8 GHz, PNY NVIDIA Quadro FX3000) with Gigabit Ethernet.

Strengert et al. [29] report 5 (resp. 8) frames per second for a $2048 \times 2048 \times 1878$ volume dataset on a 1024^2 (resp. 512^2) viewport on a 16-node (dual-AMD 1.6 GHz, NVIDIA GeForce 4 Ti 4600) PC cluster with Myrinet interconnect, and 2 frames per second for a 256^3 time-varying volume dataset on a 8-node (Pentium4 2.8 GHz, NVIDIA GeForce 4 Ti 4200) PC cluster with Gigabit Ethernet interconnect (compared to 5 frames per second with the Myrinet cluster).

Houston [4] reports compositing performance on the SPIRE [26] cluster using the binary swap SPU of Chromium [6]. The SPIRE cluster is a 16-node (Dual 2.4 GHz P4 Xeons, ATI Radeon 9800 Pro) PC cluster with Infiniband 4X interconnect. The reported compositing performances for a 1024^2 rendering area using 16 nodes are given on Table 1. However, overall performance for a 1024^3 volume dataset is reported to only 8 frames per second.

Moreland et al. [15, 14] introduced a method to efficiently perform sort-last rendering of extremely large data sets onto tile displays. Their implementation, called ICE-T, handles 450 million triangles on a 63 million pixels display at 0.06 frame per second.

Interconnect	Volume (RGBA)	Polygon (RGB + Z)
Gigabit (CPU)	9.5 fps	3.8 fps
Gigabit (GPU)	17 fps	7.2 fps
Infiniband 4x (CPU)	14 fps	6 fps
Infiniband 4x (GPU)	45 fps	11 fps

Table 1: Binary swap image compositing performance in frames per second on the SPIRE cluster (CPU and GPU blending).

1.2.2 Hardware compositors

Since 1999, several hardware architectures have been designed to support parallel image compositing for PC clusters: Sepia [3, 12] and Sepia-2 [10], Lightning-2 [27], Metabuffer [32], MPC Compositor [16, 19], Orad's DVG.

While some of them have become commercially available (Sepia-2 is available through HP and Sepia 3 is under development [25], MPC Compositor is marketed by Mitsubishi Precision Co., Ltd.), they are complex and expensive for a COTS system. We will show that our approach is highly competitive compared to these hardware solutions.

Stoll et al. [27] obtain 13 frames per second for polygon rendering on a hybrid 1280×1024 (frame buffer)/800×600 render area using a 9-node (Pentium4 1.5 GHz, NVIDIA GeForce2 Ultra) PC cluster with a Lightning-2 matrix.

Lombeyda et al. [10] report compositing performance of 24 to 28 frames per second for volume datasets on a 1024^2 render area using a 8-node PC cluster with SeverNet-II interconnect and Sepia-2 hardware compositing. Frank et al. report in a recent technical report [2] 6 frames per second on huge volume datasets for a 1280×1024 render area using 9 nodes of a 33-node PC cluster with SeverNet-II interconnect and Sepia-2 hardware compositing.

Nonaka et al. [19] obtain compositing performance of 13.8 frames per second for volume datasets on a 1024^2 render area using a 9-node (Pentium4 2.4 GHz, NVIDIA GeForce FX5950 Ultra) PC cluster with Gigabit Ethernet and a MPC compositor.

1.3 Organization of the paper

The remainder of this paper is organized as follows. In Section 2, we present a performance evaluation of standard sort-last parallel rendering methods, based on a detailed analysis of the algorithm and of recent hardware components. In Section 3, we describe our novel implementation of sort-last rendering, that fully overlaps CPU(s), GPU and network at all stages of the process. Section 4 reports performance results of our implementation on a 3 years old 32-node PC cluster and on a 1.5 years old 5-node PC cluster, both with Gigabit interconnect. Our best results show volume rendering at constant 31 frames per second and polygon rendering at constant 17 frames per second on a 1024×768 render area.

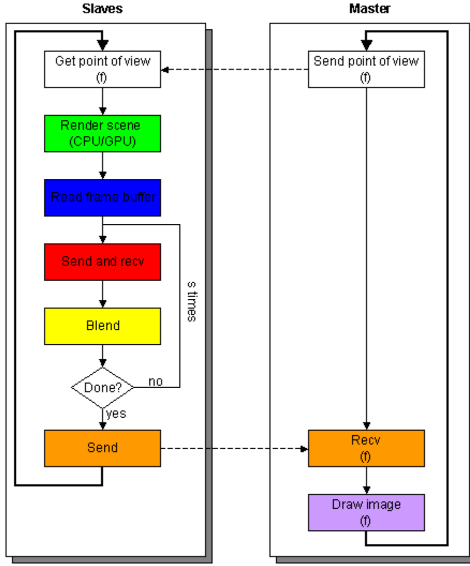

Figure 3: Flow diagram of sort-last parallel rendering.

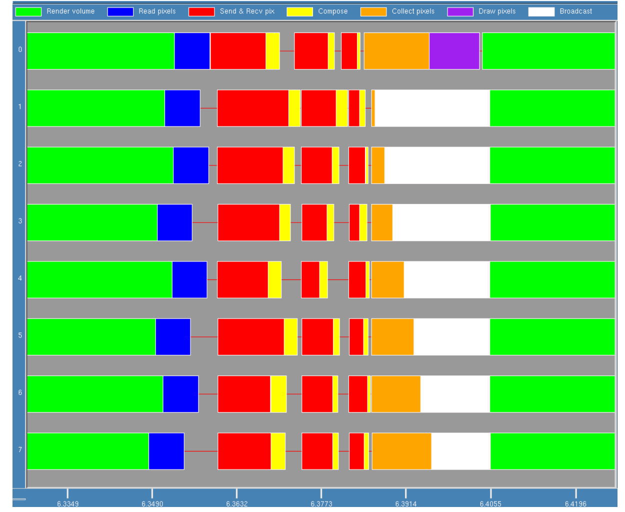

Figure 4: Typical run of a non optimized sort-last volume renderer (no Z term) using binary swap image compositing with 8 nodes (the first slave node is also the master node).

2 Performance evaluation of sort-last rendering

In this Section, we evaluate the time needed to display a frame on a cluster of $n+1$ nodes (n slaves and one master) using sort-last rendering. The frame resolution is $x \times y$ for a total of xy pixels. For each pixel, we use bpp bits to store its color and $zdth$ bits to store its depth.

The general algorithm for sort-last rendering is depicted on Figure 3. Figure 4 shows a profiling plot for the binary swap image compositing scheme. The same color coding is used in both Figures. For each new point of view, the time needed to display a frame can be decomposed in successive steps:

$$time = render + read + compose + collect + draw \qquad (1)$$

Each step will be detailed in the remaining of this Section. In the general case, the overall performance is limited by the slowest node. To simplify, we will assume that we have an homogeneous PC cluster, that the dataset is distributed in a balanced way, that no compression of any kind is applied to image and depth buffers. In other words, the amount of work for each frame is the same on all slave nodes. The Z component of each step is optional and may be removed in the case of back to front image compositing (for instance for volume rendering).

2.1 Rendering

The *render* term (green) is the time needed to render the assigned dataset on a $x \times y$ frame. Let fps denote the rendering speed in frames per second of the rendering method at the given $x \times y$ resolution. Then:

$$render = \frac{1}{fps_{xy}} \qquad (2)$$

The *render* term is an important component in the sort-last rendering approach, for two main reasons. First, it clearly impacts on the overall frame rate. Second, it consumes GPU and CPU (for instance for scene graph traversal), which can not be used for other tasks.

2.2 Reading pixels

The *read* term (blue) is the time needed to read back the color buffer and the depth buffer into main memory, and is defined as:

$$read = l_{read_{xy}} + \frac{xy \times bpp}{b_{read_{xy}}} + l_{readZ_{xy}} + \frac{xy \times zdth}{b_{readZ_{xy}}} \qquad (3)$$

where l is the latency in seconds and b is the bandwidth in bits per second of the GPU operation.

The latency l is in the order of a few us, and is negligible when reading large buffers. In theory, the peak bandwidth b of AGP x2 bus is 4.2 Gb/s and of AGP x8 bus is 16.8 Gb/s [7]. Upcoming PCI Express buses are expected to provide a theoretical peak bandwidth of 260 Gb/s (bounded by the memory peak bandwidth).

In practice, the sustained bandwidth is much lower, mostly because this functionality is not requested by game developers. Moreover, experimental measurements show that the bandwidth b is related to the size xy of the buffer [4].

For a while, the maximum readback performance commonly reported on most graphics hardware was bounded by 1.6 Gb/s (see for instance the the NVIDIA General FAQ [20]). Things have been evolving recently. Tests on the SPIRE cluster [26] report read peak bandwidth of 6.9 Gb/s for RGBA frame buffer and 2.4 Gb/s for depth buffer on the ATI Radeon 9800 Pro [4]. Our own tests on the NVIDIA 6800 Ultra report read peak bandwidth of 4 Gb/s for both RGBA frame buffer and depth buffer.

2.3 Image compositing

The *compose* term ($s\times$ red and yellow) is the time needed for the image compositing, and can usually be decomposed into s successive steps:

$$compose = \sum_{i=1}^{s} compose_i \qquad (4)$$

Several software parallel image compositing algorithms have been proposed. It can easily be shown that direct send, binary swap and parallel pipeline can be expressed in s steps with a single send and a single receive operation per node per step. The time needed for each step i is decomposed into:

$$compose_i = send_i + recv_i + blend_i \qquad (5)$$

The $send_i$ and $recv_i$ terms (red) are the time needed for a node p to send a part of the image (xy_i pixels) to a node p_1 and to receive another part of the image to be composed with (generally also xy_i pixels) from a node p_2. Then:

$$send_i = l_{send_{xy_i}} + \frac{xy_i \times bpp}{b_{send_{xy_i}}} + l_{sendZ_{xy_i}} + \frac{xy_i \times zdth}{b_{sendZ_{xy_i}}} \quad (6)$$

and:

$$recv_i = l_{recv_{xy_i}} + \frac{xy_i \times bpp}{b_{recv_{xy_i}}} + l_{recvZ_{xy_i}} + \frac{xy_i \times zdth}{b_{recvZ_{xy_i}}} \quad (7)$$

where l and b are the latency in seconds and the bandwidth in bits per second of the network operations.

Table 2 reports theoretical latency and peak bandwidth for different interconnects, including Gigabit Ethernet. In practice, latency and sustained bandwidth are different from the theoretical values: precise figures are reported in [17].

Interconnect	Bandwidth (half-full)	Latency
Gigabit	1-2 Gb/s	100-150 us
Infiniband 4x	10-20 Gb/s	3.5-7 us
Myrinet	2-8 Gb/s	3.5-7 us
SGI NUMAlink4	8-16 Gb/s	1-2 us
Quadrics	9 Gb/s	1-2 us
SCI/Dolphin	4 Gb/s	1-2 us

Table 2: Bandwidth and latency for different interconnects.

If the network and the interconnect support full duplex send and receive operations (which is the case for Gigabit Ethernet), it may be possible to overlap them, so that:

$$compose_i = \max(send_i, recv_i) + blend_i \quad (8)$$

We will now assume:

$$compose_i = sendandrecv_i + blend_i \quad (9)$$

with:

$$sendandrecv_i = \begin{cases} send_i + recv_i \\ \text{or} \\ \max(send_i, recv_i) \end{cases} \quad (10)$$

A critical point at this stage is the aggregate communication bandwidth sustained by the interconnection. If the sum of all communications is more than the aggregate bandwidth, then a severe performance degradation occurs. To simplify, we will assume for now an infinite aggregate communication bandwidth, allowing full scalability in terms of nodes and communications.

The $blend_i$ term (yellow) is the time need to blend the two subimages at step i and is defined as:

$$blend_i = l_{blend} + \frac{xy_i \times (bpp + zdth)}{b_{blend}} \quad (11)$$

where l_{blend} is the latency of the blending operation and b_{blend} is the number of bits (color and depth) per second the blending method can handle. We assume in this Section that blending is completely done in the CPU (GPU blending will be discussed in Section 3.4): in general, the latency l_{blend} is close to zero.

The theoretical peak value of b_{blend} is hard to evaluate. It clearly depends on the CPU clock speed and of the memory efficiency. Strengert et al. [29] provide an optimized MMX code but no performance figures. As an example, our SSE2 implementation of alpha blending (no depth) described in Section 3.4 can handle 4 Gb/s.

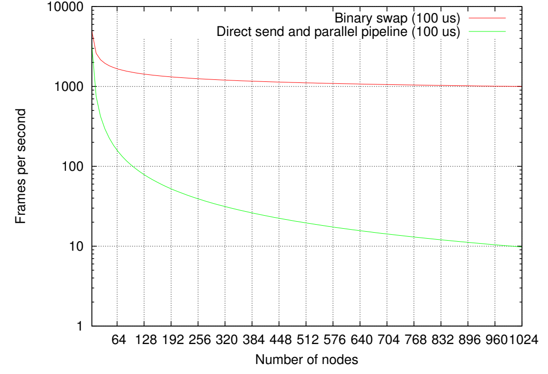

Figure 5: Scalability of binary swap versus direct send and parallel pipeline. This Figure shows an upper bound of the obtainable frame rate due to latency (100 us) in the (Gigabit Ethernet) interconnect (full duplex case, infinite aggregate bandwidth, no depth, independently of the resolution of the rendering area).

To summarize, the *compose* term is equal to:

$$\begin{aligned} compose &= \sum_{i=1}^{s} (send_i + recv_i + blend_i) \\ &= s \times (l_{sendandrecv} + l_{sendandrecvZ} + l_{blend}) + (\sum_{i=1}^{s} xy_i) \times \\ & \left(\frac{bpp}{b_{sendandrecv}} + \frac{zdth}{b_{sendandrecvZ}} + \frac{bpp + zdth}{b_{blend}} \right) \quad (12) \end{aligned}$$

The number of steps s and the amount of data transferred at each step xy_i depend on the chosen compositing algorithm. We will study two cases, first binary swap, second direct send and parallel pipeline, and we will show that all these methods are mostly equivalent on a moderately sized COTS cluster.

2.3.1 Binary swap image compositing

Classical binary swap image compositing requires $n = 2^k$ slave nodes. Then, the algorithm completes in $s = k = log_2(n)$ steps. At each step i, the amount of transferred data is $xy_i = \frac{xy}{2^i}$. Then:

$$\sum_{i=1}^{s} xy_i = \sum_{i=1}^{log_2(n)} \frac{xy}{2^i} = xy \times (1 - \frac{1}{n}) \quad (13)$$

2.3.2 Direct send and parallel pipeline image compositing

Direct send and parallel pipeline algorithms complete in $s = n - 1$ steps. At each step i, the amount of transferred data is $xy_i = \frac{xy}{n}$. Then:

$$\sum_{i=1}^{s} xy_i = \sum_{i=1}^{n-1} \frac{xy}{n} = xy \times (1 - \frac{1}{n}) \quad (14)$$

which is exactly the same as Equation 13 for the binary swap case.

2.3.3 Comparison

In terms of performance on a PC cluster, binary swap differs from direct send and parallel pipeline only in the number of calls to the network functions. In the binary swap, each node makes $log_2(n)$ calls, while in the direct send and parallel pipeline, each node makes $n - 1$ calls. Considering a latency of 100 us (Gigabit Ethernet), Figure 5 gives an upper bound of the maximum obtainable frame

rate for the different methods, only due to latency problems. The binary swap algorithm clearly scales better with a high latency (100 us) and a high number of nodes (over 128).

The choice between the different algorithms should be motivated by the total number of nodes. For instance binary swap is limited to using a power of two slave nodes. For a large number of nodes (over 128) and a high latency for network operations (about 100 us), binary swap would be the best choice.

2.4 Collecting pixels

The *collect* term (orange) is the time needed to collect the n subimages that compose the final image:

$$\begin{aligned} collect &= \sum_{i=1}^{n} \left(l_{collect_{xy/n}} + \frac{\frac{xy}{n} \times bpp}{b_{collect_{xy/n}}} \right) \\ &= n \times l_{collect_{xy/n}} + \frac{xy \times bpp}{b_{collect_{xy/n}}} \end{aligned} \quad (15)$$

where l and b are again the latency in seconds and the bandwidth in bits per second of the network operations (see Section 2.3). We assume here that we do not collect the depth buffer on the master node, although it might be necessary for some applications. On moderately sized COTS cluster, this time is independent of the number of slave nodes.

2.5 Drawing pixels

The *draw* term (purple) is the time needed by the master node to draw the color buffer from main memory into the graphics memory, and is defined as:

$$draw = l_{draw_{xy}} + \frac{xy \times bpp}{b_{draw_{xy}}} \quad (16)$$

where l and b are again the latency in seconds and the bandwidth in bits per second of the GPU operation. They have the same theoretical values as for the *read* term in Section 2.2

In practice, the sustained bandwidth to send data from the main memory to the GPU is higher than the one to read the data back. Once again, this is due to game developers, that require this functionality.

For instance, the NVIDIA General FAQ [20] reports writes performance of 5.44 Gb/s on the Quadro FX family (probably with AGP x8), and even 13.6 Gb/s (AGP x8) and 7.68 Gb/s (AGP x4) when using the `NV_pixel_data_range` extension. Our own tests on the NVIDIA 6800 Ultra report draw peak bandwidth of 7.6 Gb/s for RGBA frame buffer (without using the extension).

2.6 Compression

Some compression (bounding rectangles, RLE, ...) can be applied before sending the frame buffer and the depth buffer to the network (as mentioned in Section 1.2), in order to save bandwidth and speed-up the sort-last rendering. We have chosen not to implement compression in order to keep the network load constant: this is the worst case situation, and performance could be enhanced (at the price of instable frame rates) if the following is respected.

Let us assume a compression speed of b_c Gb/s, a compression ratio of r_c, a decompression speed of b_d Gb/s. Then, the time needed to send xy pixels with bpp bits per pixel is defined as:

$$l_c + \frac{xy \times bpp}{b_c} + l_{send} + \frac{(xy \times bpp) \times (1 - r_c)}{b_{send}} + l_d + \frac{xy \times bpp}{b_d} \quad (17)$$

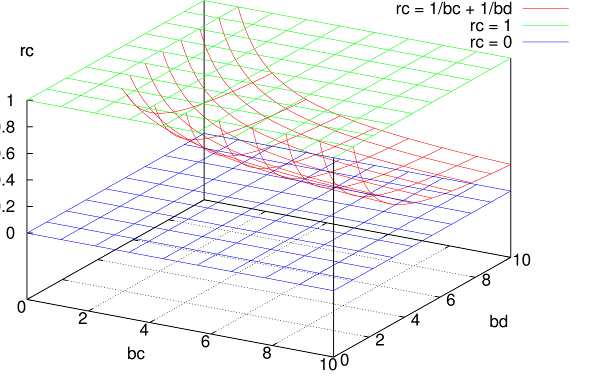

Figure 6: Dependencies between compression ratio r_c, compression speed b_c (Gb/s) and decompression speed b_d (Gb/s) on a Gigabit Ethernet interconnect.

and has to be compared to the time needed to send the uncompressed pixels:

$$l_{send} + \frac{xy \times bpp}{b_{send}} \quad (18)$$

Then, in order to benefit from compression, we need to ensure (assuming that latencies are negligible) that:

$$\frac{1}{b_c} + \frac{1 - r_c}{b_{send}} + \frac{1}{b_d} < \frac{1}{b_{send}} \quad (19)$$

in other words:

$$\frac{b_{send}}{b_c} + \frac{b_{send}}{b_d} < r_c \quad (20)$$

In the case of a Gigabit Ethernet network, this leads to:

$$\frac{1}{b_c} + \frac{1}{b_d} < r_c \quad (21)$$

Figure 6 plots a curve showing the dependencies between the three variables. As an illustration, our SSE2 implementation of alpha blending (no depth) described in Section 3.4 can handle 4 Gb/s: the compression ratio at this speed should be over 50%!

2.7 Summary

To illustrate this Section, we will use a sort-last volume rendering application (no Z term) on an ideal COTS cluster with no latencies. Then the time to render an image of $x \times y$ pixels is:

$$\begin{aligned} time &= render + read + compose + collect + draw \\ &= \frac{1}{fps} + (xy \times bpp) \times \left(\frac{1}{b_{read}} \right. \\ &\quad + (1 - \frac{1}{n}) \times (\frac{1}{b_{sendandrecv}} + \frac{1}{b_{blend}}) \\ &\quad \left. + \frac{1}{b_{collect}} + \frac{1}{b_{draw}} \right) \end{aligned} \quad (22)$$

If we now assume that our ideal COTS cluster is equipped with 3 GHz processors (using our SSE2 implementation of alpha blending described in Section 3.4, $b_{blend} = 4$ Gb/s), full duplex Gigabit Ethernet with infinite aggregate bandwidth, AGP x8 graphics, then:

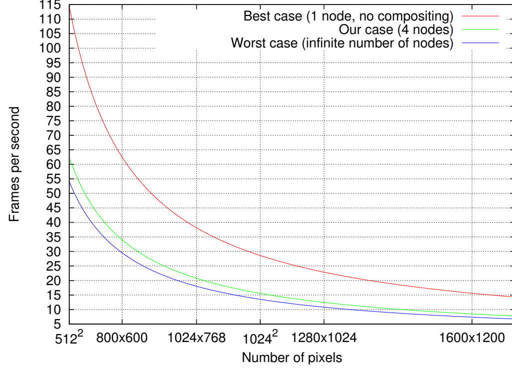

Figure 7: Theoretical optimal rendering speed in frames per second in the best and worst cases (volume rendering case).

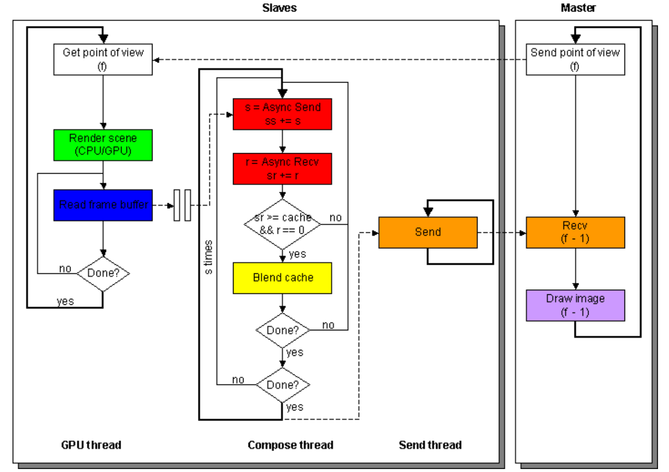

Figure 8: Flow diagram of pipelined sort-last parallel rendering.

$$time = \frac{1}{fps} + (xy \times bpp) \times (\frac{1}{16.8Gb/s} + (1 - \frac{1}{n}) \times (\frac{1}{1Gb/s} + \frac{1}{4Gb/s}) + \frac{1}{1Gb/s} + \frac{1}{16.8Gb/s}) \quad (23)$$

So, the overhead in second to render a frame (with a classical RGBA 32 bits per pixel), only due to sort last rendering, is bounded by:

$$32 \times xy \times \frac{18.8}{16.8Gb/s} \leq overhead \leq 32 \times xy \times (\frac{18.8}{16.8Gb/s} + \frac{5}{4Gb/s}) \quad (24)$$

The corresponding rendering speed in frames per second are given on Figure 7 for different resolutions: they give an upper bound of the obtainable frame rate on our COTS cluster.

3 Pipelined sort-last rendering

We present in this Section our new sort-last rendering algorithm - “pipelined sort-last” - that fully overlaps CPU(s), GPU and network usage at all stages of the parallel process. This idea is similar in the spirit to the hardware pipelining introduced in Sepia [12], but on a COTS cluster without dedicated hardware.

In this Section, we will focus on pipelined sort-last volume rendering. Pipelined sort-last polygon rendering (with depth compositing) can easily be extended from the volume version. Figure 8 gives an overview of our new algorithm, using the same color coding as Figures 3 and 4. Each slave nodes is composed of three concurrent threads: the GPU thread (in charge of GPU), the Compose thread (in charge of parallel image compositing) and the Send thread (in charge of sending the final subimage to the master node).

3.1 Multi-threading the slave nodes

In a classical implementation, the master node proceeds as follows. It starts by sending the point of view for the image f to be rendered. The n slave nodes then compute the subimage they have been assigned to, before sending it to the master node (orange). The master node receives the n subimages (orange), draws the final image to the GPU (purple), and then sends a new point of view for image $f+1$. During this time, the slave nodes are idle, waiting for the new point of view.

Our first optimization consists in starting the rendering of image f for the next point of view while the master node is receiving the subimages and drawing the final image $f-1$ to the GPU for the current point of view.

This is done by splitting the work of each slave node into three concurrent threads. The GPU thread repeatedly gets a new point of view, renders the image, and reads back the frame buffer. The Compose thread is in charge of parallel image compositing once a frame buffer has been read and is available in memory. For now, we can assume that these two threads work sequentially. The Send thread just wait for a local subimage to be computed by the Compose thread and sends it to the master node, while the GPU thread is already working on the next point of view.

3.2 Overlapping reading pixels with image compositing

In a classical implementation, the first step of parallel image compositing (red) starts when the readback operation (blue) is completed, as described in Section 2.3.

When reading back the frame buffer from GPU to main memory, the CPU and the network are not busy. The ideal solution would be to send the frame buffer content directly to the network (not passing through main memory). This is unfortunately not possible with actual technologies.

Our second optimization consists in overlapping the readback operation (blue) with the network send and receive operations (red) of the first step of parallel image compositing. This way, sending and receiving subimages can start before the readback operation is completed.

The frame buffer is read by regions by the GPU thread, and the regions are sent and received through the network by the Compose thread as soon as one region is available (using a producer/consumer model). This is possible because the graphics card and the network card are connected to different buses (respectively AGP and PCI-X) via the north and the south bridges.

A read operation actually consists in two read operations. The first reads the region to be sent, the other reads the region corresponding to the subimage to be received. As soon as the subimage has been received, subimages blending (yellow) can be done.

3.3 Overlapping sending, receiving and blending

As described in Section 2.3, parallel image compositing is really efficient when a send and a receive operation can be done simultaneously. This supposes that the network and the interconnect support full duplex (which is the case for Gigabit Ethernet), but also that the application (*i.e.* the Compose thread) does the job correctly.

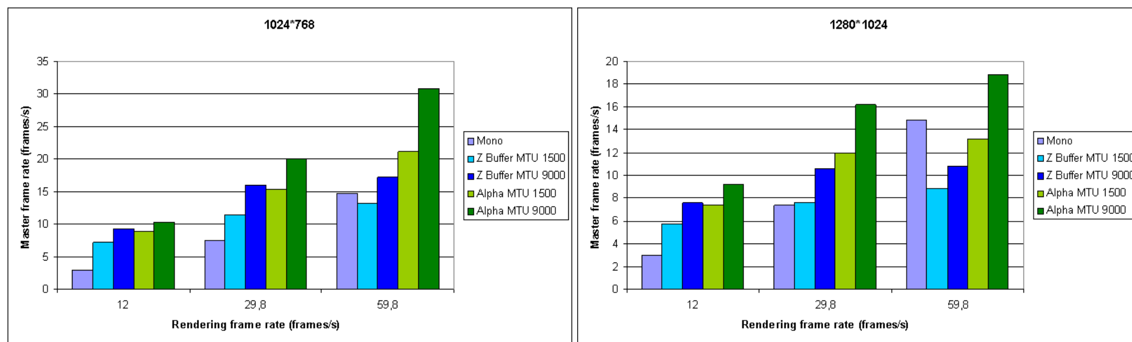

Figure 9: Frame rates observed on the master node for different rendering speeds (fps) on the 4 slave nodes.

Achieving full duplex send and receive operations could be done using two threads (one for the send, one for the receive). We have implemented this solution and we have found that, since each slave node already uses three threads, adding one more thread becomes difficult to handle for the operating system, even with hyperthreading enabled.

We have chosen to implement the full duplex send and receive operations with another strategy. The Compose thread repeatedly calls an asynchronous send and an asynchronous receive, that returns (quasi) immediately the number of bits they have been able to send or receive. If an asynchronous receive call returns zero (no packets have been received) and that enough data is available for blending, then the blending operation is done on a cache line, before the next asynchronous calls. This allows to overlap send, receive and blend operations.

3.4 Blending optimization

Another optimization concerns the blending of subimages (yellow) at each step of the parallel image compositing method, as described in Section 2.3.

Several optimizations have been proposed in the literature, including using the GPU accelerated blending [4] or software compositing using MMX [29] instructions. Other optimizations could include using multiple threads.

GPU accelerated blending has been proven to be very efficient [4] thanks to very fast read and draw operations. As shown on Table 1, the parallel image compositing performance jumps from 14 frames per second with software blending to 45 frames per second with GPU accelerated blending. However, using the GPU for the blending makes it unavailable for rendering. When rendering is overlapped with blending, this solution has to be avoided, except if a second GPU unit can be dedicated to blending.

Using multiple threads for image blending is highly efficient, since the blending algorithm is trivially parallel. However, when several threads are already running on the machine, the speed-up is not so high. This solution could be applied if one ore more CPUs are available on the machine.

A lower level parallelism can be obtained with vector instructions. Using MMX operations, Strengert et al. [29] blend two pixels together in 20 operations. We have implemented a similar blending using SSE2 operations, and we can blend 2 pixels together 2 times in 21 operations. On a 3 GHz Pentium4 processor, we can blend 13 millions pairs of 32-bit RGBA pixels per second.

4 EXPERIMENTATION

We have implemented our pipelined sort-last algorithm in C using OpenGL. Several rendering applications have been plugged in it, including volume and polygon rendering.

We have run our tests on a 3 years old 32-node (dual-Xeon 1.7 GHz, NVIDIA GeForce3 NV20) COTS cluster with Gigabit Ethernet interconnect on a Extreme Networks 6816 BlackDiamond switch, and on a 1.5 years old 5-node (dual-Pentium4 3 GHz, NVIDIA 6800 Ultra) COTS cluster with Gigabit Interconnect on a Cisco 3750 switch.

On the newest cluster, we have been able to test two different values of the Maximim Transmission Unit (MTU): 1500 and 9000 (Jumbo frames). On the oldest one, we have unfortunately not been able to change the default 1500 MTU (due to hardware limitations).

We have performed several runs of both polygon and volume rendering using our pipelined sort-last method. Figure 9 reports the performance obtained on our 1.5 years old 5-node cluster, for different MTU (1500 and 9000), different rendering areas (1024×768 and 1280×1024), and different values of fps (*i.e.* the frame rate on each slave). The results are clearly better for volume rendering, because much less data has to be transferred over the network.

Using a bigger MTU not only increases the raw performance, but also ensures the stability of the frame rate. Using a MTU of 9000 ensures a constant frame rate, while the MTU of 1500 gives unstable frame rates, due to the overusage of the CPUs.

On the 3-years old 32-node cluster, the obtained performances are two to three times lower than on the newer cluster. This is mostly due the MTU of 1500, as simple experiences have proven. The maximum bandwidth for a half duplex communication is 0.9 Gb/s. The maximum cumulated bandwidth for a full duplex communication is 1 Gb/s (which is too low compared to the expected 2 Gb/s) with a CPU usage of 40% on the send side and 100% on the receive size.

The scalability of our method is hard to demonstrate, since our newer cluster has only 5 nodes and our older cluster does not support a MTU of 9000. On the 32-node cluster, it clearly does not scale, since for volume rendering on a 1024×768 rendering area (with $fps = 59.8$), the obtained frame rates are respectively 13, 6 and 3 frames per second with 5, 9 and 17 nodes, compared to 31 frames per second on our 5-node cluster. However, our theoretical analysis of the work balancing between CPU(s), GPU and network gives us reasons to be optimistic.

Compared to other existing works described in Section 1.2, our performances on the 1.5 years old 5-node cluster are clearly better, even for hardware compositing solutions. The two only exceptions are the compositing performance of 24 frames per second reported in [10] and the 45 frames per second reported in [4]; however, when taking into account the rendering time, their frame rates drop respectively to 6 and 8 frames per second, which is way below to our results.

Let us now compare to the theoretical optimal rendering speed computed for our cluster and shown in Figure 7. We recall that the

plotted curves correspond to a non optimized (no pipelining) sort-last volume renderer running on an ideal cluster, not taking into account the rendering time ($render = 0$ in Equation 23). The upper bound of the obtainable frame rates for a 5-node cluster are respectively of 16 and 13 frames per second for a 1024^2 and a 1280×1024 rendering area. Our best results as shown by Figure 9 report best performance of respectively 31 and 19 frames per second for the same rendering areas (with $render = 1/60$ in Equation 23). This definitely proves the efficiency of our pipelined algorithm.

5 CONCLUSION

We have presented a new pipelined sort-last parallel rendering method that fully overlaps CPU(s), GPU and network usage at every level. By overlapping the different computations and communications, we have clearly reduced the overhead of parallel image compositing. Our best results show volume rendering at constant 31 frames per second and polygon rendering at constant 17 frames per second on a 1024×768 render area.

As part of our future work, we want to benefit both from our detailed analysis of sort-last rendering methods and our experimentations to specify and build a new 16-node graphics cluster, that should be able to prove the scalability of our method. Experimentations on other research groups graphics clusters would also be welcome. Future work also include the implementation of dynamic load balancing [23], SLIC image compositing [28] and volume compression [29].

ACKNOWLEDGMENTS

We would like to thank Joe Kniss for providing the volume rendering code used in our experimentations. We also want to thank to the anonymous reviewers for their helpful comments. This work is supported by grants from the Inria and the Region Lorraine (Pôle de Recherche Scientifique et Technologique “Intelligence Logicielle”/CRVHP).

REFERENCES

[1] AHRENS, J., AND PAINTER, J. Efficient sort-last rendering using compression-based image compositing. In *Proc. of the 2nd Eurographics Work. on Parallel Graphics and Visualization* (1998).

[2] FRANK, S., AND KAUFMAN, A. Massive volume rendering on a volume visualization cluster. Tech. rep., Stony Brook University, 2004.

[3] HEIRICH, A., AND MOLL, L. Scalable distributed visualization using off-the-shelf components. In *Proc. of the 1999 IEEE Symp. on Parallel Visualization and Graphics* (1999).

[4] HOUSTON, M. Designing graphics clusters. Parallel Rendering Work. - IEEE Vis 2004, 2004.

[5] HSU, W. M. Segmented ray casting for data parallel volume rendering. In *Proc. of the 1993 Symp. on Parallel Rendering* (1993).

[6] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proc. of SIGGRAPH '02* (2002).

[7] INTEL. *AGP V3.0 Interface Specification*, 1.0 ed., September 2002.

[8] KIRIHATA, Y., LEIGH, J., XIONG, C., AND MURATA, T. A sort-last rendering system over an optical backplane. In *CITSA 2004* (2004).

[9] LEE, T.-Y., RAGHAVENDRA, C. S., AND NICHOLAS, J. B. Image composition schemes for sort-last polygon rendering on 2d mesh multicomputers. *IEEE Transactions on Visualization and Computer Graphics 2*, 3 (1996).

[10] LOMBEYDA, S., MOLL, L., SHAND, M., BREEN, D., AND HEIRICH, A. Scalable interactive volume rendering using off-the-shelf components. In *Proc. of the IEEE 2001 Symp. on parallel and large-data visualization and graphics* (2001).

[11] MA, K.-L., PAINTER, J. S., HANSEN, C. D., AND KROGH, M. F. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl. 14*, 4 (1994).

[12] MOLL, L., SHAND, M., AND HEIRICH, A. Sepia: Scalable 3D compositing using PCI Pamette. In *Proc. of the Seventh Annual IEEE Symp. on Field-Programmable Custom Computing Machines* (1999).

[13] MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (1994).

[14] MORELAND, K., AND THOMPSON, D. From cluster to wall with VTK. In *Proc. of IEEE 2003 Symp. on Parallel and Large-Data Visualization and Graphics* (2003).

[15] MORELAND, K., WYLIE, B., AND PAVLAKOS, C. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proc. of IEEE 2001 Symp. on Parallel and Large-Data Visualization and Graphics* (2001).

[16] MURAKI, S., OGATA, M., MA, K.-L., KOSHIZUKA, K., KAJIHARA, K., LIU, X., NAGANO, Y., AND SHIMOKAWA, K. Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In *Proc. of the 2001 ACM/IEEE conf. on Supercomputing* (2001).

[17] Netpipe, 2003. `http://www.scl.ameslab.gov/Projects/NetPIPE/`.

[18] NEUMANN, U. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *Proc. of the 1993 Symp. on Parallel Rendering* (1993).

[19] NONAKA, J., KUKIMOTO, N., SAKAMOTO, N., HAZAMA, H., WATASHIBA, Y., AND LIU, X. Hybrid hardware-accelerated image composition for sort-last parallel rendering on graphics clusters with commodity image compositor. In *IEEE/SIGGRAPH Symp. on Volume Visualization and Graphics 2004* (2004).

[20] *nVIDIA Developer General FAQ*. `http://developer.nvidia.com/object/General_FAQ.html`.

[21] Raptor. `http://spire.stanford.edu/raptor/`.

[22] REINHARD, E., AND HANSEN, C. A comparison of parallel compositing techniques on shared memory architectures. In *Proc. of the Third Eurographics Work. on Parallel Graphics and Visualisation* (2000).

[23] SAMANTA, R., FUNKHOUSER, T., AND LI, K. Parallel rendering with k-way replication. In *Proc. of the IEEE 2001 Symp. on Parallel and Large-data Visualization and Graphics* (2001).

[24] SANO, K., KOBAYASHI, Y., AND NAKAMURA, T. Differential coding scheme for efficient parallel image composition on a PC cluster system. *Parallel Comput. 30*, 2 (2004).

[25] Scientific/engineering visualization collaboration (Sepia). `http://www.hp.com/techservers/hpccn/sci_vis/index.html`.

[26] SPIRE. `http://spire.stanford.edu/`.

[27] STOLL, G., ELDRIDGE, M., PATTERSON, D., WEBB, A., BERMAN, S., LEVY, R., CAYWOOD, C., TAVEIRA, M., HUNT, S., AND HANRAHAN, P. Lightning-2: a high-performance display subsystem for PC clusters. In *Proc. of SIGGRAPH '01* (2001).

[28] STOMPEL, A., MA, K.-L., LUM, E. B., AHRENS, J., AND PATCHETT, J. SLIC: Scheduled linear image compositing for parallel volume rendering. In *IEEE Symp. on Parallel and Large-Data Visualization and Graphics 2003* (2003).

[29] STRENGERT, M., MAGALLON, M., WEISKOPF, D., GUTHE, S., AND ERTL, T. Hierarchical visualization and compression of large volume datasets using GPU clusters. In *Eurographics Symp. on Parallel Graphics and Visualization* (2004).

[30] TAKEUCHI, A., INO, F., AND HAGIHARA, K. An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Comput. 29*, 11-12 (2003).

[31] YANG, D.-L., YU, J.-C., AND CHUNG, Y.-C. Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *J. Supercomput. 18*, 2 (2001).

[32] ZHANG, X., BAJAJ, C., AND BLANKE, W. Scalable isosurface visualization of massive datasets on cots clusters. In *Proc. of the IEEE 2001 Symp. on Parallel and Large-data Visualization and Graphics* (2001).