Fourth Eurographics Workshop on Parallel Graphics and Visualization (2002)
D. Bartz, X. Pueyo, E. Reinhard (Editors)

The Parallelization of the Perspective Shear-Warp Volume Rendering Algorithm

Jürgen P. Schulze† and Ulrich Lang†

High Performance Computing Center Stuttgart (HLRS), Germany

Abstract

The shear-warp algorithm for volume rendering is among the fastest volume rendering algorithms. It is an object-order algorithm, based on the idea of the factorization of the view matrix into a 3D shear and a 2D warp component. Thus, the compositing can be done in sheared object space, which allows the algorithm to take advantage of data locality. Although the idea of a perspective projection shear-warp algorithm is not new, it is not widely used. That may be because it is slower than the parallel projection algorithm and often slower than hardware supported approaches.

In this paper, we present a new parallelized version of the perspective shear-warp algorithm. The parallelized algorithm was designed for distributed memory machines using MPI. The new algorithm takes advantage of the idea that the warp can be done in most computers' graphics hardware very fast, so that the remote parallel computer only needs to do the compositing. Our algorithm uses this idea to do the compositing on the remote machine, which transfers the resulting 2D intermediate image to the actual display machine. Even though the display machine could be a moderately equipped PC or laptop computer, it can be used to display complex volumetric data, provided there is a network connection to a high performance parallel computer. Furthermore, remote rendering could be used to drive virtual environments, which typically require perspective projection and high frame rates for stereo projection and multiple screens.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture/Parallel Processing; I.3.12 [Computer Graphics]: Graphics Systems/Distributed and Network Graphics; I.4.10 [Image Processing and Computer Vision]: Image Representation/Volumetric.

1. Introduction

Although interactive volume rendering today is mostly done with specialized computer graphics hardware, which is usually high end graphics equipment with fast 3D texturing and large texture memory, this technique has its limitations. In today's graphics workstations, a maximum size of 128 MB is available for texture data. Larger volumes have to be swapped in and out of texture memory, which prevents interactivity. But for software based volume rendering approaches, single workstations are not fast enough to display large volume datasets interactively.

Another bottleneck of the texture based approach is the pixel fill rate. It currently does not suffice to reach interactive frame rates on a 1000^2 pixels screen. Display screens of this resolution are quite common in virtual environments, which are the motivation for the developments presented in this paper. In many installations, a large visualization machine drives multiple display screens with stereoscopic images to create the effect of immersion. The two most widely used approaches to drive virtual reality environments are high-end multi-pipe graphics machines or networked PCs. Networked PCs suffer from the same limitations for volume rendering as single graphics workstations, and the current high-end hardware does not provide that much additional functionality to justify its cost, at least in the field of volume rendering.

In the recent past, clusters of off-the-shelf personal computers have gained importance in the field of parallel computing. These clusters are usually linked with Fast Ethernet

† Allmandring 30, 70550 Stuttgart, Germany
Email: {schulze|lang}@hlrs.de

or Myrinet, both of which provide high bandwidth and low latency. Many clusters are competitive to massively parallel machines. Due to their relatively low price, they are sometimes installed in non-central places, for instance directly in departments at a university, which had to share a much more expensive machine with many other departments before. This de-centralization of parallel computing power increases the chances of getting interactive compute time on a parallel architecture for volume rendering.

The better availability of interactive nodes on parallel computers makes it worthwhile to think about using them for volume rendering in connection with a visualization machine which provides the functionality of driving multiple displays in stereo. The shear-warp algorithm is a very fast algorithm, which does not rely on special graphics hardware, and it was shown that it scales well on parallel computers for the case of parallel projection [5].

The shear-warp algorithm processes volume data arranged on regular grids. Its idea is to factorize the viewing matrix into a 3D shear and scale, and a 2D warp component. It was proved that the projection can be done before the warp [11]. After applying the shear and scale matrices, the volume slices are projected and composited to a 2D sheared image. The shear step enables the algorithm to operate in object space with high memory locality, which optimizes the usage of RAM caching mechanisms and other hardware accelerations. Since the warp can be performed in 2-space, the computational complexity is decreased considerably, compared to a 3-space operation.

2. Previous Work

The fastest implementation of the parallel projection shear-warp volume rendering algorithm was done by Lacroute [6]. He also derived the perspective projection algorithm, but never presented an implementation. This was done later in [11]. Algorithms based on the shear-warp factorization have often been compared to hardware accelerated volume rendering techniques, such as general purpose graphics boards with texturing acceleration [1], or specialized volume rendering hardware [4, 8, 7]. In [7] the idea of a texture hardware supported warp is applied to the parallel projection shear-warp algorithm.

Although on single processor machines the shear-warp algorithm is usually slower than hardware supported solutions, the good scalability of the shear-warp algorithm allows it to be competitive on multiprocessor machines. The first parallelization of the parallel projection algorithm was presented in [5].

Standard PC graphics hardware can be used for volume rendering directly. Even for the case that only 2D texturing hardware is available, Rezk-Salama et al. [10] describe an approach to generate high quality volume images. Westermann and Ertl [12] describe improvements for texture based volume rendering. Compared to the shear-warp approach described in this paper, these approaches require specific OpenGL extensions which are not part of the OpenGL standard, or they are limited by the size of the texture memory. Furthermore, all of them lack the flexibility of a software-only approach, like an arbitrary number of light sources. Using them with active stereo multi-display virtual environments requires graphics drivers that support genlocking.

One of the most recent developments in the field of using clusters for visualization is the WireGL [2] library, which acts as an OpenGL driver to an application but distributes the data which is to be displayed among a cluster of PCs. Due to the large amount of data that has to be transferred for each frame before it can be displayed, the Chromium library [3] was developed. For volume rendering, it allows the distribution of the volume dataset among all cluster nodes, each node rendering only its assigned partition. The drawback of this approach is that it requires a cluster of PCs with graphics cards, while for the volume rendering approach presented in this paper a PC cluster without graphics hardware, or a massively parallel high performance computer can be used. These systems are typically acquired for simulations in science and engineering.

3. The Rendering System

The development of the parallelized perspective projection shear-warp algorithm is based on our work in [11]. We used the object oriented Virvo volume renderer which was well suited as a framework for the required parallel processing extensions. Especially useful was the plug-in mechanism, which allowed us to add a remote renderer to the existing local rendering algorithms.

The extensions had to be done in two areas: first, the perspective projection algorithm had to be parallelized, and second, a new remote renderer had to be written, which runs on a parallel machine and communicates with the local display machine via a network connection (see Figure 1). The network connection is established directly between the renderer plugin and the root node of the parallel computer.

3.1. The Parallelized Shear-Warp Algorithm

In [5], Lacroute parallelizes both the compositing and the warp. The compositing is parallelized by partitioning the object space into sections of intermediate image scanlines, which are distributed among the available processors. Additionally, dynamic task stealing is supported for better load balancing. The warp is parallelized using static interleaved partitions without dynamic approaches.

So far, our algorithm only parallelizes the compositing, but not the warp. The warp was not parallelized because, as shown in [11], it can be done very efficiently in graphics hardware, even if only 2D texturing is supported. If 2D texturing

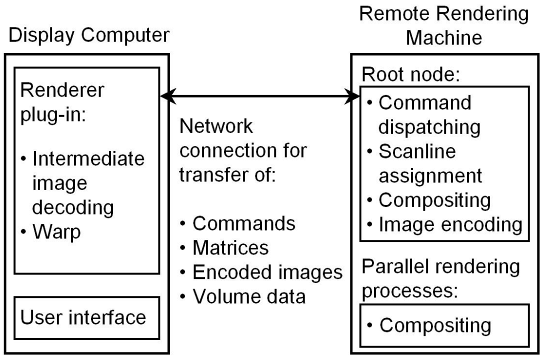

Figure 1: *Remote rendering system components.*

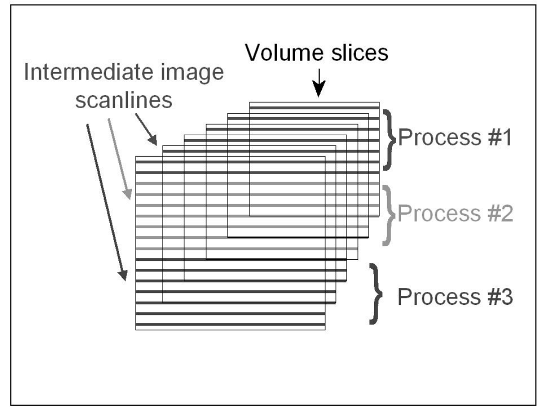

Figure 2: *Intermediate image task distribution with sections of the same size.*

acceleration is not supported by the display computer, the warp can still be done fast for small output images, but the overall performance degrades considerably for large output images. In this case the warp could be done on the parallel computer and the final image could be sent to the display machine.

The compositing was parallelized by partitioning the intermediate image into sections of scanlines, similar to Lacroute's approach, but task stealing was not implemented yet. The idea is illustrated in Figure 2. Each process is assigned an equally sized section of the intermediate image. If the scanlines cannot be distributed evenly, the root node is the first to be assigned less lines than the other nodes, because it has to do the additional work of collecting all rendered sections and sending the result to the display machine.

For perspective projection, the compositing is more expensive than for parallel projection, because every intermediate image scanline does not only require data from two voxel lines, like in the case of parallel projection. It needs to look at multiple voxel lines, depending on the degree of the perspective. In extreme cases, a single intermediate image pixel might even have to process an entire voxel slice from the back of the volume to be computed correctly. In general, the further away the slice that is currently processed, the more voxels have to be accumulated for an intermediate image pixel. This does not necessarily affect rendering speed, because less pixels have to be drawn per slice.

This feature of the perspective projection, and the fact that shear-warp rendering expects three datasets in memory, sorted by the coordinate axes, prevents the distribution of the volume data on distributed memory machines. Each node must have a copy of the entire volume dataset. If a large number of nodes are available, but memory is short, the only reasonable memory distribution would be to split the available nodes into three parts, one for each principal axis. Although the maximum usable volume size would be three times as high, this also means that only one third of the nodes can be used for rendering at a time.

3.2. The Plug-In

Since the intermediate image generation is decoupled from the actual drawing of the final image, the rendering plug-in for the existing volume rendering software is fairly simple. All it has to do is to pass the current view matrix to the remote renderer, wait for the intermediate image, and warp the image to the screen. Of course, all changes of image generation parameters also have to be passed to the remote renderer. For instance, these can be transfer functions, interpolation mode, or image quality.

The rendering plug-in does not have to know anything about the compositing, but it requires the respective warp matrix for every intermediate image it receives.

3.3. The Remote Renderer

At startup, the remote renderer must first receive the volume data. Depending on the volume size and the network connection, this can take a few seconds. Then the three RLE encoded versions of the volume data (one for each principal axis) are generated and stored on each node. After that, the remote renderer is ready to receive commands from the renderer plug-in.

The following pseudo-code shows the flow of control for the root node and the other nodes in the parallel algorithm. The root node both distributes the commands and collects the resulting intermediate image sections. The reception is done by an MPI_Recv() command with the correct memory address for the destination of the sections, so no additional copying is necessary. When all sections have arrived at the root node, the intermediate image is RLE encoded and then transferred to the renderer plug-in, along with the respective warp matrix.

procedure rootNodeRenderingLoop()

```
{
    Receive the view matrix from the plug-in().
    Compute the appropriate section partitioning().
    Pass the section partition parameters to the other nodes.
    Render own section.
    Receive the rendered sections from the other nodes.
    Encode the intermediate image.
    Transfer the intermediate image to the plug-in.
}

procedure otherNodesRenderingLoop()
{
    Receive section parameters from the root node.
    Render the section.
    Transfer the rendered section to the root node.
}
```

The remote renderer is a batch mode program with no direct user interaction after startup. This was an important requirement, because the renderer should run on as many different platforms as possible, even if there was no X Window support. In addition to the number of processes which is passed to the MPI startup tool, the remote renderer expects two command line parameters: the port number and the display host address for the socket connection. Everything else is transferred from the display host.

3.4. Data Transfer

All data communication between the renderer plug-in and the remote renderer is done with one bidirectional TCP socket connection. It is established at startup and lasts until the application is closed. The TCP connection turned out to be fast enough for our purposes, because the bottleneck is the compositing on the remote machine.

When the parallel projection shear-warp algorithm is used, the intermediate image pixels are usually mapped to voxels 1:1. This can be done because the slices are only sheared and not scaled. In the case of perspective projection, the additional scaling makes the slices smaller the further back they are. Thus, we use more than one pixel per voxel for the front volume slice. This ensures that the smaller slices map to enough pixels on the image, so that enough detail can be retained.

For this reason, the intermediate images for perspective projection are larger than for parallel projection. Furthermore we constrain the intermediate image size to edge lengths of powers of two, so the warp can be done without resizing the image - this is a 2D texturing hardware requirement. Typical 1024^2 pixel RGBA images require 4 megabytes (MB) of memory. An interactive frame rate of 10 frames per second would require a data transfer rate of 40 MB per second, which is far beyond the bandwidth of Fast Ethernet (100 Mbit/s).

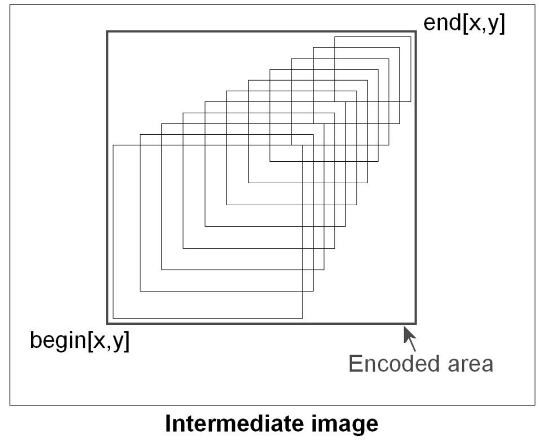

Figure 3: *Encoding of actually used intermediate image window.*

Fortunately, the intermediate image usually contains large transparent regions, which can efficiently be run length encoded (RLE). We implemented two RLE algorithms: the first algorithm encodes the entire intermediate image, the second encodes only the rectangular window which was actually touched in the compositing step, see Figure 3. It turned out that for large window sizes the first algorithm is faster, but in most cases the second algorithm is faster. You will find some performance numbers in section 4.3.

An important issue with the compression algorithm was to make sure that no memory is unnecessarily copied, allocated or deallocated in the process of encoding and decoding. This goal was reached by not reallocating memory space when the intermediate image size remains the same or becomes smaller. Only for images larger than the allocated space a reallocation is done. Furthermore, the intermediate image data is stored only once, so just a pointer to it is passed among the functions that work with it.

3.5. Overall Algorithm

The overall message flow for the rendering of one frame is shown in Figure 4. It is important to note that the display computer does not have to keep the volume data in memory. When the volume is transferred to the remote renderer upon startup, this could be done directly from disk.

3.6. Rendering Front-End

Figure 5 shows a picture of the desktop front-end. Various parameters can be set in the application. The most important are image quality (i.e., intermediate image size), interpolation mode (bilinear or nearest neighbor), and the color and opacity transfer functions.

The front-end is a hybrid C++ and Java application using

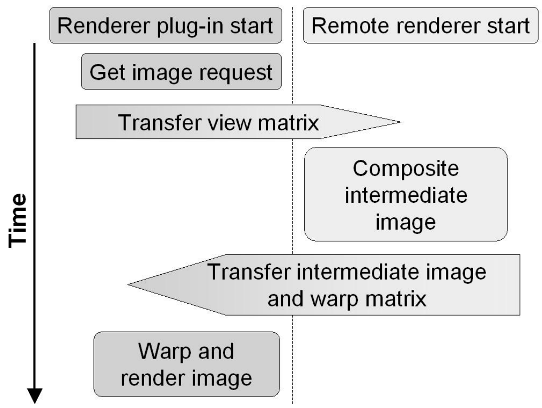

Figure 4: *The remote rendering message flow.*

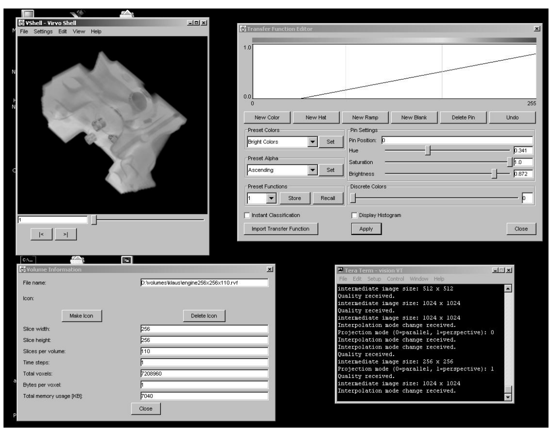

Figure 5: *The rendering front-end with the engine dataset.*

the Java native interface (JNI). The user interface was entirely programmed in Java, using the Swing widget library. Everything else like rendering, network communication, and file handling was written in C++. The rendering window is a Java canvas of which the C++ part knows the OpenGL handle so it can draw on it. The input device handling is done by Java routines that call the appropriate C++ routines if the action happened in the OpenGL canvas.

4. Results

The parallelized perspective projection rendering algorithm was tested on the following three systems: The first system is an SGI Onyx2 with 16 195 MHz R10000 processors and 16 GB RAM. The second system is a SUN Fire 6800 node with 24 UltraSparc III 750 MHz processors and 96 GB RAM. Up to 8 processors on the SUN system are available for interactive use. These two systems have shared memory architectures. The third system is a cluster of 32 Linux PCs with 64 Pentium4 2.4 GHz processors and Myrinet links. Obviously, for the shared memory machines the algorithm could have been written in OpenMP or with threads. But since it had to run on both architectures, we chose MPI.

The display machine is an SGI Onyx2 with 4 250 MHz R10000 processors, 4 GB RAM and Infinite Reality 2 graphics. It is linked to the above Onyx2 by a 1 Gbit/s Ethernet connection and to the PC cluster by a 100 Mbit/s Ethernet. Both Onyxes and the PC cluster are located in the same building at HLRS. The SUN is located about 100 km away in the city of Ulm, and it is connected to the display machine by a 100 Mbit/s Ethernet connection.

The dataset which was used to test the performance of the parallelized algorithm is the General Electric CT engine (see dataset in Figure 5). It was used in two different sizes: "large" is a 256x256x110 voxels version, "small" is a 128x128x55 voxels version which was created by downsampling the large engine. The opacity transfer function was set to a linear ramp from zero to full opacity. The image generation was performed in a 24 bit RGB color space. Whenever the large engine was used, the intermediate image size was 1024^2, for the small engine it was 512^2 pixels. The intermediate image was transferred using RLE encoding only for the actually used window.

For all tests the volume was rotated 180 degrees about its vertical axis. The rotation was done in 90 steps of 2 degrees, the rendering times were accumulated over all 90 steps.

4.1. Overall Rendering Performance

In the following three subsections, the rendering performance of our multi-processing test platforms is displayed. For each graph the remote renderer was executed with increasing numbers of processes. The initialization of the MPI environment ensured that each process could run exclusively on its own processor. The length of the bars reflects the entire rendering time needed for the above described 90 steps rotation test. The sections of the bars display how the total rendering time was distributed to specific tasks. The idle time of the renderers is for the most part the time the display machine needed to decode the intermediate image, transfer it to texture memory, and display it on the screen. During this time the renderer waits for the next view matrix. In all three performance tests, image decoding took about 2.6 seconds and drawing took 1.4 seconds. Idle times that occur due to processes waiting during compositing are included in the total compositing time. In each of the three performance tests the large engine dataset was used.

4.1.1. SUN Fire

Figure 6 shows the rendering performance of the SUN Fire. The compositing step takes most of the total time, while image encoding and image transfer both account only for very little time: encoding takes 0.91 seconds and the transfer takes 0.89 seconds.

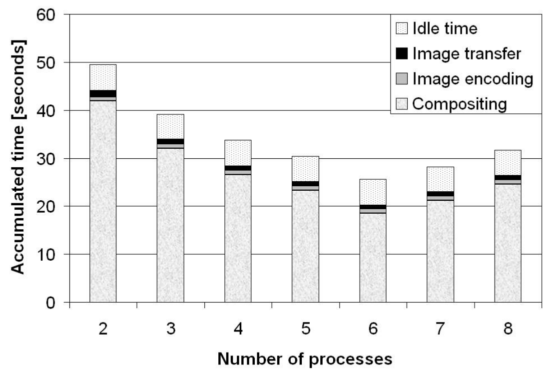

Figure 6: *SUN Fire rendering performance.*

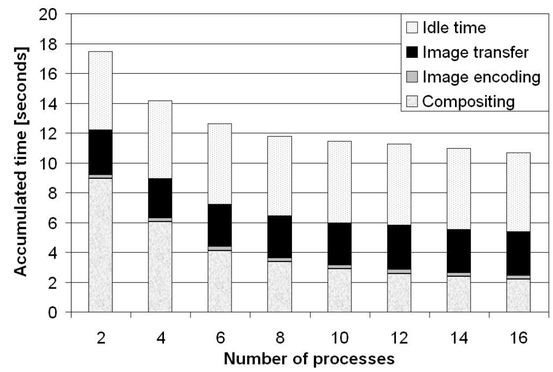

Figure 8: *PC cluster rendering performance.*

4.1.2. Onyx2

Figure 7 shows the rendering performance of the SGI Onyx2 system. Due to the fast network connection to the display machine, the image transfer takes less than 0.2 seconds in all the tests and is hardly visible in the diagram. Image encoding takes about 2.67 seconds.

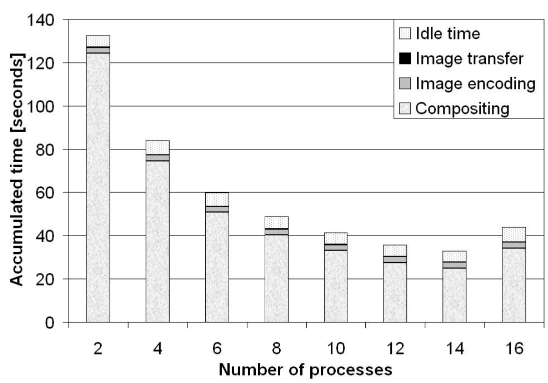

Figure 7: *SGI Onyx2 rendering performance.*

4.1.3. PC Cluster

The rendering performance of the PC cluster is displayed in Figure 8. It differs significantly from the previous two machines. The PC cluster's computing power makes it the fastest tested machine with a minimum rendering time of 10.7 seconds for 90 frames. Furthermore, the algorithm seems not to have reached its maximum performance with the tested 16 processes, although it might be very close to that point. Image encoding took about 0.27 seconds.

4.2. Compositing

Section 4.1 showed that the compositing is the most time consuming rendering step. This is why it was parallelized. Its performance can be judged by comparing the times of the total compositing, i.e. the time it takes before all processes are done with compositing, with the average compositing time of the processes. With perfect load balancing these values would be equal. Table 1, which reflects the performance of the Onyx2, shows that the numbers are not equal. The first column contains the number of processes used for the compositing. The second column shows the total compositing time, and the third column shows the average time it actually took the processes to composite their sub-tasks. The rightmost number is the result of a division of the latter two numbers, which equals the factor by which the compositing speed would improve if perfect load balancing was reached. Figure 9 is a graphical representation of the compositing times. The figure shows that with perfect load balancing, the algorithm could reach its highest performance with 16 or more processes, while the current implementation is fastest with 14 processes.

# processes	Total comp.	Section comp.	Factor
2	124	108	1.15
4	74.6	54.2	1.38
6	50.8	36.4	1.40
8	40.3	28.2	1.43
10	33.6	23.5	1.43
12	27.6	19.0	1.45
14	24.9	16.8	1.48
16	34.3	16.5	2.08

Table 1: *Accumulated compositing times [seconds].*

4.3. Transferring the Intermediate Image

The comparison of the (non-parallelized) RLE-encoding, transfer, and decoding times for the three implemented encoding types (see Figure 10) shows the great advantage of window encoding, where only the part of the image that was

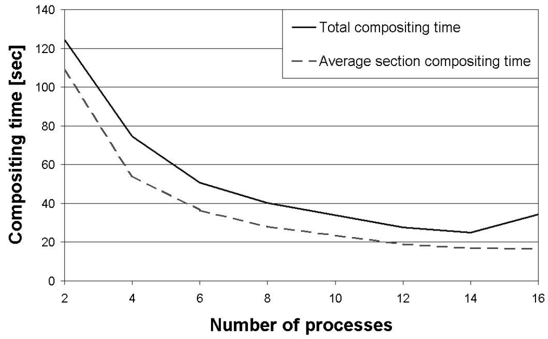

Figure 9: *Total compositing vs. average section compositing.*

actually composited is RLE encoded. In the test, the encoding was done on the SUN Fire, then the image was transferred to the SGI Onyx2, where it was decoded. The encoding and transfer times of window encoding occurred already in Figure 6. For this test, the large engine dataset was used and the intermediate image size was 1024^2 pixels.

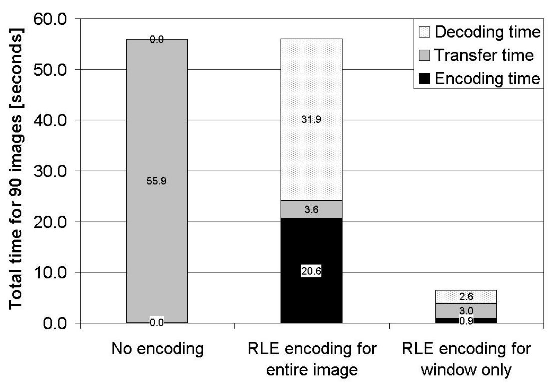

Figure 10: *RLE intermediate image encoding graph.*

4.4. Shear-Warp vs. 3D Texture Hardware

[11] showed that the rendering speed of the shear-warp algorithm is almost independent of the output image size, when the warp is done in texture hardware.

However, the 3D texturing hardware volume rendering approach is highly dependent on the output image size due to its pixel fill rate limitation. In Figure 11, the rendering times for output image sizes from 300^2 to 900^2 pixels are shown for both algorithms, using the small engine dataset. The texture hardware algorithm was used on the Onyx, the perspective shear-warp algorithm was used for the compositing on the SUN Fire using 4 processors and the Onyx did the warp. The graph shows that for an image size of 900^2 pixels, both algorithms are about equally fast.

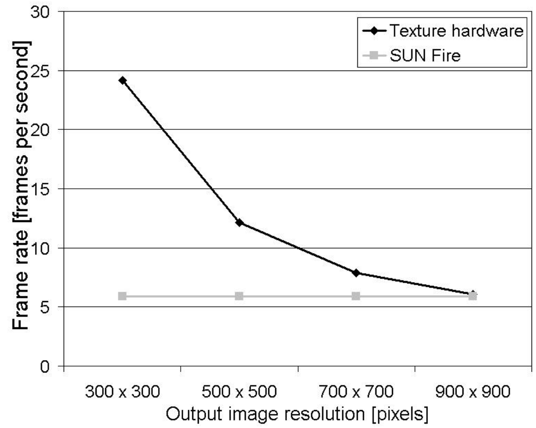

Figure 11: *Texture hardware vs. shear-warp algorithm.*

4.5. Discussion

In this section, the performance numbers from the previous section are discussed, and ideas on how to further improve the performance are given.

4.5.1. Performance Comparison

The fastest rendering rates achieved by each system are listed in Table 2. The PC cluster is fastest with 8.4 images per second. The image transfer rates are similar for the two machines which are linked to the display computer by 100 Mbit/s connections with firewalls in-between. The direct gigabit connection between the two Onyxes pays off, it allows the shortest transfer time in the test. The PC cluster's Pentium4 processors are so much faster than the other two architectures that the compositing is not the dominant factor in the rendering process anymore. Now image transfer and idle time, although roughly the same for the SUN Fire, are the most time consuming parts.

Machine	# processes	images per second
SUN Fire	6	3.5
SGI Onyx2	14	2.7
PC cluster	16	8.4

Table 2: *Maximum rendering speed of the tested machines.*

4.5.2. Idle Time

A comparison of the performance numbers of the three tested systems shows that for the SUN and the SGI, the compositing time dominates, while the PC cluster spends a large fraction of the time transferring the intermediate image to the display machine and waiting for the display machine to send a new view matrix.

While the image transfer time could be reduced by a faster network connection, the idle time could almost entirely vanish with the following idea: as soon as, or even before, the display computer receives the intermediate image, it sends the view matrix for the next image to the rendering system. This would allow the processors to keep busy, since they could start working on the next image right away. This pipelining approach, coupled with asynchronous communication, would allow a better usage of the available compute power.

4.5.3. Image Decoding Time

A significant part of the rendering processes' idle time comes from the display machine decoding the intermediate image. The decoding is not parallelized, since it is not supposed to run on a parallel computer. Our Onyx decodes with a 250 MHz R10000 processor, which should easily be outperformed by current PCs. So we used a Windows PC with a Pentium4 at 1.4 GHz, equipped with a 3Dlabs Wildcat II 5110 as the display computer.

With the Windows PC, the intermediate image decoding time went down from 2.5 seconds on the Onyx to now 0.61 seconds. Looking at the overall performance, it was unexpected that the idle time grew, as it can be seen in Figure 12. Obviously the compositing and image encoding times did not change compared to the previous test in section 4.1.1.

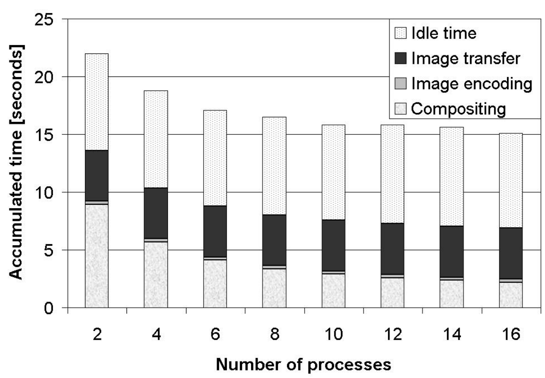

Figure 12: *Windows PC as display machine, PC cluster renders.*

Looking at the performance numbers, it can be seen that the time it takes to draw the intermediate image with texture hardware, which was 1.5 seconds on the Onyx, rose to 7.3 seconds on the PC. This is due to the lower speed of the image transfer to texture memory on the Wildcat.

4.5.4. RLE Encoding

Section 4.3 showed that RLE encoding only the actually used part of the intermediate image before transfer results in the best overall image transfer performance. Interestingly RLE encoding- and decoding the entire image takes about as long as transferring the image unencoded.

The adaptive window approach is generally so much faster than the other two that it can be used for all intermediate image transfers. Only in cases of extreme perspectives, the overhead introduced by skipping parts of the scanlines can become high enough that the other encoding schemes could be faster. However, perspectives like these do not occur in real-life applications.

5. Conclusion and Future Work

We developed an implementation of the perspective projection shear-warp algorithm for parallel computers using MPI. Any architecture which supports MPI can be used as a platform for the remote renderer. The remote rendering process scales well for up to 16 processors, depending on the hardware used. The remotely rendered volume images can be displayed on any graphics capable computer. If 2D graphics hardware is available on the display machine, the warp will be very fast. The transfer speed of the remotely computed intermediate image was optimized.

Lacroute's work [6] showed that dynamic load balancing improves the performance significantly for larger numbers of processors for the case of parallel projection, so this will be done for perspective projection in the future. Furthermore, although not critical for rendering but potentially well parallelizable, some other rendering steps like intermediate image compression and decompression could be addressed for parallelization. Also, parallel image transfer with more than one socket connection could improve the overall performance.

Another goal is to integrate the remote rendering algorithm into our virtual reality environment. The challenge is to efficiently place the socket communication in the rendering pipeline. Our virtual reality renderer COVER [9] is based on SGI Performer. Since we are using a four pipe Onyx2 for rendering, there are four draw processes. A first test showed that we can open four sockets to remote rendering processes, each of which can consist of multiple MPI processes. This promises that we can achieve high scalability, for instance by routing the communication across multiple Gigabit Ethernet connections in parallel.

6. Acknowledgments

This work has been funded by the collaborative research center (SFB) 382 of the German Research Council (DFG).

References

1. K. Akeley. *RealityEngine Graphics*. ACM SIGGRAPH 93 Proceedings, pp. 109–116, 1993.
2. G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. *WireGL: A Scalable Graphics*

System for Clusters. ACM SIGGRAPH 2001 Proceedings, 2001.

3. G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P.D. Kirchner, and J.T. Klosowski. *Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters*. ACM SIGGRAPH 2002 Proceedings, 2002.
4. G. Knittel and W. Strasser. *Vizard - Visualization Accelerator for Real-Time Display*. Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, ACM Press, pp. 139–147, 1997.
5. P. Lacroute. *Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization*. IEEE Parallel Rendering Symposium '95 Proceedings, pp. 15–22, 1995.
6. P. Lacroute and M. Levoy. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. ACM SIGGRAPH 94 Proceedings, pp. 451–457, 1994.
7. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. *The VolumePro Real-Time Ray-Casting System*. ACM SIGGRAPH 99 Proceedings, pp. 251–260, 1999.
8. H. Pfister and A. Kaufman. *Cube-4 - A Scalable Architecture for Real-Time Volume Rendering*. ACM/IEEE Symposium on Volume Visualization '96, pp. 47–54, 1996.
9. D. Rantzau, K. Frank, U. Lang, D. Rainer, and U. Woessner. *COVISE in the CUBE: An Environment for Analyzing Large and Complex Simulation Data*. Proc. 2nd Workshop on Immersive Projection Technology (IPTW '98), Ames, Iowa, 1998.
10. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization*. Proc. Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000 (HWWS00), 2000.
11. J.P. Schulze, R. Niemeier, and U. Lang. *The Perspective Shear-Warp Algorithm in a Virtual Environment*. IEEE Visualization '01 Proceedings, pp. 207–213, 2001.
12. R. Westermann and T. Ertl. *Efficiently Using Graphics Hardware in Volume Rendering Applications*. ACM SIGGRAPH 98 Proceedings, pp. 169–179, 1998.