Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT

Parallel Computing 29 (2003) 339–354

PARALLEL COMPUTING

www.elsevier.com/locate/parco

The parallelized perspective shear-warp algorithm for volume rendering

Jürgen P. Schulze *, Ulrich Lang

High Performance Computing Center Stuttgart (HLRS), Allmandring 30, 70550 Stuttgart, Germany

Received 22 April 2002; accepted 30 November 2002

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 so far. 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 display machine. Even though the display machine can be a mid range 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.

Keywords: Volume rendering; Shear-warp algorithm; Remote computing

* Corresponding author.
E-mail addresses: schulze@hlrs.de (J.P. Schulze), lang@hlrs.de (U. Lang).

0167-8191/02/$ - see front matter
doi:10.1016/S0167-8191(02)00250-8

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 PC graphics cards, a maximum size of 128 megabytes (MB) is available for texture data. Larger volumes have to be swapped in and out of texture memory, which affects 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 is currently not high enough to reach interactive frame rates on a 1024^2 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 enough additional functionality to justify its cost, at least in the field of volume rendering.

In the recent past, clusters of off-the-shelf PCs have gained importance in the field of parallel computing. These clusters are usually linked with Fast Ethernet or Myrinet, both of which provide high bandwidth and low latency. Many clusters are competitive to massively parallel machines. Due to their lower price, they do not need to be installed in central places, but they can be located where they are used, for instance in a department of a university. This de-centralization of parallel computing power increases the possibilities of getting interactive compute time on a parallel architecture for volume rendering.

The availability of large enough numbers 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 need 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 two dimensions, the computational complexity is decreased considerably, compared to a 3D 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 on 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 [4,7,8]. 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 multi-processor 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 [13] 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 or clipping planes.

A previous development for volume rendering on a parallel computer is VFleet [12], which uses a raycasting renderer and a compositing tree, but it does not offer shear-warp rendering. 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. Chromium [3] implemented an improved handling of the large amount of data that has to be transferred for each frame before it can be displayed . 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. The latter systems are typically acquired for science and engineering simulations, but not necessarily volume rendering.

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.

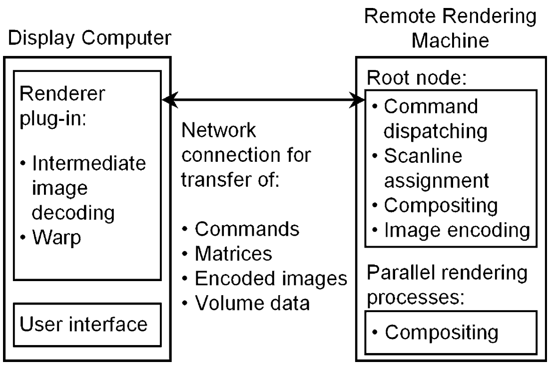

Fig. 1. Remote rendering system components.

The parallel 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 Fig. 1). The network connection is established 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.

Our algorithm only parallelizes the compositing, but not the warp. This is because, as shown in [11], the warp can be done very efficiently in graphics hardware, even if only 2D texturing is supported. If 2D texturing 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 without task stealing. The idea is illustrated in Fig. 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, an entire voxel slice from the back of the volume has to be processed to compute a single intermediate

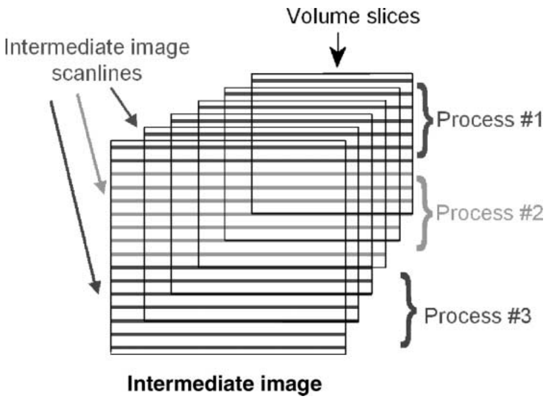

Fig. 2. Intermediate image task distribution with sections of the same size.

image pixel. 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 requires the storage of three datasets in memory, one for each principal axis, 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 is available, but there is not enough memory on each node to store the volume data, the available nodes could be split into three parts and get one volume dataset 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 de-coupled 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 pass the current view matrix to the remote renderer, wait for the intermediate image, and warp the image to the screen. Additionally, all changes of image generation parameters have to be passed to the remote renderer, which includes, for instance, transfer functions, interpolation mode, and 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 may take a few seconds. Then the three run length encoded (RLE) 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 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 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 self-assigned 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 is 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 1:1 to voxels. 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 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).

Fortunately, the intermediate image usually contains large transparent regions, which can efficiently be RLE encoded. 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 Fig. 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 de-allocated 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 decreases. A reallocation is done only for images larger than the allocated space. Furthermore, the intermediate image data is stored only once, so just a pointer to it is passed among the functions that work with it.

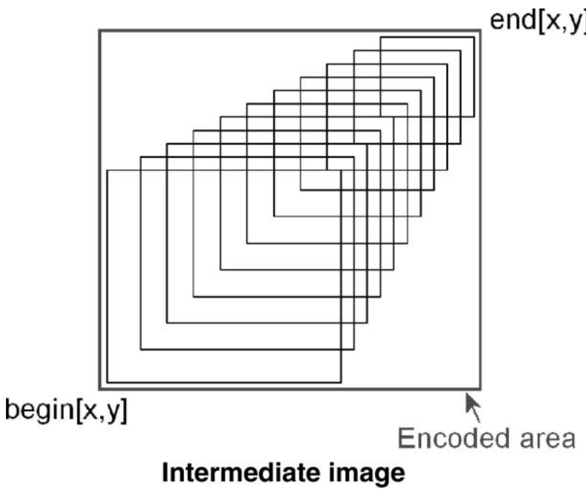

Fig. 3. Encoding of actually used intermediate image window.

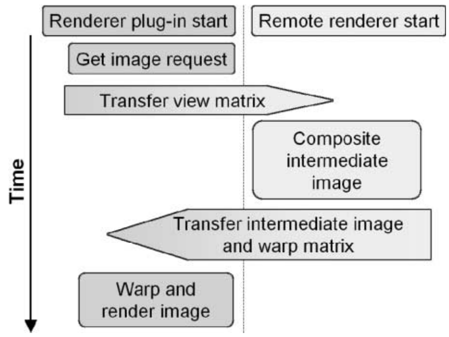

Fig. 4. The remote rendering message flow.

3.5. Overall algorithm

The overall message flow for the rendering of one frame is shown in Fig. 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 can be done directly from disk.

3.6. Rendering front-end

Fig. 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),

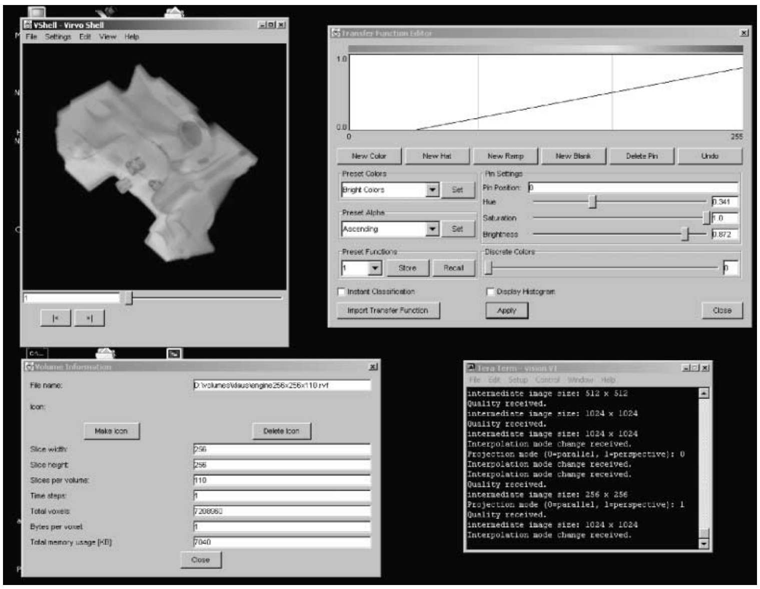

Fig. 5. The rendering front-end with the engine dataset.

interpolation mode (bilinear or nearest neighbor), and the color and opacity transfer functions.

The front-end is a hybrid C++ and Java application using the Java native interface. 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 parallel machines:

- An SGI Onyx2 with 16 195 MHz R10000 processors and 16 GB of shared memory.
- A SUN Fire 6800 node with 24 UltraSparc III 750 MHz processors and 96 GB of shared memory. Up to eight processors are available for interactive use.
- A cluster of 32 Linux PCs with 64 Pentium 4 Xeon processors at 2.4 GHz and Myrinet links.

Apparently, for the shared memory machines the algorithm could have been written in OpenMP or with thread support. But since the program had to run on any distributed architecture, we use MPI.

The display machine is an SGI Onyx2 with 4 R10000 processors at 250 MHz, 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.

The dataset which was used to test the performance of the parallelized algorithm is the General Electric CT engine (see dataset in Fig. 5). It was used in two different sizes: "large" is a $256 \times 256 \times 110$ voxels version, "small" is a $128 \times 128 \times 55$ voxels downsampled version. The opacity transfer function was set to a linear ramp from zero to full opacity. The image generation was performed in 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 for the actually used window only.

For all tests the volume was rotated 180° about its vertical axis in 90 steps of 2°.

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 different numbers of processes, each process running on a separate CPU. The initialization of the MPI environment ensured that each process could run exclusively on its own processor. The length of the bars reflects the average rendering time per frame needed for the above described 180° rotation. 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 29 milliseconds (ms) and drawing took 16 ms for each frame. 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

Fig. 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 9.9 ms and the transfer takes 11.1 ms.

4.1.2. Onyx2

Fig. 7 shows the rendering performance of the SGI Onyx2 system. Due to the fast network connection to the display machine, the image transfer takes only 1.7 ms in all the tests and is hardly visible in the diagram. Image encoding takes 29.2 ms.

4.1.3. PC cluster

The rendering performance of the PC cluster is displayed in Fig. 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 132 ms per frame. Image encoding took 3.0 ms, the image transfer accounts for 31.4 ms. The rather

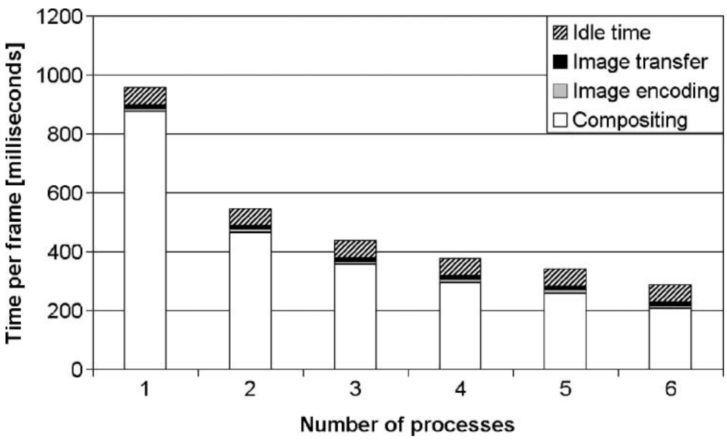

Fig. 6. SUN Fire rendering performance.

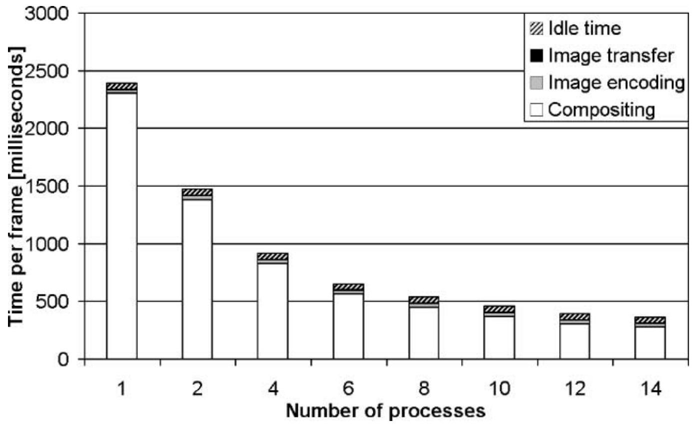

Fig. 7. SGI Onyx2 rendering performance.

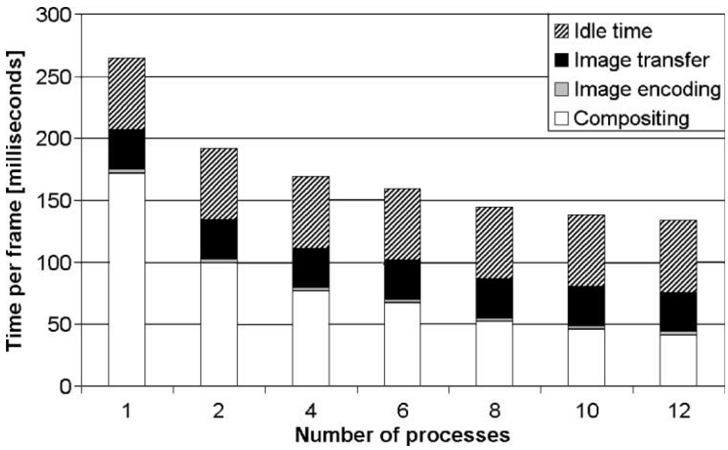

Fig. 8. PC cluster rendering performance.

slow transfer is due to the endianness adaptation that is required between the PCs and the SGI.

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. Fig. 9, which reflects the performance of the Onyx2, shows that the numbers are not equal. The solid line shows the total compositing time, and the dotted line shows the average time it actually took the

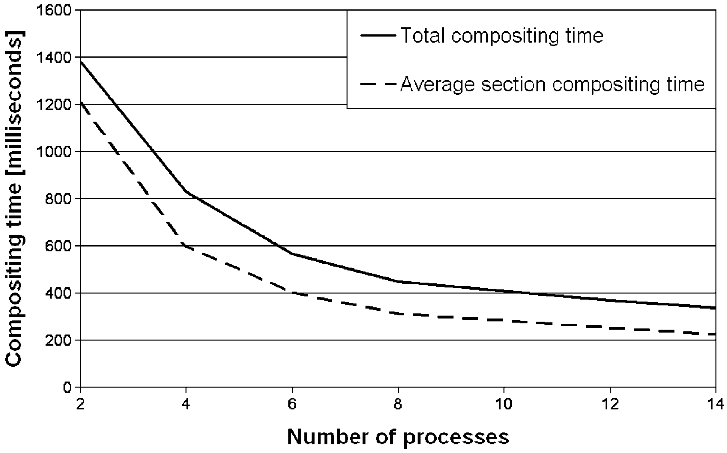

Fig. 9. Total compositing vs. average section compositing.

processes to composite their sub-tasks. The space between the lines reflects the improvement that would be possible by optimum load balancing.

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 Fig. 10) shows the great advantage of window encoding, where only the part of the image that was 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. For this test, the large engine dataset was used and the intermediate image size was 1024^2 pixels.

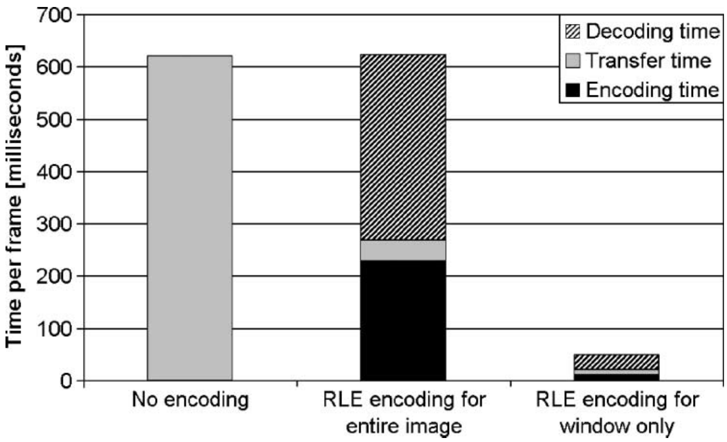

Fig. 10. RLE intermediate image encoding graph.

4.4. Shear-warp vs. 3D texture hardware

In [11] we 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 Fig. 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 four 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.

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 1. 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. Here

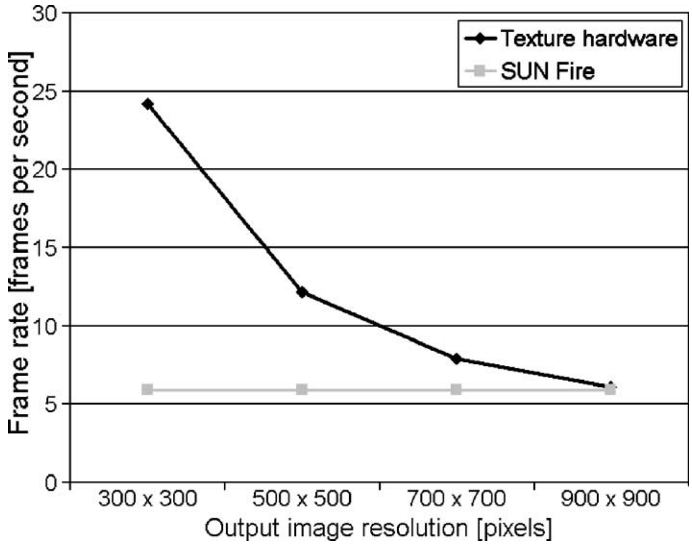

Fig. 11. Texture hardware vs. shear-warp algorithm.

Table 1
Maximum rendering speed of the tested machines

Machine	Number of processes	Images per second
SUN Fire	6	3.5
SGI Onyx2	14	2.7
PC cluster	16	8.4

image transfer and idle time, although roughly the same for the SUN Fire, are the most time consuming parts.

4.5.2. Latency hiding

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 can be used to begin the computation of the next image: as soon as, the display computer receives the intermediate image, it sends the view matrix for the next image to the rendering system. This pipelining approach leads to effective latency hiding, and it was implemented for optional use. However, for the performance tests in this paper, no pipelining was used in order to show the actual time usage of the system.

4.5.3. Image decoding time

A significant part of the rendering processes' idle time results in the display machine decoding the intermediate image. The decoding is not parallelized, since it usually does not run on a parallel computer. Our Onyx decodes with a 250 MHz R10000 processor, which is easily outperformed by current PCs. In another test we used a Windows PC as the display computer. It contains a Pentium4 at 1.4 GHz, and a 3Dlabs Wildcat II 5110 graphics board.

With this PC, the intermediate image decoding time went down from 28 ms on the Onyx to now 6.8 ms. Looking at the overall performance, it is remarkable that the idle time grew, as it can be seen in Fig. 12. Obviously the compositing and image encoding times did not change compared to the previous test in Section 4.1.1.

Looking at the performance numbers, it can be seen that the time it takes to draw the intermediate image with texture hardware, which was 17 ms on the Onyx, increased to 81 ms 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 of 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.

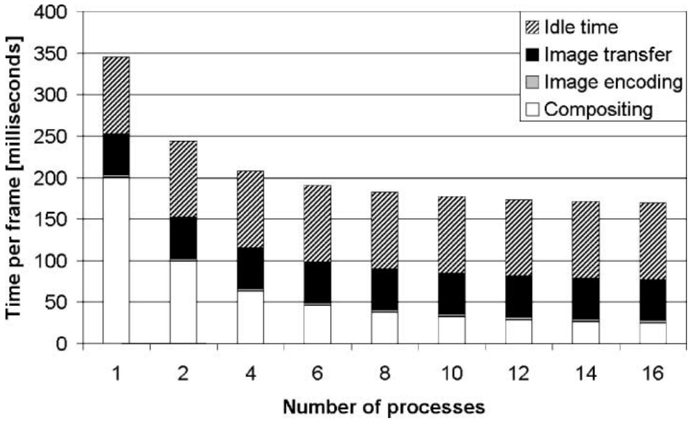

Fig. 12. Windows PC as display machine, PC cluster renders.

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 can be faster. However, perspectives like these occur only rarely 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 12 processors in our experiments, 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 in 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 de-compression should 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.

Acknowledgements

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

References

[1] K. Akeley, Reality engine graphics, in: ACM SIGGRAPH 93 Proceedings, 1993, pp. 109–116.
[2] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, P. Hanrahan, WireGL: a scalable graphics system for clusters, in: ACM SIGGRAPH 2001 Proceedings, 2001.
[3] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P.D. Kirchner, J.T. Klosowski, Chromium: a stream-processing framework for interactive rendering on clusters, in: ACM SIGGRAPH 2002 Proceedings, 2002.
[4] G. Knittel, W. Strasser, Vizard—visualization accelerator for real-time display, in: Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware, ACM Press, 1997, pp. 139–147.
[5] P. Lacroute, Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization, in: IEEE Parallel Rendering Symposium '95 Proceedings, 1995, pp. 15–22.
[6] P. Lacroute, M. Levoy, Fast volume rendering using a shear-warp factorization of the viewing transformation, in: ACM SIGGRAPH '94 Proceedings, 1994, pp. 451–457.
[7] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, L. Seiler, The VolumePro Real-Time Ray-Casting System, in: ACM SIGGRAPH '99 Proceedings, 1999, pp. 251–260.
[8] H. Pfister, A. Kaufman, Cube-4—a scalable architecture for real-time volume rendering, in: ACM/IEEE Symposium on Volume Visualization '96, 1996, pp. 47–54.
[9] D. Rantzau, K. Frank, U. Lang, D. Rainer, U. Woessner, COVISE in the CUBE: an environment for analyzing large and complex simulation data, in: Proceedings of 2nd Workshop on Immersive Projection Technology (IPTW '98), Ames, Iowa, 1998.
[10] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl, Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization, in: Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware 2000 (HWWS00), 2000.
[11] J.P. Schulze, R. Niemeier, U. Lang, The perspective shear-warp algorithm in a virtual environment, in: IEEE Visualization '01 Proceedings, 2001, pp. 207–213.
[12] J. Welling, VFleet, Available at: http://www.psc.edu/Packages/VFleet_Home/
[13] R. Westermann, T. Ertl, Efficiently using graphics hardware in volume rendering applications, in: ACM SIGGRAPH '98 Proceedings, 1998, pp. 169–179.