

# Parallel Direct Volume Rendering on PC Networks

Michael Meißner<sup>1</sup>, Tobias Hüttner<sup>1</sup>, Wolfgang Blochinger<sup>2</sup> and Andreas Weber<sup>2</sup>  
WSI, University of Tübingen, Germany

<sup>1</sup>[meissner,thuettne]@gris.uni-tuebingen.de

<sup>2</sup>[blochinger,weber]@informatik.uni-tuebingen.de

**Abstract** *Volume rendering is becoming a key technology in the field of scientific computing, medical imaging and many other fields with non-invasive scanner technology. The process of generating images from 3D volumetric data is highly demanding and pushes systems to their computational limits. Many approaches have been presented in the past using networks of workstations to distribute the workload. In contrast, we present parallel direct volume rendering on a network of PCs. To enable communication between processes running on different machines we use an object oriented thread library, called DOTS. This library enables the user to rapidly parallelize its code without getting involved in network issues.*

**Keywords:** Volume Rendering, Parallel Computing, Adaptive Sampling

## 1 Introduction

Volume rendering puts high demands on systems which claim to be capable of rendering data sets at reasonable frame-rates. Typical sizes of volume data sets range from  $128^3$  to  $512^3$  voxel elements each holding values from one to four bytes. Hence, the overall size of such data sets can be up to 512 MBytes. To obtain high image quality all this data has to be processed. It is obvious that this is far beyond nowadays desktop computers. For larger data sets, even supercomputers cannot provide enough power to achieve real-time frame rates.

We can classify the previously done work into three categories. The first makes use of

supercomputers or massive parallel architectures [1] [2] [3]. These architectures are not widely accessible and not suited for the average user. The second category includes numerous special purpose hardware, aiming to provide the required computational power [4] [5] [6] [7]. Many of those architectures are still in experimental stage, have certain limitations, or will be too expensive for the average user. Furthermore, it is not clear how some of these architectures handle oversampling, high image quality, and perspective projection. Finally, the third category uses the widely available computing power on networks and aims to make this power usable for volume visualization [8] [9] [10] [11].

Our parallel direct volume rendering (DVR) implementation belongs to the latter category. Besides earlier presented approaches which use networks of workstation we exploit the more widely available resources of PCs. Our approach addresses high image quality volume rendering of large data sets which includes oversampling. Unfortunately, oversampling is computation intense and therefore, we apply oversampling such that only few areas are sampled at cost intense high sampling rate while other areas are rendered at a much lower sampling rate. This adaption scheme is precisely the way, the perception of the human eye works.

The remaining challenges are to overcome the limited transfer bandwidth of the network, to establish an even workload, to avoid eventually arising stall conditions, and to effectively exploit the limited computational power. In

this context it is important to mention that most of the physicists, chemists and other scientists working in the field of scientific visualization are not familiar with parallelization schemes. Ideally, they would like to simply run their code on many machines in parallel. Data exchange over the network involves knowledge in the underlying structure and paradigms. To keep this knowledge highly abstract, we use an object oriented and distributed thread library which removes the burden of distributing and starting remote tasks. Hereby, data which has to be exchanged between machines can simply be marshaled using the serialization mechanism (<< and >> operator of C++). The simplicity of this library makes it more suitable for our purpose than other packages.

In Section 2 we briefly survey the DOTS package which we use to distribute the tasks. Our direct volume rendering algorithm (DVR) is presented in Section 3. Section 4 illustrates the parallelization of the algorithm, the load balancing, and the task scheduling. Finally, we present some results achieved with our implementation.

## 2 The Distributed Object-Oriented Threads System DOTS

DOTS [12] is a programming environment for the parallelization of irregular and highly data-dependent algorithms. It extends support for *fork/join* parallel programming from shared memory threads to a distributed memory environment. DOTS works on the principle that a forked thread can be executed on a remote machine provided its input and output parameters can be copied over the network and it does not engage in shared memory communication. This programming paradigm can be characterized as SPMD (*single program, multiple data*) with asynchronous remote procedure calls.

In DOTS, each network node is a multi-threaded SMP (possibly consisting of a single processor). Under DOTS, the network again forms a symmetric multiprocessor. To-

gether they form a hierarchical multiprocessor. The idea is to support a uniform parallelization paradigm on this machine. Large grained threads are to be distributed over the network, small grained ones over the processors in a shared memory multiprocessor. Since each network node is to be over-saturated with threads, communications latency is hidden implicitly by running another thread.

DOTS is an object-oriented redesign of the Distributed Threads System (DTS) [13] which was based on PVM [14]. The main problem of the use of PVM on the DTS system has been that all of the software bulk and the instabilities and non-portabilities of PVM resulted in a corresponding problem of DTS. Additionally, the way how the DTS copying functions for parameter (un-) marshalling had to be provided by the user was felt to be somewhat user unfriendly by many developers outside of computer science who used DTS to distribute their scientific simulation computations.

Therefore, a redesign of the system resulting in DOTS was necessary. DOTS is no longer based on PVM but its present implementation uses the *Adaptive Communication Environment* (ACE) toolkit [15], a C++ class library. ACE is a system-independent, object-oriented platform for developing communication software. There are implementations of ACE for all major UNIX systems (e.g. Solaris 2.x, AIX, SGI IRIX), for Windows NT (Win32) and for MVS Open-Edition. The substitution of PVM by ACE lead to the following advantages:

- ACE libraries are linked whereas PVM is started separately. So code is not bloated by unneeded functionality.
- The platform independence of ACE makes DOTS available on a wide range of operating systems and hardware platforms. Currently we are using implementations of DOTS on Solaris 2.x, IRIX 6.x and on Windows NT.
- Performance measurements presented in [16] show that the additional abstraction

layers of ACE do not cause any significant performance loss compared with directly using operating system calls.

- ACE’s class encapsulation of the C based APIs of the different operating systems substantially improves the type-safety of the implementation. This can prevent the occurrence of run time errors and therefore improves the correctness and software quality.

In our newly designed DOTS system, archive classes are provided for the serialization and deserialization of objects. The serialization of objects that will be distributed over the network is encapsulated in the class definition, and is thus compatible with the class abstractions provided by C++.

The system is configured using an ASCII file containing all machines which are used for parallelization, the number of threads to be forked on each machine, and the path of the file to be executed remotely. Additionally, a remote shell daemon must be running on each machine listed in this file. It is used by DOTS during its system initialization phase. Using *dts\_fork()* and *dts\_join()*, processes can be forked and results received.

### 3 DVR Algorithm

Volume rendering algorithms can be divided into object space algorithms and image space algorithms. Our algorithm belongs to the latter category. The view plane position is calculated using given rotation and zoom parameters. Once the view plane is located, rays are casted from the eye point through each pixel of the view plane into the view frustum. Figure 1 illustrates this process. To detect whether a ray hits the volume data or not, we use a three dimensional Cohen-Sutherland algorithm. Rays which miss the volume can be discarded. Otherwise samples along the ray are taken by interpolating values of the neighboring voxels. We use trilinear interpolation of the eight neighboring voxels. Additionally, for

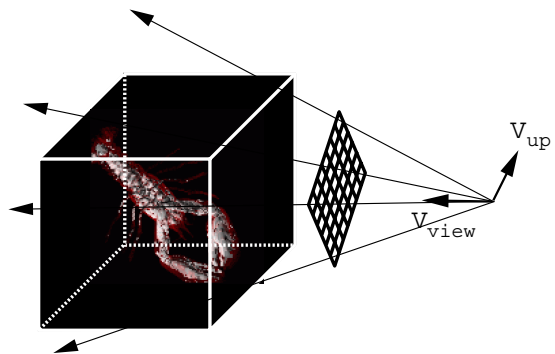


Figure 1: Rays are casted from the view plane into the view frustum.

each sample a gradient is computed out of the gradients at neighboring voxel locations. Calculating the gradient from neighboring samples [6] would lead to sampling dependent gradients which would require further treatment of the gradients. Unfortunately, the trilinear interpolations needed for gradient components and sample generation cannot be circumvented without compromising image quality. Since we aim for high image quality with strong level of detail, trilinear interpolations are unavoidable.

Classification is the stage of assigning a RGBA component to a sample [17] [18]. This is done using a look-up table containing the RGBA values for all voxels. After classification, each colored sample has to be shaded corresponding to its gradient. We apply Phong shading [19] resulting in high image quality.

To obtain a final pixel on the view plane, shaded samples along each ray have to be composed to a single pixel value. This is done by composing the values along a ray in a front-to-back or back-to-front order.

### 4 Parallelization

In this section we will briefly survey the parallelization of our DVR algorithm, the implemented load-balancing and the scheduling of the tasks.

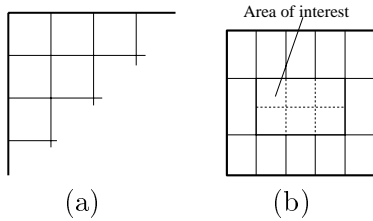


Figure 2: Possible tiling of the view plane: (a) Regular tiling, (b) irregular tiling scheme.

### Parallelization

The algorithm described in Section 3 has to be parallelized to exploit the available resources on the network. To keep network traffic low we chose an image space subdivision scheme. The view plane is subdivided into tiles which can be calculated independently. Hence, on each client machine the dataset resides within main memory. The resulting image tiles have to be combined into a single image by the master thread. Fortunately, this is only an operation in 2D and does not represent a bottle-neck.

For each tile, necessary information, i.e., eye point, view direction, light sources, etc. is provided. We combine this data into a *Tile-Data* object. Each thread is then performing the direct volume rendering algorithm as described in Section 3 on a single tile. As a result we obtain a *Tile-Image* object from the slave containing the rendered image of the tile. *Tile-Data* and *Tile-Image* are implemented as classes including serialization and deserialization from an archive which is send over the network by DOTS.

The tiling scheme is not limited to a regular tile structure, as illustrated in Figure 2(b). Non-regular tiling simply requires further treatment due to the imbalanced work-load of each tile. Furthermore, we allow varying sampling rates and possibly different transfer functions for each tile.

Tiles rendered at lower sampling rate have to be scaled to fit together with the other tiles. Neighboring tiles rendered at different sampling rates can produce visible artifacts along their border. To prevent this, sampling rates of

neighboring tiles should not vary by more than a factor of two. Additionally, applying bilinear interpolation to the finale image to double its size reduces the visibility of those artifacts to an absolute minimum.

### Load Balancing

To possibly balance the work-load, we calculate a cost factor for each tile. This factor is determined by multiplying the number of rays casted within a tile by the maximum amount of samples which can be taken along a ray. Unfortunately, not all rays leave the volume through the opposite face of the volume from where they entered it. Even worse, for perspective projection or fly throughs many rays will be composed of fewer samples. Therefore, for each tile the time share needed for computation is stored. During calculation of the cost factor for each tile of the next frame, we use this value to correct the estimated cost factor. As long as the point of view only changes incrementally, this scheme fulfills our needs. Still, to prevent an imbalanced work-load, the cost factor of each tile should not be larger than twice the cost factor of the smallest cost factor. Therefore, we split *large* tiles and merge *small* tiles, as illustrated in Figure 2(b).

### Tile Scheduling

DOTS is currently under development and the so far existing system does not allow the user to fork threads on a specific host. So far, the scheduling of tasks is done by DOTS itself. Therefore, scheduling strategies can only be incorporated to a certain limit.

We implemented three different strategies to schedule tiles. The first simply forks all threads and waits for their completion. Once all tiles are done, all results are composed into the final image and finally new threads are forked. We refer to this strategy as number I. Strategy II first saturates all clients before composing the results of the last frame into the final image. Hence, idle time on the clients inbetween frames is reduced.

Unfortunately, both presented strategies can stall once one of the machines delays the re-

sults of a single tile. This can be caused by a high load on a machine or even worse by a system crash. Therefore, we introduce strategy III which uses a redundant self-scheduling (RSS) scheme similar to [8]. Tasks are organized in a list and the largest tasks are at the head of the list. Initially as many tasks are forked as there are available machines. A task is not removed from the task list until its completion. For each completed task, a new task is forked from the list in round robin order. Once all tasks have been forked but yet not all are finished, the uncompleted tasks are redundantly re-forked unless all tiles are completed. The major drawback of RSS is the high redundancy introduced by this scheme. In the worst-case, all machines render the same tile.

To prevent this high over-head, we propose a *reliability factor RF* which is configurable by the user. The amount of tasks which might get lost or delayed within the network are assumed not to be larger than *RF*. Hence, once the master starts to redundantly redistribute tasks, the redistribution occurs only up to a maximum of *RF* times. This greatly reduces the redundancy and frees computing capacity within the network by still preventing stall conditions. Figure 3 illustrates the communication scheme. Ideally, strategy III should be combined with a frame interleaved scheduling scheme such that idle machines start calculating tiles of the consecutive frame. This has yet not been implemented.

## 5 Results

The data set chosen is a  $128^3$  down-sampled lobster residing inside a block of resin. Density values of resin range from zero through 40, for the shell from 40 through 80, and for the skin over 80. Figure 4 shows a high quality image of this dataset.

### 5.1 Adaptive Sampling

As mention in Section 1, we apply adaptive sampling rates. Ideally, the incorporation of

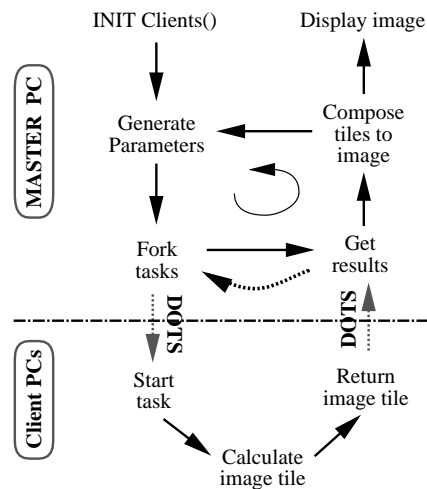


Figure 3: Communication between master and clients.

eye-tracking into the system would greatly enhance the perceived image quality by sampling the area of gaze higher than other areas of the view plane. Unfortunately, this requires real-time frame rates which we do not yet achieve. So far, this can only be simulated by defining an area of gaze. Hence, the possibly irregular tiles and their lower sampling rate are automatically generated. Figure 5 shows an image rendered using different sampling rates.

The acceleration gained by applying adaptive sampling rates depends on the users selection, the area of gaze. Assuming that only four tiles of 16 are rendered at two-folded sampling rate, the computation can be reduced to one third since two-folded oversampling in each dimension requires eight-folded computation. Since the user influences the sampling rates, it is hard to make a quantitative estimation of the real speed-up gained by using adaptive sampling.

### 5.2 Parallelization Speed-up

The achieved speed-up for the image shown in Figure 4 has been linear due to the granularity of the problem. Rendering time is about 15 minutes per frame on a Pentium Pro (200 MHz). To get a better analysis of the com-



Figure 4:  $128^3$  data set of a lobster residing in a block of resin, rendered with 512 by 512 rays using parallel projection.

munication and network overhead, we downsized the problem by reducing the amount of casted rays to  $128^2$  and no oversampling in ray direction. Figure 6 illustrates the timing depending on the amount of tiles. Obviously, the number of tiles should not be larger than 32. Smaller tiles simply increase communication overhead, and hence, saturate the system. Figure 7 show the speedup depending on two different granularities of the problem. The speedup in Figure 7(a) is almost linear due to the large size of the tasks. Due to the heterogeneous configuration of our network, the speedup exceeds the number of CPUs used. We use up to four Pentium II running at 300 MHz and five Pentium Pro running at 200 MHz.

Once the granularity is small enough, i.e., we reach almost one frame per second, the speedup drops drastically. Profiling the network traffic has shown that the bandwidth of the network is not the limiting factor. The problem seems to be related to the NT platform and less likely to the DOTS package. Saturating CPUs with more than one thread results in a loss of packages. We will investigate this problem. Once this is removed, we should get within the range of interactivity. In general, scheduling II shows better performance due to the delay-free forking scheme of the threads.

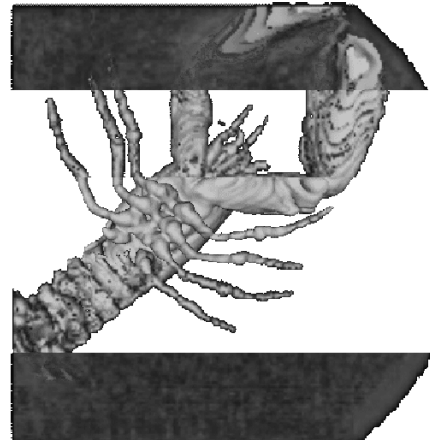


Figure 5: Same data set as in Figure 4 rendered using varying sampling rate and different transfer functions rate. There are  $5 \times 5$  tiles and the middle row is sampled higher than the other rows.

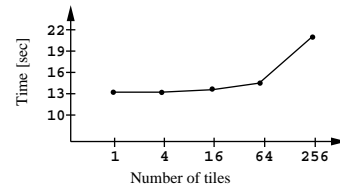


Figure 6: Time per frame casting  $128^2$  rays into a  $128^3$  data set. The results have been averaged over 40 frames and were taken on a Pentium Pro (200 MHz).

Figure 8(a) shows the amount of tiles being redundantly calculated using the RSS as proposed in [8]. It can easily be seen that the redundancy increases the more machines are used. Truly, the percentual redundancy drops by splitting the view plane into smaller tiles. Unfortunately, each tile introduces network traffic and hence, the number of tiles cannot be increased arbitrary, see Figure 6.

In contrast, Figure 8(b) illustrates the reduced redundancy achieved by our strategy III. The RF value mainly influences the amount of redundantly calculated tiles. Fortunately, it turned out that an RF value of two was sufficient for our maximum configuration of

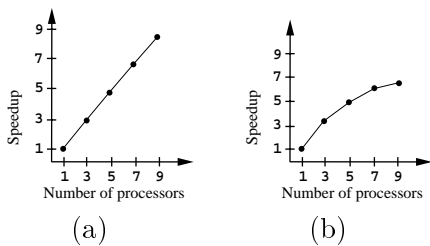


Figure 7: (a)  $256^2$  rays are casted into the  $128^3$  data set applying oversampling in ray direction. (b)  $128^2$  rays are casted without oversampling.

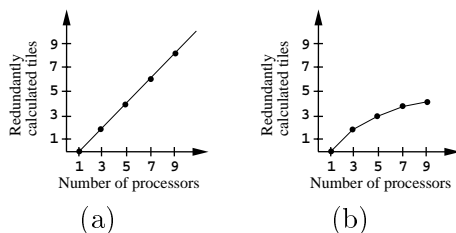


Figure 8: All values are averaged over 20 frames (a) Redundantly calculated tiles using RSS. (b) Same as (a), using RSS with varying RF.

ten machines. In general, our proposed minimal RSS using a  $RF$  value reduces the amount of redundantly calculated tiles to  $RF * Num\_Blocked\_Slaves$ . This is essentially less over-head than produced by RSS. Usually, only a very small number of clients is temporarily blocked.

## 6 Conclusions

We presented a parallel DVR algorithm being capable of rendering large data sets using adaptive sampling rates providing high image quality and strong level of detail. An image space based parallelization scheme of the algorithm, keeping the network traffic low, has been presented.

Furthermore, we presented different scheduling strategies and their timing results. On a network of PCs we achieved a substantial

speed-up for our direct volume rendering algorithm reducing the time per frame from the range of minutes to seconds. For real-time DVR, the capacity of the network has not been the limiting factor of our system. The problem seems to be related to NT but we will further investigate this issue.

Our system has been extended to successfully prevent stall conditions, such as the above mentioned loss of packages. Resulting overhead could be reduced to a minimum using a redundant self scheduling scheme with a configurable reliability factor for the network.

Finally, using the object oriented distributed thread system DOTS not only quickened the implementation process, but proved to be easy to use for scientists like physicists, etc.

## 7 Future Work

The current system is not yet optimized for speed. Our main goal is to achieve high image quality which is the most important issue for scientists to understand the details of their simulations. Nevertheless, we will speed-up the implementation by using look-up based gradient generation and shading. This can be done without compromising image quality.

## References

- [1] G. Cameron and P. E. Underhill. Rendering volumetric medical image data on a simd architecture computer. *Proceedings of the Third Eurographics Workshop on Rendering*, 1992.
- [2] P. Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. *Proceedings of the ACM/IEEE '95 Symposium on Parallel Rendering*, pages 15–22, oct 1995.
- [3] T. S. Yoo, U. Neumann, H. Fuchs, S. M. Pizer, J. Rhoades T. Cullip, and R. Whitaker. Direct visualization of volume data. *IEEE Computer Graphics & Applications*, 12(4):63–71, 1992.

- [4] T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Maenner, H.-P. Meinzer, and H.-J. Baur. VIRIM: A massively parallel processor for real-time volume visualization in medicine. In *Proceedings of the 9th Eurographics Hardware Workshop*, pages 103–108, Oslo, Norway, September 1994.
- [5] J. Lichtermann. Design of a fast voxel processor for parallel volume visualization. In *Proceedings of the 10th Eurographics Hardware Workshop*, pages 83–92, Maastricht, The Netherlands, 1995.
- [6] H. Pfister and A. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. *1996 Symposium on Volume Visualization*, October 1996.
- [7] G. Knittel and W. Straßer. Vizard - visualization accelerator for realtime display. In *Proceedings of the 1997 SIGGRAPH/Eurographics Hardware Workshop*, Los Angeles, CA, 1997.
- [8] G. Knittel. A parallel algorithm for scientific visualization. In *Proceedings of the 1996 International Conference on Parallel Processing*, Bloomington, USA, aug 1996.
- [9] C. Giertsen and J. Petersen. Parallel volume rendering on a network of workstations. *IEEE Computer Graphics & Applications*, 13(6):16–23, November 1993.
- [10] K.-L. Ma and J. S. Painter. Parallel volume visualization on workstations. *IEEE Computer Graphics*, 17(1), 1993.
- [11] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh. A data distributed, parallel algorithm for ray-traced volume rendering. In *Proceedings of the 1993 Parallel Rendering Symposium*, pages 15–22, San Jose, CA, October 1993.
- [12] W. Blochinger, W. Kuchlin, and A. Weber. The distributed object-oriented threads system DOTS. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel (IRREGULAR '98)*, Lecture Notes in Computer Science, Berkeley, CA, U.S.A., August 1998. Springer-Verlag. To appear.
- [13] T. Bubeck, M. Hiller, W. Kuchlin, and W. Rosenstiel. Distributed symbolic computation with DTS. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems, 2nd Intl. Workshop, IRREGULAR'95*, volume 980 of *Lecture Notes in Computer Science*, pages 231–248, Lyon, France, September 1995.
- [14] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *Sixth Annual Distributed-Memory Computer Conference*, pages 258–261, Portland, Oregon, May 1991. IEEE.
- [15] Douglas C. Schmidt. The adaptive communication environment: An object-oriented network programming toolkit for developing communication software., 1993. <http://www.cs.wustl.edu/~schmidt/ACE-papers.html>.
- [16] Douglas C. Schmidt. Object-oriented components for high-speed network programming, 1995. <http://www.cs.wustl.edu/~schmidt/ACE-papers.html>.
- [17] R. A. Drebin, L. Carpenter, and P. Harahan. Volume rendering. *Computer Graphics*, 22:65–74, aug 1988.
- [18] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(3):29–37, May 1988.
- [19] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 1990.