Parallel Volume Rendering

Rüdiger Westermann
Scientific Visualization Group
German National Research Center for Computer Science
Sankt Augustin, Germany

Abstract

During the last years serveral methods have been developed to visualize three dimensional volume data on multiprocessor computer systems. Most of these algorithms were strongly adapted to the underlying target architecture, and often did not take account of requirements such as scalability, portability, or memory minimization. The following paper outlines a new approach to rendering large scale volume data on distributed memory parallel computers. Within this framework, the only limiting factor concerning the resolution of the processed data is the node local memory. Furthermore, porting to other multiprocessor systems is straight forward, and scalability with increasing numbers of processors is achieved.

Keywords and Phrases: *volume rendering, massively parallel computer, MIMD*

1 Introduction

The visualization of three dimensional volume data is one of the most time consuming and memory intensive techniques in computer graphics. Although there exist fast back-to-front projection methods (splatting [6][17] or cell-projection [18]) that allow for almost interactive speed on high perfomance single processor workstations, the general front-to-back evaluation of the transport equation integral [5] results in rendering times far from interactivity. This is due to the numerical complexity of the solution process. Only the use of acceleration techniques, taking advantage of the underlying data, overcomes these difficulties in some cases.

A further restriction becoming more and more important is the limited memory capacity of single processor machines. The rapidly increasing resolution of the available data sets makes it almost impossible to process these sets on the whole. With the development of massively parallel supercomputers, the possibility of rendering high resolution volume data in an acceptable amount of time has arrived. Based on specialized multiprocessor architectures, a variety of volume rendering algorithms have been developed during recent years. Most of these algorithms were strongly wedded to particular systems and neither portability onto other architectures nor scalability with increasing number of processors was accounted for. In addition, a rather restricted and memory intensive evaluation of the physically based rendering integral has been applied in many implementations.

1.1 Goals

Since we are interested in a general framework for the visualization of scalar volume data sets on message-passing based, distributed memory multiprocessor architectures, some basic requirements have to be taken into account throughout the implementation:

- portability => message-passing based
- scalability
- load balancing
- memory minimization
- coarse grain parallelization
- communication minimization

To reach these aims, a minimum of machine dependent features is integrated into the presented approach.

2 Volume Rendering Integral

The basic problem one is confronted with in the visualization of three dimensional volume data is the accurate evaluation of the well known volume rendering integral

$$I(t_o, t_1) = \int_{t=t_0}^{t_1} q(t) e^{-\int_{s=t_0}^{t} \sigma(s)ds} dt \qquad (1)$$

where σ(s) defines the attenuation function, $q(t)$ is the volume source term, and t_0 and t_1 are the start and end points on the view ray. Starting at the point of view, the light along a ray scaled by a material dependent attenuation factor is summed up to get the final intensity. The general formulation, which also takes into account multiple and higher order scattering within the volume, along with the lighting effects of external sources, yields a more complex equation. As we are interested in comparing our results with other parallel rendering algorithms based on equation (1), this reduced

formulation will also be used in our implementation. Nevertheless we will show, that a general solution of the complete transport equation integral, considering second and higher order rays, can be easily integrated into our approach. The general solution of equation (1) can be calculated using one of the standard integration rules as instances of the more general Newton-Cotes-Quadrature formulae [15]. Using an Eulerian sum for the evaluation of the outer integral, and assuming the source term and the attenuation factor for a certain segment i as constants q_i and α_i, the integration is reduced to a finite sum over the accumulated opacity

$$I = \sum_{k=1}^{n} q_k \alpha_k \prod_{i=0}^{k-1} (1 - \alpha_i) \qquad (2)$$

which can be evaluated in time linear in the length of the ray. The volume functions, which are defined on the grid points of a regular, three-dimensional, discrete grid, are continued between the grid points using tri-linear or higher order interpolation methods.

To speed up the complex evaluation of the rendering integral, many acceleration techniques are available, taking advantage of the properties of the underlying signal. The most frequently used acceleration techniques, which can now be considered standard, are presence-acceleration, homogeneity-acceleration and α-termination [2][7]. The former takes advantage of a reorganization of the volume data into a pyramid data representation [19], allowing fast traversal of empty or homogeneous subblocks, while the latter stops the integration after a certain attenuation is reached, and the contribution of further values is negligible.

Due to the fact that these acceleration methods are becoming more and more a standard in current implementations, they should be integrated into parallel rendering approaches as well.

2.1 Previous Work

The majority of parallel volume rendering algorithms can be classified into one of the two ray tracing variants:

- object space driven
- image space driven

The former technique is based on a volume data transformation to align the volume coordinate axis with the view coordinate axis, while the latter resamples the volume along the view rays.

Two object space methods, which utilize factorings of rotations into a sequence of one-dimensional shears [3], where proposed by Schröder et al. [13] and Vezina et al. [16]. Both implementations take advantage of fast nearest neighbor communications between processing nodes on the CM2 respectively the Mas-Par MP-1.

Schröder and Stoll [14] introduced another image space method that describes the ray tracing step as discrete line drawings, reducing the communication patterns to slice-wise toroidial shifts along specific axis.

The above mentioned implementations are intimately related to specialized SIMD architectures. As a result the former uses large amounts of memory together with an effective pre-filtering of the data during the shear steps, while the latter exhibits rather poor scalability with increasing number of processors.

Both methods are closely connected to the fast nearest neighbor communication primitives of the underlying network implying fine grain parallelization only.

Hsu [4] proposed a distribution of smaller subvolumes of the original data set to the processing elements of a DECmpp 12000/Sx, where the rendering is performed on each node separately. According to the actual view direction, a front-to-back sorting of the subblocks has to be performed, along with a final global communication step to merge the resulting subpictures.

A high speed implementation on a DASH shared memory architecture was presented by Nieh et al. [12]. Due to the fast memory access, coupled with first and second order level data caches and the utilization of spatial data coherences, impressive results were achieved. In terms of the overall rendering times, a comparison with other parallel implementations appears rather difficult. This is due to the close adaption of the implementation to the specialized shared memory architecture and the hardware supported handling of data cache structures. A drawback of this method, even for the interconnection of large numbers of processors, is the competetive memory access which leads to non optimal scalability. Additionally the hardware does not take advantage of explicit knowledge of the underlying data domain and caches neighboring data segments only.

Montani et al. [10] proposed to replicate the volume data along processor clusters on a MIMD nCUBE architecture, taking advantage of the fast communication primitives between processing elements inside a certain cluster. Each cluster is responsible for a certain region of the screen, processing only the corresponding rays. The bottleneck of this implementation appears to be scalability with the size of the cluster as well as the immense overhead, arising from the replication of the data across clusters.

A fully message passing based algorithm was proposed by Ma et al. [9] on a CM5 MIMD architecture. The subdivision of the volume data into smaller subblocks together with their assignment to different nodes allows for independent rendering of these blocks. Due to the associativity of the *over* operator – used to compute the color and opacity accumulation during the rendering process – the sub pieces of the final screen image, which exist on each node, can be merged into the final image. With respect to the independent but complete rendering of each subblock, the size of the screen tiles is strongly related to the extend of the subblocks. Furthermore, a large region of the whole screen has to be stored, and merged with other subregions on different nodes, to get the final image.

2.2 Basic Ideas

A major drawback of the majority of the mentioned parallel volume rendering algorithms, with the exception of the one proposed by Nieh, is the impossibility to perform a general front-to-back evaluation of the transport integral. In this context, the integration of multiple scattering effects, volume shading, and any of the standard volume rendering acceleration techniques, seems to be rather difficult, if not impossible. Additionally, great effort is required to adapt these implementations to other architectures. Even if this

is possible, the resulting performance will often be unacceptable, due to the close relationship between the applied methods and specialized hard- or software architectures.

The present paper introduces a new general approach to parallel volume rendering on distributed memory multiprocessor systems, which avoids the disadvantages of the previously described methods. Taking into account the growing distribution of massively parallel MIMD systems that are based on the standard message-passing communication primitives, a general approach should fit into this class of systems. In this context it is most important to rid the implementation of any machine dependent features, as well as to adapt the used communication patterns to the characteristics of distributed memory parallel computers.

Instead of traversing the volume based on the original volume elements (voxels), a much coarser resolution is assumed in our implementation, treating subblocks of many voxels as the new volume primitives. These macro voxels are equally distributed over the nodes. Based on the new larger voxel size, fast ray-generators can be applied, determining the subblocks that are hit by a certain view ray. These blocks have to be fetched from other nodes, forcing a global communication step, if they do not exist in the local memory of the requesting node.

A further subdivision of the screen into smaller regions and the utilization of the spatial coherence of neighboring rays within these regions, leads to a reduced number of macro voxels that have to be fetched from other nodes. These macro voxels can be stored in local memory, simulating an explicit data cache structure.

This approach is closely related to the method used by Nieh for parallel volume rendering on a shared memory DASH architecture. The advantage of our method is its portability to other distributed memory multiprocessor systems together with a more rigorous exploitation of data domain knowledge during the rendering process.

In the following the applied data distribution scheme used in our implementation will be introduced, followed by a description of the so called ray-generator process. Some implementation dependent details will be given, and the results of our approach will be discussed.

3 Data Distribution

One of the major goals of our approach is portability onto other distributed memory multiprocessor systems, together with a minimization of the amount of memory needed during the rendering process. In this context a complete and coherence free distribution of the data incurred during the rendering process should be achieved, avoiding dependency of a special machine architecture. To reach the highest possible efficiency, the distribution of the volume data as well as the screen data all over the nodes is performed in our implementation.

3.1 Volume Data Distribution

The volume data is divided into smaller but connected subblocks or macro voxels of equal size. The subblock resolution should be equal for each major axis so as to be independent of arbitrary view directions. To support interpolations and

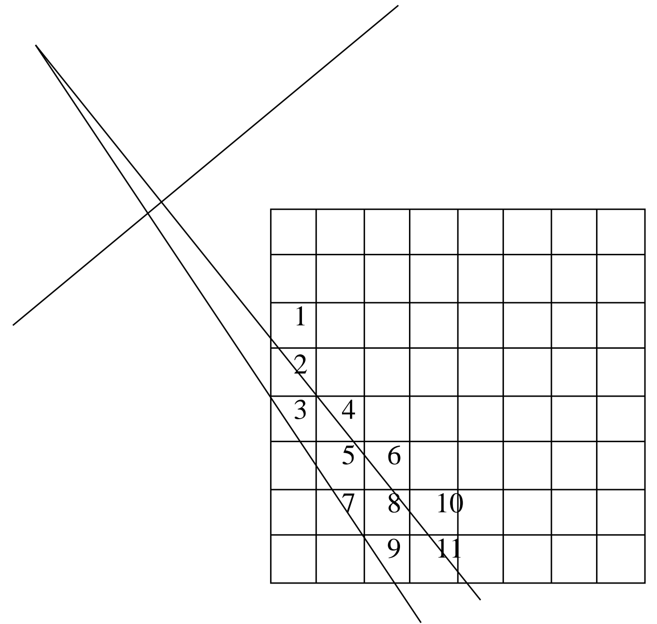

Figure 1: Sequence of subblocks that are traversed while processing a certain screen region

gradient computations, an additional shell with an extent of two voxels has to be stored with each subblock.

In general, more than one block exists on every node depending on the macro voxel resolution. For each of these blocks the integration of the previously mentioned presence- or homogeneity-acceleration can be applied easily, resulting in the separate extraction of empty or homogeneous subregions from within each block. In general, this will force some communication with other nodes, but it has to be done only once as a pre-processing step. Of course this kind of data extension leads to a more memory consuming representation, but speeds up the rendering process to some extent.

If we assume an independent evaluation of the rendering integral for each ray, and each data item has to be fetched from other nodes, the total number of original voxels sent over the network can be calculated as

$$N^2 * k * RES \tag{3}$$

where N is the number of rays, RES the resolution of the original volume data set, and k a view direction dependent factor. This number, not taking into account further voxels needed for interpolations, increases to

$$N^2 * k * \frac{RES}{SUBRES} * SUBRES^3 \tag{4}$$

for our subdivision scheme, where $SUBRES$ is the resolution of the macro voxel. This number is a multiple of the original amount, but this comparison only makes sense, if the communication costs for sending a block of n voxels is equal to the costs for sending n times one voxel. Todays multiprocessor architectures achieve their highest communication rates when transfering large, connected blocks. This is due to startup overhead including address calculations and global routing mechanisms. As a result the somewhat larger

amount of data transmitted in our implementation does not lead to a proportional increase of the overall communication times, taking into account that we transfer large, connected data segments only. From the point of view of distributed multiprocessor systems with local memory, this is the prefered approach to using global communication. Whenever possible, many small block communication steps should be replaced by few large block communication steps. Even for our implementation we find that for high resolution subblocks the communication times are almost negligible, compared to the overall rendering times.

In a typical host-node environment the host reads the data slice by slice or subblock by subblock, and distributes partitions to the nodes on which the corresponding macro voxel "lives". This method turns out to be rather slow but does not depend on a globally accessible file system, from which each node can read his subblocks separately.

Apart from the above, the other main advantage of the described distribution scheme lies in clustering and distributing the screen data. Due to the spatial coherence of neighboring rays and information about the actually fetched subblock, overall communication costs can be minimized.

3.2 Screen Data Distribution

In addition to the volume data layout across the nodes, a subdivision and distribution of the screen data is also performed. It is more correct to talk about task distribution than data distribution in this case, because the pixel data will be generated after the distribution process. This strategy implies the assignment of a certain number of screen regions to each node, which have to compute the intensity distribution for the corresponding pixels.

To obtain optimal load balancing, a non-static distribution strategy for the screen regions has to be found. Assuming that each node processes more than one region, an optimal distribution can be computed in a pre-processing step. Starting with a static assignment, every node shoots one or more rays that are equally distributed over the subscreen areas, through the volume, and sumes up the lengths each ray stays within the volume. Finally, each node carries a list with the accumulated lengths, which can be used as a load estimator for the processing of a given region. Each node sorts his local list in decreasing load order and broadcasts the list along with a specification of the corresponding subscreens to the host. The host merges all incoming lists to obtain an ordererd global load list for all subscreens which have to be processed. At each time step a node sends a request to the host announcing its willingness to process another region. The host sends back the identification of the actual subscreen determined by a pointer into the load list, and increments this pointer by one. Due to the max-to-min order of the list, computationally complex screen regions will be processed earlier, which results in an almost optimal load balancing as a function of the overall number of screen regions and processors.

Based on the described clustering of the view rays, their spatial coherence can be utilized during the rendering process. In many cases, neighboring rays hit the same macro voxel on their way through the volume. The global communication step to fetch these blocks need be done only once. To determine which subblock is hit by a given ray, efficient methods (so called ray-generators) can be applied.

4 Ray-Generators

Based on the described distribution strategies, the front-to-back traversing of the volume data to evaluate the rendering integral has to take place. At any time, every node processes a certain subscreen, storing all necessary information for the corresponding pixel values and view rays respectively. For each pixel the ray from the view point through that pixel as well as the intersection with the volume has to be computed. To obtain the final intensity for a certain pixel, all voxels which are hit consecutively by the corresponding ray have to be determined. The generation of this sequence can be accelerated drastically using so called ray-generators [1] [11], which are based on a uniform subdivision of the traversed space.

Instead of applying the ray-generator process based on the subdivision given by the original data resolution, the virtual macro voxel resolution will be taken into account (see Fig. 2). As a matter of fact, the next hit with a macro voxel can be determined using only additions and comparisons.

4.1 Traversing

For each ray the actually hit subvolume can be extracted according to the described method. In Figure 1 a sequence of subvolumes is outlined, which have to be fetched for a certain screen region. Due to the spatial coherence of rays belonging to the same region, each subvolume that is hit will, in most cases, be traversed by several rays. Once the macro voxel has been fetched, the integration process for all these rays can be performed, avoiding multiple fetch operations for the same subblock. The traversal of the macro voxel can be done using equation (2), interpolating between the original voxel values within this block. If there is a need for different blocks on the same node, only the nearest one in relation to the viewpoint will be fetched. If this block does not exist in the local memory of the node, it will be fetched from another node in a general communication step. For each ray that determines an intersection with the actual subvolume, the traversal of that block takes place and the next cell to traverse will be determined immediately.

The same method works equally well for the integration of higher order rays which might be shoot from a given sample into other directions, taking advantage of the spatial coherence of n-th order rays starting on a certain ray of order n-1. A disadvantage is that the integration of higher order rays generally requires more than one temporary subblock on each node. Also the spatial coherence will be lost with increasing ray order.

Due to the applied image space parallelism coupled with the front-to-back traversal of the volume subblocks, all previously described acceleration techniques can be easily integrated into this approach. Totally empty or homogeneous macro voxels do not have to be sent, and the integration step size can be adapted to the underlying subblock quantities or the accumulated opacity.

Another well known and often used technique is the extraction of iso-surfaces from within the volume [8]. Based on a certain threshold, only regions in which this threshold is hit are extracted and visualized. The visualization of these iso-surfaces can be accomplished by applying different shading models to the surface gradients. The gradient vector for

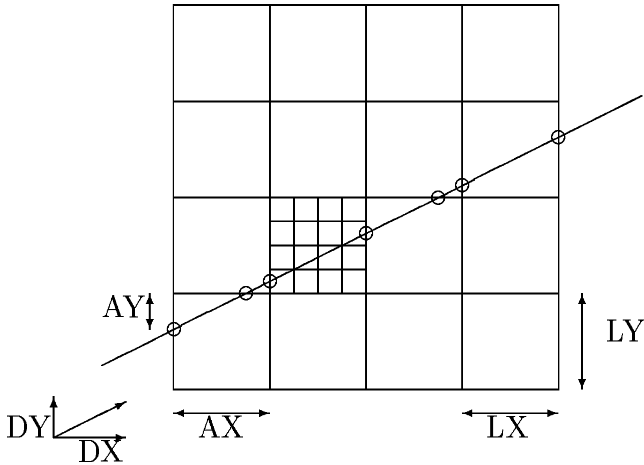

Figure 2: Uniform subdivision and cell traversal process based on macro voxel resolution. LX, LY and LZ are the distances from plane to plane. DX, DY and DZ specify the normalized ray direction. AX, AY and AZ are the distances from the entrance point of the ray to the planes parallel to the axes. Furthermore the distances from the entrance point to the planes on the ray are PX, PY and PZ. Given $TX = \frac{LX}{DX}, TY = \frac{LY}{DY}$ and $TZ = \frac{LZ}{DZ}$, and $PX = \frac{AX}{DX}, PY = \frac{AY}{DY}$ and $PZ = \frac{AZ}{DZ}$, the minimum of the distances PX, PY and PZ is the actual distance to the exit point. The new values of PX, PY and PZ can be computed from their old values and the values of TX, TY and TZ.

a arbitrary location in space can be computed by interpolations between the gradients defined at the discrete voxel values. These gradient vectors $g = (g_x, g_y, g_z)$ for a lattice point (x, y, z) are estimated from the neighboring voxel values using central differences:

$$g_x = f(x-1, y, z) - f(x+1, y, z) \quad (5)$$

$$g_y = f(x, y-1, z) - f(x, y+1, z) \quad (6)$$

$$g_z = f(x, y, z-1) - f(x, y, z+1) \quad (7)$$

If a certain threshold is given, each node can process his local subblocks, checking if a hit with the surface is possible or not. If there is no hit possible, the corresponding subblock can be skipped, and needs never be transferred during the iso-surface extraction process.

5 Memory Requirement vs. Speed-up

Although one major goal of our approach is the memory minimization during the rendering process, even for implementations on loosely connected workstation clusters, provided with large scale local memory, this requirement becomes less important in relation to the overall speed-up. A more memory intensive but faster extension of our approach is the implementation of an explicit data cache structure with adjustable number of cache segments. Before the processing of a certain screen region takes place, each processor computes all macro voxels which will be fetched during the traversal process. While a processor is waiting idle for the transmission of a requested subblock, it sends further requests to other nodes for subblocks that will be needed later. Upon transmission of these blocks, they are stored in the segments of the data cache structure. Each time a given macro voxel is required, the node traverses the cache list first, checking whether this block has already been fetched and transmitted. Of course this method increases the total amount of memory to some extent, but minimizes the overall idle times and prevents bottlenecks due to requests from different nodes for the same subblock. This strategy can be extended efficiently if the data cache structure is large enough to take up all subblocks that will be fetched during the processing of a certain screen region. All requested blocks are stored within the cache segments and can be used during the processing of the next screen region. If the host tries to distribute neighboring screen regions to the same node, a large number of cache segments can be used for the evaluation of the next region. As another possibility the nodes themselves could find and process the next region within their neighborhood. In this case the host would be responsible to store a flag for each screen region, indicating whether this region has already been processed.

The explicit data cache structure reduces the total number of global communication steps, and benefits from the spatial coherence of rays belonging to neighboring screen regions.

6 Implementation

Our approach has been implemented on a 64 node CM5 (TMC) with partitions of 16, 32 or 64 nodes. Each partition is managed by a local partition manager, which is also responsible for the host program in our host-node implementation. Each node is provided with a Sparc 2 processor along with 32 MB of local memory.

The host is responsible for receiving the processed subscreens from the nodes and flushes the evaluated intensity values to a X/Motif window. Additionally, the host broadcasts visualization parameters, that can be changed interactively by the user, to the nodes. All communication was perfomed using the CMMD message-passing library with the restriction to only a few communication primitives, facilitating fast and easy implementation on other message-passing based systems. In this context, the only primitives we used in our implementation were *CMMD-send-async* and *CMMD-receive-async*. The routines perform a non-blocking sent/receive operation.

A node which wants to fetch a subblock from another node, first sends an asynchronous request to the destination node. Two bytes have to be transferred that specify the number of the subblock on the destination node and the definite identification of the source node within the actual partition. Subsequently, the source node has to poll for the transmission of the requested block or for requests from other nodes. If a node receives a request for a certain subblock, an asynchronous send operation with the requested block is performed.

The disadvantage of this protocol is that there have to be distinguished synchronization points, at which the nodes perform the polling to check the network for incoming messages from other nodes. This task is performed at every time a node starts traversal of a new subblock.

7 Results

The implementation has been tested with three data sets consisting of 256^3, 512^3 and 1024^3 8 bit voxel values. All measurements have been performed with a zero length data cache structure, storing only the actual subblock which has to be processed. To show the full scalability of the presented method, the implementation has been tested on all available partitions of the CM5. Tables 1-7 show the speed up due to the increasing number of processors. To be most flexible, the resolutions for both the macro voxels (SB) and the screen regions (SS) have been changed independently. To streamline the behavior of the proposed method in more detail, all partitions have been compared using a fine to coarse grain parallelization. For each partition the rendering times as well as the communication times for different resolutions have been measured. The final times are average times for a 360 degree rotation with a 5 degree increment around the y-axis.

It turns out that a larger subblock resolution leads to shorter rendering times in general, profiting from the fact that only a few hits with the macro voxel have to be determined. With constant subblock resolution, the overall rendering times decrease with growing resolution of the screen regions. This depends on the implementation of the ray-generator which takes advantage of the spatial coherence of the rays, and the fact that fewer subscreens have to be processed on every node. In the case of much larger subscreen extents than subblock extents, the rendering times grow rapidly, because more and more coherence between neighboring rays is lost. We see the opposite behavior for much larger subblock extents than subscreen extents. In this case the rendering times decrease due to the spatial coherence. Nevertheless, this leads to an increase of the communication times. Large subblocks together with relatively small subscreens result in a large number of screen areas on each node, for which the blocks have to be fetched independently. On the other hand a smaller subscreen size takes better advantage of the spatial coherence and for a given region fewer subvolumes have to be fetched.

Due to the complete and coherence free distribution of all the used data segments, the only limiting factor for the resolution of the processed volume data sets is the local available memory on every node. Even for a 16 nodes partition, this allows us to render data sets up to 512^3 (see Table 1-2).

The full adjustability of either the subvolume size or the subscreen size enables us to determine precisely the optimal resolutions in relation to the underlying target architecture. For larger subvolumes, the amount of memory transferred during the traversing process increases rapidly, while neighboring rays are much more able to take advantage of their spatial coherence. On the other hand, larger subscreens change the behavior to the opposite, also forcing frequent communication steps in addition to decreasing subblock sizes.

This adjustibility makes the implementation fully adaptive to other system architectures, taking advantage of the special structure of the nodes as well as a given interconnection network. Using high performance nodes with large local memory but rather poor transfer rates favors increasing subblock sizes. For computationally weak nodes, which are connection via high performance networks, a decrease of the segment resolution leads to better performance.

We also tested the explicit data cache structure, and adjusted the number of cache segments to take up all subblocks which are traversed for a given screen region. The overall rendering times remained constant, but the communication times decraesed approximately 20 - 30 %, due to the minimized global fetch operations and idle times.

8 Conclusion

A fully scalable and less memory intensive approach for the visualization of three-dimensional data sets on distributed memory multiprocessor systems has been presented. Due to the applied data distribution schemes, only the locally available memory on each node limits the resolution of the volume to be visualized. In addition to the distribution of the original data set across the nodes, only one further memory segment has to be allocated, to store the subblock that is actually fetched from another node. Based on the straight forward front-to-back processing together with the subdivision of the original data set into connected subblocks, a general evaluation of the volume rendering integral can be performed. Furthermore, many acceleration techniques based on subblock quantities can be applied.

The flexibility to accommodate the macro voxel resolution as well as the subscreen resolution, allows one to switch between less or more communication intensive strategies, based on a used architecture.

Due to the fact, that only a few standard message passing primitives have been used, the presented approach is fully portable onto other message passing based machines or workstation clusters. In general, only the moduls in which the communication takes place have to be adapted. They can also be implemented on shared memory architectures. The introduced data cache structure allows a switch between more or less memory intensive strategies.

9 Acknowledgement

The author wish to thank the TMC-Team of GMD for their valuable advices concerning the programming of the CM5, and Bernd Froehlich for fruitful discussions and comments.

REFERENCES

[1] Amanatides, John and Andrew Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing". Proceedings of Eurographics 1987, (Bruessels, Belgium, August 24-28, 1987).

[2] Danskin, John and Pat Hanrahan, "Fast Algorithms for Volume Rendering", Proceedings of the 1992 Workshop on Volume Visualization, (Boston, MA, October 19-20, 1992).

[3] Hanrahan, Pat, "Three-Pass Affine Transforms for Volume Rendering". *Computer Graphics*, Vol.24, No. 5, 1990.

[4] Hsu, W.M. "Segmented Ray Casting for Data Parallel Volume Rendering", Parallel Rendering Symposium, San Jose, October 1993, pp. 7-14.

[5] Krueger, Wolfgang. "The Application of Transport Theory to the Visualization of 3-D Scalar Fields". *Computers in Physics*, July 1991, pp. 397-406.

[6] Laur, David and Pat Hanrahan. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering". *Computer Graphics*, Vol. 25, No. 4, July 1991.

[7] Levoy, Marc. "Efficient Ray Tracing of Volume Data". *Transactions on Graphics*, Vol. 9, No. 3, July 1990.

[8] Levoy, Marc. "Display of Surface from Volume Data". *Computer Graphics and Applications*, Vol. 8, No. 3, May 1988.

[9] Ma, L-K.; Painter, J.; Hansen, C.D. and M. Krogh. "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering". Parallel Rendering Symposium, San Jose, October 1993, pp. 15-22.

[10] Montani, C; Perego, and R. Scopignio. "Parallel Volume Visualization on a Hypercube Architecture". Workshop on Volume Visualization, Boston 1992, pp. 17-24.

[11] Mueller, Heinrich. "Ray Tracing Complex Scenes by Grids", Karlsruhe 1989.

[12] Nieh, Jason and Marc Levoy. "Volume Rendering on Scalable Shared-Memory MIMD Architectures". Workshop on Volume Visualization, Boston, October 19-20, 1992, pp. 17-24.

[13] Schröder, Peter and J.B. Salem. "Fast Rotation of Volume Data on Data Parallel Architectures". Visualization'91, San Diego, October 1991.

[14] Schröder, Peter and Gordon Stoll. "Data Parallel Volume Rendering as Line Drawing". Workshop on Volume Visualization, Boston, October 19-20, 1992. pp. 25-32.

[15] Stoer, Josef and Roland Bulirsch. "Introduction to Numerical Analysis". Springer Verlag, New York 1980.

[16] Vezina, Guy; Fletcher Peter and Philip Robertson. "Volume Rendering on the MasPar MP-1". Workshop on Volume Visualization, Boston, October 19-20, 1992, pp. 3-8.

[17] Westover, Lee. "Footprint Evaluation for Volume Rendering", *Computer Graphics*, Vol. 24, No. 4, August 1990.

[18] Wilhelms, Jane and Allan van Gelder. "A Coherent Projection Approach for Direct Volume Rendering", *Computer Graphics*, Vol. 25, No. 4, July 1991.

[19] Williams, L. "Pyramidal Parametrics". *Computer Graphics*, Vol. 17, No. 3, July 1983, pp. 1-11.

Table 1: Timings in seconds on 16 node partition. Data resolution: 256^3. **Picture resolution:** 256^2

SB/SS	16/16	32/16	64/16	16/32	32/32	64/32
render	29.3	26.6	25.3	34.2	26.8	25.8
comm	2.0	2.6	4.6	1.5	1.8	2.0
final	31.3	29.4	29.9	35.7	28.6	27.8

Table 2: Timings in seconds on 32 node partition. Data resolution: 256^3. **Picture resolution:** 256^2

SB/SS	16/16	32/16	64/16	16/32	32/32	64/32
render	13.8	12.7	12.1	16.9	13.3	12.4
comm	1.2	1.5	3.2	1.0	1.0	1.3
final	15.0	14.2	15.3	17.9	14.3	13.7

Table 3: Timings in seconds on 64 node partition. Data resolution: 256^3. **Picture resolution:** 256^2

SB/SS	16/16	32/16	64/16	16/32	32/32	64/32
render	6.3	6.1	5.7	7.7	6.4	6.0
comm	0.6	0.7	2.0	0.4	0.3	0.6
final	6.9	6.8	7.7	8.1	6.7	6.6

Table 4: Timings in seconds on 16 node partition. Data resolution: 512^3. **Picture resolution:** 512^2

SB/SS	32/32	64/32	128/32	32/64	64/32	128/64
render	230.6	214.6	210.3	245.3	215.2	209.2
comm	18.6	20.1	37.0	18.6	16.2	18.3
final	248	234	247	263	231	227

Table 5: Timings in seconds on 32 node partition. Data resolution: 512^3. **Picture resolution:** 512^2

SB/SS	32/32	64/32	128/32	32/64	64/32	128/64
render	114.1	106.0	104.1	121.3	106.7	103.4
comm	10.8	12.2	28.0	9.2	9.7	12.1
final	124.9	118.2	132.1	130.5	116.4	115.5

Table 6: Timings in seconds on 64 node partition. Data resolution: 512^3. **Picture resolution:** 512^2

SB/SS	32/32	64/32	128/32	32/64	64/32	128/64
render	55.1	52.2	51.6	58.1	52.0	51.3
comm	4.0	4.7	13.2	3.6	3.0	4.9
final	59.1	56.9	64.8	61.7	55.0	56.2

Table 7: Timings in seconds on 64 node partition. Data resolution: 1024^3. **Picture resolution:** 1024^2

SB/SS	32/64	64/64	128/64	32/128	64/128	128/128
render	437	417	411	455	419	409
comm	32	35	100	32	26	40
final	469	452	511	487	445	449