Data Parallel Volume Rendering as Line Drawing

Peter Schröder
Department of Computer Science
Princeton University

Gordon Stoll
National Information Display Laboratory
David Sarnoff Research Center

Abstract

In this paper we describe an algorithm for volume raytracing in a data parallel framework. The algorithm uses the idiom of line drawing to traverse the data set when evaluating the path integrals corresponding to a raytracing of the volume. Since the rays of a parallel projection correspond to a single line instanced multiple times across the viewing plane the approach lends itself well to implementation on massively parallel computers. We have implemented this algorithm on the Princeton Engine (PE) and the Connection Machine CM2 computers and achieved interactive performance.

1 Introduction

With the increasing availability of parallel computers, algorithms have been developed in many application domains to exploit their processing power [1]. This can be used both to admit more computationally expensive models and to move applications from batch processing to interactive computing. One such domain is volume visualization. It has been acknowledged for some time that interactivity is very desirable in volume visualization [24; 14]. While relatively fast algorithms for volume raytracing exist for high end workstations [15; 12], very little work has been published describing algorithms which run on massively parallel platforms [20; 28; 17].

In the present paper we describe an algorithm for data parallel volume raytracing which is both more memory efficient and less communications bound than previous work [20]. We describe its implementation on two different platforms: the Princeton Engine and the Connection Machine CM2 computer.

1.1 Assumptions and goals

The basic assumptions which influenced the algorithm design were

- **SIMD data parallel programming model**
  Due to the SIMD nature of the underlying hardware it is difficult to incorporate such optimizations as early view ray termination without requiring a large communication overhead to perform load balancing.
- **Distributed memory**
  The relatively small amount of memory per processor precludes such measures as duplication of the data set at every processor[1]. Thus communication needs to be considered explicitly.
- **Regular communication versus irregular communication**
  When communication cannot be avoided, such as when accessing values of voxels not held in a processor's local memory, regular communication patterns are much preferable over irregular ones.

Interactive performance was the main motivation behind the development of the present algorithm. Scalability of the algorithm was also deemed important. We desire an algorithm which scales to large numbers of processors. A given implementation may well take advantage of a particular number of processors. However, the algorithm itself needs to exhibit scalability to continue to be useful as future architectures incorporate increasing numbers of processors.

1.2 Previous work

Previous approaches to volume rendering can be classified into three groups

- Ray tracing, with two variants
  1. Object space methods, which resample the volume at sample points along rays [13].
  2. Image space methods, which transform the volume to be axis aligned with the view direction [6].
- Splatting [24; 25; 12]
- Cell projection [26; 16; 21]

While both splatting and cell projection can have efficient parallel implementations we are concerned only with ray tracing methods in the present discussion. Parallel implementations of object and image space methods exploit different forms of parallelism. The former can be characterized by ray parallelism and the latter by voxel parallelism. They are also distinguished by the communication patterns which they imply on distributed memory architectures.

In image space methods the volume is treated as a set of planes which are subjected to two dimensional transformations [8]. The viewing transformation is decomposed into a set of affine transformations on the volume. These transformations have the advantage of being constant across planes of the volume and only involve simple one dimensional resampling operations. For parallel viewing projections of the volume it is sufficient to consider rotations only. The fact that rotations can be decomposed into a sequence of pure

[1] Duplication at every processor is not desirable in any case since the amount of wasted memory grows with the number of processors.

shears [18; 22] was exploited in [20] for a massively parallel volume rendering algorithm. If the individual transformations are pure shears, only regular communication—shifting by constant amounts along each row—is necessary. One disadvantage of this approach in a data parallel framework is the high memory consumption of the algorithm. Semantically each operation on the volume is executed everywhere at once, hence every intermediate variable must be instanced at each voxel[2]. For the same reason the regular communication used is fairly costly, since every voxel participates.

In object space methods the volume is left in its local coordinate frame and instead rays are subjected to the inverse viewing transform. Each ray parameterizes a set of sample points along its length, some of which will lie inside the volume. In order to resample using tri-linear interpolation the eight grid points surrounding a given ray-sample need to be accessed. Since the mapping of samples onto voxels is not one-to-one—some voxels contain more than one ray-sample, others none—a SIMD implementation appears to require general communication. The present algorithm addresses this concern by describing a mapping of ray-samples onto the volume which is one-to-one. This mapping uses the idiom of line drawing on discrete grids [3]. Instead of transforming the volume once for a given rotation, we step the rays through the volume for each rendering. Our algorithm has three advantages

- Small memory requirements, since we do not transform the volume
- Low communication requirements, since only regular communication on the rays, not the voxels, is necessary.
- Lock step movement of all the rays, allowing for a SIMD implementation

We describe the algorithm in detail in the following section.

2 Volume path integrals as line drawing

When tracing a single ray through the volume we are interested in evaluating a path integral along that ray [11]. This is typically accomplished by discretizing the integral into a sum based on some step size along the ray. At each one of these ray-sample points we need to identify a set of surrounding grid points for resampling. This is equivalent to enumerating the sequence of voxels in which successive ray-sample points lie. Notice that some voxels may have more than one ray-sample point interior to them, while other voxels, although pierced by the ray, do not contain any ray-sample point. Enumerating these voxels is reminiscent of the approaches used for voxel traversal in non-volumetric ray tracing [7].

To simplify the discussion we first explain the algorithm in 2D. All arguments used readily generalize to the 3D case. Each ray starts in the image line (plane) and steps towards the volume. If we start the rays with a spacing in the image line such that their *vertical distance* is 1 (see Figure 1), then all rays will enter the frontmost column (face) of the image (volume) with the same pixel (voxel) local coordinate. We ignore for a moment those rays which do not enter the image in the front (such as the rays on the top face in Figure 1). Since the rays have a vertical distance of 1, the pixels they pierce perfectly tile the image without holes or overlaps (see the two non-adjacent rays whose pixels are grayed in Figure 1). As a consequence the spacing of the rays in the image line is $\cos\theta$ (see Figure 1) and not 1 as demanded by the inter pixel distance of the final picture. Considering the rotation matrix for an arbitrary 3D rotation this decomposition into a scaling component in the image plane and the

[2] An implementation on a specific number of processors may be able to achieve the correct semantics while instancing variables only at each *physical* processor, rather than each voxel.

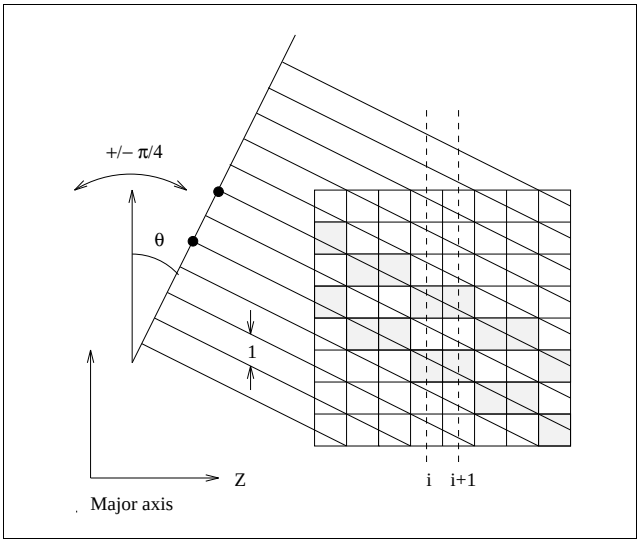

Figure 1: *Line drawing in 2D. All rays start from the image "line" and are chosen such that they intersect the outermost column of pixels at the centers. Each ray enumerates the voxels it steps through. An example for two non-adjacent rays is given with the shaded pixels. All rays move in lock step along the major axis. The sample points at step i and $i+1$ respectively are given by the intersection of the dashed lines with the rays.*

operations on the volume can be written as

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & e & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{b}{a} & \frac{c}{a} \\ \frac{d}{e} & 1 & \frac{f}{e} \\ g & h & i \end{pmatrix} \quad (1)$$

So far the stepping along the rays is reminiscent of Bresenham line drawing. However, in contrast to Bresenham lines we want the step length along the ray to be independent of the view angle θ. Notice that this requirement does not change our argument above. If the step length along the major axis (Z in Figure 1) is less than 1 it is possible for two successive steps i and $i+1$ to occupy the same column. All sample points (shown in Figure 1 as the intersection of the rays with the dashed lines) still have the same voxel local coordinates. The pseudocode below embodies this approach. Z is the major axis with length `N` and the keyword `Parallel` introduces declarations of variables instanced across all processors. For simplicity we assume for now that there are as many processors as pixels

```
Parallel float vol[N], sample;
Parallel Rgba integral, color;
Point3D coord;
Vector3D ray;
int i;

ray = compute_view_parameters();

/* voxel local coordinates of the sample points */
coord.x = coord.y = coord.z = 0;

while( coord.z < N ){
  i = ( int )coord.z;
  sample = interpolate( coord, vol[i], vol[i+1] );
  color = shade( sample );
  composite( integral, color );
  step( ray, coord, integral );
  coord.z += ray.z;
}
```

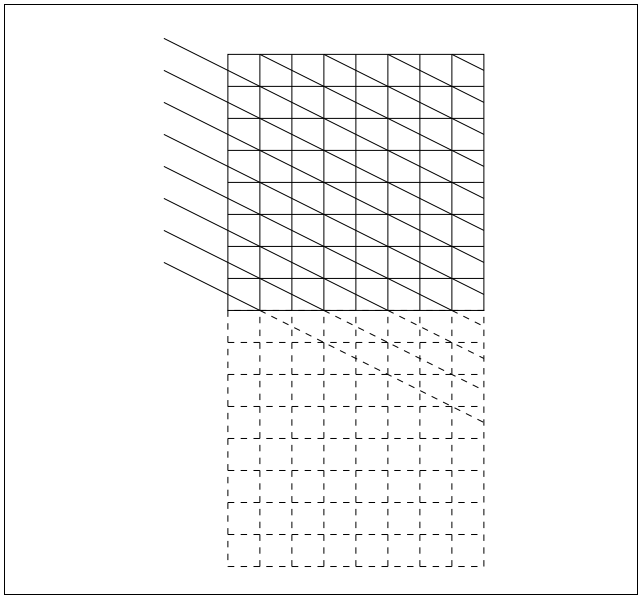

Figure 2: *When only considering rays entering through the frontmost face, the rays entering at a later point (on top) correspond to initial rays leaving through the bottom in a one-to-one fashion. This can be understood as continuing the exiting rays through a periodic tiling of the original set of pixels.*

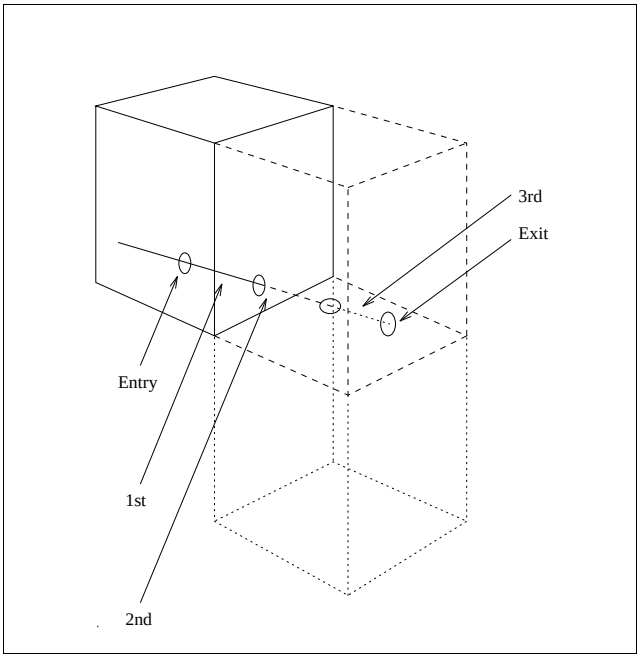

Figure 3: *In 3D it is possible for a ray to have three segments going through the volume. In the picture one such case is shown with arrows indicating the entry point, first, second, and third segments and the exit point.*

`vol` is an in-processor array of length `N` instanced across all processors and holds the scalar variable to be visualized. The function `interpolate` executes a tri-linear interpolation, while `color` maps a sample value onto r, g, b, and α; `composite` is typically just the *over* operator [19]. The `step` function increments the sample coordinates and tests the voxel local coordinates along the minor axes against overflow and underflow to determine whether to take a step orthogonal to the major axis.

So far we have neglected the rays which do not enter the volume on the frontmost face as well as the question of what happens to those rays leaving the volume before reaching the far end. The answer is based on the observation that rays entering the image on the top (see Figure 2) correspond in a one-to-one fashion to rays leaving on the bottom. If we tile space with copies of the image there are only rays which start on the frontmost face and continue until they exit the far face. In the 2D example any ray has at most two parts—an initial segment—and then, if and when it leaves—a segment corresponding to a new ray entering. For volumes there are at most three segments to a ray, since a ray can pierce up to three copies of the volume before exiting the far plane (see Figure 3). All of these considerations rely on the fact that we can limit ourselves to angles $\theta \in [-\pi/4, \pi/4]$ about the major axis (see Figure 1).

Tiling space with virtual copies of the volume is easiest to accomplish using coordinate arithmetic modulo the side length of the data set. The desired modulo arithmetic finds its physical implementation in communications hardware with toroidal topology, as can be found both on the CM2 and the PE. Hence if the next step is such that a ray leaves the volume (as does the rightmost ray in Figure 4) a toroidal shift moves the data holding the finished integral to the opposite side, where a new ray enters (the dotted ray in Figure 4). With this in mind, the pseudocode for the `step` function is as shown below. The function `shift` executes a toroidal shift, while `save` stores the finished integral and reinitializes the variable `integral` to begin integration along the new ray segment.

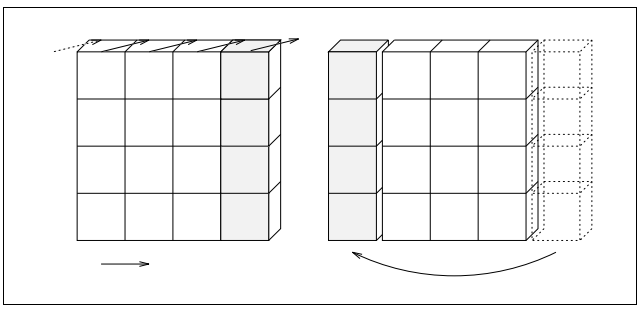

Figure 4: *Whenever a shift occurs due to the next step a column of processors has finished its segment and a new column is needed on the opposite face, to start the next segment. Toroidal shifts provide the right semantics.*

```
coord.x += ray.x; coord.y += ray.y;

if( coord.x < 0 ){
  save( integral, LEFT );
  shift( integral, LEFT );
  coord.x += 1;
}else if( coord.x >= 1 ){
  save( integral, RIGHT );
  shift( integral, RIGHT );
  coord.x -= 1;
}

/* similarly for Y */
```

At this point only the rescaling operation in the image plane is left to generate the final image (see equation 1).

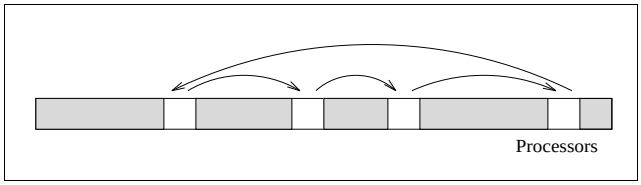

Figure 5: *Bypassing makes it possible for a subset of all processors to execute a circular shift among themselves.*

3 Implementation

3.1 Architecture overview

CM2 The CM2 consists of 4k to 64k bitslice processors with one floating point chip for every 32 processors. The processors are organized in a hypercube with a toroidal topology [9; 23]. Each processor has from 256 to 1024 kilo-bits of local memory. The programming model supports *virtual processors*, allowing the programmer to abstract from the actual number of physical processors available. The mapping of virtual processors to physical processors is performed by the runtime system, allowing the same executable to run on machines with differing numbers of processors. The particulars of this mapping can be influenced by the programmer, optimizing communication performance for a given application. The concept of virtual processors is especially useful for volume rendering since volumes often have more than 64k voxels.

Princeton Engine The Princeton Engine [5] consists of 64 to 2048 16-bit DSP processors arranged in a ring. The processing elements operate at 14 MHz and have up to 640 kilo-bytes of memory each. Standard communication consists of a right or left shift of a 16-bit word from each processor to its neighboring processor, and is performed in a single instruction cycle. The communication system also supports *bypassing* and *broadcasting*. During bypassing only a selected subset of processors participates in the ring shift (see Figure 5). Broadcasting allows a single processor to transmit a value to all other processors. The cost for these operations is at most seven instruction cycles (depending on the particular communication pattern).

3.2 Implementation on the CM2

The implementation on the CM2 computer operates chiefly on two parallel variables, one holding the accumulated integral and another holding the volume data set. Due to the virtual processor mechanism we always assume that there are (at least) as many processors as there are rays.

In the actual code we declare the processor set for the volume data to be three dimensional, rather than two dimensional with an in-processor array as in the code above. This allows us to take advantage of all processors in the machine as long as there are at least as many voxels as there are processors. When laying out the mapping of virtual processors onto physical processors we take advantage of the facilities to preferentially map one of the axes of the shape—Z in our case—along the *in-processor* dimension. This gives the code a simple uniformity and allows us to abstract from the actual number of processors and the particulars of the mapping.

Assuming for the moment that the volume has side length n and n^3 voxels and we have n^2 processors the entire Z axis would be mapped in-processor. This still holds if there are less than n^2 processors, which is the typical case. We have therefore optimized the code for this case. This optimization also implies that we need to transpose the volume when Z is not the major axis. For another implementation alternative (pipelining) see the PE implementation below.

If the number of processors is larger than n^2 we divide the ray into pieces along its length and compute the subintegrals as before. Parallelism assures that the above code remains invariant—only the runtime mapping of virtual processors onto the actual hardware changes. After the individual subintegrals have been computed in this fashion a parallel prefix operator [2] is used to combine the results of the sub integrals. As the number of processors approaches the number of voxels the length of the sub integrals goes to one, in effect evaluating the entire integral with the parallel prefix operators (for details of this approach to rendering see [20]).

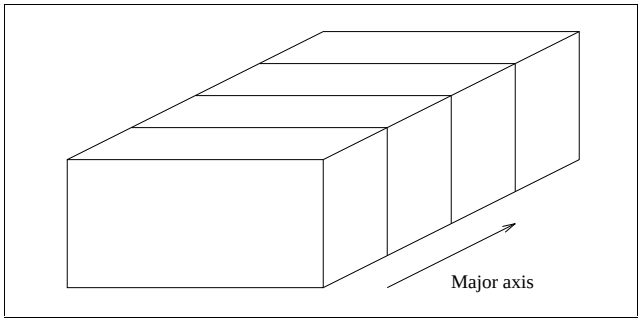

Figure 6: *Volumes whose side lengths are not equal are accommodated by dividing them into smaller pieces and combing the sub integrals.*

Non-even sided volumes can be accommodated as well. On the CM2 all processor layouts need to be rounded up to a power of 2 multiple of the machine size, since only such shapes are supported[3]. This implies that we may treat any parallelepiped as a set of smaller parallelepipeds along the major axis, such that each has a depth less than or equal to its smallest side. This way we can again guarantee that no ray will pierce its volume more than 3 times (see Figure 6). The combining of the sub integrals occurs just as above.

In this framework of virtual processor layouts supersampling is accommodated easily. Nothing needs to change for supersampling *along* the rays. In the image plane it can be accomplished through two dimensional serial arrays in each processor and iterating over these. On the Connection Machine, with its hypercube connectivity, however, it is more efficient to use two of the hypercube dimensions to parallelize along the supersample dimensions as well. As a result of using the layout mechanism to accomplish supersampling, the code need not be changed.

The final rescaling of integrals in the image plane does use a general communication step in the current CM2 implementation. While it could be accomplished with regular communication the implementation was judged too complicated. The complexity arises since the subsections of the rays[4] will have shifted in rather complicated patterns when rotating about more than one axis.

3.3 Implementation on the PE

On the PE we equate the processor array with the X axis of the volume data set such that each processor holds a YZ slice of the volume. In the first implementation of volume rendering on the PE [10], only rotation about the X axis is supported, making rotation a completely local operation. The immediate goal for the new implementation is to provide rotation about the Y axis. Later integration of X and Z axis rotation (both of which operate quickly) will allow for full freedom of motion.

[3] One can of course compute with less processors, but only at the price of leaving some idle due to the SIMD nature of the machine.

[4] See the discussion regarding each ray piercing the volume in at most three sections.

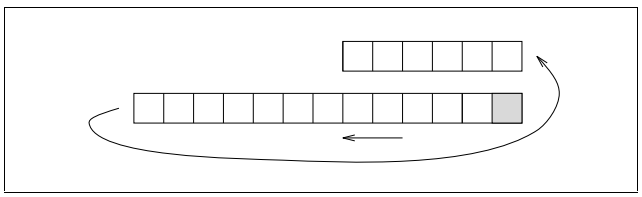

Figure 7: *If a data element is shifted off the end of the processor array, it wraps around to the first processor, where it is saved in an auxiliary variable. This frees up the first processor to start another ray.*

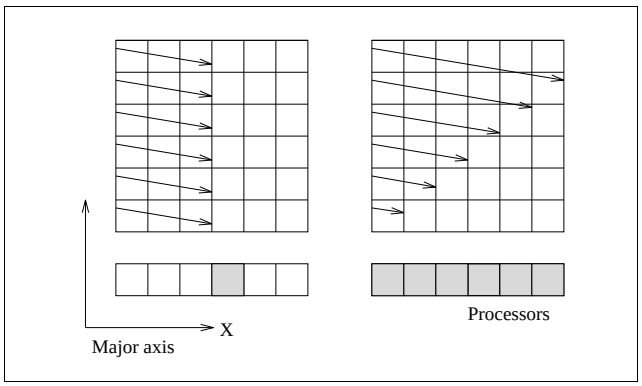

Figure 8: *If the major axis is along the array of processors, the regular version of the code leads to only one processor active (grayed) at a given time (left side). Furthermore this processor has to service all rays. A pipelined version (right side) advances rays in a staggered fashion resulting in every processor computing on one ray at a time.*

In the following description of the implementation we first assume that the number of slices, n, is equal to the number of processors, p. When the Z axis is the major axis of the projection, the implementation on the PE is very similar to the pseudocode given in Section 2. The array of processors holds the integrals corresponding to a single scan line of the projection, and the computation proceeds along the Z axis through each processor's local memory. The accumulated integrals are shifted right or left as dictated by the `step` logic. The ray which wraps around after a shift is stored (although it continues to be shifted) and a new ray is initialized on that processor. Once the projection is completed, the integrals are distributed as shown in Figure 7, and rescaling must be performed in the image plane.

Rescaling requires an irregular communication step, but since the communication involves only the number of elements in the projection, rather than the volume, it is not prohibitively expensive. This two-dimensional projection and rescaling operation is repeated for each scan line in order.

When the X axis, which is along the processors, is the major axis of the projection, a different approach is required. If we step as before only the processor which currently contains the "front" of all rays can do any computation, and it needs to do computation for all rays (see the left side of Figure 8). In order to avoid this problem and maintain approximately one intersection calculation per processor per constant-time iteration, the intersections are evaluated in a pipelined fashion. By staggering the rays it can be assured that each processor has one and only one ray to work on at a time (see the right side of Figure 8). The loop which moves the rays forward must be executed an extra n times to fill the pipeline, but if there are approximately n^2 rays to be processed this overhead is negligible. Shifting orthogonal to the major axis (function `shift` in the pseudocode in Section 2) is now a processor-local operation.

In contrast to the code used when Z is the major axis, the pipelined version completes the evaluation of all rays in the same processor. For rescaling and display purposes it is desirable to redistribute the rays across the processors. Currently the pipelined version distributes the final results in the same pattern as the non-pipelined code. Each processor calculates whether it is the appropriate destination of the ray whose evaluation is completed in the current iteration. All processors except the end processor and the appropriate destination processor *bypass* themselves, so that the transmission can be performed in a single shift (see Figure 5).

In the case that p is greater than n, high processor utilization can be maintained by the same method used in the previous implementation of volume rendering on the PE [10]: if the number of slices, n, is less than the number of processors, p, each successive group of n processors is given a copy of the volume. In this case each processor computes the rotation and projection for a horizontal strip with height n/p of the total height. This division achieves a linear increase in speed at the expense of a linear increase in total memory consumption. The non-pipelined algorithm iterates over all scan lines of the projection independently. This allows groups of n processors to compute the horizontal strips of the projection independently. In the pipelined algorithm, each group of n processors is used as an independent pipeline which operates on a single strip. On each iteration p/n rays are completed and shifted to their destinations using bypassing.

4 Performance

4.1 Performance of the CM2 implementation

In order to measure the performance of the CM2 implementation we have run the algorithm on data sets of size 64^3 and 128^3 on machines from 8k to 32k processors in size. An example of an image generated is given in Figures 9 and 10. The clock frequency was 8MHz. The rendering used a volume source term, one lambertian light source and gradient magnitude surface source terms [11] for the engine block. The spine was rendered with a volume source term for the discs and a surface source term for the bone which consisted of a lambertian light source at the eye and a Phong light source. Integration was performed by using the conventional *over* operator. The runtime of the algorithm does depend on the rotation angle, which influences the number of nearest neighbor communications. We have found this influence to be very weak (less than ten percent) on the overall runtime and have used a rotation of $\pi/6$ for all timings below, giving the average time.

Table 1 gives timings for different machine sizes using a data set of 128^3 voxels. As can be seen from the table not all components of the algorithm scale linearly with the machine

Mach. size	Integr. time	Shad. time	Rescal. time	Total time	M vox. per sec.
8k	500	(297)	84	584	3.42
16k	332	(148)	43	375	5.33
32k	216	(68)	34	250	8.00

Table 1: *Performance for the CM2 implementation for various machine sizes. The last column gives the number of Mega voxels per second. The data set size was* 128^3*. Integration time includes tri-linear interpolation at the sample points and composition along the rays. Interpolation was performed on the shaded values (pre-shading) in grayscale. The shading times were not included in the total time. All times are given in ms.*

Data size	Integr. time	Shad. time	Rescal. time	Total time	M vox. per sec.
full color, tri-linear interpolation, inner loop shading					
64^3	89	41	58	188	1.33
128^3	546	316	61	923	2.17
grayscale, tri-linear interpolation, inner loop shading					
64^3	76	30	34	140	1.79
128^3	481	231	43	755	2.65
full color, tri-linear interpolation, pre-shading					
64^3	89	(28)	58	147	1.70
128^3	546	(202)	61	607	3.29
grayscale, tri-linear interpolation, pre-shading					
64^3	53	(19)	34	87	2.87
128^3	332	(148)	43	375	5.33
full color, no interpolation, inner loop shading					
64^3	35	41	58	134	1.87
128^3	178	316	61	555	3.60
grayscale, no interpolation, inner loop shading					
64^3	22	30	34	86	2.90
128^3	113	231	43	387	5.17
full color, no interpolation, pre-shading					
64^3	34	(28)	58	92	2.61
128^3	176	(202)	61	237	8.44
grayscale, no interpolation, pre-shading					
64^3	22	(19)	34	56	4.46
128^3	111	(148)	43	154	12.99

Table 2: *Performance of various rendering options on a 16k processor CM2. Integration time includes tri-linear interpolation at the sample points and composition along the rays. All times are given in ms.*

size. The clearest linear scaling can be seen in the shading. Since it is a pre-process from the point of view of the projection, it is performed on the volume (voxel parallel). This implies a high vp-ratio (number of virtual processors per physical processor) during this part of the computation. Whenever a program is run at high vp-ratios the instructions are relatively more efficient, since any startup overhead of the runtime system can be amortized over more iterations. If the vp-ratio falls below eight the overhead of various instructions bears more heavily on their timing. Thus the scaling does not appear linear anymore. Integration, which includes interpolation, is performed at vp-ratios of one and two, since only a "plane" of processors—one for each ray—is active at any time, and the scaling of the times reflects this. Since the image size was constant for a given data set size, the times for the final rescaling of the image do not vary linearly with machine size either. In this case the vp-ratio continually goes down as the machine size increases.

Table 2 gives performance numbers for different modes of operation. We have experimented with tri-linear interpolation versus no interpolation (simply using the nearest neighbor). For some of the high resolution data sets which we have examined this lack of interpolation makes no appreciable difference in the image. When shading is performed in the inner loop it executes at a lower vp-ratio and hence is not as efficient as pre-shading. The tradeoff between grayscale and full color is more subtle. If pre-shading is enabled the entire inner loop reduces to working on two channels, grayscale and α. If on the other hand shading is performed in the inner loop four values, the volume data as well as its gradient, have to be worked on in either case and the influence of grayscale versus full color is not felt as strongly.

These factors combine to give a rather large range of voxels per second rendering performance data for the various rendering options. For data sets of 64^3 these range from 1.33 to 4.46 Mega voxels per second, while for 128^3 voxels the range is 2.17 to 12.99 Mega voxels per second. In particular, if perfect scaling would occur the number of voxels per second should be independent of the data size, which is clearly not the case. The best time we have measured so far was for a 128^3 data set on a 32k machine with grayscale, no interpolation, and pre-shading at 18.73 Mega voxels per second.

Data size	Integr. time	Rescal. time	Total time	M vox. per sec.
non-pipelined				
128^3	17	11	28	71.43
256^3	129	86	215	74.42
pipelined				
128^3	40	10	50	40.00
256^3	293	94	387	41.34

Table 3: *Preliminary timings of single axis rotation on the PE with 1024 processors. Integration time includes tri-linear interpolation at the sample points and composition along the rays. All timings are given in ms.*

4.2 Performance of the PE implementation

Table 3 shows preliminary timings for 128^3 and 256^3 data sets on a 1024 processor PE. Rendering is performed by integration of RGB values (i.e. there is no shading in the inner loop). The *over* operator is used for integration, with linear interpolation used for resampling. The timings listed are averaged over angles of rotation $\theta \in [0, \pi/4]$. As in the CM2 implementation the angle of rotation influences the performance of the reorganization and rescaling step, but the effect is weak. From the timings it is clear that the difference between the 128^3 and 256^3 figures is very close to the desired factor of eight. As expected, the rescaling step is expensive but not prohibitively so. Currently the pipelined code is approximately 80% slower than the non-pipelined version. The inner loop of the current pipelined version is significantly more complex than that of the non-pipelined version, but much of the computation could be moved outside the loop by pre-computing tables for voxel and processor addressing. Also, the amount of communication required for rescaling can be reduced in the pipelined version by changes in the processor bypassing step.

5 Discussion

We have described a data parallel algorithm for volume ray-tracing. It uses the idiom of line drawing to enumerate the voxels along every ray. The algorithm has been implemented both on the PE and the CM2 and renders arbitrary parallel projections at interactive rates.

The performance numbers indicate that the considerably simpler communication system of the PE is sufficient to implement the current algorithm efficiently. The communication network of the CM2 is much more general and not as optimized for nearest neighbor shifts as the PE. Conse-

quently the communication performance as needed by this algorithm is more efficient on the PE than on the CM2. This greater generality, on the other hand, provides higher level data parallel programming primitives. These primitives, as well as the virtual processor abstraction, make a fully general implementation of the algorithm much simpler on the CM2.

One comment is in order regarding the scaling properties of the algorithm itself. For simplicity assume that the side length of the volume is n. If we increase n the amount of data will grow as n^3 while the number of rays only grows as n^2. The present algorithm is characterized by its ray parallelism. As a consequence the work per processor increases linearly in n, assuming that we have n^2 processors and n^3 voxels. If, on the other hand, object space parallelism is exploited (as is the case in [20]), the work per processor grows as $\log n$. This is due to the fact that for n^3 processors working on n^3 voxels the combining along the rays can be executed with a parallel prefix operator in logarithmic time [2].

Generally, however, the number of available processors is less than n^2. In this case the present algorithm scales as expected.

Note: It has come to the authors' attention that algorithms similar to that described in this paper have been developed independently by Roni Yagel [27] in the context of hardware design for volume rendering and by George Cameron [4] for a software implementation on the AMT DAP massively parallel processor.

Acknowledgments

The first author would like to thank Thinking Machines and Lew Tucker for access to "unlimited" CM2 cycles. Part of the research was performed while the first author was a guest scientist at the Gesellschaft für Mathematik und Datenverarbeitung, Sankt Augustin, Germany. The color maps used for the engine block rendering were provided by John M. Danskin and the data for the human spine was provided by Dr. A. Kern, Radiological Institute of FU Berlin.

The second author would like to thank Jim Kaba, Jim Matey, Herb Taylor and Ann-Marie Lanzillotto for their help in the volume visualization work at DSRC; Stan Knight for managerial support; and the National Information Display Laboratory and the David Sarnoff Research Center for financial support.

Special thanks to Pat Hanrahan who first suggested the idea of volume rendering as line drawing to us.

References

[1] James Bailey. *Implementing Fine-grained Scientific Algorithms on the Connection Machine Supercomputer*, tr89-1 ed. Thinking Machines Corporation, 1990.

[2] Guy Blelloch. *Vector Models for Data Parallel Computing*. Artificial Intelligence Series. MIT Press, Cambridge, MA, 1990.

[3] C. Braccini, and G. Marino. Fast Geometrical Manipulations of Digital Images. *Computer Graphics and Image Processing 13* (1980), 127–141.

[4] George G. Cameron, and P. E. Undrill. Rendering Volumetric Medical Image Data on a SIMD-Architecture Computer. In *Proceedings of Third Eurographics Workshop on Rendering* (May 1992).

[5] D. Chin, J. Passe, F. Bernard, H. Taylor, and S. Knight. The Princeton Engine: A Real-Time Video System Simulator. *IEEE Transactions on Consumer Electronics 34*, 2 (1988), 285–297.

[6] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. *Computer Graphics 22*, 4 (August 1988), 65–74.

[7] Akira Fujimoto, Tanaka Takayuki, and Kansei Iwata. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications* (April 1986), 16–26.

[8] Pat Hanrahan. Three-Pass Affine Transforms for Volume Rendering. *Computer Graphics 24*, 5 (November 1990), 71–78.

[9] Danny W. Hillis. *The Connection Machine*. MIT Press, 1985.

[10] J. Kaba, J. Matey, G. Stoll, H. Taylor, and Pat Hanrahan. Interactive Terrain Rendering and Volume Visualization on the Princeton Engine. submitted to Visualization 92 conference, April 1992.

[11] Wolfgang Krüger. The Application of Transport Theory to the Visualization of 3-D Scalar Fields. *Computers in Physics* (July 1991), 397–406.

[12] David Laur, and Pat Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Computer Graphics 25*, 4 (July 1991), 285–288.

[13] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications 8*, 3 (May 1988), 29–37.

[14] Marc Levoy. Design for a Real-Time High-Quality Volume Rendering Workstation. In *CH Volume Visualization Workshop* (May 1989), ACM Siggraph, ACM Press, pp. 85–92.

[15] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261.

[16] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics 24*, 5 (November 1990), 27–33.

[17] Ulrich Neumann. Interactive Volume Rendering on a Multicomputer. In *Proceedings of 1992 Symposium on Interactive 3D Graphics* (March 1992), ACM, ACM Siggraph, pp. 87–93.

[18] Alan W. Paeth. A Fast Algorithm for General Raster Rotation. *Proceedings Graphics Interface* (May 1986), 77–81.

[19] Thomas Porter, and Tom Duff. Compositing Digital Images. *Computer Graphics 18*, 3 (July 1984), 253–260.

[20] Peter Schröder, and James B. Salem. Fast Rotation of Volume Data on Data Parallel Architectures. In *Proceedings Visualization 1991* (October 1991), G. M. Nielson and L. Rosenblum, Eds., IEEE, IEEE Computer Science Press, pp. 50–57.

[21] Peter Shirley, and Allan Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics 24*, 5 (November 1990), 63–70.

[22] A. Tanaka, M. Kameyama, S. Kazama, and O. Watanabe. A Rotation Method for Raster Image Using Skew Transformations. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition* (June 1986), pp. 272–277.

[23] Connection Machine Model CM-2 Technical Summary. Tech. Rep. HA 87-4, Thinking Machines Corporation, April 1987.

[24] Lee Westover. Interactive Volume Rendering. In *CH Volume Visualization Workshop* (May 1989), ACM Siggraph, ACM Press, pp. 9–16.

[25] Lee Westover. Footprint Evaluation for Volume Rendering. *Computer Graphics 24*, 4 (August 1990), 367–376.

[26] Jane Wilhelms, and Allen Van Gelder. A Coherent Projection Approach for Direct Volume Rendering. *Computer Graphics 25*, 4 (July 1991), 275–284.

[27] Roni Yagel. *Efficient Methods for Volumetric Graphics*. PhD thesis, SUNY Stony Brook, December 1991.

[28] Terry S. Yoo, Ulrich Neumann, Henry Fuchs, Stephen M. Pizer, Tim Cullip, John Rhoades, and Ross Whitaker. Achieving Direct Volume Visualization with Interactive Semantic Region Selection. In *Proceedings Visualization 1991* (October 1991), G. M. Nielson and L. Rosenblum, Eds., IEEE, IEEE Computer Science Press, pp. 58–65.

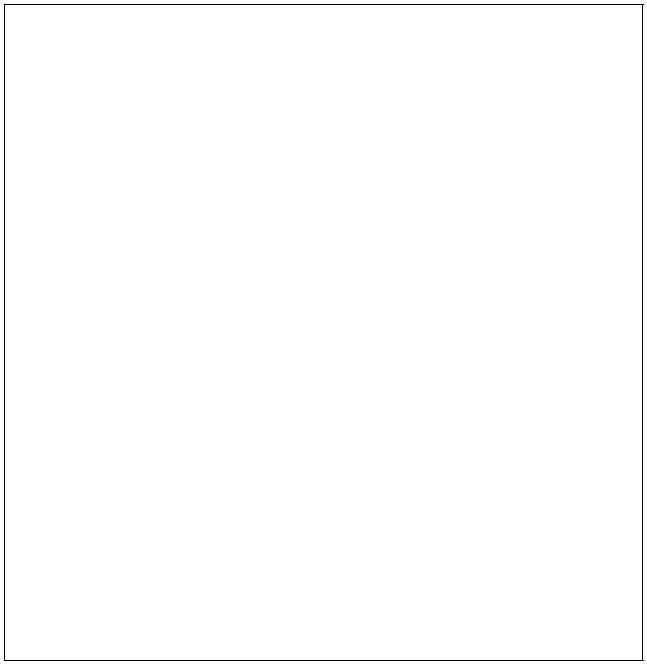

Figure 9: *CT data of an engine block. 128 by 128 by 110 voxels rendered with a supersampling of 16 rays per voxel.*

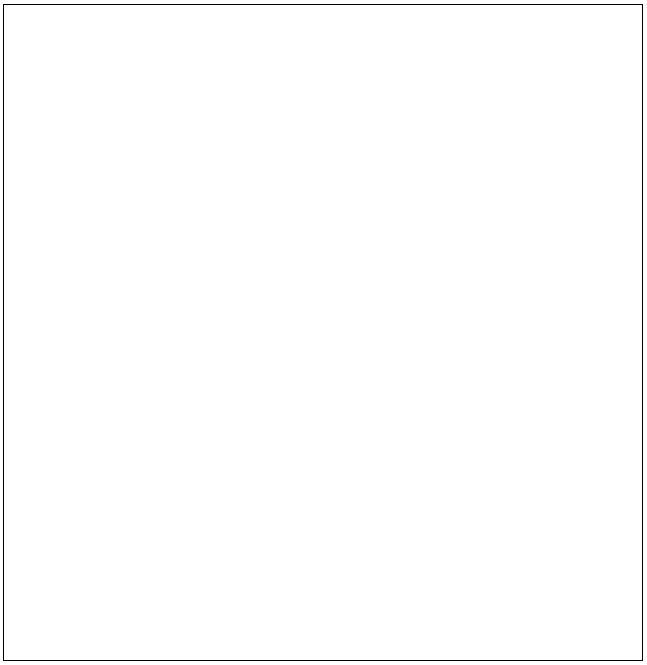

Figure 10: *CT data of a human spine. 128 by 128 by 107 voxels rendered with a supersampling of 16 rays per voxel.*