Fast Volume Rendering Using an Efficient, Scalable Parallel Formulation of the Shear-Warp Algorithm

Minesh B. Amin[1] Ananth Grama [1] Vineet Singh[2]
Hewlett Packard Labs, Palo Alto

Abstract

This paper presents a fast and scalable parallel algorithm for volume rendering and its implementation on distributed-memory parallel computers. This parallel algorithm is based on the shear-warp algorithm of Lacroute and Levoy. Coupled with optimizations that exploit coherence in the volume and image space, the shear-warp algorithm is currently acknowledged to be the fastest sequential volume rendering algorithm. We have designed a memory efficient parallel formulation of this algorithm that (1) drastically reduces communication requirements by using a novel data partitioning scheme and (2) improves multi-frame performance with an adaptive load-balancing technique. All the optimizations of the Lacroute-Levoy algorithm are preserved in the parallel formulation. The paper also provides an analytical model of performance for the parallel formulation that shows that it is possible to sustain excellent performance across a wide range of practical problem sizes and number of processors. Our implementation, running on a 128 processor TMC CM-5 distributed-memory parallel computer, renders a 256^3 voxel medical data set at 12 frames/sec.

Keywords: volume rendering, raytracing, algorithms, shear-warp algorithm, performance modeling and analysis, scalability, adaptive load-balancing

1 Introduction

Volume rendering is an important tool for visualizing three dimensional volume data. This data is gathered from such varied sources as medical imaging and modeling physical phenomena using finite element and finite difference methods. The rendering problem involves the projection of this volume data on to a two dimensional image plane that can be viewed on a screen.

[1] Visiting from Computer Science Department, University of Minnesota, EE/CS 4-192, 200 Union Street, S.E., Minneapolis, MN 55455; {amin, ananth}@cs.umn.edu
[2] HP Labs, 1501 Page Mill Road, Palo Alto, CA 94304; vsingh@hpl.hp.com

Most of these applications require the generation of a sequence of images for different orientations of the volume. This places a requirement on the rendering algorithm that images should be generated in a reasonable amount of time. Ideally, it is desirable that images be generated in real-time. This enables a continuous visualization of the volume as its orientation is changed. A number of approaches have been adopted towards achieving this goal. These include advances in algorithms, using dedicated hardware, and general purpose parallel processors.

A number of algorithms have been proposed for volume rendering. They can be broadly classified as raytracing [5], splatting [4], cell-projection [8], or shear-warp algorithms [3]. Raytracing algorithms, specifically those based on object space methods, trace rays through each point in the image plane as they pass through volume elements. The contribution of each volume element to an image pixel is computed by trilinearly interpolating neighboring voxels. This contribution is composited with accrued opacity and color of the intermediate image. Since interpolation weights change for each ray and each slice, these have to be computed repeatedly. In contrast, splatting is an object space technique in which object contributions are projected onto the intermediate image plane and composited there. The shear-warp algorithm can be viewed as an image space technique capable of utilizing both image and object space coherence. This approach has shown particular promise. Lacroute and Levoy [3] combine this technique with optimizations such as early termination and run-length encoding to obtain single processor performance that exceeds that of most parallel formulations of volume rendering thus far. Improvements in rendering speed come at the expense of a second resampling step. Its implications on image quality are discussed by Lacroute and Levoy [3].

The fastest sequential time for rendering a typical volume of size $256 \times 256 \times 256$ using the shear-warp algorithm is approximately one second on a R4000 Indigo. The desired goal of achieving real time rendering requires a further thirty-fold speedup over this time. Commodity microprocessor speeds are not expected to go up by that factor in the near future. Until that happens, parallel computing provides a viable and cost effective means of achieving this. Finally, as uniprocessor speeds increase, the need for larger datasets may increase also, and we may still require parallel processing.

This paper explores the application of parallel processing to volume visualization with the objective of achieving real time performance. Volume rendering presents conflicting scenarios for exploiting parallelism. Faster sequential algorithms incorporate a higher degree of coherence in the object and image space. This limits the extent of available concurrency. Conversely, approaches offering higher degrees of concurrency tend to be significantly slower. In this paper, we present a parallel formulation of the shear-warp algorithm. The critical aspects of this parallel formulation are the partitioning techniques for the object and image spaces. We demonstrate near real-time performance for our formulations. We also present analytical performance models for our formulation to indicate the performance trends as problem sizes and number of processors vary.

In Section 2, we discuss the sequential shear-warp algorithm in greater detail. In Section 3, we present possible partitioning techniques and comment on their relative performance. Section 4 describes the various steps of the parallel algorithm in more detail and includes an analytical model. Section 5 gives experimental results on a 128 processor CM-5. Section 6 summarizes the findings and describes some directions for future research.

2 Shear-Warp Algorithm for Volume Rendering

Volume rendering techniques project a set of 3-D volume slices on to a two dimensional viewing plane. Before describing the shear-warp algorithm, let us consider raytracing algorithms based on object space methods [5]. For each pixel in the viewing plane, a ray is driven through the volume slices and their color and opacity accrued in a front to back order. Since the volume slice samplings may not coincide with the point at which rays are driven, they are resampled using trilinear interpolation of neighboring volume elements. The resampling weights for each of these interpolations are different. Consider an $n \times n$ pixel viewing plane and an $n \times n \times n$ voxel dataset. The raytracing algorithm will drive $\Theta(n^2)$ rays through the volume. Each of these rays is driven through n slices. This requires a total of n^3 interpolations and $k \times n^3$ resampling weight computations. Here, k is the number of weights that need to be computed for each iteration and is either four or eight depending on the number of interpolation planes. This operation count can be reduced by using various optimizations discussed later.

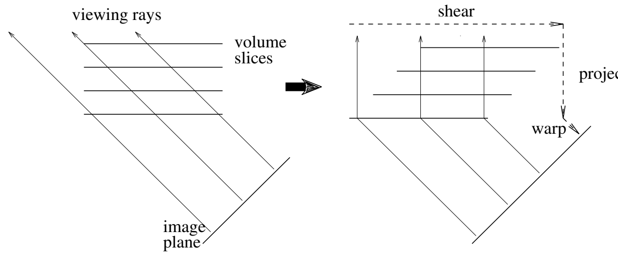


Figure 1: Shear-warp algorithm for volume rendering.

Raytracing algorithms spend a significant amount of time computing resampling weights. If the incident rays are perpendicular to the volume slice, the resampling weights are identical across the slice. This is the basic premise of shear-warp algorithms. When viewed at an orientation other than perpendicular to the volume slices, the volume slices can be sheared in such a way that the rays can be assumed to be perpendicular to the slices. This is illustrated in Figure 1. Each slice can now be translated and resampled using weights that are invariant across the slice. The slices can be composited in a front to back order to yield an intermediate image. This intermediate image must now be warped to yield the final image. This requires a second resampling phase. For each pixel in the final image, the four nearest neighbors in the intermediate image are located. The color value of the pixel in the final image is determined by interpolating from the color values of these neighbors. For generating an $n \times n$ pixel image from an $n \times n \times n$ voxel dataset, this technique would require $\Theta(n^3)$ interpolations and $\Theta(n)$ weight computations. We can see that the operation count for the shear-warp algorithm is less than that of the raytracing algorithm.

Algorithm Component	Time (msecs)
Shade Table	70
Composite	900
Warp	120
Total	1100

Table 1: Breakdown of sequential time on SGI Indigo R4000 workstation

However, the algorithm uses an additional resampling step in the warp phase instead of the single resampling step in raytracing algorithms. Fortunately, it has been shown that this does not lead to a significant deterioration in the quality of rendering [3]. Our parallel algorithm maintains the image quality of this sequential algorithm.

The operation count can be further reduced by using a variety of optimizations. These optimizations are based on two key observations: once the accrued opacity of the intermediate image crosses a certain threshold, subsequent volume slices have negligible impact on the final image; and, transparent voxels have no impact on the intermediate image. The first observation forms the basis for early termination of rays. It is implemented by having a forward pointer associated with each intermediate image pixel that points to the next un-saturated pixel. Composition of volume slices follows this chain of pointers and saturated pixels are skipped. The second optimization is implemented using a run-length encoded representation of the volume. Using this, runs of transparent voxels are skipped. In this way, the algorithm follows two pointer chains computing interpolations only for non-transparent voxels and unsaturated intermediate image pixels. This is illustrated in Figure 2. For the implementation in [3], three run-length encodings are precomputed, one for each possible principal viewing direction. The encoding used for a particular frame is the one with slices most perpendicular to the viewing rays. For the medical datasets used in [3], total memory required for the volume representation is lower than an uncompressed 3D array representation despite the need for three run-length encodings. Our parallel algorithm uses the same run-length encoded data-structure used in this sequential algorithm. Furthermore, our parallel formulation is memory efficient; *i.e.* the sum total of memory used across all the processors is approximately the same as that used by the sequential formulation.

The shear-warp algorithm can be viewed as consisting of three main functional units: lookup table computation (shading information), compositing (projecting volume slices onto intermediate image), and warping intermediate image. Although the bulk of the time may be spent on the compositing step, the other steps are not negligible. Table 1 shows the breakdown of the time taken for the sequential algorithm on a single SGI Indigo R4000 workstation for a 256x256x167 voxel "brain" data set (from a MRI scan of a human brain) and image size 256x256 pixels [3]. This implies that each one of these functional units must be effectively parallelized.

Although Lacroute and Levoy [3] discuss versions of their algorithm to deal with perspective projections and run-time change in the opacity transfer function, we do not discuss them in our paper. Our paper deals exclusively with parallel projections of classified volumes for which run-length encodings have been pre-computed.

3 Parallel Algorithms for Volume Rendering

The shear-warp algorithm augmented with early termination and run-length encoding yields excellent per-frame rendering times. This algorithm forms the basis of our parallel formulations. Parallelizing this algorithm presents considerable challenges. The crit-

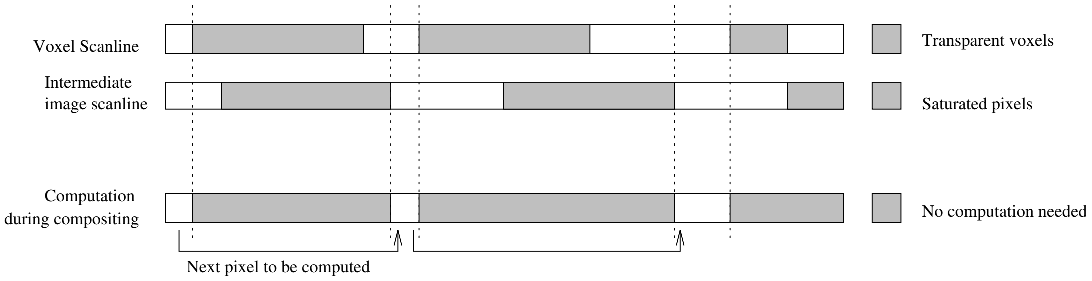


Figure 2: Compositing volume slices: skip transparent voxels and saturated intermediate image pixels.

ical issues in any parallel algorithm are concurrency, minimizing communication overhead and balancing load among processors. The fast rendering times of the shear-warp algorithm are derived by exploiting coherence in image and object space. In the parallel context, these optimizations limit the amount of concurrency. Since runs of image and object data span scanlines, an efficient parallel formulation should avoid cutting across scanlines. Therefore, a single scanline (of the intermediate image) can be viewed as a unit of work.

Parallel formulations must assign scanlines of the intermediate image and volume slices with the combined objective of minimizing communication and balancing load. If it is indeed possible to replicate the entire volume at each processor, the communication problem can be solved relatively easily. However, such an assumption is unreasonable since it increases the overall memory requirement significantly. Each scanline has a widely differing amount of computation associated with it. Therefore, naively assigning equal scanlines to each processor can lead to significant load imbalances. Furthermore, it is impossible to evaluate accurately the amount of computation for each scanline. Efficient load balancing techniques must therefore be designed for assigning scanlines to processors.

The two critical aspects of parallelizing the algorithm are partitioning the volume data and partitioning the computation. Fixing either of these automatically induces a partitioning of the other. Let us explore possible partitioning techniques and their performance. We first discuss different types of data partitioning and then an adaptive load balancing mechanism for the chosen data partitioning type.

3.1 Types of Data Partitioning

3.1.1 Volume Space Partitioning

In volume space partitioning, the volume data is partitioned among various processors. The simplest instance of volume space partitioning is sliced partitioning. This is illustrated in Figure 3 for four processors. The volume data is sliced into p parts where p is the number of processors. Each processor drives rays through the volume segment assigned to it. The resulting intermediate images I1, I2, I3, and I4 are composited to yield the final intermediate image that can be warped. This algorithm works well when there is no shear in the volume. This is illustrated in Figure 3(a). The intermediate volumes are non-overlapping and no compositing is necessary. However, a shear in the volume results in significant overheads. The most obvious overhead results from communication of overlapping areas of the intermediate image (Figure 3(b)). Here, the intermediate images have to be composited in a front to back order, i.e. I1 over I2 over I3 over I4. Consider a case in which each processor gets two scanlines and the shear is 45 degrees. If there are a total of n volume slices, the shear translates to n scanlines. In this case, after computing two scanlines, each processor will have to communicate intermediate image data corresponding to n scanlines. This is a significant overhead. A second source of overhead is wasted computation. A part of intermediate image I2 is obscured by image I1. It is likely that many of the rays would get saturated in intermediate image I1. This results in wasted computation in overlapped areas. This is a major overhead when the number of processors becomes large.

An alternate partitioning technique for volume space partitioning is illustrated in Figure 3(c). Here, in addition to striping each slice, the slices themselves are partitioned among processors. In the example, the top $n/2$ scanlines of the first $n/2$ slices are assigned to the first processor, and of the last $n/2$ slices to the second processor, and so on. The processors generate intermediate images I1, I2, I3, and I4 that are composited in a front to back order. This partitioning technique allows us to use a larger number of processors. However, the overhead due to wasted computation is more severe for this partitioning scheme.

3.1.2 Image Space Partitioning

One simple technique for partitioning the computation (and required volume data) is by slicing the final image along its scanlines and assigning them to different processors. A horizontal slicing plane through the final (warped) image is however not a horizontal slicing plane through the volume data and intermediate image if the volume is sheared. This implies that the partitions cut across scanlines. Cutting across scanlines reduces coherence in the image and volume space. As the number of processors increases, scanlines might be cut into a large number of segments and eventually, all benefit of spatial coherence in volume and image may be lost. As pointed out in [3], coherence in volume and image space is responsible for reducing the computational complexity of this problem from $O(n^3)$ to $O(n^2)$ and exploiting this coherence leads to significant performance gains in practice.

3.1.3 Sheared Volume Space Partitioning

A principal drawback of volume space partitioning is the excess computation resulting from an inability to incorporate early termination effectively. One way of remedying this is using partitioning planes parallel to the rays. Combining this with our earlier restriction that any partitioning plane must be parallel to the scanlines yields sheared volume space partitioning illustrated in Figure 4.

In this partitioning technique, the volume is first sheared and then partitioned by slicing orthogonal to volume slices. Each processor can now drive rays through its assigned volume segment. The resulting intermediate images at different processors are disjoint and can be independently warped. Since the partial intermediate images are disjoint, the corresponding partial warped images are also disjoint. In this way, no compositing is required across processor boundaries. The resulting final image segments are then assembled at a single processor where they can be displayed.

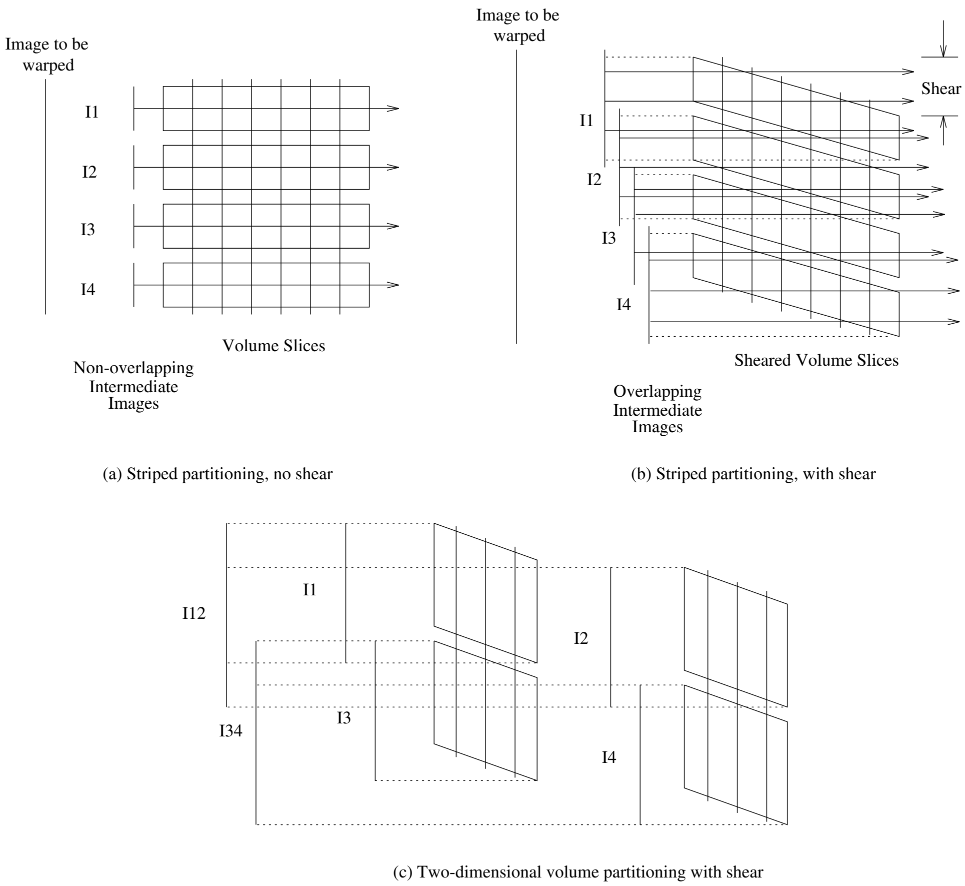


Figure 3: Volume space partitioning for four processors.

The principal overhead of this algorithm results from communication of volume data when the volume is sheared. This communication is illustrated in Figures 4(b,c). When the volume shear is changed from a to b, the shaded scanlines must be communicated. The volume of communication is a function of the relative shear between two orientations. Since the objective of real time rendering is to generate a smooth motion of the object, the relative shear between successive orientation is not expected to be significant. Therefore, the communication overhead incurred by this partitioning technique is not expected to be significant.

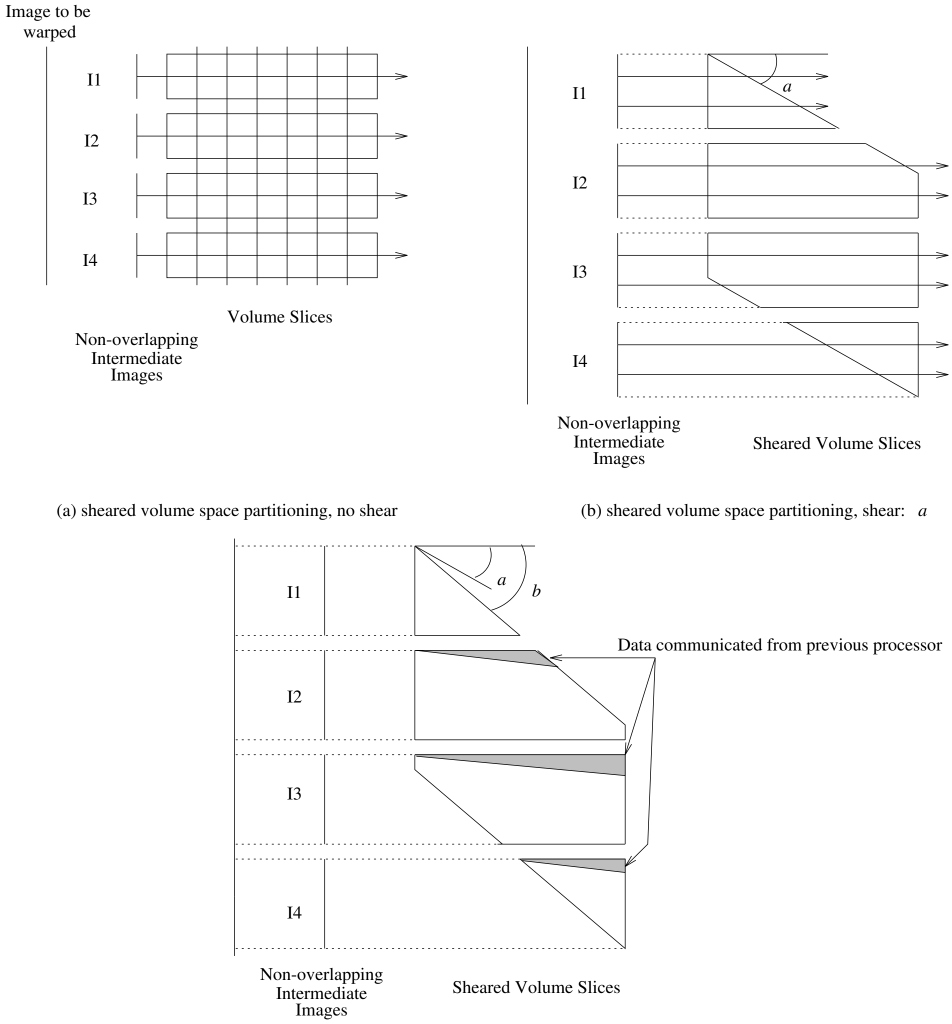
Note that for neighboring processors that share a boundary in the sheared volume space partitioning, one scanline per slice is replicated on both sides of the boundary. No additional compositing computation is required with this scheme. The entire volume is traversed only once exactly like the original sequential algorithm [3], except for the replicated scanlines. When processor boundaries change (due to relative shear between subsequent frames or due to load balancing) additional communication is required for replication of volume scanlines on processor boundaries.

To further illustrate the reduction of communication overhead in using sheared volume space partitioning compared to volume space partitioning, we note that the algorithm described in [6, 1] uses volume partitioning. They have a phase they call "image composition". This is identical to what we have to do for compositing of the overlap areas of the image if we use volume space partitioning (as described in Section 3.1.1). For a 128^3 voxel dataset and 256^2 pixel image using 128 processors on a CM-5, this phase takes 128.7 msecs in [6, 1]. As we will see later, our entire algorithm (not just this particular communication overhead) based on sheared volume space partitioning takes less than this time for a larger volume and identical image size using the same number of processors of the same parallel computer.

3.2 Load Balancing

Run-length encoding of scanlines combined with early termination makes it possible to skip empty voxels and saturated pixels. This implies that different scanlines may have widely differing amounts of computation associated with them. Therefore, naively partitioning the scanlines by assigning an equal number to each processor results in significant load imbalances. Furthermore, it is impossible to accurately determine the computational load of a scanline a-priori. Therefore, load balancing techniques must use heuristics to determine load and balance load accordingly.

We augment our parallel formulation with an adaptive load balancing scheme. Since the volume does not move significantly between two successive frames, it is possible to use load information from one iteration as an estimate of the load at the processor during the next iteration. The time to process a scanline is used as an indication of the load. Each processor keeps track of the time taken to compute the scanlines assigned to it. This time is made available to all the processors through a single all-to-all broadcast. We also assume that neighboring scanlines have similar loads associated with them. Therefore the load is assumed to be uniformly distributed among the scanlines assigned to a processor. Based on this assumption, each processor can determine the load at each scanline. Knowing the total load and load at each processor, a processor can compute the new partitions independently. This is used



(a) sheared volume space partitioning, no shear

(b) sheared volume space partitioning, shear: *a*

(c) relative shear of *b-a* and resulting communication

Figure 4: Sheared volume space partitioning for four processors.

in conjunction with the new value of shear to determine the new destination of each scanline assigned to a processor in the current destination. In this way, the communication due to shear and load balancing is integrated.

It is not necessary to balance load at each frame. This frequency is a function of the dataset. Although, we adjust this frequency of load balance manually, it can be triggered when the load difference between the highest and the least loaded processor as a ratio of total load crosses a certain threshold. In certain instances, we noticed that the best performance was obtained when load was balanced just once (at the start) in 89 frames.

The load balancing strategy described above can be viewed as belonging to a larger class of lookahead load-balancing schemes. Instead of using just one frame to predict loads for the next frame, it is possible to use more than one frame and extrapolate to obtain load information for subsequent frames. Furthermore, it is possible to store loads of scanlines separately rather than assuming them to be constant across partitions. This increases the communication associated with the all-to-all broadcast of loads and may not result in a significant improvement in performance.

4 Analytical Model for Performance Estimation

The objective of designing an algorithm whose performance scales to a large number of processors requires that all steps of the algorithm are effectively parallelized. These include the construction of the shading-table, volume data communication, compositing, and warp. In this section, we discuss each step and associated time and space requirements.

4.1 Notation and Basic Assumptions

The following terminology will be used in this section:

- The volume dataset is assumed to be of size $n \times n \times n$ voxels.
- The final rendered image is assumed to be of size $n \times n$ pixels.
- The parallel computer is assumed to have p processors connected using a direct network. The network is assumed to have an $O(p)$ bisection width. (Networks such as fat tree and hypercube fall in this class. This assumption is made for

analytical purposes only. The algorithm is suited for lower degree networks such as meshes also.)

- The time taken to send a message of size b bytes from one processor to another on an uncongested network is $T_s + T_w b$ time units where T_s is a start-up time constant and T_w is the inverse of the network link bandwidth. Messages using the same network link, must be serialized. We explicitly account for the serialized time in our analysis.

4.2 Analytical Model

In this section we analyze the runtime of various components of the algorithm:

Shade-Table Computation The shading table is a lookup table of approximately 8K (single precision floating point) entries. These entries can be computed independently in parallel. Therefore, it is possible to parallelize this phase by partitioning the table entries equally among the processors and computing them independently. However, after computing the assigned entries, each processor requires the entire shading table. This is accomplished using a single all-to-all broadcast operation[3]. If each message is of length $b = \frac{8\text{K} \times 4}{p} = \frac{32\text{K}}{p}$ bytes, the time taken on a CM-5 hyper-tree network is $T_s \log p + T_w bp$.

From this expression, it is clear that both computation and the communication overhead increase linearly with the size of the table for a fixed number of processors. However, the time taken to compute a single entry of the table is significantly higher than the time to communicate it. Therefore, as the table size increases or the number of processors decreases, the time spent in computation increases as a fraction of total time resulting in higher speedups.

Communication for Sheared Volume Partitioning As the orientation of the volume changes, the shear may change. This requires communication of scanlines between processors (Figure 4(c)). Typically the relative shear, θ radians, ($= b - a$ in Figure 4(c)) between successive frames is small. For an absolute shear of β radians, slice i is displaced downwards by $i \tan \beta \approx i\beta$. Therefore, a relative shear of $\theta = b - a$ results in a downward displacement of $i(\tan b - \tan a) \approx i(b - a) \approx i\theta$ for the ith slice. The total downward displacement for all the slices is a summation of this expression from $i = 1$ to n or $\frac{n(n+1)}{2}\theta$ scanlines. These scanlines are shifted to neighboring processors. The last scanline gets displaced the most (by $n\theta$). We make a simplifying assumption that each processor is assigned equal scanlines. We will discuss the implications of relaxing this assumption when we discuss load balancing techniques. If each processor gets n/p scanlines, the maximum downward displacement corresponds to communication between processor p_i and processor p_{i+w}, where $w = \frac{n\theta}{n/p} = p\theta$. Since θ is small, $w = p\theta$ is a small constant close to 1. Finally, each scan line has from $O(1)$ to $O(n)$ non-transparent voxels each of which is represented by 4 bytes. Therefore, the total time taken is

$$\begin{aligned} &O(wT_s + \frac{n(n+1)}{2}\theta n \times 4 \times T_w) \\ &= O(wT_s + 2n^3\theta T_w) \end{aligned}$$

To get an idea of the value of θ, consider a relative shear of one degree between two frames. This corresponds to a shear of $\theta = 0.017$. For upto 128 processors, $w = p\theta \approx 2$. From this, it is clear that the communication overhead of our parallel formulation is minimal. Our performance results later support this observation.

Compositing Slices to form Intermediate Images As pointed out in Section 3 the compositing phase in sheared volume space partitioning can proceed independently at various processors. We assume that each processor has equal computational load. Although there are a total of $O(n^3)$ voxels in the dataset, a significant fraction of them are transparent. Furthermore, a large number of them are never visited due to early termination of rays. Lacroute and Levoy [3] estimate that the computational complexity of this phase of the algorithm is $O(n^2)$. Assuming ideal load balance, the parallel runtime of this phase is given by $O(\frac{n^2}{p})$.

Warping In the warp phase, for each pixel in the final image, the four nearest intermediate image pixels are located and interpolated. Since there are n^2 such pixels and each pixel requires a constant amount of computation, the computational complexity of this phase is $\Theta(n^2)$. We assume that the final image is equally partitioned among the processors. It is not possible to ensure this while balancing load during the compositing phase. However, the computational phase is much more expensive than the warp phase and we have observed that balancing load in the compositing phase does not lead to significant imbalances in the warp phase. Since this phase is perfectly parallelizable and requires no communication, the parallel warp time with good load balancing is $O(\frac{n^2}{p})$.

Image Assembly After the warp phase, each processor has a segment of the final image. This must be assembled at one processor for display. Note that more than one processor may contribute to a particular pixel's value in the final image if the four nearest intermediate image pixels are spread among those processors. Since addition is associative, the final image pixel's value remains the same as in the original sequential algorithm without requiring any additional computation or communication for pixels on the boundary between neighboring processors. For the sake of analysis, we assume that each processor has an equal number of final image pixels, n^2/p. Image assembly requires that $O(\frac{n^2}{p})$ pixel data (one byte per pixel for gray scale) at each processor be assembled at a single processor. This is done by sending a message with $O(\frac{n^2}{p})$ bytes from each processor to the final destination processor. Communication time taken is $O(p \times (T_s + T_w \frac{n^2}{p}))$. In typical renderings, a significant part of the image corresponds to empty space (i.e. background color). The amount of data communicated can be reduced by eliminating these pixels. In our implementation, these pixels are not sent in the messages to the final destination processor.

5 Experimental Results

In this section, we report on the implementation of a massively parallel volume renderer. The parallel formulation is based on the sheared volume space partitioning. The message passing implementation, using the CMMD message-passing library, was run on up to 128 processing elements of a Thinking Machines CM-5 parallel computer. The code itself is portable to such other platforms as workstation clusters, nCUBE 2 and the Cray T3D among others. Each processing element of the CM-5 multicomputer is a Viking Sparc microprocessor running at 40 MHz. We determined that for this application, each of these processing elements is approximately three times slower than an Indigo R4000 workstation (by comparing the runtimes on the two machines for the dataset used in Table 1). The processing elements are connected using a

[3] All-to-all broadcast requires that each processor starts with some data. At the end of the all-to-all broadcast, each processor contains all the data from all the processors. This can be done in a tree of communication messages with message sizes doubling at each level of the tree. See [2] for more details.

Fat Tree interconnection network. The interconnection network is capable of yielding per-processor bandwidths in the range of 10 - 20 Megabytes per second.

We tested our parallel formulation on a number of datasets. We report on the rendering time of two datasets here: the "brain" dataset of dimensions $256 \times 256 \times 167$ and the smaller 1:2 subsampled version of this brain dataset of dimensions $128 \times 128 \times 84$. The "brain" dataset is the same as the one used in [3]. In each case, the final image size was 256×256. Since the communication overhead changes as the dataset is rotated, the per-frame times are computed as an average of 89 frames displaced from each other by a rotation of one degree. Figure 5 shows an example of parallel rendering for the larger brain dataset.

Figure 5: Parallel rendering of the large brain dataset of 256x256x167 voxels

The program consists of four main functional modules. The construction of the lookup table, load balance and shear, compositing and warp, and assembly. The lookup table is a table of approximately 8K entries which are computed independently. Each processor computes an equal number of entries. These are then made available to everyone else through a single all-to-all broadcast operation [2]. Shearing the volume results in communication between processors. This communication is integrated with the communication resulting from load balance. Compositing and warp are completely local operations. Each processor is now left with segments of the final image. These are put together using an optimized assembly operation that eliminates blank pixels on either extremity of the image. The load balanced version of the program starts by assigning equal number of scanlines to all processors. For each successive iteration, load is balanced using load estimates from the previous iteration.

We study the implementation of our parallel formulation with a view to establishing raw performance, scalability and impact of load balancing (in descending order of importance). In particular, we do not consider speedup results to be important by themselves. Table 2 presents runtimes (in msecs) of the parallel formulations with and without load balance. Figure 6 presents the corresponding speedup curves for the larger dataset. The smaller dataset is not run with 128 processors because there is insufficient parallelism for this case. A number of observations are evident from these results:

P	Volume			
	No Load Balance		Load Balance	
	LARGE	*SMALL*	*LARGE*	*SMALL*
1	3193	976	3193	976
2	1627	548	1625	551
4	892	309	910	310
8	620	197	593	196
16	345	127	327	121
32	216	86	204	81
64	142	74	118	70
128	103		85	

Table 2: Rendering time (in msecs) per frame on CM-5 parallel processor (in msecs). The $256 \times 256 \times 167$ dataset is indicated by *LARGE*, and the $128 \times 128 \times 84$ dataset is indicated by *SMALL*.

- Per-frame times of 85 ms have been demonstrated for the larger dataset. These were obtained without using the vector units on the CM-5 nodes. This leads us to believe that with state of the art processors (which are significantly faster than CM-5 nodes) and interconnects, real time rendering is within reach for 256^3 voxel datasets.
- The speedup increases consistently up to 128 processors for the larger dataset. This implies that provided there is available concurrency (scanlines), the parallel formulation scales up to at least 128 processors.
- Load balancing helps reduce the overall time of execution in some cases by up to 16%. The gains from load balancing are a function of the dataset and the number of processors.

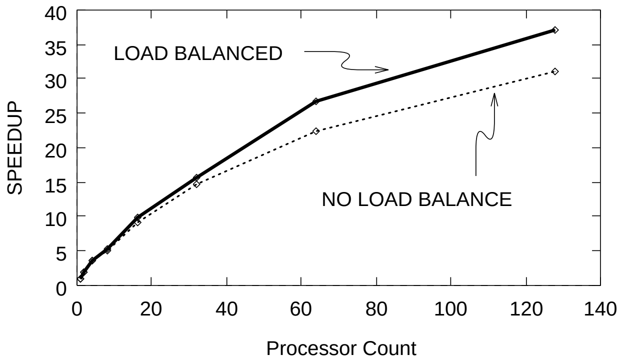


Figure 6: Speedups obtained from the load balanced and load imbalanced versions of parallel rendering for up to 128 processors on CM-5 for the larger dataset

These results compare very favorably with existing results of other researchers. Lacroute and Levoy achieved 10 frames/sec on an SGI Challenge 16 processor shared-memory computer for a comparable data set by parallelizing the resampling and compositing steps (but apparently not the warp step which can take 10-20% of the time by their estimate). Since no details are available for this implementation, we are unable to do a detailed comparison. However, we can make the following limited observations. The data set used by them was a 256x256x225 voxel "head" generated from a CT scan. Each node of a Challenge multiprocessor is significantly faster than an Indigo R4000 workstation (43% faster for our large dataset on a Challenge and Indigo available to us). This implies that each node of a CM-5 is 4.2 times slower than one node of a Challenge multiprocessor (and 3 times slower than an Indigo R4000 workstation as mentioned earlier). Our formulation achieves a performance of 12 frames per second on a 128 processor CM-5. The implications of the result are however deeper than that. The scalability properties of the shared memory implementation are not understood. In fact, performance on shared bus multicomputers is rarely known to scale to a large number of processors.

There have been other attempts at parallelizing volume rendering. The best rendering times reported in these papers range from a few hundred milliseconds [7] to a few seconds [1, 6]. Our results are over an order of magnitude better than most of these results and more than two orders of magnitude better than some of them [1, 6].

6 Conclusions and Future Work

In this paper, we have presented a parallel formulation of the shear-warp rendering algorithm and demonstrated scalable performance up to 128 processors of a CM-5. Novel data partitioning and load-balancing methods were instrumental in achieving the best performance reported so far on this problem.

In our continuing work, we are exploring the following improvements. First, we are developing alternate formulations capable of larger degrees of parallelism. Second, we are reducing communication overhead in the algorithm. In the parallel formulation presented here, entire slices of the volume are sheared irrespective of whether they are required or not. This is because it is impossible to estimate a priori the depth at which a ray terminates. One way of improving the communication overhead of this formulation is to fetch voxel data as and when they are required. These first two improvements will reduce the communication overhead due to sheared volume partitioning and allow us to use more than $O(n)$ processors. Third, we are exploring the use of a distributed frame buffer to reduce communication overhead. One major source of overhead in this formulation is the process of putting the sub-images generated at various processors together into a single image. This overhead can be circumvented using a distributed frame buffer.

Processor speeds have increased significantly beyond the processors used in the CM-5. This is accompanied by an increase in communication speeds. All of this implies that the same formulation implemented on a state-of-the-art machine is capable of yielding real time rendering. We are currently exploring platforms such as other MPPs, the Convex Exemplar (based on the PA-RISC architecture), low-cost workstation clusters using high-speed interconnects (such as ATM and Myrinet), and multiprocessors based on dedicated digital signal processors.

Acknowledgements

We are grateful to the following people: Philippe Lacroute and Marc Levoy for giving us access to their original sequential code; Tom Malzbender for pointing us in the right direction; Fred Kitson and Chris Hsiung for their encouragement and support.

REFERENCES

[1] HANSEN, C. D., KROGH, M., AND WHITE, W. Massively Parallel Visualization: Parallel Rendering. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing* (Feb. 1995), pp. 790–795.

[2] KUMAR, V., GRAMA, A., GUPTA, A., AND KARYPIS, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, 1994.

[3] LACROUTE, P., AND LEVOY, M. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proceedings of the SIGGRAPH 94 Conference* (July 1994), pp. 451–457.

[4] LAUR, D., AND HANRAHAN, P. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Computer Graphics 25*, 4 (July 1991), 285–288.

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

[6] MA, K.-L., PAINTER, J. S., HANSEN, C. D., AND KROGH, M. F. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering. In *Proceedings of the 1993 Parallel Rendering Symposium* (Oct. 1993), ACM, pp. 15–22.

[7] NIEH, J., AND LEVOY, M. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *Proceedings of the Boston Workshop on Volume Visualization* (October 1992).

[8] WILHELMS, J., AND GELDER, A. V. A Coherent Projection Approach for Direct Volume Rendering. *Computer Graphics 25*, 4 (July 1991), 275–284.