Parallel Performance Measures for Volume Ray Casting

Cláudio T. Silva and Arie E. Kaufman

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We describe a technique for achieving fast volume ray casting on parallel machines, using a load balancing scheme and an efficient pipelined approach to compositing. We propose a new model for measuring the amount of work one needs to perform in order to render a given volume, and use this model to obtain a better load balancing scheme for distributed memory machines. We also discuss in detail the design tradeoffs of our technique. In order to validate our model we have implemented it on the Intel iPSC/860 and the Intel Paragon, and conducted a detailed performance analysis.

1 Introduction

As researchers and engineers use volume rendering to study complex physical and abstract structures they need a coherent, powerful, easy to use visualization tool, that lets them *interactively* change all the necessary parameters. Unfortunately, even with the latest volume rendering acceleration techniques running on top-of-the-line workstations, it still takes a few seconds to a few minutes to volume render images. This is clearly far from interactive. With the advent of parallel machines, scanners and instrumentation, larger and larger datasets (typically from 32MB to 512MB) are being generated that would not even fit in memory of a workstation class machine. Even if rendering time is not a major concern, big datasets may be expensive to hold in storage, and extremely slow to transfer to a typical workstations over network links.

These problems lead to the question of whether the visualization should be performed directly on the parallel machines that generate the simulation data or sent over to a high performance graphics workstation for post-processing. First, if the visualization software was integrated in the simulation software, there would be no need for extra storage and visualization could be an active part of the simulation. Second, large parallel machines can render these datasets faster than workstations can, possibly in real-time or at least giving the possibility of achieving interactive rates. Finally, if real integration between the simulation and the visualization tool is possible, one could interactively "steer" the simulation, and possibly terminate simulations that are wrong or uninteresting at an earlier stage instead of performing long and expensive archiving operations for the generated datasets. In this paper we focus on the architecture and performance measures of visualization algorithms that are running directly on the parallel machines.

Clearly, an algorithm that runs on a parallel machine has to be efficient and should be able to make good use of the computing power. A conservative tradeoff between scalability and actual processing speed is very important. Also, the algorithm has to be space efficient, and for the case of a distributed memory MIMD machine, memory duplication should be avoided. In this paper we propose a space efficient, fast parallel algorithm that addresses these issues. This algorithm will be the basis of a visualization library in the molds just described using the VolVis system [1] as its front end.

A large number of parallel algorithms for volume rendering have been recently proposed. Schroeder and Salem [13] have proposed a shear based technique for the CM-2 that could render 128^3 volumes at multiple frames a second, using a low quality filter. The main drawback of their technique is low image quality. Their algorithm had to redistribute and resample the dataset for each view change. Montani et al. [10] developed a distributed memory ray tracer for the nCUBE, that used a hybrid image-based load balancing and context sensitive volume distribution. An interesting point of their algorithm is the use of clusters to generate higher drawing rates at the expense of data replication. However, their rendering times are well over interactive times. Using a different volume distribution strategy but still a static data distribution, Ma et al. [9] have achieved better frame rates on a CM-5. In their approach the dataset is distributed in a K-d tree fashion and the compositing is done in a tree structure. Others [6, 3, 11] have used similar load balancing schemes using static data distribution, for either image compositing or ray dataflow compositing.

Nieh and Levoy [12] have parallelized an efficient volume ray caster [8] and achieved very impressive performance on a shared memory DASH machine.

In this paper we concentrate on the parallelization of a simple but fast method for ray casting, called PARC (polygon assisted ray casting) [2]. Our parallel implementation uses a static data decomposition and an image compositing scheme. We have implementations that work on the Intel iPSC/860 and the Intel Paragon. In Section 2 we explain the important issues in designing and writing a parallel ray caster, followed by Section 3, where we study a new method for measuring the work done by a ray caster. In Section 4 we describe our algorithm and its implementation.

2 Performance Considerations

In analyzing the performance of parallel algorithms, there are many considerations related to the machine limitations, like for instance, communication network latency and throughput [11]. *Latency* can be measured as the time it takes a message to leave the source processor and be received at the destination end. *Throughput* is the amount of data that can be sent on the connection per unit time. These numbers are particularly important for algorithms in distributed memory architectures. They can change the behavior of a given algorithm enough to make it completely impractical.

Throughput is not a big issue for methods based on volume ray casting that perform static data distribution with ray dataflow as most of the communication is amortized over time [10, 6, 3]. On the other hand, methods that perform compositing at the end of rendering or that have communication scheduled as an implicit synchronization phase have a higher chance of experiencing throughput problems. The reason for this is that communication is scheduled all at the same time, usually exceeding the machines architectural limits. One should try to avoid synchronized phases as much as possible.

Latency is always a major concern, any algorithm that requires communication pays a price for using the network. The start up time for message communication is usually long compared to CPU speeds. For instance, in the iPSC/860 it takes at least 200μs to complete a round trip message between two processors. Latency hiding is an important issue in most algorithms, if an algorithm often blocks waiting for data on other processors to continue its execution, it is very likely this algorithm will perform badly. The classic ways to hide latency is to use pipelining or prefetching [5].

Even though latency and throughput are very important issues in the design and implementation of a parallel algorithm, the most important issue by far is *load balancing*. No parallel algorithm can perform well without a good load balancing scheme.

Again, it is extremely important that the algorithm has as few inherently sequential parts as possible if at all. Amadahl's law [5] shows how speed up depends on the parallelism available in your particular algorithm and that *any*, however small, sequential part will eventually limit the speed up of your algorithm.

Given all the constraints above, it is clear that to obtain good load balancing one wants an algorithm that:

- Needs low throughput and spreads communication well over the course of execution.
- Hides the latency, possibly by pipelining the operations and working on more than one image over time.
- Never causes processors to idle and/or wait for others without doing *useful work*.

A subtle point in our requirements is in the last phrase, how do we classify *useful work* ? We define useful work as the number of instructions I_{opt} executed by the best sequential algorithm available to volume render a dataset. Thus, when a given parallel implementation uses a suboptimal algorithm, it ends up using a much larger number of instructions than theoretically necessary as each processor executes more instructions than $\frac{I_{opt}}{P}$ (P denotes the number of processors). Clearly, one needs to compare with the best sequential algorithm as this is the actual speed up the user gets by using the parallel algorithm instead of the sequential one.

The last point on useful work is usually neglected in papers on parallel volume rendering and we believe this is a serious flaw in some previous approaches to the problem. In particular, it is widely known that given a transfer function and some segmentation bounds, the amount of useful information in a volume is only a fraction of its total size. Based on this fact, we can claim that algorithms that use static data distribution based only on spatial considerations are presenting "efficiency" numbers that can be inaccurate, maybe by a large margin.

To avoid the pitfalls of normal static data distribution, we present in the next section a new way to achieve realistic load balancing. Our load balancing scheme, does not scale linearly as others claimed before, but achieves very fast rendering times while minimizing the "work" done by the processors.

3 Load Balancing

This section explains our new approach to load balancing, which is based on the PARC (polygon assisted ray casting) algorithm [2]. The section presents a short description of PARC and describes different approaches to using it as a load balancing technique. PARC can be characterized as a *presence acceleration*

technique [4], like the octree decompositions of Levoy [8]. Instead of stepping through the whole volume for rendering, only the parts that contain relevant data are used, this can save an enormous amount of rendering time, not only in volume stepping, but also because it greatly decreases the number of compositing and shading calculations one needs to perform.

The rational behind PARC is simple. As one needs to calculate the integral $I = \int_{t_0}^{t_1} e^{-\int_{t_0}^{t} \sigma(s)ds} I(t)dt$ during rendering, PARC finds tighter bounds for t_0 and t_1, thus, substantially lowering the rendering time. PARC does this by enclosing the volume with a rough polygonal approximation, which is transformed and scan converted into front and back Z buffers. For each ray the front one gives us a conservative estimate for t_0, and the back one gives the t_1 estimate.

In order to skip over empty space inside volumes, our implementation of PARC uses pre-calculated cubes aligned with the primary axes to bound cubes inside the volume. For each particular view, we scan convert the cubes into a Z buffer (implemented in software) to obtain closer bounds on the intervals where the ray integrals need to be calculated. This method achieves speeds comparable with the fastest high quality volume renderers.

One can specify the number of cubes in the subdivision of the original dataset. This determines the accuracy of the t_0 and t_1 estimates; the higher the number of cubes, the closer to the exact intersection points they are. If the estimates are accurate, we perform less work on the ray, but on the other hand the scan conversion time is higher as the number of cubes grows very fast. For instance, one can ask for a *level* 4 *PARC approximation*, this means the dataset is partitioned to 2^4 intervals in *each* of the coordinate directions, for a total of 4096 small cubes. Depending on the low and high threshold specified, one usually gets a much lower number of such cubes. For instance, with a level 4 PARC approximation of a CT 3D reconstructed head at a 20-200 threshold, only 38% of the cubes are non-empty.

The cubes generated by PARC are the basic units for our load balancing. As the cubes are very close approximation of the amount of work one has to perform during ray tracing, we use the number of cubes a processor has as the measure of how much work is performed by that particular processor. Let P denote the number of processors, and c_i the number of cubes processor i has. To achieve a good load balancing we need a scheme that *minimizes* the following heuristic function for a partition $X = (c_1, c_2, \ldots)$:

$$f(X) = \max_{i \neq j} |c_i - c_j|, \forall i, j \leq P \qquad (1)$$

The main problem in implementing this approach is that for ray casting to be efficient the dataset part of a particular processor need to be contiguous. Not only this makes compositing easier but it also reduces the number of intersection calculations required. Once one decides what shape to assign to each processor, one just needs to use either Equation 1 or a variation of it. For the rest of the paper we describe an implementation of our load balancing scheme that uses slabs, which are consecutive slices of the dataset aligned on two major axes, as the basic partition blocks of the dataset for load balancing. Slabs are very easy to implement and we show that they provide a good sense of load balance. In the case of slabs, the PARC algorithm produces an ordered list of number: $b_1, b_2, \ldots, b_n$, which are the number of cubes in each slab. We need to find indices pairs $(k_1^1, k_1^2), (k_2^1, k_2^2) \ldots, (k_P^1, k_P^2)$, that minimizes the following expression:

$$f(X) = \max_{i \neq j} \left| \sum_{m=k_i^1}^{k_i^2} b_m - \sum_{m=k_j^1}^{k_j^2} b_m \right|, \forall i, j \leq P \qquad (2)$$

The problem of computing the optimal (as defined by our heuristic choice) load balance partition indices can be solved naively as follows. We can compute all the possible partitions of the integer n, where n is the number of slabs, into P numbers, where P is the number of processors. For example, if $n = 5$, and $P = 3$, then $1 + 1 + 3$ represents the solution that gives the first slab to the first processor, the second slab to the second processor and the remaining three slabs to the third processor. Enumerating all possible partitioning to get the optimal one is a feasible solution but can be very computationally expensive for large n and P. At this time we have a Prolog implementation of a slightly revised algorithm. Instead of calculating the minmax Equation 2, we choose the permutation with the smallest square difference from the average.

In order to show how well our approach works in practice, let us work out the example of using our load balancing example to divide the *neghip* dataset (the negative potential of a high-potential iron protein of 66^3 resolution) for four processors, using a level 4 PARC decomposition with a 10 to 200 value threshold. After running PARC we get the following 16 numbers, one for each slab, out of the 1570 total cubes:{12, 28, 61, 138, 149, 154, 139, 104, 106, 139, 156, 151, 129, 62, 29, 13}. The naive approach of other volume renderers has been to assign an equal part of the volume to each processor, resulting in the following partition: {12+28+61+138=239, 149+154+139+104=546, 106+139+156+151=552, 129+62+29+13=233}, where processors 2 and 3 have twice as much work than processor 1 and 4. Our approach based on Equation 2 gives us {388, 397, 401, 384}, clearly a much more balanced solution.

One can see that some configurations will yield better load balancing than others but this is a limitation

of the particular space subdivision one chooses to implement, the more complex the subdivision one allows, the better load balancing but the harder it is to implement a suitable load balancing scheme and the associated ray caster. Figure 1 plots the examples just described for the naive approach. Figure 2 shows how well our load balancing scheme works for a broader set of processor arrangements. By comparing both plots, one can see that our algorithm generates much smoother curves, thus leading to better load balancing.

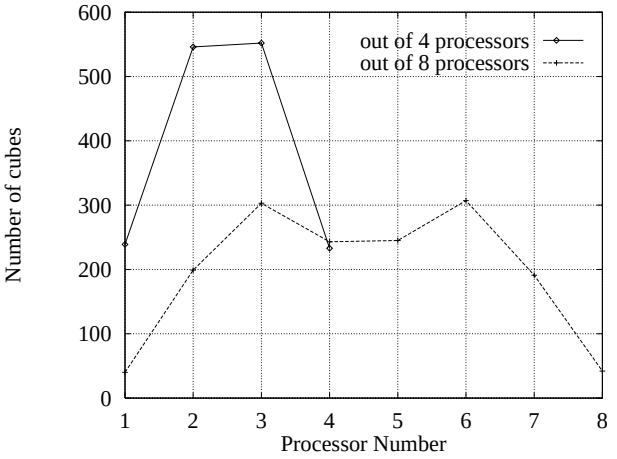

Figure 1: The graph shows the number of cubes per processor under naive load balancing.

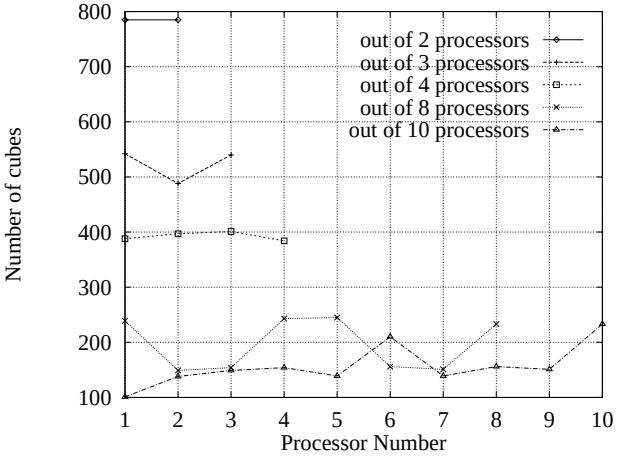

Figure 2: Load balancing measures for our algorithm. The graph shows the number of cubes the processor receives in our algorithm.

Figures 3 and 4 show the rendering times on the Intel Paragon, showing the correlation between the number of cubes a processor has and the amount of work it has to perform. By comparing these graphs

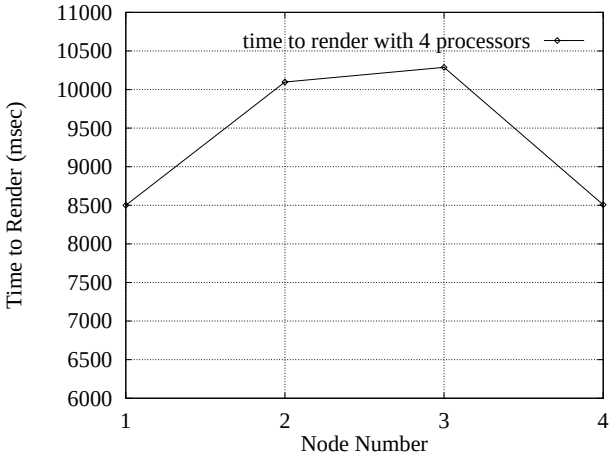

Figure 3: Naive load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using the naive load balancing.

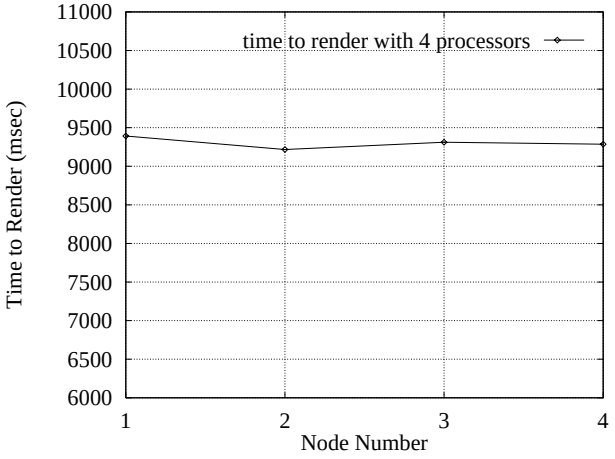

Figure 4: Our load balancing on the Paragon. The graph shows the actual rendering times for 4 processors using our load balancing.

and those in Figures 1 and 2, one can observe that our load balancing is effective and accurate, compared to the naive approach of equally subdividing the dataset. If one was calculating a single image, the total rendering time of the image subparts would be the maximum of every processor plus the compositing time. As will be seen in the next section, we use a pipeline approach to optimize image generation performance, by amortizing compositing over time.

4 Parallel Ray Casting

The version of our parallel PARC-based volume ray caster described here uses the NX/2 library on the iPSC/860 and the Paragon, although previous versions also ran under TCP/IP on workstations. We plan to release a production level version of this code on the iPSC/860, Paragon, PVM, network workstations (TCP/IP) together with a distribution version of VolVis [1].

In order to avoid the processors having direct access to the dataset description files, we chose to broadcast once the necessary information for the rendering, like the dataset, processor assignments, transfer functions, and so on, and have all the processors synchronize during this phase. Clearly, this may make our implementation unsuitable for someone that needs to generate only a single image, specially because some machines (like the iPSC/860) have slow processor access to NFS mounted files. The best scenario is one where the datasets are generated on the parallel machine, and not moved in and out at all.

After initialization, user commands representing different viewing angles are sent to all the processors by broadcast messages. Only information like transformation matrices and image sizes are sent at this time to minimize the communication cost per image. In order to avoid flooding the parallel machine with requests, a feedback synchronization technique is used. It basically balances the requests rate with the machine power available. The flow of messages in the algorithm is shown in Figure 5.

The feedback synchronization techniques we use are based on work by Van Jacobson [7], who designed a set of techniques to avoid congestion in TCP/IP networks. We use a variation of his *slow-start* and *round-trip-time estimation* technique, where the host slowly sends requests and adaptively changes the rate of requests with the feedback it receives from the network. This is implemented by having the host keep the number of outstanding image render requests, and setting a maximum on this number based on the number of processors and the amount of memory each has. At the start of the computation the host begins sending image requests to the processors, and for every image received it sends two requests to the processors until the maximum is achieved. Also the host keeps a running average of the time taken to compute an image,

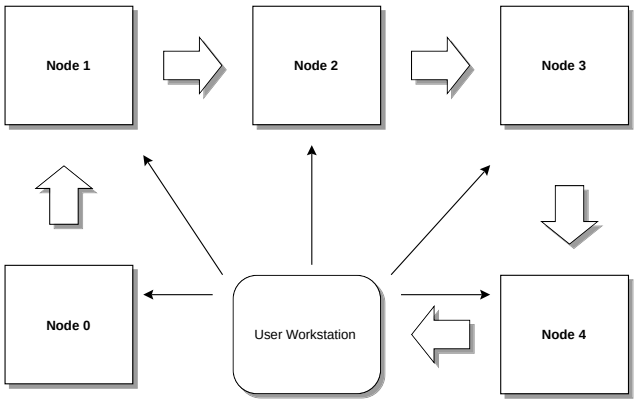

Figure 5: Overview of communication flow in the algorithm. Arrow width represents the expected bandwidth necessary in each communication link.

computed as $T_f = \alpha T_i + (1 - \alpha)M$, where T_f is the new estimate, T_i was the initial estimate, M is the time measured in the last image computation, and α is an amortization constant. By changing α we can make the host more or less responsive to changes in rendering times. By using this procedure, when this time increases the host can adaptively decrease the rate of requests or increase the rate if the processors begin computing images faster.

In the computing processors, a set of working requests are queued and serviced on demand. Basically, there are two different kinds of requests, *rendering requests* and *compositing requests*. The first type of request is received directly from the host (where supposedly the user is waiting for images to show up), while the second comes from other *slab neighboring* processors. The computing processor keeps servicing both types of requests, by picking a message from each queue.

While servicing a *rendering request* the processor allocates enough memory for it, renders it, and keeps the rendered image around until a *compositing request* for that particular image comes from its respective neighboring processor, then composites its part of the image and sends it over to the other neighboring processor, and continues working on a new *rendering request*. The last processor on a chain, sends the whole image back to the user's workstation. If a *compositing request* is received for an image that is not rendered yet we take the approach of computing it right away rather then delaying it as this could double our memory requirements for images. Once requests are serviced the memory is immediately freed.

This approach is simple and effective. One of the clear advantages is that if we disregard the message and synchronization overhead for a moment, we see that we are maximizing the computation overlap among processors and getting a much better utiliza-

tion of the communication network as messages are being sent during the whole course of the image computation time instead of just at a certain point in time.

One may claim that other, tree based, compositing schemes [9] may yield better results, however, the drawbacks of these schemes (low processor utilization during compositing and high network utilization during the peak of compositing) are major. Even though the tree approach would give a final image in $O(\log n)$ time steps, it still needs asymptotically the same number of messages. Therefore, it does not save any computation time, but it actually wastes it when some of the processors become idle.

The use of a pipelined compositing approach, where images are asynchronously generated and saved in buffers, requires the use of the feedback synchronization technique to avoid increasing the memory overhead without bounds. An interesting side effect of this technique is that our algorithm automatically adjusts itself to the rendering times of the particular machine and/or configuration being used, like the number of processors and network performance.

5 Performance Analysis

In this section we present a few performance figures of our algorithm and demonstrate that our approach is sound and fast. The main points that we are discussing are: the effectiveness of PARC load balancing, the communication overhead of the compositing scheme, algorithm behavior under different shading models, and overhead as compared to a sequential implementation. The effectiveness of our PARC load balancing was studied extensively in the last section, but to complete our choice of using PARC as our ray casting algorithm, it is interesting to compare its advantages to a more naive ray casting approach where no presence accelerations are adopted.

A conventional ray caster where the rays are cast from start to end by calculating intersections with the bounding box of the object is only slightly different from a PARC ray caster. A PARC ray caster actually does more work than a naive one, as it needs to scan convert and to find t_0 and t_1 from the Z buffer. The place that a PARC ray caster really gains performance is in the fact that it better approximates the volume bounds. It should be clear that the higher the cost of the shading function per step, the more advantageous it is to calculate these bounds well. In Figure 6, we can see how a PARC based ray caster performs against a naive ray caster under different shading functions. For our purposes we consider "light" shading a method that uses 5-10 instructions per sample, "medium" a method that uses 50 instructions per sample, and "heavy" shading functions require about 300 instructions per sample. Nieh and Levoy [12] have reported that trilinear interpolating a ray sample takes 320 instructions. One can see from Figure 6 that not only times but also the rate of increase of cost decreases as one computes more samples.

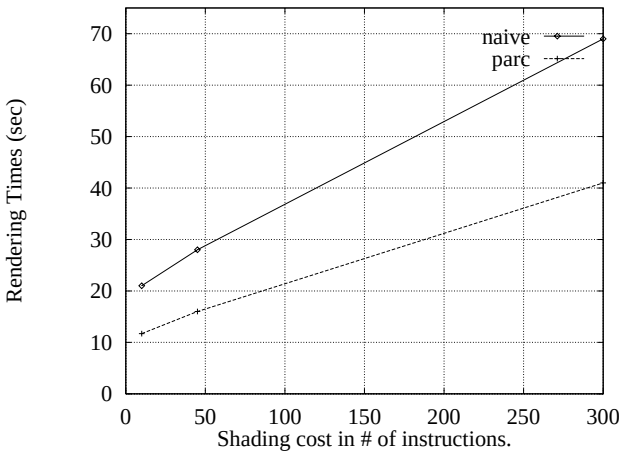

Figure 6: PARC versus naive ray casting. Times were calculated on a Sparc1000.

The work performed during rendering each ray can be broken into I_r, the initialization work, and W_r, the work performed to calculate and shade the samples along the ray. If perfect load balancing is achieved for every ray, each processor will perform $\frac{W_r}{P} + I_r$ work per ray, that is, the initialization time is replicated for every ray. If $W_r \gg I_r$, then we can achieve very high scalability with the algorithm, otherwise, as the number of processors increases the amount of work done on the initialization by all the processors PI_r gets larger than W_r, thus limiting the performance. This makes optimization of the initialization time critical to the performance of the algorithm.

Initialization time is composed of several components, the most time consuming being the PARC projection time and the transformation time. Right now it takes anywhere from 350 msec to 1600 msec to scan convert a level 4 PARC approximation on the machines we used. We believe scan conversion itself can be done on the order of 20 times faster when the code is re-written in a more efficient way, possibly in i860 specific code. By broadcasting at the beginning only the necessary PARC polygons we can also avoid increasing the number of polygons that need to be scan converted in each processor, and at the same time decrease the memory requirement. Until these changes get incorporated in our code, our timings are going to be around the 1 second mark, even if the rest of the algorithm takes no time at all. However, overall rendering times decreased substantially after we optimized our transformation time by factoring out all common matrix multiplications and inlining the ones inside tight loops.

All of the performance numbers presented in the rest of the section are for the Intel Paragon. The Intel

Paragon uses an Intel i860XP, a 50MHz superscalar microprocessor, and a 2D mesh interconnection network. Every processor of the machine actually contains two i860XPs, but only one is used for computation.

Figures 7 and 8 show the average time to composite different image sizes in three different machine configurations and for five different screen sizes, ignoring completely the rendering time. We use a slow start technique for these measures (only when pipelining). It is interesting to compare the figures as one can see that our pipelining method can very well hide the effects of the network and the work done to composite the image. For instance, every processor has to spend around 52 msec to composite a 300^2 image (only compute time), if we consider 6 processors, it will take over 300 msec of CPU time to generate this image, still with our pipelining approach the user only sees 65 msec as opposed to 330 msec a sequential composite requires.

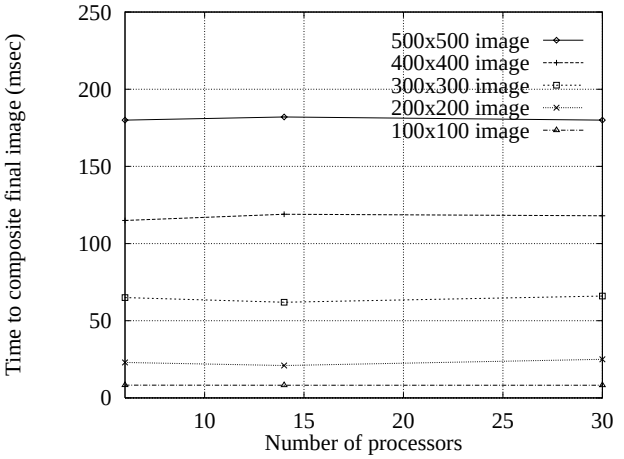

Figure 7: Timing as seen by a user of the arriving of images using our pipelining approach.

In Figure 9, we present some of our rendering times. These are rendering times for our first implementation and should not be regarded as what we are expecting for the production level code. One can see from the graph that our algorithm scales well as the number of processors increases. Also our prediction that the higher the shading cost, the better the parallel scalability can be seen from the graphs. We have filtered out the PARC rendering time from the numbers. We expect to speed the PARC projection times up by at least 20 times with the new scan conversion routines, and with a new set of fast PARC projection techniques being designed we anticipate getting scan conversion to under 25 msec. At this time, the best rates attainable by our algorithm are about 1.5 frames/sec on a 32 processor configuration of our Intel Paragon for a 256^2 image size. This is very competitive and even

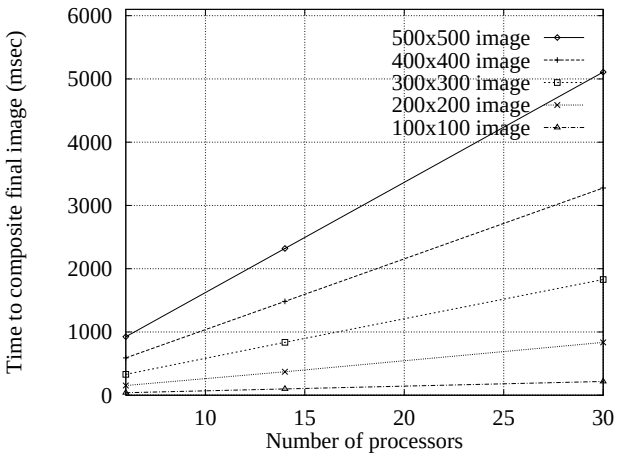

Figure 8: Timing as seen by a user of the arriving of images using sequential composite.

better than other rendering times published for machines with this number of processors.

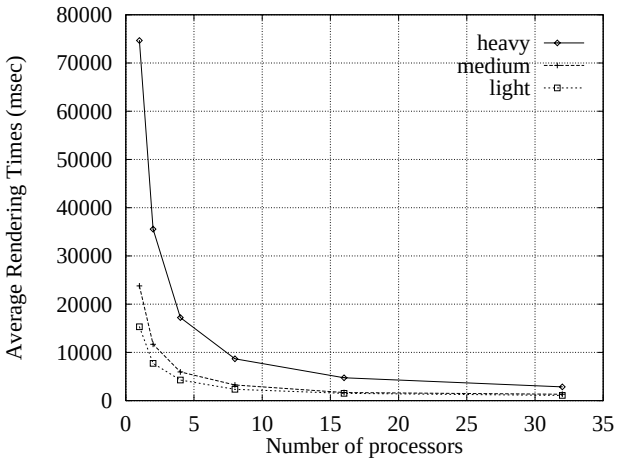

Figure 9: Rendering times on an Intel Paragon.

6 Conclusions and Future Work

We have shown that using PARC cubes for measuring useful work generates an intuitive way to load balance volume ray casting on distributed memory parallel machines. This not only generates a method that is theoretically sound but its preliminary implementation seems to present a method that is both efficient and scalable.

We have also proposed a new method for compositing that achieves better throughput than previous methods and that can be used to generate better refresh rates. If one cannot accept the delay pipelining imposes, one can always make judicious replication of volume data, for instance, one volume for every 16

processors to avoid long image delay times and still keep high refresh rates.

We believe our method is simple, fast, uses coherency and achieves high resource utilization on a given machine. As we use PARC, we achieve a high utilization of the compute processors and thus a very fast rendering time on every processor. Because of our pipelined compositing scheme, we achieve a much higher network utilization than other methods. Finally, our feedback synchronization image request technique guarantees a constant flow of information that adapts itself to different configurations of processor performance and network utilization.

Our current implementation can be greatly improved and optimized. One of our main concerns is to smoothly integrate all the parallel code into VolVis, so our users can take advantage not only of its intuitive and flexible user interface, but also of greater speed provided by parallel machines. Other plans include the porting of our algorithm to other architectures and a more detailed performance analysis of the whole algorithm. We are also planning on introducing optimization that would allow the system to use data replication and sharing whenever allowed. This way users with multiple processor shared-memory machines, like a network of Sparc1000s would be able to get better performance.

Another direction of future work is the extension of our load balancing technique to non-slabs partitions. The major problem is that computing optimal partitions in one dimention (the slab case) is already hard and computationally expensive. Another interesting question is whether this method can be extended to irregular shaped grids.

Acknowledgments

This research has been supported by the National Science Foundation under grants CCR-9205047 and DCA 9303181 and by the Department of Energy under the PICS grant. Special thanks to Rick Avila and Lisa Sobierajski for several enlightening discussions about PARC, volume rendering, and the implementation of VolVis. We are grateful to Juliana Freire for implementing the efficient Prolog algorithm described in Section 3 in the XSB system developed at Stony Brook by David Warren.

References

[1] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. Volvis: A diversified volume visualization system. In *Visualization '94 Proceedings*. IEEE CS Press, October 1994.

[2] R. Avila, L. Sobierajski, and A. Kaufman. Towards a comprehensive volume visualization system. In *Visualization '92 Proceedings*, pages 13–20. IEEE CS Press, 1992.

[3] E. Camahort and I. Chakravarty. Integrating volume data analysis and rendering on distributed memory architectures. In *1993 Parallel Rendering Symposium Proceedings*, pages 89–96. ACM Press, October 1993.

[4] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *1992 Workshop on Volume Visualization Proceedings*, pages 91–98. ACM Press, October 1992.

[5] J. Hennesy and D. Paterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann, 1990.

[6] W. Hsu. Segmented ray casting for data parallel volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 7–14. ACM Press, October 1993.

[7] V. Jacobson. Congestion avoidance and control. *Computer Communication Review*, 18(4):314–29, 1988.

[8] M. Levoy. Efficient ray tracing of volume data. *ACM Transations on Graphics*, 9(3):245–261, 1990.

[9] K. Ma, J. Painter, C. Hansen, and M. Krogh. A data distributed parallel algorithm for ray-traced volume rendering. In *1993 Parallel Rendering Symposium Proceedings*, pages 15–22. ACM Press, October 1993.

[10] C. Montani, R. Perego, and R. Scopigno. Parallel volume visualization on a hypercube architecture. In *1992 Workshop on Volume Visualization Proceedings*, pages 9–16. ACM Press, October 1992.

[11] U. Neumann. Parallel volume-rendering algorithm performance on mesh-connected multicomputers. In *1993 Parallel Rendering Symposium Proceedings*, pages 97–104. ACM Press, October 1993.

[12] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24. ACM Press, October 1992.

[13] P. Schroeder and J. Salem. Fast rotation of volume data on data parallel architectures. In *Visualization '91 Proceedings*, pages 50–57. IEEE CS Press, 1991.