Parallelizing the ZSWEEP Algorithm for Distributed-Shared Memory Architectures

Ricardo Farias[1] and Cláudio T. Silva[2]

[1] Department of Applied Mathematics and Statistics
State University of New York at Stony Brook
Stony Brook, NY 11794-3600
rfarias@ams.sunysb.edu

[2] AT&T Labs-Research
180 Park Avenue, Room D265
Florham Park, NJ 07932
csilva@research.att.com

Abstract. In this paper we describe a simple parallelization of the ZSWEEP algorithm for rendering unstructured volumetric grids on distributed-shared memory machines, and study its performance on three generations of SGI multiprocessors, including the new Origin 3000 series.
The main idea of the ZSWEEP algorithm is very simple; it is based on sweeping the data with a plane parallel to the viewing plane, in order of increasing z, projecting the faces of cells that are incident to vertices as they are encountered by the sweep plane. Our parallel extension of the basic algorithm makes use of an image-based task partitioning scheme. Essentially, the screen is divided in more tiles than the number of processors, then each processor performs the sweep independently on the next available tile, until no more tiles are available to render.
Here, we detail the modifications necessary to efficiently extend the sequential algorithm to work on shared-memory machines. We report on the performance of our implementation, and show that the *tile-based* ZSWEEP is naturally cache friendly, achieves fast rendering times, and substantial speedups on all the machines we used for testing. On one processor of our Origin 3000, we measure the L2 data cache hit rate of the *tile-based* ZSWEEP to be over 99%; a parallel efficiency of 83% on 16 processors; and rendering rates of about 300 thousand tetrahedra per second for a 1024×1024 image.

1 Introduction

In this paper, we describe a parallel extension of our ZSWEEP [4] algorithm for rendering unstructured grids on distributed shared-memory machines. Despite the substantial progress on the state-of-the-art in rendering of irregular grids, high-quality renderings of very large grids still take a substantial amount of time. Our goal is to explore the availability of small and mid-size parallel machines for rendering (and also to provide a path for exploring much larger machines). We focus on distributed-shared memory hardware, since these capabilities are quite common in servers sold by major vendors, including SGI, SUN, and IBM.

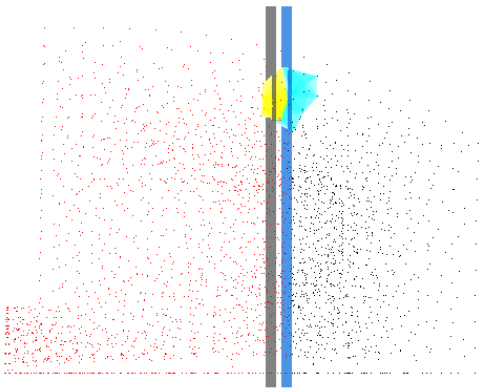


Fig. 1. ZSWEEP algorithm in action. The plane sweep is shown in blue while the plane determined by the *target Z* is shown in light-gray. The sweeping direction is from the right to the left and the swept vertices are shown in black while the still untouched vertices are shown in red. Faces in the *use set* of the current vertex are identified and shown as previously projected faces (light-blue) and faces to be projected (yellow), the ones that lie ahead of the plane sweep.

Although the programming model for shared-memory parallelization is quite trivial, achieving good performance on actual machines is usually hard. Even *embarrassingly* parallel algorithms, such as ray casting irregular grids [6] usually do not scale well beyond a few processors. Several issues such as proper load balancing need to be taken into account for good performance. Quite possibly, the hardest issue to deal with in distributed-shared memory machines is memory coherence and related issues. The problem comes from the fact that access to memory is non-uniform, since often the data one processor needs actually resides in physical memory that belongs to another processor. Hardware designers have developed intricate techniques for optimizing memory access (such as the deployment of large caches and aggressive memory prefetching strategies) but still software has to be carefully developed to collaborate with the hardware, and avoid performance killers such as unnecessary sharing of data. In general, one needs algorithms with a high degree of cache coherence to perform well on distributed shared-memory machines.

Direct volume rendering is a term used to define a particular set of rendering techniques which avoids generating intermediary (surface) representations of the volume data. Instead, the scalar field is generally modeled as a cloud-like material, and rendered by computing a set of lighting equations. In general, while evaluating the volume rendering equations [12], it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells intersected by the ray, so that the overall integral in the rendering equation can be evaluated.

ZSWEEP [4] is an algorithm for the computation of the sorted order of the cells intersected by all the rays in a given image. The main idea of the ZSWEEP algorithm is very simple; it is based on sweeping the data with a plane parallel to the viewing plane (shown in blue on Fig. 1), in order of increasing z, *projecting* the faces of cells that are incident to vertices as they are encountered by the sweep plane. ZSWEEP's face projection is different from the ones used in projective methods, e.g. [14]. During face projection, we simply compute the intersection of the ray emanating from each pixel, and store their z-value, and other auxiliary information, in *sorted* order in a list of intersections for the given pixel. The actual lighting calculations [12] are deferred to a later phase (b). Compositing is performed as the "target Z" plane (shown in gray on Fig. 1) is reached. The efficiency arises from: (1) the fact that the algorithm exploits the implicit (approximate) global ordering that the z-ordering of the vertices induces on the cells that are incident on them, thus leading to only a very small number of ray intersection are done out of order; (2) the use of early compositing which makes the memory footprint of the algorithm quite small. The key properties for the efficiency of ZSWEEP is the fact that given a mesh with v vertices and c cells, the amount of sorting ZSWEEP does is $O(v \log v)$ (in practice), i.e., depending on the number of ray intersections, this is substantially lower than the amount of sorting necessary to sort all the intersections for each pixel.

Contributions:

- We propose a simple parallel extension of the basic algorithm using an image-based (i.e, tiling) task partitioning scheme. Following Nieh and Levoy [13], our algorithm is based on an adaptive image-based task scheduling scheme. Basically, we divide the screen into tiles, which are dynamically assigned to the processors.
- We describe the changes that need to be performed to the original algorithm to efficiently implement a *tile-based* ZSWEEP.
- We perform a detailed analysis of the memory characteristics of the *tile-based* ZSWEEP. In particular, we show that the image tiling strategy improves the memory coherency of ZSWEEP, and can lead to the whole set of ray intersections fitting in the secondary level (L2) cache. On the Origin 3000 this leads to better than 99% hit rate and greatly improved rendering rates.
  Even on single-processor machines the *tile-based* ZSWEEP is considerably more efficient than the original algorithm.
- Finally, we study load balancing and efficiency of the parallel ZSWEEP on three generations of SGI multiprocessors, including the new Origin 3000 series.

The paper is organized as follows. In Sec. 2, we briefly describe related work. In Sec. 3, we present the parallel algorithm. Then in Sec. 4, we present our experimental results on three different kinds of SGI multiprocessors. Sec. 5 ends the paper with final remarks, and future work.

2 Related Work

We keep our related work section short and focus on parallel rendering algorithms for irregular grids and other work directly relevant to our work. The original ZSWEEP pa-

per [4] contains references to previous work in volume rendering of irregular grids. (In that paper, we failed to mention two relevant publications by Westermann and Ertl [17, 18] describing fast rendering techniques which are also based on the sweep paradigm. These papers describe techniques which are able to exploit the graphics hardware to achieve fast rendering.) For a discussion of computational complexity issues in rendering of irregular grids, we point the reader to [15].

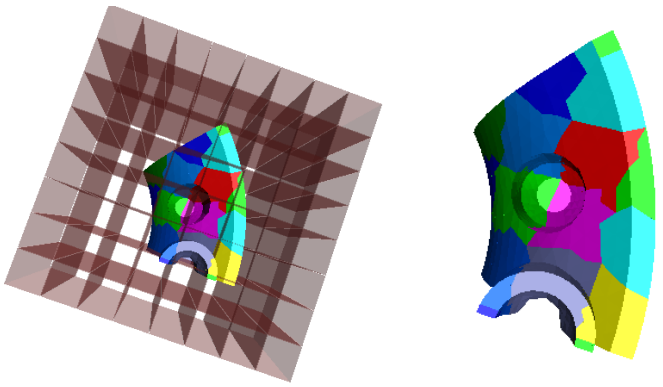
As we said before, for evaluating the volume rendering equations, it is necessary to have, for each line of sight (ray) through an image pixel, the sorted order of the cells intersected by the ray, so that the overall integral in the rendering equation can be evaluated.

One solution to this problem is to compute the intersections of rays with each cell in the mesh independently, then sort each list of intersections before compositing is performed. This is essentially the approach proposed by Ma and Crockett [10]. In more detail, their technique distributes the cells among processors in a round-robin fashion. For each viewpoint, each processor independently computes the ray intersections, which are later composited in a second phase of the algorithm. One of the potential shortcomings of this technique is that it requires the storage of a very large number of ray intersections. Ma and Crockett cleverly avoid this potentially crippling shortcoming by scheduling the computation using a k-d tree. As shown on [10, 11], their algorithm has been shown to be very scalable on message-passing machines, including the IBM SP-2 and the Cray T3D. Recently, Hofsetz and Ma [5] have developed an efficient shared-memory version of this algorithm, which they demonstrate on a 16-processor SGI Origin 2000. They showed that a naive port of the original algorithm lead to poor performance, but with substantial changes to the original implementation, very good performance was achieved.

One of the advantages of the Ma and Crockett technique is that no mesh connectivity is necessary. At the same time, by completely ignoring connectivity, this algorithm does not exploit a lot of the coherence intrinsic in the mesh, which both raises its memory requirements, and forces it into having to sort potentially very large lists. Most other algorithms for rendering irregular grids actually attempt to use mesh coherence (in the form of connectivity among cells), and try to get the sorting cost as close to linear as possible.

Hong and Kaufman [6] proposes a very efficient ray-casting based rendering algorithm for curvilinear grids. Their work is similar in some ways to [1], but optimized for curvilinear grids, which makes it faster and use far less memory than [1]. Our interest in their work for the purposes of this paper is the fact that they parallelized their fast ray caster on a 16-processor SGI machine using an image-based task scheduling scheme similar to the one we use in this paper. The speedups achieved were on the order of 11.88 on 16 processors, or 74% efficiency. The parallelization of a ray casting technique has also been studied by Uselton [16] with very good results.

Challinger [2] and Wilhelms et al [19] propose similar scanline rendering algorithms (similar in several respects to [15]). Both paper report on parallelizations, which is the main focus of [2]. Challinger also uses an image tiling scheme for parallelization with very good results, which are reported separate for different phases of the algorithm, and



(a) (b)

Fig. 2. A 8-by-8 tiling decomposition is shown. In (a) we show the shafts generated by each tiling region. In (b), we give a close up of the decomposition on the dataset. Each region was computed by intersecting the octree with the shafts shown in (a).

when taken all into account, amount to impressive speedups of a little over 70 on 100 processors of a BBN TC2000.

Still on shared-memory machines, Williams [20] reports parallelizing his rendering algorithm for an 8-processor SGI 4D/VGX. Other notable papers (which focus on rendering regular grids) include Nieh and Levoy [13] and Lacroute [7, 8]. We would like to note that irregular grid rendering algorithms tend to be hard to parallelize with screen-space parallelism because of their object-space disparate resolution. That is, possibly, a large number of cells project into a small area of the screen.

3 The Parallel ZSWEEP Algorithm

In this section we describe our parallelization of the ZSWEEP algorithm. The sequential algorithm is highly efficient, and uses little extra memory on top of the original dataset. It is based on computing ray intersections with the faces of the cells, which are "roughly" pre-sorted in depth by using a sort of the vertices of the cells. Each time a vertex is found during a z-sweep, the faces incident on it are marked, and the ray intersections for the pixels that overlap with the faces are computed, and inserted on intersection lists. In order to avoid having the lists get arbitrarily large, ZSWEEP employs a scheme for early compositing. See [4] for full details.

Following previous works, including Nieh and Levoy [13] and Hong and Kaufman [6], our parallelization is based on breaking the screen into tiles. Then placing the tiles into a work queue which processors compete for work. Each processor continously fetches a tile from the work queue, and computes the subimage corresponding to that tile until all the tiles have been rendered. In order for a given processor to compute the

image for a tile using ZSWEEP, we must determine all the vertices from any face that intersects the "shaft" emanating from that tile. This is similar to the parallel view sort of Challinger [2], and is primarily the main difference between the sequential and parallel ZSWEEP, since in the sequential algorithm the vertices are known apriori (that is, all of them are sorted in depth).

For efficiency purposes, we made a small data structure change. While in the sequential ZSWEEP implementation the **use set** of a vertex is the list of cells incident on it, in the parallel version we decided to break the cells into its faces and keep them in the use set of the vertices. The reason for the modification comes from the fact that the projection of a face requires a somewhat expensive setup (see [1] for details). Since faces might intersect multiple tiles of the screen, the setup time would be replicated multiple times. By actually having a list of the faces, we are able to parallelize these computations as a first phase in the parallel render.

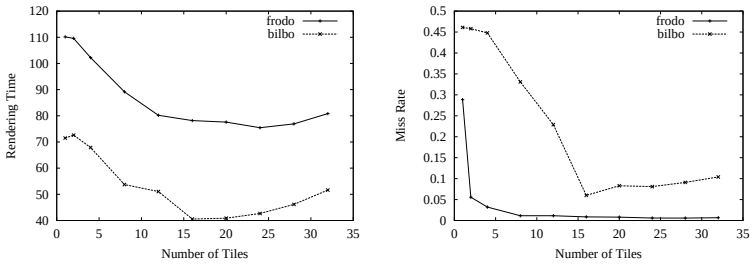
We separate the computation of the vertices that belong to an image tile into a view independent phase which is performed only once when the data is first loaded, and a view dependent phase which is performed by each processor when rendering a given tile. The view independent phase consists of (1) constructing an octree of the vertices of the mesh; (2) computing for each octree "leaf" the faces which intersect that leaf (in the implementation we use the bounding box of the faces, which is a conservative estimate); (3) record for each leaf the list of faces which have non void intersection. From such list we are able to determine the vertices that must be considered in the sweeping phase, for each leaf. The view dependent phase uses the octree to find which leaves intersect the shaft corresponding to the tile, then uses the union of all the vertices assigned to those leaves as the input for the rendering routine. (See Fig. 2.)

The actual rendering algorithm is fairly simple. Given p processors and f faces, each processor transforms $\frac{f}{p}$ faces. The image is divided into tiles, and each processor will incrementally grab a tile, and render the subimage corresponding to that tile. Rendering a tile is performed by (a) finding the leaves of the octree which project inside the particular tile, (b) computing vertices of all the faces which intersect any of the leaves found, and (c) projecting the faces in order (that is, the last phase is simply the sequential ZSWEEP applied to the subset of the vertices which have faces projecting inside the tile). In our implementation, we are careful to clip the projection of the faces to within the tile being computed.

As shown in Sec. 4, we achieve very good load balancing with this simple scheme. The cost of rendering a tile is dependent both on its area, and the number of points which project into it. Experimentally, we have found that the area cost is considerably larger than the cost associated with the number of points. The number of points can vary as much as by a factor of five, and have little impact on the running time of the region.

4 Experimental Results

In this section we summarize our findings about the performance of our algorithm. We ran our experiments on three difference machines, all manufactured by SGI:



(a) (b)

Fig. 3. Sequential tile-based ZSWEEP results for rendering the small SPX dataset at 1024×1024 resolution under different tiling. In (a), we see the rendering times. In (b), the L2 data cache miss rate. By using tiling, the miss rate drops considerably to under 1% on **frodo**.

- **bilbo**: 12-processor SGI Onyx. The processors are 194Mhz MIPS R10000. Eight of them have 1 MB of secondary cache, and the other four have 2 MB of secondary cache. Bilbo has 2 GB of memory. This machine is a snoop-based multiprocessor [3, Chapter 6], a design which is popular in small parallel machines, but does not scale well.
- **smaug**: 24-processor SGI Origin 2000. It is equipped with sixteen 250 Mhz MIPS R10000 and eight 300 Mhz MIPS R12000. Each R10000 has a 4MB secondary level cache, while each R12000 has a 8 MB secondary level cache. Smaug has 14 GB of memory. This machine is based on a scalable shared-memory system, and it uses directory-based cache coherence [3, Chapter 8].
- **frodo**: 16-processor SGI Origin 3000. It is equipped with sixteen 400 Mhz MIPS R12000 and it has 12 GB of memory. This machine is a faster and more scalable directory-based distributed shared-memory system. In particular, each parallel "node" has four processors (compared to two for the O2K), and higher memory bandwidth, and much lower latency.

4.1 Sequential Tile-Based ZSWEEP

An interesting fact is that the tile-based ZSWEEP is faster by almost 50% than the original. This is somewhat counter intuitive, since it actually does more work: it needs to sort vertices multiple times (the actual number depends on tiling and resolution of octree), and it definitely touches faces multiple times, although the actual pixel calculations are quite similar. A potential advantage of the tile-based approach is that the "target Z" used for early compositing is likely to be more accurate. But when we first noticed this speedup from tiling, we suspected that these performance gains actually arise from better memory coherency.

We used `perfex`, an SGI IRIX tool which is able to configure and retrieve the MIPS R10K hardware counters, to validate our hypothesis. In Fig. 3, we show some

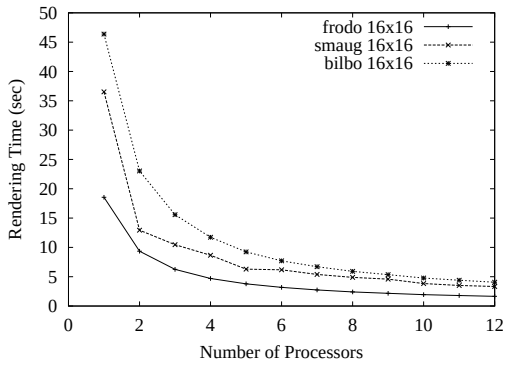


Fig. 4. Running times on up to 12 processors for the Post dataset for images of size 512×512 with 16-by-16 tiling.

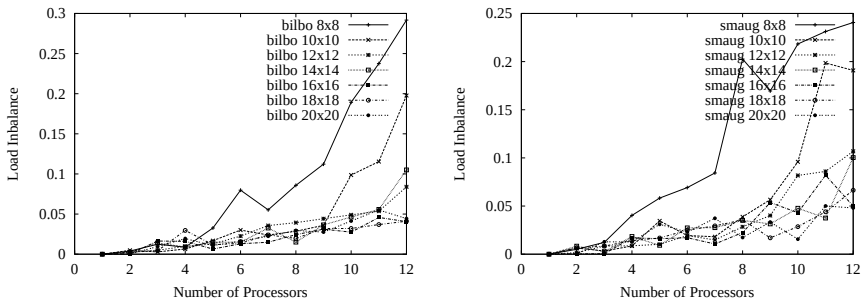
of our findings. In particular, we can see that the L2 data cache hit rate [1] is greatly improved with caching, and there is a corresponding improvement in rendering times. On **frodo**, we get better than 99% hit rates, and on **bilbo** they were improved from just a bit over 50% to over 90%. Another interesting statistics is the number of TLB misses which changes by a factor of 300 on some of the runs, thus indicating the considerable better data locality of the tile-based approach.

4.2 Load Balancing

We ran a battery of tests for studying the scalability of our algorithm on all these machines. We tried to use the machines when they were free, although this was virtually impossible for smaug which is used for heavy batch processing of data. We ran jobs on smaug at times of lightest load. Unfortunately, given the heterogenous nature of the CPUs, it is really not possible to make very accurate measurements on that machine. The other two machines were used at idle times. We generated 512×512 images under different conditions, and changing the tiling granularity. We use the term X-by-Y tiling decomposition to mean that the image was subdivided into X times Y regions. That is, an 8-by-8 tiling decomposition means that the image was divided into 64 tiles.

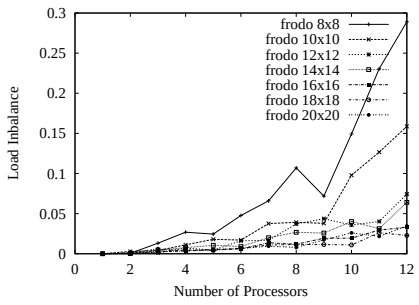
Fig. 4 shows the running times for the Post dataset on the different machines on up to 12 processors. As can be seen from the picture, the rendering times are quite fast, and improve as the number of processors increase. As expected, frodo is considerably faster than the other two machines, and the parallel efficiency is about 93% with 12 processors (11.2 speedup). It is interesting to note that even on the **bilbo**, which has considerably inferior memory system, our parallel algorithm is able to scale quite nicely. Part of the credit might go to the fact that ZWEEP tends to minimize data movement.

[1] L2 (secondary) data cache hit rate is the fraction of data accesses that are satisfied from a cache line already resident in the secondary data cache. It is calculated as 1.0 - (secondary data cache misses divided by primary data cache misses). This is the exact definition from the `perfex` man page.



(a) **bilbo**

(b) **smaug**



(c) **frodo**

Fig. 5. Load imbalance with different tiling parameters for the Post dataset for images of size 512×512.

The tiling granurality has an impact on the performance. We use Ma's load imbalance metric to study the impact on load balancing of different tiling sizes. Given a set of processors where the average rendering time is t_{avg} and the maximum rendering time is t_{max}, Ma [9] defines the imbalance to be:

$$1 - \frac{t_{avg}}{t_{max}};$$

basically, his metric measures the spread of the running times among the different processors around the mean. In Fig. 5, we plot the imbalance. As can be seen in the picture, **bilbo** and **frodo** behave almost exactly the same, while smaug, due to its different speed processors, exhibits more load imbalance. The worst load imbalance happens for 8-by-8 tiling decomposition, and can be as high as 30%. Part of the problem is that because the dataset is not uniform, some parts of the screen might have a very large number of faces, that need to be rendered. With a 16-by-16 tiling decomposition, things get substantially better, and the load imbalance is lower than 5%.

Rendering Times												
	512×512				1024×1024				2048×2048			
	SPX	SPX1	SPX2	SPX3	SPX	SPX1	SPX2	SPX3	SPX	SPX1	SPX2	SPX3
1	4.51	9.95	38.10	186.05	15.39	27.86	73.40	267.46	82.89	145.74	298.78	731.99
2	2.32	5.09	19.65	97.96	7.78	13.97	36.85	135.13	41.85	73.08	150.23	375.3
4	1.18	2.62	10.37	50.55	3.99	7.08	18.47	68.12	21.17	36.71	75.65	188.3
8	0.63	1.39	5.60	26.63	2.11	3.71	9.71	36.27	11.04	19.07	39.20	98.09
16	1.38	1.93	10.30	27.56	1.28	2.17	5.61	21.72	6.50	10.96	22.07	56.27

Table 1. Rendering times on **frodo** as the dataset and image size increases.

Datasets Information		
Dataset	# of vertices	# of cells
Oxygen Post	109K	513K
SPX	2.9K	13K
SPX1	20K	103K
SPX2	150K	830K
SPX3	1150K	6620K

Table 2. Main datasets used for benchmarking. The first four are tetrahedralized versions of the well-known NASA datasets. SPX is an unstructured grid composed of tetrahedra. We have subdivided each tetrahedron into 8, for each version of the last three, that is, SPX3 is 512 times larger than SPX. The number of vertices and tetrahedra are listed in thousands.

On **frodo**, for the Post, using 16-by-16 tiling decomposition, the speedups are 12.3 for a 512×512 image, and 13.5 for a 1024×1024 image, or approximately 84% efficiency. The best rendering times for the Post are 1.5 seconds for a 512×512 image, and 4.44 seconds for a 1024×1024 image. In general, it is possible to improve the load balancing by simply incresing the tiling resolution. In fact, we were able to get efficiencies of almost 90% by tweaking the parameters. A better solution would be to have an adaptive technique which automatically fine tunes the load balancing. We have actually implemented such a scheme, but were not able to make it work consistently yet.

4.3 Data and Image Scalability

Finally, we present some results related to the data and image scalability of our parallel code. We took the SPX dataset and subdivided it multiple times (by breaking each tetrahedra into eight). For each version of the dataset, we rendered it ten times along a uniform rotation of the y-axis. The images were computed at different resolutions, and the full results are reported in Table 1, and some subset are plotted in Fig. 6.

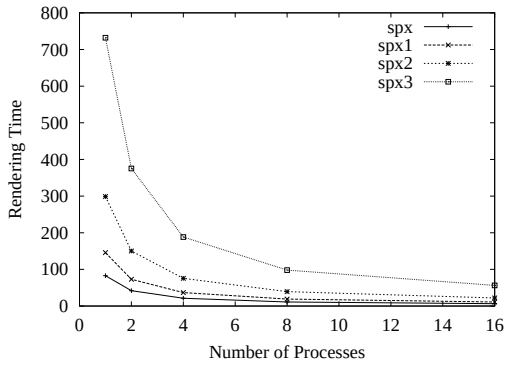


Fig. 6. Plot for the 2048×2048 data from Table 1.

5 Conclusion

In this paper we present a simple parallelization of the ZSWEEP algorithm for distributed-shared memory machines. Other than changes to the actual code to make it more modular, and to isolate shared variables, we only had to perform one major architectural change to the algorithm to make it parallel: the introduction of an octree for the vertices so we can efficiently find which faces project into a given tile. In this work, we were able to keep all the nice features of ZSWEEP, i.e., the fact that it is very simple to implement, robust, and memory efficient.

We were able to achieve a parallel efficiency of 84% on 16 processors on an SGI Origin 3000 machine. The complexity of rendering a tile is dependent both on the number of primitives which project on the tile, and the area of the tile. In order to further speed up the code for more processors, we believe we might need a more fine grain load balancing scheme which is able to dynamically partition regions when we discover that we have too many primitives that project in it.

It would be useful to run our code on larger SMP machines. The reported results are for a version of the code parallelized with the **m_fork** calls of SGI IRIX. We have ported this code to POSIX Pthreads, which runs quite well on Linux, but we have not performed detail analysis of the Pthread version performance yet.

Acknowledgements

We are grateful to Anne Rogers (AT&T) for suggestions and help in analyzing the memory characteristics of the tile-based ZSWEEP, Kwan-Liu Ma (UC, Davis) for suggestions that greatly improved our presentation, Joseph Mitchell (Stony Brook) and Brian Wylie (Sandia) for their collaboration on this research. NASA has gracefully provided the Liquid Oxygen Post dataset. Peter Williams (LLNL) gave us the SPX dataset. This work was made possible with the generous support of Sandia National Labs and the Dept of Energy Mathematics, Information and Computer Science Office. R. Farias acknowledges partial support from CNPq-Brazil under a PhD fellowship.

References

1. P. Bunyk, A. Kaufman, and C. Silva. Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization, Proceedings of Dagstuhl '97*, pages 30–36, 2000.
2. J. Challinger. Scalable parallel volume raycasting for nonrectilinear computational grids. In *ACM SIGGRAPH Symposium on Parallel Rendering*, pages 81–88, November 1993.
3. D. Culler, J. Singh, and A. Gupta. Parallel Computer Architecture, A Hardware-Software Approach. Morgan-Kaufmann, 1999.
4. R. Farias, J. Mitchell, and C. Silva. ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. In *2000 Volume Visualization Symposium*, pages 91–99. October 2000.
5. C. Hofsetz and K.-L. Ma. Multi-threaded rendering unstructured-grid volume data on the sgi origin 2000. In *Third Eurographics Workshop on Parallel Graphics and Visualization*, 2000.
6. L. Hong and A. Kaufman. Accelerated ray-casting for curvilinear volumes. *IEEE Visualization '98*, pages 247–254, October 1998.
7. P. Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. *IEEE Parallel Rendering Symposium*, pages 15–22, October 1995.
8. P. Lacroute. Analysis of a parallel volume rendering system based on the shear-warp factorization. *IEEE Transactions on Visualization and Computer Graphics*, 2(3), September 1996.
9. K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. *IEEE Parallel Rendering Symposium*, pages 23–30, October 1995.
10. K.-L. Ma and T. Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. *IEEE Parallel Rendering Symposium*, pages 95–104, November 1997.
11. K.-L. Ma and T. Crockett. Parallel visualization of large-scale aerodynamics calculations: A case study on the Cray T3E. *Symposium on Parallel Visualization and Graphics*, pages 15–20, October 1999.
12. N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.
13. J. Nieh and M. Levoy. Volume rendering on scalable shared-memory mimd architectures. In *1992 Workshop on Volume Visualization Proceedings*, pages 17–24, October 1992.
14. P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics (San Diego Workshop on Volume Visualization*, vol. 24, pages 63–70, November 1990.
15. C. Silva and J. Mitchell. The lazy sweep ray casting algorithm for rendering irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, 3(2), April - June 1997.
16. S. Uselton. Volume rendering for computational fluid dynamics: Initial results. Tech Report RNR-91-026, Nasa Ames Research Center, 1991.
17. R. Westermann and T. Ertl. The vsbuffer: Visibility ordering of unstructured volume primitives by polygon drawing. *IEEE Visualization '97*, pages 35–42, November 1997.
18. R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. *Proceedings of SIGGRAPH 98*, pages 169–178, July 1998.
19. J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. *IEEE Visualization '96*, pages 57–64, October 1996.
20. P. Williams. Parallel volume rendering finite element data. In *Proceedings of Computer Graphics International*, 1993.