Experimental Evaluation of Load Balancing Strategies for Ray Tracing on Parallel Processors

Tong-Yee Lee, C.S. Raghavendra
School of EE/CS
Washington State University
Pullman, WA 99164

John B. Nicholas
Molecular Science Research Center
Pacific Northwest Laboratory
Richland, WA 99352

Abstract

Ray tracing is one of the computer graphics techniques used to render high quality images. Unfortunately, ray tracing complex scenes can require large amounts of CPU time, making the technique impractical for everyday use. Parallel ray tracing algorithms could potentially be used to reduce the high computational cost. However, pixel computation times can vary significantly, and naive attempts at parallelization give poor speedups due to load imbalance between the processors. In this paper, we evaluate the performance of three load balancing schemes for ray tracing on parallel processors, and propose two new load balancing strategies. To evaluate the performance, we implement all these strategies on the 512 processor Intel Touchstone Delta at Caltech.

1 Introduction

The rendering of high quality images of complex 3-D scenes requires sophisticated graphics algorithms. Ray tracing is one method that is capable of generating such high quality images[1]. Ray tracing is based on simple models of light shading, reflection, and refraction. The main rays are sent from the eye through pixels on an imaginary viewplane and are traced as they are reflected and transmitted by objects. The reflected and transmitted rays are in turn traced until a maximum tracing depth is reached or the rays no longer hit any object. The amount of computation time required for each pixel depends on the number of objects hit by that particular ray. Since the computation of each pixel is independent of the others, ray tracing is potentially highly suitable for parallel processing. However, because pixel calculation time can vary significantly, the key to achieving efficient parallel ray tracing is to ensure that the processors have closely equivalent amounts of computation to perform. There have been several approaches to implementing ray tracing on parallel processors. Most of these approaches can be classified according to whether parallelism in *object space*[2, 3, 4, 5, 6, 7] or *image space*[8, 9, 10, 11, 12] is being exploited.
In the following sections, we evaluate the performance of five image space load balancing schemes for ray tracing on a 2-D mesh parallel processor. We have selected three commonly used load balancing schemes. We also propose two new load balancing strategies, called Global Distributed Control (GDC) and Local Distributed Control (LDC). The GDC scheme achieves load balancing through global redistribution of workload, whereas the LDC scheme shares work load only among local neighbor processors. We will show that our new methods achieve better performance than previous schemes. In order to evaluate the load balancing methods, we have implemented them on the Intel Touchstone Delta and have tested them on 1 to 512 nodes. The Delta is a high-speed concurrent multicomputer, consisting of an ensemble of 512 computational nodes arranged as a 16 x 32 mesh[13]. Groups of nodes can work on the same problem and communicate with each other by message passing.

2 Load Balancing Strategies

In this section, we briefly discuss the five load balancing strategies for ray tracing. Of the five methods, two are static and three are dynamic load balancing methods. The methods also differ in how the image screen space is divided and how assignment of screen regions to processors is initially done.

2.1 Direct Mapping

The direct mapping (DM) strategy is the simplest of the five strategies. In DM, the 2-D screen is divided into equal size (square or rectangular) regions and each $region_i$ will be assigned to a $processor_i$. This approach is probably the most naive parallel implementation. DM is a static load balancing method; no adjustment of the workload is done after the initial assignment. Since load balancing is static, DM will only work well if each screen region is fortuitously equivalent in terms of pixel computations. However, this situation is unlikely in most real applications. Thus, DM is vulnerable to load imbalance and exhibit poor scalability.

2.2 Interleaved Mapping

Another commonly used strategy is interleaved mapping (IM). In IM, the image screen space is divided into horizontal (or vertical) strips or square (or rectangular) blocks that are assigned in an interleaved fashion to available processors. In its basic implementation, IM is also a static load balancing method, as no reassignment of interleaved regions is made during the calculation. However, IM generally works better than DM since the interleaving tends to give reasonable load balancing. A thorough mathematical analy-

Proceedings of the International Conference on Parallel Processing (ICPP'94)
0-8493-2493-4/94 $0.00 © 1994 IEEE

sis of the IM strategy (also termed scatter decomposition) was given in [15].

2.3 Master/Slave method

The master/slave (MS) strategy uses a master node that is responsible for dynamically assigning the screen regions (lines or blocks), and many slave nodes that perform the pixel computations. The MS method would be most effective when the variation in the computational requirements of different screen regions is large. One might naively assume that the MS method has the potential to achieve almost perfect load balancing if the size of the screen regions is very small (approaches one pixel). Unfortunately, as the number of regions is increased, the cost of the communication required to schedule the work also increases, which can severely affect the parallel performance. Thus, perfect load balancing is difficult or impossible to achieve with the MS approach. We used the central node of mesh topology on the Intel Delta as the master node to schedule the task requests and used a single horizontal line as the workload unit.

2.4 Local Distributed Control Method

In the LDC[16] method, we divide the screen into a number of 2x2 pixel regions and initially assign these regions to the processors in an interleaved manner. As previously mentioned, interleaved assignment can often provide roughly equal computational loads to processors. However, to achieve better performance, in LDC we redistribute work from active nodes to idle nodes during the computation, in an attempt to keep all processors as busy as possible. In LDC we logically extend the 2-D mesh of the Intel Delta into a torus topology, allowing every node to have four neighbors, and limit load sharing to these nearest neighbors. These neighbors are numbered in a cyclic fashion. The neighbor node closest to the center of the mesh is numbered *1*, with the others numbered *2*, *3*, and *4* in a clockwise fashion. When a node (say P_i) becomes idle, it requests work from its neighbors starting with neighbor *1*. If the first node has work to pass to node P_i, it does so. If no work is available at node *1*, node P_i continues through its neighbors in a cyclic fashion until it either finds more work or has checked all neighbor nodes. If it finds an active neighbor (say *2*), it will receive half of the remaining workload from that neighbor. If node P_i becomes idle again, it will search for additional work, starting from node *2*. If all neighbor nodes are idle, node P_i will stop requesting work. When all nodes finish their work, the ray tracing computation is completed.

2.5 Global Distributed Control Method

In GDC method, we logically organize the N processors in a ring topology. Again, we divide the screen into a number of 2x2 pixel regions and assign the regions to the processors in an interleaved fashion. A processor will first ray trace its assigned regions. When a particular node, say P_i, becomes idle, it will request extra work from successive nodes on the ring, $P_{i+1}, \ldots, P_N, P_1, \ldots, P_{i-1}$, until it finds a node that is active. This active node, say P_j, dispatches a 2x2 pixel region to this request. The node P_i will remember that node P_j was the last node from which a request was made. The next time it is idle, it will start requesting work from node P_j, bypassing nodes between P_i and P_j. An additional feature of this strategy is that, if in the meantime, node P_j becomes idle and P_j remembers that it received work from node P_k, node P_i will jump from P_j directly to P_k without asking for work from nodes between P_j and P_k. In this strategy, node P_i stops its search for work if it ends up at itself or at some node that it has already visited or skipped in a search step. This ensures that the search will end when all nodes are idle.

Scene	Primitives	Light Sources
Ball	7382	3
Mountain	8196	1
Tree	8191	7

Table 1: Database characteristics

Scene	Max.	Min.	Ave.	PSD
Ball	6496.2	120.2	1649.1	101.2%
Mountain	11007.2	1042.6	4273.2	43.6%
Tree	2928.5	277.8	1716.4	50.7%

Table 2: Load distribution characteristics (Unit: milliseconds)

3 Experimental Results

We used a set of standard scenes from Eric Haines's database to perform our experimental evaluation[14]. Table 1 shows the geometric characteristics of three scenes (see Figures 1,2 and 3). These test scenes have been used in many previous studies and are believed to be a good representation of real data. To show the wide variation in pixel computation times, we first ray traced each scene at 512x512 resolution with one subray anti-aliasing. We then summed the computation time for 256 32x32 subregions of the screen. Table 2 shows load distribution characteristics for three test scenes based on this screen division. We can estimate the load imbalance among these subregions by taking the standard deviation of the subregion execution times divided by the average execution time. Although, this percentage standard deviation (PSD) depend on the manner in which screen space is divided, it is potentially useful indicators of the difficulty one would expect in load balancing a particular scene. For example, the "Balls" scene has the largest PSD. The large PSD suggests that this scene should be the most difficult for which to achieve good load balancing. The data we will present in the following sections show that this is the case.

We experimentally evaluated each load balancing strategy using from 1 to 512 nodes. We ray traced each test scene at 512x512 resolution with 16 subray anti-aliasing, to produce a high-quality image. We consider the results for parallel speedup, which we present in

Figure 1: "Tree" scene

Figure 2: "Balls" scene

Figures 4, 5, and 6. The GDC method is consistently the best performer; it gives almost linear speedup for all three scenes. It is also generally the fastest method in absolute terms. LDC performs the next best; it is slightly faster than GDC for less than 64 processors and is very comparable overall. However, the parallel speedup of LDC deteriorates for large numbers of processors, a result of the limited load balancing that is possible when load sharing is restricted to immediate neighbors only. Both LDC and GDC perform noticeably better than the IM method. Since LDC and GDC are dynamic load balancing methods applied to the basic IM strategy, the data proves that the incorporation of dynamic load balancing gives a significant improvement in parallel performance. However, considering that IM is a static method, it still gives relatively good parallel speedup. IM performs poorest on the "Balls" scene, which is not surprising considering the very uneven computational distribution of that scene. Note that IM almost always performs better than MS, even though MS is a dynamic load balancing method. The overhead of scheduling jobs of the master node overwhelms any potential benefit from the dynamic load balancing in the MS method. This is particularly apparent for the "Tree" scene; MS does quite poorly on 256 and 512 nodes. As expected, DM has the worst parallel speedup and the slowest overall performance of all the methods. The large load imbalance in the "Balls" scene causes DM to give very poor

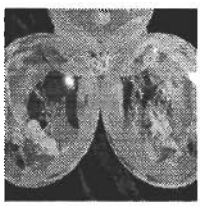

Figure 3: "Mountain" scene

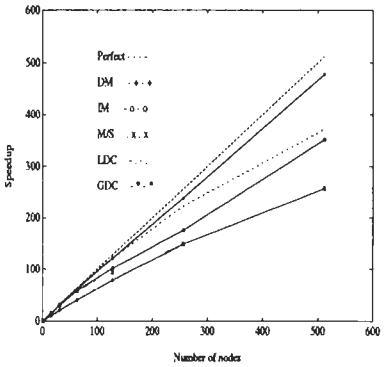

Figure 4: Parallel Speedup for the "Tree" scene

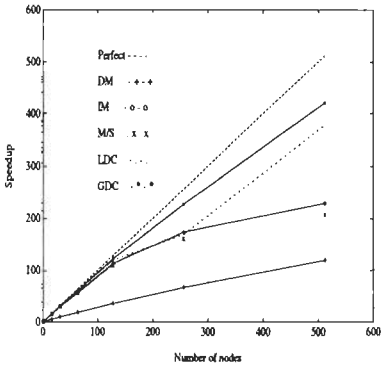

Figure 5: Parallel Speedup for the "Balls" scene

results; almost four times slower than GDC and LDC. As predicted by the PSD in Table 2, all the methods do most poorly on the "Balls" scene, which has the greatest variation in pixel computation times. While one might expect that the static methods would perform the best on the "Mountain" scene, since it has the lowest PSD, they do slightly better on the "Tree" scene. As expected, the dynamic methods perform the best on the "Mountain" scene. This scene has the lowest PSD, and also the largest amount of computation per pixel.

4 Conclusions and Future Work

In this paper, we evaluated five different load balancing strategies for parallel ray tracing. We tested each strategy on three scenes, to give a fair representation of parallel performance. Our new methods, LDC and GDC, consistently gave a better parallel speedup than the other methods for rendering high quality images. In future work we will investigate ways to ray trace much larger and more complex scenes. We will have to consider ways of partitioning data among processors, as the data will be too large to duplicate on each processor. We will need to consider ways in which data can be accessed or moved among processors as part of the load balancing strategy.

0-8493-2493-4/94 $0.00 © 1994 IEEE

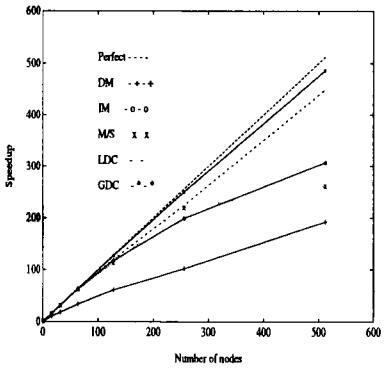

Figure 6: Parallel Speedup for the "Mountain" scene

Acknowledgment

This research is supported in part by the NSF Grant No. MIP-9296043 and the Boeing Centennial Chair Professor funds. This research was performed in part using the Intel Touchstone Delta System operated by California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest Laboratory, which is operated for the U.S. Department of Energy by Battelle Memorial Institute under Contract DE-AC06-76RLO 1830. We also like to give our special thanks to Mark VandeWettering for providing the source code for his invaluable sequential MTV ray tracer.

References

[1] Turner, "An Improved Illumination Model for Shaded Display", Comm. ACM, Vol. 23, No. 6, June 1980.

[2] Cleary, J.G and et. al., "Multiprocessor Ray Tracing.", Tech. Rept. 83/128/17, Univ. of Calgary, 1988.

[3] Dippe, M., and Swensen, J., "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", Computer Graphics, July, 1984.

[4] Keiji Nemoto and Takao Omachi, "An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing", In Proceedings of Graphics Interface '86, 1986.

[5] Hiroaki Kobayashi and et. al., "Parallel Processing of an Object Space for Image Synthesis Using Ray Tracing", The Visual Computer, Feb., 1987.

[6] Kadi Bouatoch and Terry Priol, "Parallel Space Tracing: An Experience on an iPSC Hypercube", In N. Magnenat-Thalmann and D. Thalmann, editors, New Trends in Computer Graphics, 1988.

[7] David A. J. Jevans, "Optimistic Multi-processor Ray Tracing", , In R.A. Earnshaw and B. Wyvill, editors, New Advances in Computer Graphics, Springer Verlag, 1989.

[8] Jamie Packer, "Exploiting Concurrency: A Ray Tracing Example", Inmos Technical Note 7, Inmos Ltd., Bristol, 1987.

[9] Kobayashi H, Nishimura S, Kubota H, Nakamura T, Shiegi Y, "Load Balancing Strategies for a Parallel Ray Tracing System based on constant subdivision", The Visual Computer, 1988, 4(4).

[10] Salmon J, Goldsmith J, "A Hypercube Raytracer", Proc 3rd Conf. Hypercube Computers and Applications, 1988.

[11] Green, S. and et. al., " Exploiting Coherence for Multiprocessor Ray Tracing", IEEE Computer Graphics and Applications, Nov., 1989.

[12] Badouel and et. al., "Ray Tracing on Distributed Memory Parallel Computers", SIGGRAPH 1990.

[13] "Touchstone Delta User's Guide", Intel Corporation, October, 1991.

[14] E. Haines, "A Proposal for Standard Graphics Environments", IEEE Computer Graphics and Applications, July, 1987.

[15] J. Salmon., "A mathematical analysis of the scattered decomposition", In The Third Conference on Hypercube Concurrent Computers and Applications, Volume I, p239-240, Jan., 1988.

[16] Tong-Yee Lee, C.S. Raghavendra, and John B. Nicholas, "A Fully Distributed Parallel Ray Tracing Scheme on the Delta Touchstone Machine", Proceedings of 2nd International Symposium on High Performance Distributed Computing, July, 1993.

Proceedings of the International Conference on Parallel Processing (ICPP'94)
0-8493-2493-4/94 $0.00 © 1994 IEEE