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, USA

John B. Nicholas

Molecular Science Research Center
Pacific Northwest Laboratory
Richland, WA 99352, USA

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. Since the ray tracing calculations that determine the values of individual pixels are independent, this appears to be an easy problem to parallelize and parallel algorithms have been proposed. However, pixel computation times can vary significantly, and naive attempts at parallelization give poor speedup due to load imbalance between the processors. The key to achieving high parallel efficiency is to ensure that the computational load is evenly balanced. In this paper, we propose two new load balancing schemes and evaluate the performance of ours for ray tracing on parallel processors. We term both new methods Local Distributed Control (LDC) and Global Distributed Control (GDC). Our new strategies are complementary: GDC performs well for high computational complexity images and LDC works well for low computational complexity images.

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. For each pixel, the ray tracing algorithm calculates the (R,G,B) value at the intersection points between rays and objects. The amount of computation time required for each pixel depends on the number of objects hit by that particular ray. Ray-tracing is a computationally intensive task; it can take several hours to render an image on current workstations. 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, some processors will have much more computation to perform than others. 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* or *image space* is being exploited[2]. In the object space methods, the 3-D object space world is divided into regular or irregular subvolumes. The subvolumes are then allocated to the processors such that adjacent subvolumes are associated with neighboring processors. During ray tracing, the paths of rays will traverse the subvolumes. This information must be communicated to the processors assigned to the particular subvolumes through which the rays propagate. There have been many works devoted to the object space method[3, 4, 5, 6]. The main issue in these works is how to divide object space and how to dynamically balance the workload in parallel computations by adjusting the size of the subvolumes.

In contrast, the image space methods do not consider the objects to be rendered, and instead divide the 2-D image screen pixels into a number of distinct regions. Each processor is assigned some number of regions to compute. During ray tracing, dynamic load balancing can be attempted by redistributing regions from heavily loaded nodes to lightly loaded ones[7, 8, 9, 10, 11, 12]. Similar to the object space methods, previously published techniques have focused on different ways to divide the screen space and balance the load among processors.

In this paper, we propose two new image space load balancing strategies and evaluate the performance of both for ray tracing on a 2-D parallel processor. These two strategies are called Global Distributed Control (GDC) and Local Distributed Control (LDC). 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. We will show that our new methods achieve almost linear speedup performance, with GDC performing best for high computational complexity images (complex scenes, high anti-aliasing effect, high resolution, etc.) and LDC performing best for low complexity images.

The Intel Touchstone Delta is a high-speed concurrent multicomputer, consisting of an ensemble of 512 computational nodes arranged as 16 × 32 mesh. The nodes are Intel i860 microprocessors, each with its own memory space. Groups of nodes can work on the same problem and communicate with each other by message passing.

In the following sections, we describe our strategies for dynamic load balancing of parallel ray tracing on 2-D mesh parallel processors.

2 Local Distributed Control Method

In the local distributed control (LDC) 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 mentioned in[12], interleaved assignment can often provide roughly equal computational loads to processors. However, to

Figure 1: Tree scene

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 4 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.

3 Global Distributed Control Method

In the Global Distributed Control (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, $P_{i+1}, \dots, P_N, P_1, \dots, P_{i-1}$, until it finds a node which 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. Node P_i stops its search for work if it ends up at itself or at some node which it has already visited or skipped in a search step. This ensures that the search will end when all nodes are idle.

4 Experimental Results for High Quality Images

We used a set of standard scenes from Eric Haines's database to perform our experimental evaluation[13]. These test scenes have been used in many previous studies and are believed to a good representation of real data (Figures 1, 2, and 3).

Figure 2: Balls scene

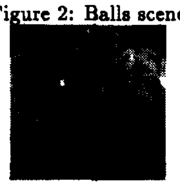

Figure 3: Mountain scene

Table 1 shows the load distribution characteristics for each scene. This table is obtained by raytracing each scene at 512x512 resolution with 1 subray anti-aliasing and then by summing the computation time for 256 32x32 subregions of the screen. In this table, we create an estimate of the load imbalance, (PSD), by taking the standard deviation of the subregion execution times divided by the average execution time. PSD is a useful indicator of the difficulty one would expect in load balancing a particular scene. For example, the "Balls" scene, in which we altered the viewpoint to create a load imbalance, has the largest PSD. The large PSD suggests that this scene should be the most difficult for which to achieve good load balancing.

In our evaluation, we first ray traced each test scene at 512x512 resolution with 16 subray anti-aliasing, to produce a high-quality image. We will present results for the parallel speedup to evaluate the effectiveness of the load balancing. Here, speedups are computed for raytracing time only, and do not include data parsing and final image display timings. The results for the parallel speedup are given in Figures 4, 5, and 6. The GDC method is consistently the better performer; it gives almost linear speedup for all three scenes. LDC performs well also; it is slightly faster than GDC for less than 64 processors and 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.

As predicted by the PSD in Table 2, both methods do the most poorly on the Balls scene, which has the greatest variation in pixel computation times, and

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 1: Load distribution characteristics (Unit: milliseconds)

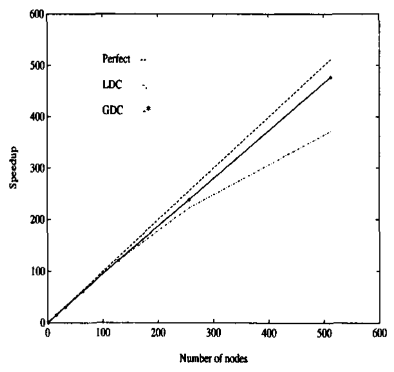

Figure 4: Parallel Speedup for the Tree scene. Ray tracing time on a single processor = 6687 seconds.

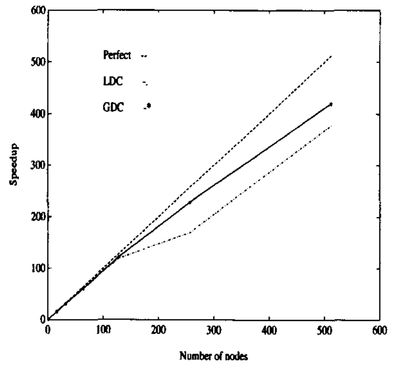

Figure 5: Parallel Speedup for the Balls scene. Ray tracing time on a single processor = 6428 seconds.

perform the best on the Mountain scene, which has the lowest PSD, and also the largest amount of computation per pixel. The low PSD obviously makes load balancing an easier task, while the large amount of computation tends to mask the degradation of the parallel performance caused by the overhead costs associated with the dynamic methods.

5 Experimental Results for Low Quality Images

Our initial test show that LDC and GDC provide good parallel performance for all three scenes. However, since we ray traced each scene at 512x512 resolution using 16 subray anti-aliasing, the actual ray tracing computations generally required considerably more time than the overhead tasks This large amount of computation tends to mask the relatively high communication cost associated with the dynamic load balancing methods. We should note that previous workers also tested their algorithms at 512x512 resolution and 16 subray anti-aliasing.[2] It is not always necessary to generate images of this high quality. If we instead ray trace the same scenes with a low anti-aliasing effect and low screen reso-

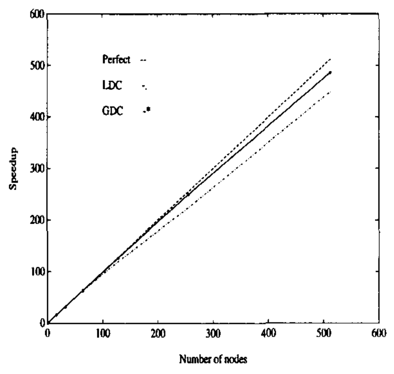

Figure 6: Parallel Speedup for the Mountain scene. Ray tracing time on a single processor = 17504 seconds.

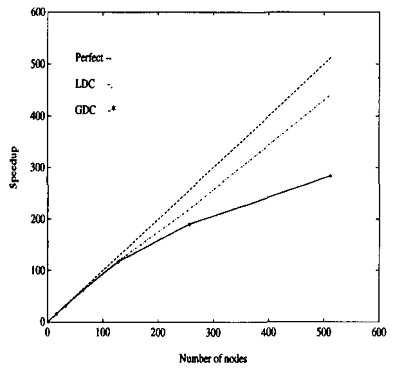

Figure 7: Parallel Speedup for the Tree scene with 512x512 resolution and 1 ray anti-aliasing. Ray tracing time on a single processor = 431 seconds.

lution, we will create a poorer quality image, but with much less computation effort. This level of quality might be appropriate for many applications. To test the effectiveness of LDC and GDC in producing lower quality images, we experimented with the "Tree" scene. We first decreased anti-aliasing from 16 to 1, which decreased the computation time of each pixel by approximately a factor of 16. We also tested performance with the screen resolution decreased from 512x512 to 256x256, which approximately decreases the time by an additional factor of 4. Figures 7 and 8 show the results for LDC and GDC. In contrast to the previous tests, LDC now performs better than GDC. The speed of the two methods is comparable up to 128 processors, but there is a noticeable deterioration in the performance of GDC for 256 and 512 nodes. For the larger numbers of processors, GDC's load balancing gains are offset by overhead costs.

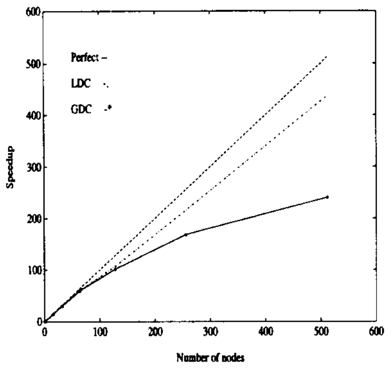

Figure 8: Parallel Speedup results for the Tree scene with 256x256 resolution and 1 ray anti-aliasing. Ray tracing time on a single processor = 110 seconds.

6 Enhancement of GDC

We have shown that the GDC method works very well for scenes that require large amounts of computation, but less well for low quality images, when the overhead associated with the global search for work becomes significant compared to the computation time. Although a global search scheme is required for maximum load balancing, we need to reduce the overhead involved in the global search if this method is to be practical for low-quality rendering. We thus present some modifications of GDC that improve parallel scalability, which are described as follows:

- There are N processors, $P_1, \ldots, P_N$.
- T_{single} is the ray tracing time on single processor and $T_{average}$ is $\frac{T_{single}}{N}$.
- The screen space is divided into R regions (2x2 pixels) and the average ray tracing time for a region is $\frac{T_{single}}{R}$ and is denoted as T_{region}.
- For each processor P_i, there are *N-1* other processors to search, $P_{i+1}, \ldots, P_N$, $P_1, \ldots, P_{i-1}$ which we order as $S_1, \ldots, S_{N-1}$.
- $T_{overhead}$ is the average cost for a single search step.
- α is $\frac{T_{average}}{T_{overhead}}$.
- β is $\frac{T_{region}}{T_{overhead}}$

α is a measure of the amount of time required by each processor to complete the computation relative to the cost of a search step. In our initial implementation of GDC, we allowed idle processors to search over all the other processors to find work. In order to achieve a reasonable tradeoff between load balancing and the overhead cost, we would like to limit the range of processors over which the search is done. If α is very large, then there is little cost in searching, and there is no need to restrict the range of the search. Thus, there is no change to the GDC algorithm as we initially presented it. However, as α decreases, it becomes beneficial to restrict the range of processors that are searched. We term this range $S_{maximum}$, and determine it from a percentage of α. Empirically, we have found that values of $S_{maximum}$ in the range of 5-20% of α provide good parallel performance. If α is very small, no global searching will be done and GDC is reduced to the static interleaved assignment method.

In addition to limiting the range of processors over which the search is conducted, we also want to avoid searching processors that are near neighbors to the idle processor. Due to image coherence, neighboring regions will have similar computational requirements. If a node become idle, there is a high probability that its neighbors may also be idle. Thus, we would also like to request work from nodes that are distant to the idle node. We define a search step increment, Δ, which is equal to $\lceil \frac{N}{S_{maximum}} \rceil$. An idle processor will search for work among the processors $S_{\Delta}, S_{2\Delta}, \ldots, S_{S_{maximum}\Delta}$ in a somewhat interleaved fashion.

We can also improve the performance of GDC by considering the amount of available work left at each node. For example, when a processor P_i makes a request from a processor located at $S_{k\Delta}$, we can assume that the processors located at $S_{\Delta}, S_{2\Delta}, \ldots, S_{(k-1)\Delta}$ are either executing their last local region or a region that they requested from other nodes. Therefore, these processors may also send a request to $S_{k\Delta}$ very soon. It is counterproductive for many idle processors to make requests from a processor which has only a few computation tasks left. To help prevent this situation, we use the following strategy: After receiving a request from an idle processor P_i, an active processor located at $S_{k\Delta}$ will send back a region for computation, as well as the number of regions it has left to do, R_{left}. If $\beta * R_{left} < (k\text{-}1)$, P_i will skip $S_{k\Delta}$ and request an extra region from the processor located at $S_{(k+1)\Delta}$ the next time it becomes idle. For the scenes rendered in this work, the β's are 25.3, 10.1 and 10.2 for the "Mountain", "Tree", and "Balls" scenes. If β became too small we could enlarge the region size (currently 2x2 pixels). In effect, we are determining whether it is cost effective for P_i to request additional work from a processor located at $S_{k\Delta}$. With this strategy, each idle node has an equal probability of getting extra work from an active processor $S_{k\Delta}$ in response to the first request. However, to make searching for additional work more cost effective, after the first request a idle processor nearer to a processor $S_{k\Delta}$ will have more probability of getting work than farther nodes. Figure 9 shows our experimental results for ray tracing the three scenes at 256x256 resolution with 1 anti-aliasing effect and the percentage of α is set to 10. The overall parallel performance shows the same trend as all the previous tests; the improved GDC performs the best on the Mountain scene and the worst on the Balls scene. However, the speedup is very good compared to the basic method. The im-

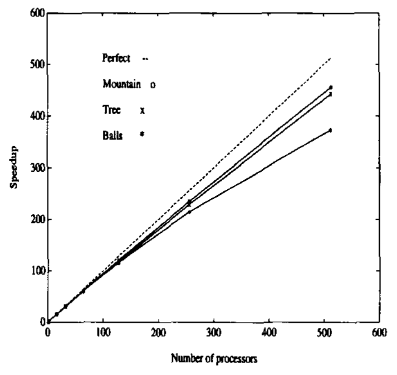

Figure 9: Parallel Speedup for the three scenes with enhanced GDC method, 256x256 resolution, and 1 anti-aliasing

proved GDC gives a speedup comparable to the LDC on the Trees scene. These results show that with enhancement the GDC method is able to efficiently render scenes with a wide range of computational requirements.

7 Conclusions

In this paper, we propose two new load balancing strategies for parallel ray tracing. Our methods, LDC and GDC, consistently achieved a very good parallel speedup for rendering high quality images. However, the cost of global searching in GDC impairs performance as the amount of computation per pixel goes down. In the cases in which we generate lower quality images, LDC gives better parallel speedup. However, by placing restrictions on the manner in which the global searching for work is done, we can greatly improve the basic GDC method. The enhanced GDC gives a parallel speedup for the low quality images that is comparable to LDC. Thus, we can potentially use GDC for rendering images with a wide range of quality.

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.

References

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

[2] Stuart Green, "Parallel Processing for Computer Graphics", MIT press, 1991.

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

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

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

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

[7] Jamie Packer, "Exploiting Concurrency: A Ray Tracing Example", Inmos technical note 7, Inmos Ltd., Bristol, 1987.

[8] 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).

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

[10] Capspary, E. et. al., "A Self-balanced Parallel ray tracing a algorithm", in Parallel Processing for Computer Vision and Display, P.M., T.R. Heywood, and R.A. Earnshaw, editors, Addision-Wesley, 1989.

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

[12] Carter, M. B. and Teague, K. A., "The Hypercube Ray Tracer", In Proc. of the 5th Distributed Memory Computing Conference, 1990.

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