The Journal of Supercomputing, 12, 57–68 (1998)
© 1998 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Competitive Analysis of Load Balancing Strategies for Parallel Ray Tracing

ALAN HEIRICH heirich@sgi.com
JAMES ARVO arvo@cs.caltech.edu
Silicon Graphics Computer Systems, 2011 N. Shoreline Blvd., Mountain View, CA 94043
Department of Computer Science, California Institute of Technology, Pasadena, CA 91125

Editor: Hamid R. Arabnia

Abstract. This paper examines the effectiveness of load balancing strategies for ray tracing on large parallel computer systems and cluster computers. Popular static load balancing strategies are shown to be inadequate for rendering complex images with contemporary ray tracing algorithms, and for rendering NTSC resolution images on 128 or more computers. Strategies based on image tiling are shown to be ineffective except on very small numbers of computers. A dynamic load balancing strategy, based on a diffusion model, is applied to a parallel Monte Carlo rendering system. The diffusive strategy is shown to remedy the defects of the static strategies. A hybrid strategy that combines static and dynamic approaches produces nearly optimal performance on a variety of images and computer systems. The theoretical results should be relevant to other rendering and image processing applications.

Keywords: load balancing, ray tracing, rendering, graphics, image processing, diffusion.

1. Introduction

High quality computer generated images can attain the clarity and realism of photographs. Techniques for photo-realistic image synthesis model the transport of light through geometrically complex scenes. The most popular of these techniques, ray tracing and radiosity [7, 11] require massive amounts of floating point calculations, comparable to the requirements of the largest problems in computational science and engineering. It has long been realized that parallel computers can be an effective way to accelerate these computations [5, 10, 14].

The advent of commodity based cluster computing has made it possible to render photo-realistic images in reasonable time on modest budgets. To take a single example, a parallel rendering system running on a $50,000 commodity cluster computer (the 3.2 GFlops NASA/Caltech Beowulf system) renders a complex image, which takes close to an hour on a high performance workstation, in under two minutes [10, 16, 17]. Further speedups can be obtained by larger clusters and high performance parallel computers.

These speedups are most important in interactive settings where a human is in-the-loop waiting for images to be rendered. These settings occur in product design, architectural walkthroughs, and in previewing commercial animation. In the face of demanding performance requirements it is necessary to use computing resources efficiently. This requires an effective load balancing strategy that can distribute work among the computers without adding significantly to the overhead of the computation.

This paper compares the effectiveness of popular strategies for load balancing ray tracing computations on large parallel computer systems. It considers static image partitioning strategies, both with and without randomization, and compares them to a dynamic scheme based on a diffusion model [9]. Using both offline simulation, and online performance measurements, the paper demonstrates that static strategies produce unacceptable levels of workload imbalance in rendering complex images. This point is underscored by an analysis of the effectiveness of randomization strategies, which shows that these strategies do not scale effectively to large numbers of processors. Under a set of generous assumptions, pixel level random assignment is shown to be ineffective for NTSC resolution images on parallel computers with 128 processors or more. Strategies in which the image is divided into tiles, which are then randomly distributed, are even less effective, with the effectiveness decreasing as the tile size increases. This analysis should be relevant to other rendering and image processing applications.

The paper concludes by measuring the online performance of a dynamic load balancing scheme based on a diffusion model. Such schemes, which can be implemented as a form of work-stealing, have previously been analyzed and shown to scale without limit [8, 9]. In this paper these predictions are tested empirically by measuring the performance of these schemes on substantial test problems. In general the dynamic scheme proved at least as effective as the best of the static schemes and was usually superior. The best results were obtained with a hybrid strategy, in which the problem was initialized using the best static scheme, and subsequently rebalanced using the dynamic scheme. With the hybrid strategy some results were within seconds of the optimal time on runs that lasted for over an hour.

2. Ray tracing on parallel computers

Ray tracing algorithms estimate the intensity and wavelengths of light entering the lens of a virtual camera in a simulated environment. These quantities are estimated at discrete points in the image plane that correspond to pixels. The estimates are taken by sending rays out of the camera and into the scene to approximate the light reflected back to the camera. This process requires finding points of intersection among rays and objects in the environment, a common geometrical problem known as ray casting. When these estimates faithfully reflect the scattering properties of materials and the emissive properties of light sources the results can be breathtaking, and even indistinguishable from photographs.

The cost to compute individual pixels can vary dramatically, depending on the complexity of the model being rendered and the algorithm employed. Different surfaces will scatter different amounts of light, requiring different amounts of computation to follow the associated paths. Light can travel long distances as it is reflected by mirrorlike surfaces, or can be rapidly attenuated by collisions with nonreflective surfaces. The positions of light sources with respect to geometry and viewing position determines the number and lengths of the paths that must be followed. In general these effects cannot be precomputed.

When ray tracing is implemented on a parallel computer these costs play an important role in determining the overall solution time. In this paper a Monte Carlo algorithm is employed which is capable of estimating complex scattering and emissive properties. The cost of this algorithm, measured in floating point operations, was evaluated for a set of images from the Materials and Geometry Format data base maintained at the Lawrence Berkeley

Laboratories. Typical values of the mean μ and variance σ^2 of these measurements were 10^6 and 10^{12} respectively. This large variance presents a challenge to load balancing, since if even one pixel has a dramatically higher cost than others this pixel will come to dominate the computation. While other ray tracing algorithms will exhibit different statistics it is common for them to share this characteristic of a large variance relative to the mean. Implementation strategies designed to speed up a computation, such as prematurely terminating a path that falls below an intensity threshold, generally increase this relative variance. This problem therefore holds practical significance for most parallel ray tracing implementations.

3. Load balancing

The load balancing problem is fundamental to parallel computing and arises in many contexts. In the present context the problem is to distribute the calculations for a set of pixels in an image in order to minimize the elapsed rendering time. Each pixel has an associated amount of computation w_i that can be modeled by the number of floating point operations required to compute the value of that pixel. The discussion that follows will assume that all images are at NTSC resolution, 640×480, for a total of 307,200 pixels. Then if there is a set of computers indexed by j, and a mapping function $\mathcal{F}(i) \rightarrow j$ that assigns each pixel to a computer, the objective of the *load balancing problem* is to find an $\mathcal{F}$ to minimize

$$T \equiv \max_j \sum_{i \in \mathcal{F}^{-1}(j)} w_i. \tag{1}$$

A minimum of T occurs when all computers complete their work in the same amount of time. If all p computers perform work at the same rate this minimum occurs at $T_{min} = \sum_i w_i/p$. This suggests an objective function to measure the effectiveness of any candidate solution $\mathcal{F}$ to any instance of the load balancing problem. The quality of any $\mathcal{F}$ can be measured by the percentage of imbalance that it produces,

$$\epsilon \equiv (T - T_{min})/T_{min}. \tag{2}$$

It has come to be commonly accepted that “embarrassingly parallel” algorithms can be implemented so that they scale linearly with respect to the number of processors that they run on, up to large numbers of processors [2]. In the best cases a speedup of p can be achieved on p processors. In these cases overheads of even a few percent can come to dominate the efficiency of a computation. In our rendering system the parallel overheads of communication and scheduling have been measured in the neighborhood of one to five percent on as many as 256 computers. We would like to achieve a load balance in this range, so that load imbalance does not become the primary source of inefficiency in the computation.

Strategies to solve the load balancing problem are usually categorized as either *static* or *dynamic*. Static load balancing strategies construct an $\mathcal{F}$ prior to the start of a calculation and never modify it. These strategies can expend considerable resources to obtain a good initial $\mathcal{F}$, but they are vulnerable to data dependent imbalances that can arise during the course of a calculation. This is particularly relevant to contemporary ray tracing algorithms

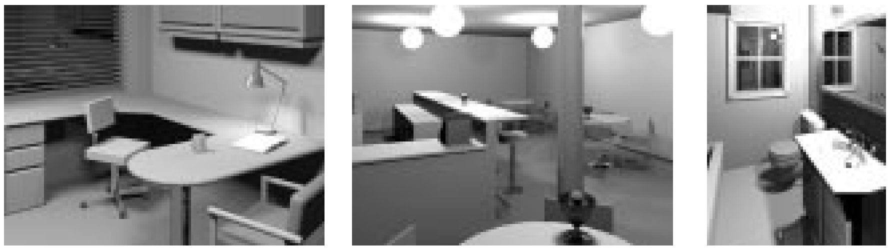

Figure 1. The three images (office, soda, and bath) used in this study.

where adaptive sampling strategies and a variety of optical effects can produce per-pixel costs that are highly variable and data dependent.

Dynamic load balancing strategies work on-line during the course of a calculation to redistribute work as the need arises. Several algorithms based on recursive partitioning have been proposed for static load balancing and some of these have also been applied to dynamic load balancing [12, 18]. Unfortunately these approaches are usually expensive because they always repartition an entire calculation. It would be desirable to have a dynamic load balancing strategy that worked locally, required no global communication, and expended computational work only in areas where it was needed. *Diffusion* has been discussed as a general solution to the problems of load balancing and mapping in distributed systems and in recent years several authors have proposed or demonstrated diffusive algorithms with these desirable properties [4, 6]. When these algorithms are constructed from finite difference approximations to differential equations they are computationally efficient and converge at rates that are independent of problem scale [8, 9].

4. Static load balancing strategies

The following discussion will assume that the geometric model is replicated within the memory of every computer in a parallel computer system. (When the geometric model is too large to replicate in this way the issues associated with load balancing are very different and will not be discussed here.) Our experience has shown that a computer with 64 MB of memory running Unix can run our rendering software and hold a model of roughly 10^6 geometric elements. By comparison, the scenes in figure 1 contain roughly 20,000 elements each. The corresponding data structures occupy roughly one MB.

A typical ray tracing algorithm samples the image plane by issuing a set of primary rays from a camera into an environment. Contemporary algorithms commonly include a bidirectional phase as well, in which rays are emitted from the light sources. In both cases the computation is highly dynamic and data dependent. Adaptive sampling strategies may vary the numbers of primary rays according to the evolving values of individual pixels. The distribution of subsequent rays (shadow, specular, or inter-reflection) may also vary dramatically according to material properties and sampling strategies.

Static load balancing strategies partition the image plane among a set of computers and then render the different portions of the image concurrently. Naive strategies that partition contiguous segments of the image fare poorly. These strategies suffer from the effects of locality in the image which can lead to wide variations in workload for different computers. A more effective strategy assigns pixels pseudo-randomly to computers in an attempt to obtain a balanced workload. The most straightforward version of this strategy assigns pixels in an alternating sequence so that $\mathcal{F}(i) = (i \bmod p)$. This well-known strategy has been dubbed the "scatter decomposition" [15].

The effectiveness of various static load balancing strategies can be predicted using data obtained from program traces generated in the course of rendering complex images. Figure 1 shows black and white versions of three images that were used for the study reported here. Each image is at NTSC resolution (640 $\times$ 480 pixels). A Monte Carlo ray tracer was instrumented to collect empirical values of w_i (required floating point operations) for each pixel. The three images were then rendered using an adaptive sampling strategy that generated from one to twenty five primary rays through each pixel. After rendering, the w_i were collected from the program traces and were used to predict workload distributions for the naive strategy and scatter decomposition on varying numbers of computers. In order to evaluate the scatter decomposition, predictions were also computed for a strategy in which $\mathcal{F}(i)$ is chosen at random. These predictions are shown in figure 2.

All three images have highly nonuniform distributions as illustrated by their histograms (left column of figure 2). These histograms show the frequency of occurrence (vertical axis) versus floating point operations (horizontal bins). For each image the predicted imbalance ϵ (right column) is shown for three static load balancing strategies (naive, scatter, and random) on increasing numbers of computers (horizontal axis). In all cases the naive strategy produces the largest predicted ϵ (vertical axis) and the random strategy produces the smallest. The scatter decomposition is generally comparable to the random strategy but suffers from isolated peaks at multiples of 320 pixels, exactly one half the width of the image. In all cases the predicted ϵ is greater than 0.10 when the number of computers is 128 or higher.

These results suggest that the naive strategy fares poorly in general, and that even a random strategy will fail to obtain $\epsilon \leq 0.10$ when the number of computers is 128 or more. They also reveal that the scatter decomposition, while often assumed to be equivalent to a random assignment, is in fact subject to pathologies related to spatial regularity in the image, as illustrated by the peaks that occur at multiples of 320 pixels. (It seems relatively easy to work around this pathology. For example, a simple strategy adds an offset k to each $\mathcal{F}(i)$, where k changes from scan line to scan line according to a pseudo-random series known at every computer. This strategy breaks up the low spatial frequencies in the image that produce the observed peaks.)

The accuracy of these predictions was tested empirically by measuring the elapsed times to render the three images on various numbers and types of computers using the naive and scatter strategies. The resulting measurements are shown in table 1. The measurements are generally in close agreement with the predictions but there are a few surprises. For example, the scatter decomposition worked well for the bath image on 16 computers but performed poorly on 15 computers. The converse was true for the image of the soda shop interior. In all cases the naive decomposition produced unacceptably high values of ϵ, as high as 2.22 for the bath image on 32 computers. Figure 2 suggests that all of these methods

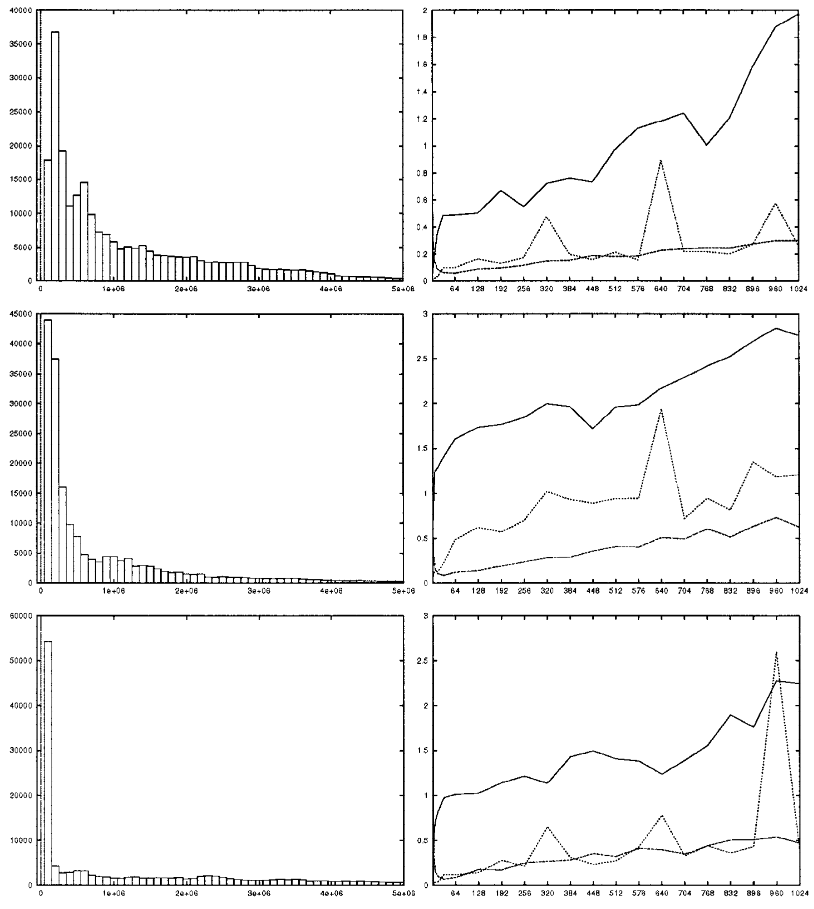

Figure 2. Distribution histograms (left) and predicted imbalances (right) for the three images. The three rows correspond to the three images, from top to bottom: office, soda, and bath. Each histogram shows the frequency of occurrence on the vertical axis, and bins of cost per pixel (measured in floating point operations) on the horizontal axis. The rightmost tails of the histograms have been truncated for the sake of appearance. The mean and variance for the images appear in table 2. In all cases the distributions exhibit a large right skew, in that they contain small numbers of extremely high-cost pixels. It is this skew that undermines all static load balancing strategies. Each predicted imbalance shows the result of an offline simulation of three static load balancing strategies. The vertical axis represents ϵ and the horizontal axis represents the number of processors. In each case the top line is the result of the naive strategy, the bottom line the result of a pure random strategy, and the middle line the result of scattering pixels.

Table 1. Empirical results measuring effective imbalance ϵ in rendering the office, soda, and bath images with different load balancing strategies on varying numbers of computers. These measurements represents the rendering core in isolation from the user interface, rasterization, and other non-scalable components. In each case one additional computer supported these non-scalable components.

Model	p	T_{naive}	ϵ	$T_{scatter}$	ϵ	$T_{diffusion}$	ϵ	T_{hybrid}	ϵ	T_{min}
Office	15	1:08:10	0.22	1:00:46	0.05	58:41	0.02	57:51	0.00	57:38
	16	1:00:53	0.08	1:00:24	0.07	57:15	0.01	57:22	0.02	56:25
	32	36:29	0.29	30:46	0.09	29:42	0.05	28:58	0.02	28:18
	64	19:44	0.39	15:56	0.12	15:49	0.11	14:35	0.03	14:12
Soda	15	1:34:08	0.30	1:16:31	0.06	1:17:22	0.07	1:13:03	0.01	1:12:13
	16	1:50:57	0.46	1:35:27	0.25	1:23:06	0.09	1:18:13	0.03	1:16:07
	32	1:32:08	1.41	48:51	0.28	48:28	0.27	40:25	0.06	38:12
	64	49:05	1.57	25:37	0.34	24:25	0.28	21:21	0.12	19:05
Bath	15	2:14:46	0.40	2:06:09	0.31	1:42:46	0.07	1:38:58	0.03	1:36:28
	16	2:24:17	0.48	1:55:15	0.18	1:46:06	0.09	1:38:35	0.01	1:37:24
	32	2:36:44	2.22	1:17:59	0.60	1:00:04	0.24	51:14	0.05	48:38
	64	40:29	0.67	38:43	0.60	31:24	0.30	26:34	0.10	24:11

will break down as p increases. The following sections discuss theoretical support for this conclusion.

5. Limits of randomization

Predictions derived from empirical measurements of w_i suggest that static load balancing strategies will be ineffective for NTSC resolution images rendered on 128 or more computers. These predictions might be criticized on the grounds that they are based on w_i for specific images and for a specific type of ray tracing algorithm. In fact these predictions are quite general as may be demonstrated by considering the expected behavior of a perfectly random strategy. The central limit theorem states that when a sufficiently large number of samples w_i are drawn from a sample population the sums of any sets of n samples will take on a normal distribution. If the sample population is uniform then it is possible to quantify the probability that the sum of any individual set of n samples is below a given bound y [13]. In the general case we have

$$P\left[\sum_{i=1}^{n} w_i \le y\right] = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{(y-n\mu)/\sigma\sqrt{n}} \left(e^{-u^2/2} du\right) = \frac{1}{2} + \frac{1}{2}\text{Erf}\left(\frac{y-n\mu}{\sigma\sqrt{2n}}\right). \quad (3)$$

In this expression μ and σ^2 are the mean and variance of the sample population. When this is applied to the problem of ray tracing for NTSC resolution images we take n equal to m/p where m is 307,200, the total number of pixels. We define y to be $(1+\epsilon)T_{min}$ to obtain a formula for a new quantity Pr.

$$Pr \equiv P\left[\sum_{i=1}^{n} w_i \le (1+\epsilon)\,T_{min}\right] = \frac{1}{2} + \frac{1}{2}\text{Erf}\left(\frac{\epsilon\mu m}{p\sigma\sqrt{2n}}\right) \quad (4)$$

The probability that all p computers will be balanced to within a factor of ϵ is the product of p instances of Pr, that is, Pr^p. This quantity can be computed for varying numbers of

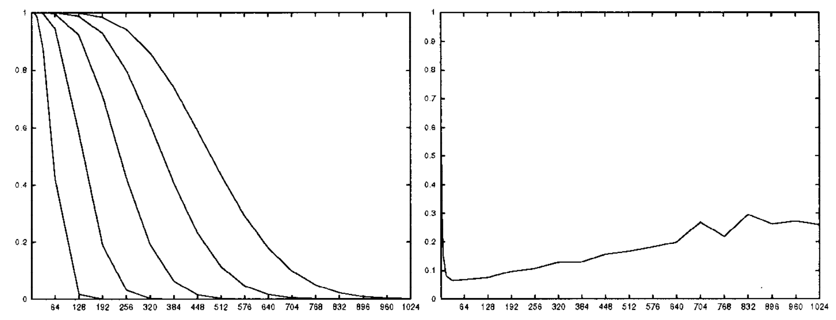

Figure 3. The result of applying a random assignment strategy to a uniformly distributed sample population with mean $\mu = 10^6$ and variance $\sigma^2 = 10^{12}$. The left graph shows Pr^p (vertical axis) versus p (horizontal axis) for (left to right) $\epsilon = 0.05, 0.10, 0.15, 0.20, 0.25$. The right graph shows the result of an offline simulation of a pure random assignment strategy with a uniformly distributed population. The assumption of uniformity makes these results optimistic in comparison to realistic cases.

Table 2. Probability (Pr^p) of obtaining $\epsilon \le 0.10$ by random assignment of pixels. See equation (4).

			Number of computers p				
Model	μ	σ^2	16	32	64	128	256
Office	9.72×10^5	1.38×10^{12}	0.99	0.98	0.87	0.34	0.00
Soda	8.70×10^5	5.10×10^{12}	0.96	0.71	0.18	0.00	0.00
Bath	1.37×10^6	8.95×10^{12}	0.76	0.30	0.01	0.00	0.00

computers and varying ϵ. The result of such a computation is shown in figure 3. This figure shows that the probability of achieving $\epsilon \le 0.10$ by a random strategy declines rapidly with increasing numbers of computers. This is not surprising since the number of pixels is fixed at NTSC resolution. As more computers are applied to the rendering problem fewer pixels are assigned to each computer and the law of large numbers breaks down.

The accuracy of equation (4) can be affected by skew in the sample population. Complex images generally do not have uniformly distributed w_i as is evident from figure 2. As a result the predictions of (4) may err on the side of optimism. For example, compare the simulated results of figure 2 with the predictions of table 2. As ray tracing algorithms become increasingly sophisticated they generally increase the skew in the sample population, decreasing the chances of success for random load balancing strategies. The absolute magnitudes of μ and σ^2 are not important since ϵ is defined as a percentage of μ rather than an absolute value. For all of the above reasons we conclude that these predictions apply to a wide range of images and ray tracing algorithms. These conclusions should also be relevant to other rendering and image processing algorithms.

These conclusions are rather ironic. It is precisely in the context of large numbers of computers that the quality of load balance becomes critically important. In order to achieve sub-second frame rates, using general purpose computers, it is necessary at the present time to employ hundreds of computers. (This situation will improve, as processors continue

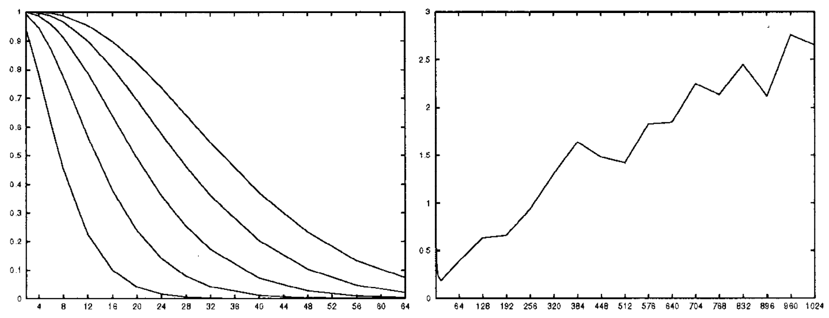

Figure 4. The result of randomly distributing a tiled partition using an 8×8 tiling strategy. These graphs have the same meaning as in figure 3.

to grow faster, and as special purpose accelerators become increasingly available.) It is unfortunate that with random assignment strategies the quality of the solution decreases as the numbers of computers increases, so that when p is 500 or more the expected value of ϵ is at least 0.25. It is clear that a static load balancing strategy based upon randomization will be insufficient for rendering NTSC resolution images on large numbers of computers. At higher resolutions like HDTV similar conclusions will obtain although the numbers would differ in absolute terms.

6. Tiling strategies

Another popular static load balancing strategy partitions an image into rectangular tiles and then distributes the tiles randomly to computers. The preceding analysis indicates that this strategy will be less successful than pixel based scattering, because it amounts to reducing the value of m in equation (4). For example, a strategy that partitions an NTSC resolution image into tiles of 8×8 pixels each reduces m from 307,200 to 4,800. Figure 4 shows the result of a static random assignment strategy under these conditions. The effective imbalance grows far more rapidly than with the pixel scattering in figure 3. Even with sixteen processors the probability of achieving ϵ below 0.10 is less than 50%. If the results on real, non-uniformly distributed problems are similar to those of pixel scattering, then tiling strategies are likely to be effective only on very small computer systems, with processor counts in the single digits. Applications that intend to achieve scalability should avoid tiling strategies.

7. Dynamic load balancing by diffusion

One way to achieve good load balance for large p is to employ a scalable strategy in order to balance workloads dynamically. *Diffusion* is one of the few load balancing paradigms that is provably scalable. Diffusion algorithms provide scalable solutions to a number of other dynamic problems in placement and partitioning, including the mapping problem [8, 9].

We have experimented with diffusive load balancing in the context of Monte Carlo image synthesis. The results were very favorable and appear in table 1. In most instances the diffusion algorithm produced a smaller imbalance than scatter decomposition, as evidenced by shorter elapsed rendering times. This is in itself remarkable since the scatter strategy incurs practically no cost, while the diffusion algorithm must operate concurrently with the rendering process and thus incurs some run time overhead. To take one example, in rendering the office model on the Beowulf computer, the diffusion algorithm added only 57 seconds to the overall elapsed time, which was approximately an hour. When the diffusion algorithm was combined with the scatter decomposition this overhead was reduced to 13 seconds, less than one percent of the elapsed time.

This pattern was also observed in other instances. In general, the combination of scatter and diffusion approaches produced the lowest values of ϵ. Under these circumstances load imbalance ceases to be the dominant term in the elapsed time for a computation, and the load balancing problem has been solved effectively. In every case a hybrid strategy that combines diffusion and scattering produced the best results. In most cases diffusion by itself produced better results that scattering. In general the problem becomes more difficult with increasing numbers of computers.

Several practical lessons were learned in the course of implementing and tuning the diffusion algorithm. The theoretical work on these algorithms assumes that dynamic load balancing occurs throughout a computation, and that the goal of the algorithm is to maintain an equilibrium workload at all times. This is an appropriate assumption for applications in which new work is created frequently and work queues are limited in size. Our initial experiments with this approach revealed that it added a significant overhead to the computation, because load balancing was occurring throughout the entire rendering process. By properly implementing the strategies for managing work queues it was possible to ensure that work queues would not overflow and that computers could proceed for indefinitely long periods without re-balancing.

As a result we adopted a “work stealing” strategy whereby an interconnected set of computers load balances only when one of its members runs out of work. The load balancing operation is initiated by the computer that runs out of work, and never spreads more than a single step through the interconnection network. The problem of computing a solution for a locally connected set of computers is very simple, since it amounts to directly solving for an equilibrium workload. The result is a simple deterministic algorithm that converges in a single step. This simple algorithm works as follows:

- Computer p_i runs out of work and sends a “request status” message to every immediate neighbor p_j.

- Each neighbor p_j sends a “status report” to p_i that reveals it’s current workload, quantified as the number of pending rays that are to be cast.

- After receiving all status reports p_i sends a “transfer work” message to every neighbor that reported more than the average workload among the set of p_i, p_j.

- Each of these neighbors p_j removes a designated amount of work from it's queue of pending work, and sends these transactions in a series of messages to p_i, which places them into it's own queue.
- After receiving all of this contributed work, p_i sends unsolicited amounts of work to every neighbor that reported less than the average workload among the set.

In the course of tuning the load balancing algorithm for performance we were forced to make certain policy decisions. A few of these decisions turned out to be critical to achieving the desired efficiencies. For example, when a p_j transfers work to a p_i it is important to select primary rays whenever possible, in order to give p_i the greatest possible amount of future work and thus minimize the number of subsequent load balancing operations that p_i will initiate. During the period when a requester p_i is waiting for status reports from the p_j the p_i should continue to cast rays. Similarly the p_j should continue ray casting after they send their status reports and while they are waiting for requests from the p_i. During this period if a p_j receives a request for status from another peer $p_{i'}$, $i \neq i'$, the p_j responds to the $p_{i'}$ with a "busy" signal. Upon receiving this signal the $p_{i'}$ ignores this p_j for the purpose of load balancing.

With an appropriate set of policy decisions the diffusion algorithm has proven to be remarkably efficient. When combined with the scatter decomposition nearly optimal performance was observed in the best instances.

8. Summary and conclusions

This paper has presented the load balancing problem for parallel ray tracers and defined a measure ϵ of the imbalance in potential solutions. It has demonstrated that for NTSC resolution images, static load balancing strategies based upon randomization result in unacceptably high values of ϵ. This was shown empirically for the office image on 64 computers, the soda image on 16, 32, and 64 computers, and the bath image on 15 computers (table 1). It was shown through simulation and analysis that this problem becomes only more severe as p increases (figures 2, 3, 4). This result was corroborated by predictions of the central limit theorem which showed that the probability of obtaining acceptable ϵ through randomization vanished for p above 128 (table 2). In the case of tiled strategies the results were even worse (figure 4). Tiling strategies do not appear to be effective for NTSC resolution images when processor counts are above single digits. A dynamic load balancing method based on a diffusion model was implemented and demonstrated to produce results that were generally superior to randomization. When a hybrid strategy that combined diffusion with randomization was tested it produced outstanding values of ϵ that were in some cases less than one percent.

From these results we conclude that static load balancing strategies are insufficient for time constrained parallel ray tracing on large numbers of computers. For NTSC resolution images unacceptable ϵ were observed on as few as 15 computers, and were predicted to predominate when p is above 128. For higher resolution images with larger numbers of pixels randomization methods will work better but will still deteriorate as p increases. For this reason it would appear necessary to use a dynamic load balancing algorithm. When a diffusive dynamic load balancing algorithm was combined with a static randomization strategy the load balancing problem was solved, in the sense that the observed imbalance ϵ represented a smaller percentage of the overall calculation than the other overheads due to parallelism.

Acknowledgments

Models were provided by Greg Ward Larson of the Lawrence Berkeley Laboratories and Silicon Graphics Computer Systems. Alan Heirich has been supported by Caltech's Center for Advanced Computing Research, the Program of Computer Graphics at Cornell University (NSF Project ASC-9523483, "MRA: Physically and Perceptually-Based Parallel Global Illumination Solutions"), the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (NSF Project ASC-8920219), the U.S. Army Research Office, grant DAAH04-96-0077, and Silicon Graphics Computer Systems. The research reported here has made use of computing facilities provided by Caltech's Center for Advanced Computing Research, the Cornell Theory Center, and the Maui High Performance Computing Center.

References

1. Anderson, T. E., Culler, D. E. & Patterson, D. A. 1995. A case for NOW (Networks of Workstations). IEEE Micro, 15: 54-64.
2. Baskett, F. & Hennessy, J. L. 1993. Microprocessors: from desktops to supercomputers. Science, 261: 864-871.
3. Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N. & Su, W. 1995. Myrinet: a gigabit per second local area network. IEEE Micro, 15: 29-36.
4. Cybenko, G. 1989. Dynamic load balancing for distributed memory multiprocessors. Journal of Parallel and Distributed Computing, 7: 279-301.
5. Delaney, H. C. 1988. Ray tracing on the Connection Machine. Proceedings of SIGGRAPH (Atlanta, August), ACM Press, pp. 659-664.
6. Dijkstra, E. W. & Schölten, C. S. 1980. Termination detection for diffusing computations. Information Processing Letters, 11: 1-14.
7. Hanrahan, P., Saltzman, D. & Aupperle, L. 1991. A rapid hierarchical radiosity algorithm. Computer Graphics, 25: 197-206.
8. Heirich, A. & Taylor, S. 1995. A parabolic load balancing method. Proceedings of the 24th International Conference on Parallel Processing (Milwaukee, August), CRC Press, III, pp. 192–202.
9. Heirich, A. 1997. A scalable diffusion algorithm for dynamic mapping and load balancing on networks of arbitrary topology. To appear in The International Journal of Foundations of Computer Science.
10. Heirich, A. & Arvo, J. 1997. Scalable Monte Carlo image synthesis. To appear in Parallel Computing.
11. Kajiya, J. T. 1986. The rendering equation. Computer Graphics, 20: 143-150.
12. Karypis, G. & Kumar, V. 1995. Multilevel graph partitioning schemes. Proceedings of the 24th International Conference on Parallel Processing (Milwaukee, August), CRC Press, III, pp. 113-122.
13. Lindgren, B. W. 1962. Statistical Theory. New York: Macmillan.
14. Priol, T. & Bouatouch, K. 1988. Experimenting with a parallel ray-tracing algorithm on a hypercube machine. Proceedings of Eurographics (Nice, September), North-Holland, pp. 243-259.
15. Salmon, J. 1988. A mathematical analysis of the scattered decomposition. Proceedings of the Third Caltech Conference on Hypercube Computers and Applications (Pasadena, January), ACM Press, pp. 239-240.
16. Sterling, T. L. 1996. The scientific workstation of the future may be a pile-of-pcs. Communications of the ACM, 39: 11-12.
17. Taubes, G. 1996. Do-it-yourself supercomputers. Science, 274: 1840.
18. Williams, R. D. 1991. Performance of dynamic load balancing algorithms for unstructured mesh calculations. Concurrency: Practice and Experience, 3: 457-481.