

Scalable Monte Carlo Image Synthesis

Alan Heirich^{a,b} and James Arvo^b

^a*Center for Advanced Computing Research*

^b*Department of Computer Science
California Institute of Technology, Pasadena, CA 91125*

This paper describes a scalable photorealistic renderer that is designed to render scenes of arbitrary complexity on computer systems of arbitrary size. The rendering algorithm is a Monte Carlo method to compute approximate solutions of the rendering equation. The software implementation uses a diffusive load balancing method coupled with a message driven concurrent pipeline. Measured performance in rendering replicated models on up to 256 computers shows scaling efficiencies as high as 99 percent. Simple extensions will partition extremely large models across physically distributed memory as well as perform out-of-core calculations.

1 Introduction

In recent years scalable parallel processors (SPPs) have become readily available to solve computationally intensive problems in science and engineering. Experience with these applications has shown that for certain classes of problems these SPPs routinely achieve speedups close to a factor of P using P computers [5,15]. Photorealistic rendering of complex scenes poses computational challenges that rival even the largest of these scientific and engineering calculations. It is therefore reasonable to explore how this problem can achieve similar speedups from SPPs.

Amdahl's law states (correctly) that only concurrent computations demonstrate significant scalability on parallel computer systems [2,12]. Therefore it is not surprising that the first successful applications of SPP technology have been to solve large scale problems in chemistry and physics that involve concurrent phenomena. The natural world abounds with such phenomena. To cite just one example, the process of human vision encompasses a concurrent flow of photons through space, and concurrent computations within the retina of the eye, optic nerve, and visual cortex of the brain. Concurrent processing is a prerequisite to real time vision.

Photorealistic rendering can be expressed as a problem of modeling photon transport through a geometric environment. Ray tracing algorithms directly simulate transport by following the paths of individual photons [22]. A full simulation of this kind can be extremely expensive since the number of paths required to produce a high quality image can be enormous, and each path can require a significant amount of geometry processing. Path tracing algorithms reduce the number of paths substantially [16]. Despite this improvement a high quality rendering task implemented by path tracing can consume many hours or even days of computing time on a high performance workstation.

This paper describes a Monte Carlo implementation of path tracing on a parallel computer system. A Monte Carlo method is well suited to parallel implementation because it is inherently concurrent. It is particularly convenient for modeling arbitrary bidirectional reflectance distribution functions (BRDFs) and thus is appropriate for models with directionally diffuse reflective surfaces. In this paper we describe software implementation methods and algorithms that achieve scalable performance on large numbers of computers and accommodate models containing large numbers of geometric primitives. The approach works on a range of scalable computing systems available at the present time and in the foreseeable future. In particular we have demonstrated the approach on networks of workstations, cluster computers, massively parallel computers, a system with global shared memory, and uniprocessors.

The initial implementation replicates the geometric model in the memory of each computer. In this configuration we have rendered models containing nearly one half million geometric primitives, and have rendered models with a few thousand primitives in a matter of seconds. When model sizes exceed one million primitives it will be necessary to partition the model data among the computers. We have specifically designed the software to support this albeit at some loss of efficiency. These extensions lend themselves easily to supporting out-of-core computations.

The remainder of this paper is organized as follows. The next section describes the Monte Carlo path tracing algorithm. For completeness we give a brief overview of path tracing with a moderately efficient Monte Carlo sampling strategy and show how its natural recursion can be expressed more conveniently as an iteration. Following this we describe software implementation methods by which we achieve scalable performance. This performance is demonstrated on a set of images using a diffusion algorithm to perform dynamic load balancing [9,13,14]. Finally a set of mapping and load balancing strategies is described for models that are partitioned among computers or between memory and disk.

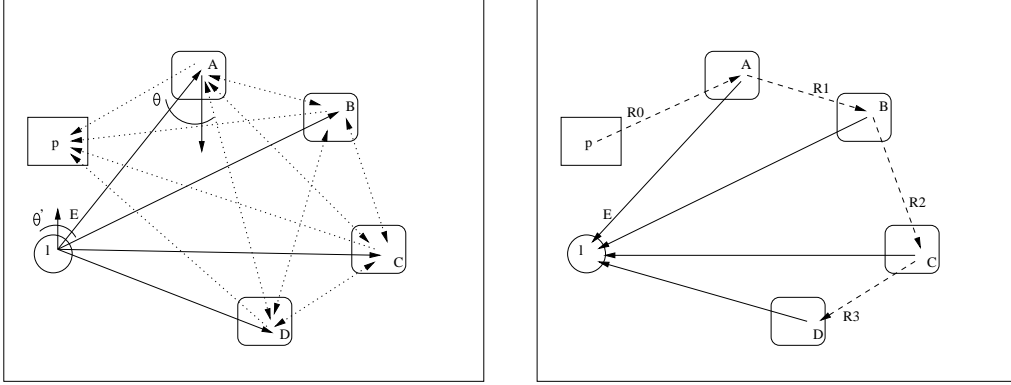


Fig. 1. Computing diffuse interreflection by path tracing. The objective is to compute the radiance L_p at a point p in the image plane. Left figure (a) illustrates light from a luminaire l scattering at points $A, B, C,$ and D on diffusely reflecting surfaces and reaching the image plane at p . Right figure (b) illustrates the path tracing algorithm to estimate the radiance L along a single path. Each step in the path is determined by shooting a ray in a random direction to reach a new point q . At each q the direct irradiance $\hat{\Phi}_q$ is computed, attenuated appropriately, and added to L_p . This procedure is typically repeated for many paths.

2 Monte Carlo path tracing

The radiance (brightness) L_p leaving a point p on a diffusely reflecting surface can be computed from the irradiance (density of incident power) Φ_p at p , which in turn can be estimated from the radiance L_{q_i} arriving from other points q_1, q_2, \dots, q_n in the environment. The radiance arriving from these other points may be estimated by path tracing.

Consider figure 1(a) which shows rays of light traveling indirectly from a luminaire l to a point p in the image plane. The irradiance Φ_A at a selected point A is a sum of the irradiance $\hat{\Phi}_A$ due to direct illumination from light sources and the irradiance due to light that is reflected from surrounding surfaces, such as $B, C,$ and D . Expressing irradiance as a sum of direct and indirect components is an application of the superposition principle, and is extremely useful for Monte Carlo calculations.

For simplicity in the following discussion we shall assume a single light source l . Thus, $\hat{\Phi}_A$ can be expressed as

$$\hat{\Phi}_A = \int_{\Omega} L_E \cos \theta d\omega, \quad (1)$$

where θ is the angle between a ray E from l to A and the surface normal at point A , and L_E is the radiance (in W/m^2sr) of E . This quantity is integrated over the solid angle Ω subtended by the surface of l , resulting in the irradiance

(in W/m^2). The differential solid angle $d\omega$ about a direction E is related to the surface of l by

$$d\omega = \frac{\cos \theta'}{r^2} dA, \quad (2)$$

where θ' is the angle between E and the surface normal of l , dA is the differential surface area of l , and r is the length of E .

To examine the effect of diffuse scattering assume the irradiance Φ_B is known at point B . The associated light energy is scattered by the diffuse surface at B , resulting in equal radiance in all directions. Each of the scattered rays has radiance L_{R_1} equal to $\alpha_B \Phi_B / \pi$, where α_B is a monochromatic reflectance coefficient at B . In this example the point B is visible to the point A in the direction ω . The radiance L_{R_1} , integrated over the solid angle subtended from A by the object containing B , contributes to the irradiance Φ_A at point A . The irradiance $\hat{\Phi}_A$ due to direct illumination from l can be calculated according to (1). The total irradiance Φ_A is the sum of $\hat{\Phi}_A$ and all indirect contributions (such as L_{R_1}),

$$\Phi_A = \hat{\Phi}_A + \int_{\square} L_R \cos \theta_A d\omega. \quad (3)$$

Here \square represents a unit hemisphere above the surface at A . As in (1) the angle θ_A represents the angle of incidence (i.e., the angle between incoming ray R and the surface normal at A). Similarly $d\omega$ represents a differential solid angle, which corresponds to an area on the surface of the sampling hemisphere. The irradiance at point A , in terms of the irradiance of surrounding Lambertian surfaces, can be expressed

$$\Phi_A = \hat{\Phi}_A + \int_{\square} \left(\frac{\alpha_B}{\pi} \Phi_B \right) (\cos \theta_A d\omega) \quad (4)$$

where the location of any point B is a function of A and the direction ω . Specifically, it is a point that is visible to A in the direction ω . (This integral is evaluated over many such points B).

In Monte Carlo path tracing the integral is estimated by tracing individual paths through a geometric environment (see figure 1) and then summing these individual contributions. At each point p along a path one or more sample points q_1, q_2, \dots, q_n from the surrounding environment are used to estimate the irradiance at p , denoted by Φ'_p . The points q_i are random variables; they are the points visible to p in n randomly chosen directions $\omega_1, \omega_2, \dots, \omega_n$ distributed

over the hemisphere above p . Thus, we have the estimator

$$\Phi'_p \approx \hat{\Phi}'_p + \frac{1}{\pi n} \sum_{i=1}^n \alpha_{q_i} \Phi'_{q_i} \cos \theta_i \quad (5)$$

It is generally more efficient to sample according to a cosine distribution over the hemisphere, which increases the density of sampling near the pole. This reduces the variance of the estimator and eliminates the factor of $\cos \theta$ from the samples. A cosine distribution is obtained by sampling the unit disk centered at p and projecting upward (along the surface normal at p) to the unit hemisphere. Because $\int_{\square} \cos \theta d\omega = \pi$, equation (5) becomes

$$\Phi'_{p_i} \approx \hat{\Phi}'_p + \frac{1}{n} \sum_{i=1}^n \alpha_{q_i} \Phi'_{q_i} \quad (6)$$

The objective of the path tracing procedure is to estimate the radiance L_p at an image point p . Equation (6) describes a recursive estimator, in which each Φ'_p is written in terms of a sum of several estimated Φ'_{q_i} . A direct implementation of (6) leads to a branching recursion which is well suited to uniprocessor implementation.

Such an implementation leads to practical difficulties on parallel computers. It is difficult in general to ensure that the result of a recursion is returned to its source of origin when data is partitioned among computers, or when work has been moved as a result of dynamic load balancing. It is far more convenient in a parallel context to implement an iterative strategy and this is what we have done. In this iteration rays follow a series of paths that are equivalent to the branching recursion of (6). Rays move forward along the paths until they reach a maximum path length. Along the path the product I_R of a series of α_{p_i} is computed which represents the accumulated attenuation of radiance along the path. At each point p_i the direct irradiance $\hat{\Phi}'_{p_i}$ is attenuated by this product and added to Φ'_p . The final quantity L_p is directly proportional to Φ'_p .

In the iterative path tracing procedure paths start at the eye point and flow into the scene, with successive hops potentially computed on different computers. We shall denote a ray by an ordered-triple (p, ω, I) , where p is the origin, ω is the direction, and I is the importance. Importance is a dimensionless scalar (per color channel) that indicates the fraction of emitted radiance that will contribute to a given pixel. Pixels of the image are incrementally refined by the loop shown below, in which rays are taken from the queue, processed, and additional rays that were spawned are added back to the queue.

Rays are initially entered into the queue as (p_0, ω_k, I_k) , where p_0 is the eye point, $\omega_1, \omega_2, \dots, \omega_s$ are s directions from p_0 through a given pixel, and I_1, I_2, \dots, I_s

are the weights according to some filter kernel; for example, $I_k = 1/s$ for a simple box filter. Each ray also carries with it the index of an associated pixel, which is either the pixel through which it was shot, or the pixel index of the ray that spawned it. When a light source is hit by a ray, a contribution is made to the associated pixel. The main loop for processing rays is as follows:

- (i) **Cast a ray:** Take ray (p, ω, I) from the queue and find the first point of intersection q on surface S . If the ray is a *shadow ray* then add $I\varepsilon/\pi$ to the associated pixel, where ε is the emissive power of the surface S .
- (ii) **Estimate direct illumination:** Set $I' \leftarrow I\alpha_q/n$, where α_q is the diffuse reflectivity at q . Issue shadow rays $(q, \omega_i, I'A_i \cos \theta_i/r_i^2)$ for $i = 1, 2, \dots, n$, where $\omega_1, \omega_2, \dots, \omega_n$ are directions pointing from q toward n stratified points over the surface of the light source, and A_i is the surface area of stratum i .
- (iii) **Estimate indirect illumination:** If the maximum path length has not been exceeded, set $I'' \leftarrow I\alpha_q/m$ and issue indirect illumination rays (q, ω_j, I'') for $j = 1, 2, \dots, m$, where $\omega_1, \omega_2, \dots, \omega_m$ are directions stratified over the outgoing hemisphere above the point q .
- (iv) **Estimate specular component:** If the surface has a specular component and the maximum path length has not been exceeded, issue the ray $(q, \omega', I\beta_q)$, where β_q is the specular reflectivity of surface S , and ω' is the direction of mirror reflection.

The values of n and m may vary with the depth of the path, the type of surface, or other factors. If $n = 1$ and $m = 1$, then the algorithm corresponds to the path tracing approach introduced by Kajiya [16].

3 A message driven computation with diffusive load balancing

Whenever a problem can be solved by a modest number of computers it is usually convenient to employ a symmetric multiprocessor (SMP) or other computer system with shared memory. Programming these systems is relatively straightforward because processes can share data and synchronize easily. Programming is more difficult on computer systems with physically distributed memory because communication and synchronization must be managed explicitly. As a matter of architectural necessity the largest parallel computer systems have these properties as do all scalable commodity network configurations [1,3,6,21].

In order to execute a parallel program efficiently it is necessary to maximize concurrency in the program and to distribute the computation evenly among all of the computers. Concurrency may be limited by explicit aspects of the algorithm being employed, for example by explicit requirements for synchro-

nization. Concurrency can also be limited by artifacts of the implementation, for example as a result of using synchronous communication.

Concurrent pipelines are a useful model for achieving efficient parallelism in iterative computations. A pipeline is constructed by breaking a monolithic iteration into a series of stages. If sufficient storage is available to hold the result of each stage then available concurrency, and therefore computational throughput, can be increased considerably, limited only by the granularity of the stages and the availability of computers to evaluate them. Individual stages can be parallelized to arbitrary degrees. Pipelines avoid synchronization between stages and as a result eliminate the need for synchronous communication.

Message driven computation is a term that is sometimes used to describe a method for implementing concurrent pipelines on parallel computer systems with distributed memory [3,4,10]. When a computer executes one stage of a pipeline it stores along with the result of that stage the identity of the next stage which is to process that result. This allows the next stage to be executed by any computer in the system. The result can be transferred to another computer deterministically or as a consequence of a dynamic load balancing operation. Such a strategy makes efficient use of a parallel computer system because the time spent in communication between pipeline stages can be overlapped with the time spent executing those stages. This strategy is effective for hiding a wide range of communication latencies and therefore can be effective on hardware platforms with widely varying communications characteristics. At one extreme this includes platforms with shared memory in which communication latency is virtually zero. At an opposite extreme this includes low cost clustered systems connected by high latency Ethernet.

We have implemented a Monte Carlo path tracing iteration as a message driven concurrent pipeline. The stages of this pipeline correspond primarily to casting bundles of rays. A typical bundle size is 16 or 32 rays which gives a good balance between the amount of computation required to execute one pipeline stage (typically over 100,000 instructions) and the cost of scheduling that stage and managing the data (which can be as little as a few hundred instructions, not including any time that might be required for communication). The implementation does not have an explicit thread of control. Instead it consists of an event loop and a series of event handlers each of which corresponds to a single pipeline stage. When messages are transferred between computers they are routed to appropriate handlers according to their message type. An event handler uses the data in the message as input to a pipeline stage, and typically produces a new message for the result of that stage, marked with an identification of the next stage to execute. For the sake of efficiency a message which corresponds to a ray bundle is processed whenever possible on the same computer as its predecessor. This may not be possible, either because

of dynamic load balancing or because a geometric model has been partitioned among a set of computers.

In a full path tracing implementation the pipeline stages do more than cast rays. Our implementation contains stages to perform global synchronization, control startup and termination, build the geometric model, define viewpoints, communicate messages to the user, handle exceptions, monitor performance, and perform a variety of mundane housekeeping chores. Taken together these other stages represent only a small fraction of the overall time spent in the computation.

In order to approach the ideal of P times speedup using P computers it is essential to have a scalable load balancing mechanism. For this reason we have employed a diffusion algorithm for dynamic load balancing [9,13,14]. Diffusive load balancing algorithms have the desirable property of converging in a fixed elapsed time that is independent of P [13]. In a later section of this paper we will propose applying a related technique [14] to the problem of partitioning the geometric model on the computer system in order to achieve similar scalable convergence in this phase of the computation.

Instead of load balancing throughout the computation we have implemented a receiver initiated strategy which becomes active only when a computer has processed all of its messages and has no more pending work. At such a time a computer polls all its neighbors in order to solicit work and requests a specific amount of work as determined by a diffusion calculation [14]. Neighbors respond by supplying the requested amount of work. This strategy produces an overhead from load balancing which is proportional to the number of computers but which does not depend on the length of time spent in computation. In order to satisfy the requirements of the diffusion calculation it is often necessary to break a large ray bundle into a set of smaller bundles. Rays are chosen as close to termination as possible (i.e., the farthest along the path) since these will spawn the smallest number of subsequent rays and thus conform most closely to the predictions of the diffusion calculation.

4 Measured performance with replicated data

Despite the variation that can occur in network and computer performance characteristics this implementation should in principle produce scalable performance on most parallel computer systems currently available or available in the near future. The implementation has been tested on a variety of uniprocessors, an Ethernet cluster of Sparcstations, a “Beowulf” cluster computer [21], a Hewlett-Packard/Convex Exemplar SPP-2000 system with global shared memory, and IBM SP1 and SP2 massively parallel computers. We expect it

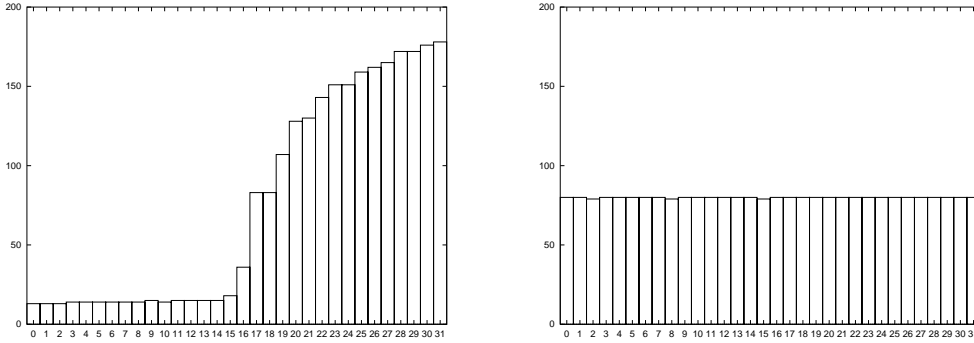


Fig. 2. The result of dynamic load balancing on 32 computers. Left, completion times for the stained glass image with no load balancing. Completion times for individual computers varied between 13 and 178 seconds. Right, completion times for the same image with dynamic load balancing by a diffusion algorithm. Completion times varied by no more than one second.

p	Office			Glass			Conference		
	t_p	s_p	e_p	t_p	s_p	e_p	t_p	s_p	e_p
1	61:18	1	1	69:00	1	1	375:30	1	1
2	29:11	2.10	1.05	39:53	1.73	0.87	180:37	2.08	1.04
9	3:50	15.99	1.78	5:04	13.62	1.51	22:44	16.52	1.84
17	1:58	31.17	1.83	2:33	27.06	1.59	11:25	32.89	1.93
33	1:02	59.32	1.80	1:20	51.75	1.57	5:53	63.82	1.93
65	0:41	89.71	1.38	0:40	103.50	1.59	3:02	123.79	1.90
129	0:33	111.46	0.86				1:38	229.90	1.78
257	0:35	105.09	0.41				2:23	157.55	0.61

Fig. 3. Times to render an NTSC draft resolution image for each of three models. The apparent superlinear scaling is an artifact of the way processes are mapped onto computers. For each set of p computers t_p reflects the elapsed time to render the image, s_p is the speedup attained using p computers, and e_p is the scaling efficiency s_p/p . Measurements are on IBM SP2 “thin nodes”, 200 MFlops peak per node.

to run effectively on systems with shared memory where the message passing primitives are replaced by operations that copy data or pointers to data in memory, and also on cluster of such machines.

Figures (3,4,5) demonstrate measured performance on NTSC resolution draft quality images for three models. The office model consists of 3,256 geometric primitives with 3 light sources. The stained glass model contains 83 primitives and 1 light. The conference room model (two images) consists of 37,220

p	Office			Glass			Conference		
	k_p	s_p	e_p	k_p	s_p	e_p	k_p	s_p	e_p
1	29:11	1	1	39:53	1	1	180:37	1	1
8	3:50	7.61	0.95	5:04	7.87	0.98	22:44	7.95	0.99
16	1:58	14.84	0.93	2:33	15.64	0.98	11:25	15.82	0.99
32	1:02	28.24	0.88	1:20	29.91	0.93	5:53	30.70	0.96
64	0:41	42.71	0.67	0:40	59.83	0.93	3:02	59.54	0.93
128	0:33	53.06	0.41				1:38	110.58	0.86
256	0:33	53.06	0.21				2:23	75.78	0.30

Fig. 4. *Times to render the same images, rendering component only, IBM SP2. Each k_p corresponds to t_{p-1} of the previous table. These times reflect the scaling of the computational kernel. One additional computer was used to support the user interface and other nonscaled overheads.*

p	Office			Glass			Conference		
	k_p	s_p	e_p	k_p	s_p	e_p	k_p	s_p	e_p
1	26:15	1	1	34:19	1	1	162:21	1	1
8	3:26	7.65	0.96	4:21	7.89	0.99	20:34	7.89	0.99
15	1:51	14.19	0.95	2:23	14.40	0.96	11:02	14.71	0.98

Fig. 5. *Times to render the same images, rendering component only, on a “Beowulf” cluster computer. Peak performance is 200 MFlops per node.*

primitives and 24 lights.

Figure 3 presents the elapsed time, speedup, and scaling efficiencies for the models on 1 to 256 “thin nodes” of an IBM SP2. The superlinear scaling (for example, speedup of 59 times on 33 computers) is an artifact of the way the processes are mapped onto the computers. The base time t_1 is measured when all tasks are processed on a single computer. In this case the single computer is performing all of the rendering tasks as well as user interface tasks and other overheads. The other times $t_2 \dots t_{257}$ are measured when these overheads are confined to a single computer and all rendering tasks are shared by all other computers.

Figure 4 presents more meaningful data showing the scaling of the rendering kernel by itself. Note that case k_1 for this table corresponds to case t_2 for the previous table. Figure 5 shows measurements for the same problems running on a Beowulf cluster computer constructed by staff at Caltech’s Center for

Advanced Computing Research. The Beowulf is a cluster of 16 Intel processors coupled by a 100 Mbps crossbar. It is interesting to see that in each test this inexpensive commodity based system slightly outperformed the MPP system.

These results demonstrate consistent scaling efficiencies in the high 90 percents at moderate numbers of computers. At larger numbers of computers the scaling efficiency remains high until the amount of computation per computer becomes as small as the fixed overheads of termination detection and barrier synchronization. Images with larger computational requirements scale to larger numbers of computers before they reach these fixed overheads. This is visible in the data for the conference room which continues to speed up beyond 128 computers. The slight increase in elapsed time for the conference room on 256 computers is probably an anomaly. We have not had an opportunity to repeat this run in order to confirm or disprove this.

On 256 processors the overheads of termination detection and synchronization have been observed to be as high as 30 seconds in the present implementation making scaling above this point of negligible benefit for the office and glass images. A more efficient implementation of these mechanisms should make it possible to achieve a factor of several times additional speedup. This would make it possible to render models like the office at NTSC draft resolution in a few seconds of elapsed time.

5 Extensions to render partitioned models

It is sometimes impossible to replicate a large model on every computer. In these cases models must be partitioned among the computers. The problem of partitioning is itself challenging (and NP complete) and has consequences for the other phases of the computation. In this section we describe our plans for and initial experiments in addressing these issues. We identify the problems to be solved in mapping and partitioning models among computers. We propose strategies to perform this partitioning, one of which employs a diffusion algorithm related to the algorithm we used for load balancing and has similar scaling properties [14]. We describe the impact that this partitioning has upon the other parts of the rendering calculation. An interesting (and unexpected) discovery is that when the necessary modifications have been made to support rendering partitioned models it becomes possible to render models out-of-core, in effect providing user-level control of an extended virtual memory. This approach can be exploited on parallel computers or uniprocessors with equal benefits.

5.1 Mapping and partitioning

The problem of mapping distributed structures onto parallel computers has been studied for many years. This problem was originally posed in terms of mapping a graph of communicating processes onto a network of computers. More recent papers have addressed the analogous problems of mapping data structures [7,14,17–19,23].

Simple and efficient methods exist to compute a static solution to this problem on a network with regular structure. These methods work by recursively subdividing a data structure into locally contiguous pieces, and assigning the pieces to contiguous regions of the network [17,23]. For the purposes of rendering these subdivisions can be obtained cheaply by sorting geometric objects according to their relative positions in space.

When a network has an arbitrary structure, such as might occur with a network of workstations, or when a problem changes dynamically, such as occurs in virtual reality environments and in animation, these static subdivision methods are inadequate. In recent years a number of papers have begun to explore concurrent *diffusion algorithms* for dynamic problems of load balancing and mapping on parallel computers.

Diffusion algorithms can address a number of quadratic minimization problems, including mapping data structures onto parallel computers. The mapping problem is derived from the (NP complete) *graph embedding* problem [20]. In the graph embedding problem the vertices of a relatively large graph, termed the *guest*, are mapped into the vertices of a smaller graph, termed the *host*. The objective of the problem is to find a map which places related guest vertices next to each other in the host, and which maps a similar number of guest vertices to each host vertex.

In the problem of mapping a geometric model onto a set of computers the model corresponds to the guest and the computer network corresponds to the host. A diffusive mapping algorithm can be understood informally as a method to diffuse the geometric elements in a space occupied by a set of computers while satisfying the objective of the graph embedding problem. Specifically, the diffusion algorithm constructs a Euclidean space and assigns regions of the space to computers in the network. It assigns a position in that space to each geometric object. The algorithm works by a simple iteration which moves objects incrementally according to a diffusion process. The algorithm converges to a condition in which related objects have been moved near each other and the distribution of objects is balanced among the computers.

Diffusion algorithms have a number of properties which make them attractive in distributed computing environments. The correctness of these algorithms,

their utility and efficiency in parallel computing, as well as a complete bibliography of related work, is given in [14]. The algorithms converge in a fixed amount of elapsed time which is independent of the number of computers on which they execute. This makes them very attractive for massively parallel implementation. They require no synchronization and communicate only among directly connected computers. They tolerate faults and communication delays without failing. For the purposes of dynamic animation or virtual reality they can reconverge a modified problem rapidly without recomputing it *ab initio*.

A diffusive mapping algorithm can adapt its solution to recognize the availability of different amounts of available memory on different computers. It can also recognize that different geometric objects consume different amounts of storage, and that it is more important for some pairs of objects to be near each other than for others. In order to implement a diffusion algorithm for mapping it is necessary to first construct a graph of nearest neighbor relations among the objects of a geometric model. While this may add substantially to the amount of storage required for the model it only needs to be computed once and thus does not impose a computational burden upon the rendering process.

5.2 *Rendering a partitioned model*

Rendering a partitioned model introduces new considerations for ray casting and load balancing. When a model was replicated, if a ray were cast that did not hit an object, it was safe to assume the ray exited the geometric environment. This is no longer the case when the model is partitioned. A ray that fails to intersect an object must be transferred to a computer containing an appropriate adjacent segment of geometry where it can be cast again.

This may be a straightforward task if each computer is assigned to manage the geometry for a single region of space. Unfortunately in an implementation with load balancing this is not likely to be the case. When uneven workloads develop in a partitioned model it becomes necessary to reassign ownership of partitions. Unless partitions are replicated it is impossible to exchange work according to the requirements of a dynamic load balancing algorithm.

A strategy must be designed to allow computers to exchange partitions as well as rays, and to manage rays which are cast between partitions. We have designed such a strategy and implemented and tested its fundamental components. The strategy involves transferring geometric data associated with a partition, i.e., the geometric objects and associated bounding hierarchy, between computers and between computer memory and disk. The strategy works as follows. If computer *A* develops an empty work queue and subsequently re-

quests work from computer B , B first offers to supply A with a copy of its partition. A compares the identity of this partition to the identity of its current partition. If these are different then A caches its partition data on disk and accepts a copy of B 's partition which is transferred as a (very large) message. After that A accepts work from B in the same way it would if the model were replicated among all of the computers.

During the time that A has a partition k cached on disk A is still the destination for rays which exit other partitions bound for partition k . A must enqueue these rays and ensure that the entire queue for partition k eventually gets evaluated.

We have implemented and tested the fundamental operations of exchanging partitions between computers and caching them to disk. The time required to cache an old partition and acquire a new one is on the order of a few seconds for partitions that are roughly 30 megabytes in size. This suggests that while these transfers should not occur frequently they are affordable in the context of a large rendering task. It also suggests that out-of-core computations are feasible, in which portions of a geometry are transferred at staged intervals between disk and memory, in effect providing an extremely large virtual memory under the control of the application. We are interested in exploring these issues as well as others as we continue to develop this software.

6 Acknowledgments

We appreciate the careful proofreading given to the section on Monte Carlo path tracing by James Patton of the Center for Advanced Computing Research. We have benefited from access to models provided by Greg Ward of the Lawrence Berkeley Laboratories. The conference room model was constructed by Greg Ward and Anat Grynberg.

The research reported here has made extensive use of computer facilities provided by the Cornell Theory Center, the Argonne High Performance Computing Research Facility, Caltech's Center for Advanced Computing Research, and the Hewlett-Packard Company.

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 NSF Science and Technology Center for Research on Parallel Computation (grant 292-3-51393 under NSF CCR-8809615), and the Army

Research Office, grant DAAH04-96-0077.

References

- [1] Aboulenen, N., Gjessing, S., Goodman, J. & Woest, P. Hardware support for synchronization in the Scalable Coherent Interface (SCI). *Proc. 8th IEEE Int. Par. Proc. Symp.* (1994).
- [2] Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conf. Proc.* (1967) 483–485.
- [3] Anderson, T. E., Culler, D. E. & Patterson, D. A. A case for NOW (Networks of Workstations). *IEEE Micro* **15** (1995) 54-64.
- [4] Athas, W. C. & Seitz, C. L. Multicomputers: message passing concurrent computers. *IEEE Comp.* **21** (1988) 9-24.
- [5] Baskett, F. & Hennessy, J. L. Microprocessors: from desktops to supercomputers. *Science* **261** (1993) 864–871.
- [6] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N. & Su, W. Myrinet: a gigabit per second local area network. *IEEE Micro* **15** (1995) 29-36.
- [7] Bokhari, S. Assignment problems in parallel and distributed computing. Boston : Kluwer (1987).
- [8] Chang, C.-C., Czajkowski, G. & von Eicken, T. Design and performance of active messages on the IBM SP2. Cornell Computer Science Technical Report 96-1572 (1996).
- [9] Cybenko, G. Dynamic load balancing for distributed memory multiprocessors. *J. Par. Dist. Comp.* **7** (1989) 279-301.
- [10] von Eicken, T., Culler, D. E., Goldstein, S. C. & Schauser, K. E. Active messages: a mechanism for integrated communication and computation. *Proc. 19th Int. Symp. Comp. Arch.*, Gold Coast, Australia (1992) 256-266.
- [11] von Eicken, T., Basu, A., Buch, V. & Voegls, W. U-Net: a user level network interface for parallel and distributed computing. *Proc. 15th ACM Symp. Op. Sys. Princ.* (1995) 1-14.
- [12] Gustafson, J. L. Reevaluating Amdahl's Law. *Comm. ACM* **31** (1987), 532–533.
- [13] Heirich, A. & Taylor, S. A parabolic load balancing method. *Proc. 24th Intern. Conf. Par. Proc.*, **III**, pp. 192–202. New York : CRC Press (1995).
- [14] Heirich, A. A scalable diffusion algorithm for dynamic mapping and load balancing on networks of arbitrary topology. *Int. J. Found. Comp. Sci.* (1997), to appear.

- [15] Hillis, W. D. & Bhogosian, B. M. Parallel scientific computation. *Science* **261** (1993) 856–863.
- [16] Kajiya, J. T. The rendering equation. *Comp. Graph.* **20** (1986).
- [17] Karypis, G. & Kumar, V. Multilevel graph partitioning schemes. *Proc. 24th Intern. Conf. Par. Proc.*, **III**, pp. 113–122. New York : CRC Press (1995).
- [18] Kung, H. T. & Stevenson, D. A software technique for reducing the routing time on a parallel computer with a fixed interconnection network. In *High speed computer and algorithm organization*, eds. Kuck, Lawrie & Sameh (1977).
- [19] Martin, A. A distributed implementation method for parallel programming. *Inf. Proc.* **80** (1980) 309-314.
- [20] Rosenberg, A. Issues in the study of graph embedding. In *Graph theoretic concepts in computer science*, Lecture Notes in Computer Science **100**, New York : Springer (1981) 150-176.
- [21] Sterling, T. L. The scientific workstation of the future may be a pile-of-pcs. *Comm. ACM* **39** (1996) 11-12.
- [22] Whitted, T. An improved illumination model for shaded display. *Comm. ACM* **23** (1980), 343-349.
- [23] Williams, R. D. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Pract. Exp.* **3** (1991) 457-481.