CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 4(2), 119–141 (APRIL 1992)

A Performance Analysis Exemplar: Parallel Ray Tracing

D. W. JENSEN AND D. A. REED
Department of Computer Science
University of Illinois
Urbana, Illinois 61801, USA

SUMMARY

Among the many techniques in computer graphics, ray tracing is prized because it can render realistic images, albeit at great computational expense. Ray tracing's large computation requirements, coupled with its inherently parallel nature, make ray tracing algorithms attractive candidates for parallel implementation. In this paper we illustrate the utility and the importance of a suite of performance analysis tools when exploring the performance of several approaches to ray tracing on a distributed memory parallel system. These ray tracing algorithm variations introduce parallelism based on both ray and object partitions.

Traditional timing analysis can quantify the performance *effects* of different algorithm choices (i.e. when an algorithm is best matched to a given problem), but it cannot identify the *causes* of these performance differences. We argue, by example, that a performance instrumentation system is needed that can trace the execution of distributed memory parallel programs by recording the occurrence of parallel program events. The resulting event traces can be used to compile summary statistics that provide a global view of program performance. In addition, visualization tools permit the graphic display of event traces. Visual presentation of performance data is particularly useful, indeed, necessary for large-scale, parallel computations; assimilating the enormous volume of performance data mandates visual display.

1. INTRODUCTION

Among the plethora of techniques for generating computer graphics, ray tracing is prized because it can render extremely realistic images, albeit often at great computational expense. Intuitively, ray tracing projects a series of rays from a viewing perspective, the viewpoint, through a defined screen of pixels (see Figure 1). Once in the scene, each ray interacts with the scene's component objects to determine the color of its associated pixel.[1] The computational complexity is a consequence of the larger number of rays needed to generate an image. For example, 256K rays must be generated to produce a modest 512 by 512 pixel image. Ray tracing's large computation requirements, coupled with its inherently parallel nature, make ray tracing algorithms attractive candidates for parallel implementation.

In this paper we explore the performance of several approaches to ray tracing on a distributed memory parallel system. However, the large number of parallel ray tracing algorithm variants make selection of a single algorithm problematic. For example, ray tracing algorithms for distributed memory parallel systems can introduce parallelism

[1] In essence, ray tracing is the inverse of nature, where rays exit the scene to intersect with the observer at the viewpoint.

1040-3108/92/020119-23$11.50

Received 20 October 1989
Revised 31 May 1991

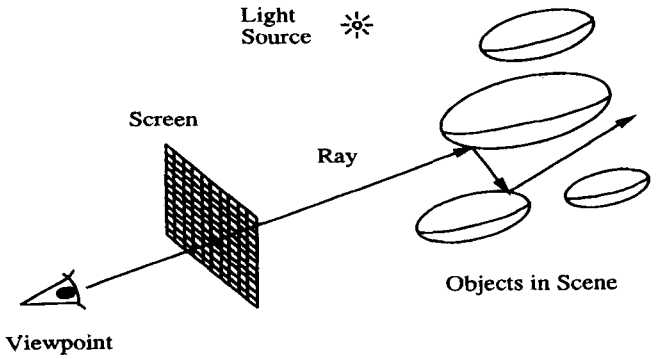

Figure 1. Ray tracing concept

based on both partitions of the rays and the objects in the scene. Ray partitioning schemes distribute image generation across multiple processors but require redundant storage of the scene data. In contrast, object partitioning distributes the scene data, allowing the processing of more complex scenes, but with increased interprocessor communication. The choice of an appropriate technique depends on the balance of parallel computation and interprocessor communication and the complexity of the scene.

Traditional timing analysis can quantify the performance *effects* of different algorithm choices (i.e. when an algorithm is best matched to a given problem), but it cannot identify the *causes* of these performance differences. The performance of a parallel computing system is the complex product of its component interactions. A complete performance characterization requires both *static* and *dynamic* analysis. Static or average behavior analysis (e.g. timing measurements), although useful for comparison of aggregate performance, may mask transients that dramatically alter system performance. Simply put, the performance of parallel system components depends on the frequency and types of their interactions; these interactions often cannot be predicted, but they can be measured.

We argue, by example, that a performance instrumentation system is needed that can trace the execution of distributed memory parallel programs by recording the occurrence of parallel program events. The resulting event traces can be used to compile summary statistics that provide a global view of program performance. In addition, visualization tools permit the graphic display of event traces. Visual presentation of performance data is particularly useful, indeed, necessary for large-scale, parallel computations; assimilating the enormous volume of performance data mandates visual display.

In Section 2, we describe the experimental environment, the Intel iPSC/2 hypercube, and the implementation constraints it imposes. We follow in Section 3 with a description of our performance instrumentation environment, including a description of the visualization tools used in our exploration of parallel ray tracing algorithms. In Section 4 we examine approaches to ray and object partitioning and their prospective balance of parallelism and interprocessor communication. Based on these algorithms, we describe in Section 5 a set of performance experiments intended to compare the performance of different partitioning schemes. Finally Section 6 summarizes our results and suggests avenues for further research.

2. INTEL IPSC/2 DESCRIPTION

Before describing our ray tracing experiments, we briefly digress to review the architecture and system software of the Intel iPSC/2, the distributed memory parallel system used in our experiments. As with any parallel system, both the iPSC/2 architecture and system software dictate the domain of feasible application algorithms (e.g. the balance of computation and communication). Thus, parallel algorithm variations and performance analysis tools must be understood in the context of the intended architecture—in this case, the Intel iPSC/2, a second generation, distributed memory parallel system.

The Intel iPSC/2 and the more recent iPSC/860 consist of processing nodes connected in a hypercube topology[1,2]. Each iPSC/2 processing node contains a 16 MHz Intel 80386 microprocessor, an 80387 floating point co-processor, up to 16 megabytes of memory, a 64K byte cache and an autonomous message routing controller called the Direct Connect Module[2]. The routing controller supports fixed path, circuit-switched communication between nodes, eliminating most of the store-and-forward latency that existed in earlier distributed memory parallel systems. The software development interface for the iPSC/2 is a standard Unix system that transmits executable programs to the nodes, accepts results from the nodes, and can, if desired, participate in the computation.

Although distributed memory avoids the problems of memory contention inherent in shared memory parallel systems, the degree of internode data sharing is limited by the latency and bandwidth of the communication network. On the Intel iPSC/2, the latency to initiate a message transmission is approximately 270 microseconds[2] with a bandwidth of 2.8 megabytes/second[3]. Large messages require the source and destination nodes to negotiate a buffer allocation strategy prior to message transmission and have the same transmission rate but higher latency. Such a system favors a small number of large messages over a large number of small messages. Thus, the performance of different algorithm variations is sensitive to the distribution of data, computation and volume of interprocessor communication. To quantify and understand these effects, performance analysis tools are needed; this is the subject of the following Section.

3. PERFORMANCE INSTRUMENTATION

Any new computer system organization raises many questions about hardware, system software, algorithms and programming; distributed memory parallel systems have proved no different. Although simple performance measures (e.g. communication latency) are readily obtained[3], the algorithm design experience of many researchers quickly revealed the difficulties in inferring system behavior from aggregate performance measures. This stimulated our development of an integrated instrumentation environment for the analysis of distributed memory parallel computations[4]. Below, we briefly review the instrumentation software and visualization components of this environment and describe its use in our evaluation of parallel ray tracing algorithms; for complete details on the instrumentation and visualization environment, see References 4 and 5.

[2] Almost all of this message transmission latency is operating system software overhead. The Intel iPSC/860 uses the same communication hardware with the faster i860 microprocessor—its message passing latency is approximately 90 microseconds.

3.1. Software instrumentation

Most users search for performance problems only when necessity dictates. Repeated experience has shown that if the user must manually insert extensive source code instrumentation to capture application performance data, there is little chance that the instrumentation system will be used. Indeed, users will embrace technically inferior tools if they are sufficiently easy to use. Thus, our software performance instrumentation includes automatic event tracing at both the program and operating system levels.

At the program level, the performance analyst can direct a modified version of the GNU C compiler[6] to automatically generate code to create a time-stamped log of procedure entries and exits; this requires no modification to the application source code. The great appeal of automatic, compile time instrumentation is its flexibility and ease of use. If other instrumentation is needed, the user is free to manually insert additional source code instrumentation that marks particular events or measures the execution time of specific code fragments.

Like the automatic instrumentation, manual instrumentation generates events that are passed to NX/2, the Intel iPSC/2 node operating system, where they are merged with three classes of operating system events: message, process and system call. These operating system events are captured by an instrumentation of the NX/2 operating system source code[5,7]. For message transmissions, the instrumentation records the parameters of the application program's call to the message passing library and the time that the node hardware began and completed physical transmission of the message. For context switches, the identity of the activated process is recorded, and for system calls, the instrumentation saves the identity of the system call and its parameters. Because the NX/2 provides only minimal services, these three event classes completely characterize system behavior.

3.2. Performance data visualization

Historically, performance measurement techniques have included profiling (i.e. where), sampling (i.e. when) and event tracing. Event tracing subsumes both profiling and sampling; each trace event includes information on *where*, *what* and *when*. Thus, tracing is the most general instrumentation method, and is, we believe, the best means to capture and analyse the time varying behavior of complex parallel systems.

Unfortunately, event tracing can quickly generate megabytes of trace data, and only rarely does one want or need to examine individual events. Given an event trace, one can construct a profile of execution time, but such a statistical summary discards much of the detail inherent in a trace. Fortunately, a graphical event timeline provides one intuitive mechanism for display of large event traces. This allows one to visually compare the time varying state of multiple nodes and to focus on particular time intervals for more detailed analysis.

Our interactive timeline display program[5], based on the X window system[8], simultaneously shows the time varying state of each hypercube node. Using the mouse, a user can interactively control the fraction of the program's execution that is displayed. At the highest level, the timeline provides a qualitative representation of the entire program's activity. By repeatedly clicking the mouse to select a smaller area, the user can observe successively smaller portions of the timeline in greater detail. In the limit, individual trace events are visible.

The timeline display algorithm divides the trace into a series of fixed size windows of time and identifies the most common activity in each window (e.g. idle time, user computation, communication, or operating system execution). On monochrome displays, each activity is represented by a particular pattern—application execution is gray or black, idle time is represented by a row of dots along the bottom of each node's timeline, message passing activities are diagonally striped, and execution of operating system code is represented by a solid line at the top of the node timeline.

As an example, Figure 2 shows the execution behavior of one parallel ray tracing algorithm on eight processors of the Intel iPSC/2. As the black regions suggest, all nodes initially are actively tracing rays. However, the timeline shows that the second half of the computation is severely imbalanced. To understand the reasons for this load imbalance, one can use the timeline to examine a much smaller interval of time, as shown in Figure 3. In the Figure, all other nodes are waiting for receipt of a message from node zero. After receipt of this message, the nodes compute for a small interval and are again quiescent. This pattern persists throughout the remainder of the computation.[3]

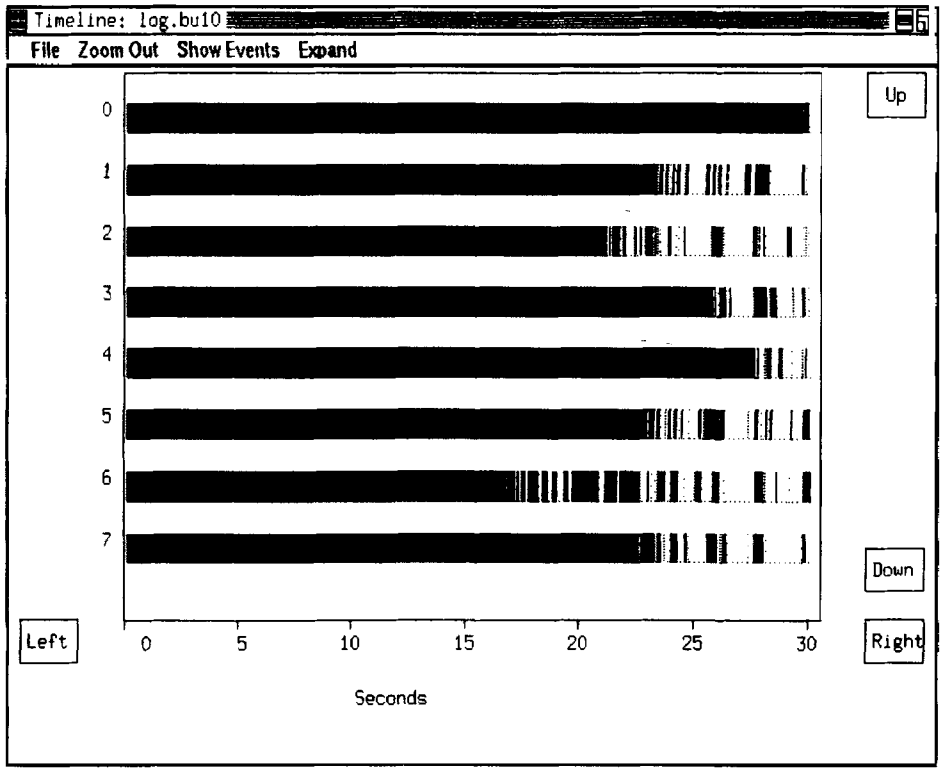

Figure 2. Ray tracing timeline

[3] For an explanation of the underlying causes of this behavior, see Section 5.3 and Figure 17.

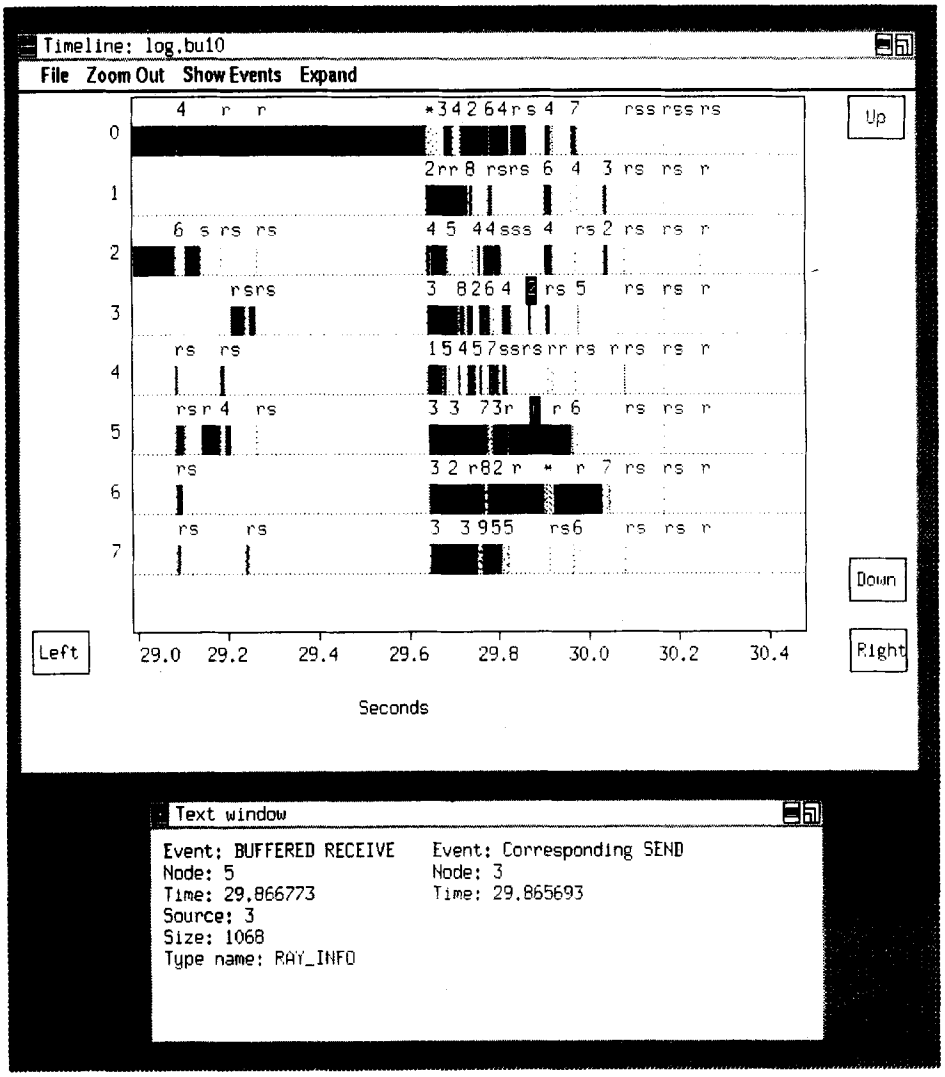

Figure 3. Ray tracing timeline (detail)

In Figure 3, the letters and numerals above each node's timeline represent the time of occurrence for selected trace events. The letters `s` and `r` represent message sends and receives, respectively. Numerals are shown when the screen resolution precludes display of all events; their magnitudes indicate the number of events present near that point in time. Clicking on any letter or numeral displays more information about the corresponding event or events, including the time it occurred and any event-specific information. For classes of events that occur in pairs (e.g. message transmission and receipt), the system shows the matched pair.

Although timelines can provide insight into the time-varying computation state, they cannot directly represent such performance measures as processor utilization or the

distribution of messages by destination or data volume. To display dynamic statistics, we developed a visualization environment[4] that supports a variety of display *views*, including dials, bar charts, LEDs, Kiviat diagrams, matrix views, *X-Y* plots, contour plots, strip charts, and a general purpose graph display.

The environment separates the calculation of dynamic statistics from graphical display, allowing the user to independently select the desired statistics and the most appropriate displays. Thus, one can create multiple views of the same statistic, display multiple statistics using several instances of the same view, or choose a unique view for each statistic. Figure 4 shows some of the currently operational display views using trace data taken from one variation of our parallel ray tracing code.

The displays in the upper left of Figure 4 show communication traffic among hypercube nodes. In the far left, the message count and volume bar charts show the relative fraction of messages and the relative number of message bytes (i.e. message volume) sent by

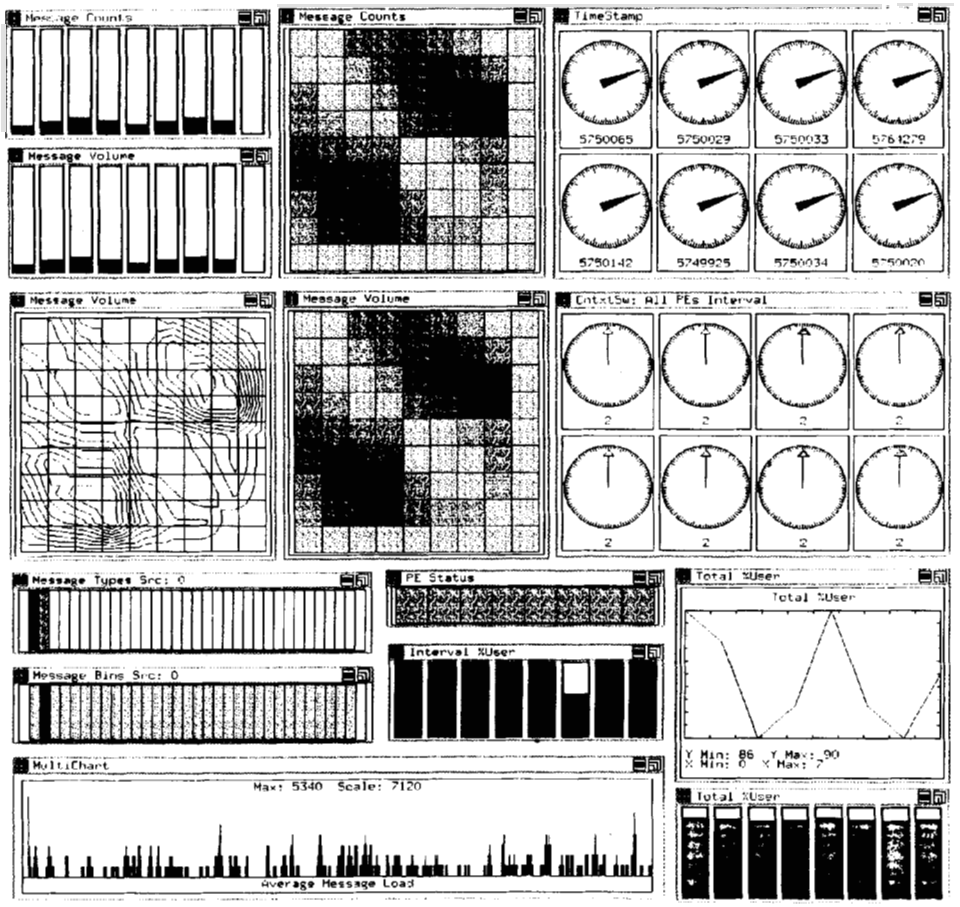

Figure 4. Performance visualization environment

each node. The two-dimensional message count and volume views reflect the spatial distribution of messages.[4] They show the total number of messages and number of message bytes, respectively, sent between each source (row) and destination (column) node; the ninth row and column denote communication traffic to and from the hypercube host.

The similarity of the message count and message volume matrices implies that the variance of message sizes is small—the pattern of communication and the number of bytes sent to each destination are similar. The message bin and message type views confirm this hypothesis; these views show histograms of message sizes and message types, respectively. Most messages are small, and there are only a few message types. Finally, the average message load display at the bottom of Figure 4 shows the sliding window average of the aggregate network communication traffic. The irregularity reflects varying numbers of ray reflections (and message transmissions) in different portions of the scene.

The processor status (PE) view in the center of Figure 4 shows the current state of each node (idle, user computation, system state or message transmission). The bar charts and *X-Y* plot in the lower left corner show both the processor utilization since the beginning of the computation and during a sliding window interval. Similarly, the context switches view shows the number of state changes by each node in a sliding window of time.

Operating system instrumentation, statistical analysis of event traces, and event trace visualization all provide the means to analyse and understand algorithm variations. The remainder of this paper illustrates the insights possible using data derived from detailed performance instrumentation of parallel ray tracing algorithms. We know of no other performance instrumentation environment capable of capturing such detailed performance data on distributed memory parallel systems.

4. PARALLEL RAY TRACING APPROACHES

As noted earlier, ray tracing projects a series of rays from a viewing perspective, the viewpoint, through a defined screen of pixels. Once a projected ray enters the scene, its path may or may not intersect an object. If a ray exits the scene without encountering an object, and it did not intersect a light source, the scene's background color is assigned to the ray's associated pixel. Conversely, if the ray intersects a light source but no objects, the associated pixel is assigned the color of the light source.

If the ray does intersect some object in the scene, the intersected object's surface characteristics determine if the ray is terminated and a pixel color assigned or if new rays are created. Any rays created by object intersection are traced from their creation point. Typical surface types and their effect on intersecting rays include:

- *Absorbing.* If the object's surface is matte, the surface color is returned, and the ray terminates.
- *Refracting.* If the object's surface refracts or bends the trajectory of the ray, a new ray is created and is projected along the new trajectory.
- *Reflective.* If a surface reflects light, such as a mirror, a new ray is created and is projected along the reflected trajectory.

[4] Recall that these displays are dynamic, reflecting the temporal *and* spatial pattern of communication.

Rays generated by refraction or reflection may recursively generate additional rays. All such rays eventually must terminate, either because they failed to intersect an object, they intersected a matte object, or a bound on the number of reflections or refractions was reached. The colors generated by these recursive rays are combined to determine the color of the parent ray and its associated image pixel. Thus, the computation required to generate an image depends on both the number of pixels and the characteristics of a scene's objects.

This approach to ray tracing does not render a perfect image; ray tracing models a continuous image with a uniform placement of discrete points, the pixels. By increasing the total number of pixels in the image, the pixel discretization errors decrease and resolution improves. Likewise, supersampling increases the number of rays traced per pixel and averages the results to determine the final pixel color. Clearly, the time to render the scene will increase in proportion to the number of rays. For a lucid introduction to ray tracing, its advantages, and its limitations, see Reference 9.

The structure of ray tracing algorithms provides many potential approaches to parallelization based on the distribution of rays and objects. Each such approach (i.e. ray or object partitioning) offers potential advantages and disadvantages. As we shall see, ray partitioning minimizes interprocessor communication but must replicate the scene's objects in the local memory of each processor. In contrast, object partitioning reduces memory requirements at the expense of interprocessor communication. Ideally, a unified approach, one that combines the best features of both ray and object partitioning, would maximize performance by matching the algorithm to the distributed memory architecture. The remainder of this Section addresses such algorithms, followed in Section 5 by an analysis of the algorithms using our performance tools.

4.1. Ray partitioning

Ray partitioning divides the pixels, and by implication, the rays, into distinct subsets and assigns each subset to a processor. Valid subsets range from one subset containing all rays, the sequential case, to one ray per subset, the maximally parallel case. Because each ray is independent of all other rays, parallel implementation is trivial.

In the simplest scheme, the screen is divided into fixed subsets (i.e. groups of contiguous pixel rows, columns or blocks)[10]. Because the viewpoint and number of objects define the computational complexity for ray tracing each subset of the screen, a simple partitioning based on tilings of the screen can create severe load imbalances. A variation of this technique ameliorates load imbalances by mapping pixel row i to processor $i \bmod P$, where P is the total number of processors. This modulo partition scatters rays from contiguous areas across many processors. Ideally, this equitably distributes the computation by scattering complex parts of the scene.

The static tiled and modulo partitions can be created *a priori* without knowledge of each ray's computational complexity. At best, this yields a statistical load balance. If the computation times for rays differ significantly, the variance in each processor's load can be high. Rather than partitioning the scene *a priori*, a dynamic ray partitioning assigns groups of rays to processors based on the current system state. These groups, hereafter called ray bundles, are distributed to processors on demand as they complete computation of earlier bundles. Processors assigned bundles that require less computation will process more bundles, balancing processor utilization.

In principle, optimum load balance results from bundles containing a single ray (i.e. the computation of each ray is dynamically distributed). However, the high latency of current message passing systems, including the Intel iPSC/2, constrains the minimum size of the ray bundles. The overhead of smaller bundles can offset the performance gains from the improved load balance.

4.2. Object partitioning

Although good load balance and significant speed-up can be achieved with either the static scattered or dynamic bundled ray partitioning schemes, both schemes require each processor to maintain the entire scene in its local memory. Why? Regardless of the ray partitioning scheme used, each ray and its descendants potentially can intersect any object in the scene. Ray partitioning distributes ray processing but not the objects of the scene. As noted earlier, one attractive feature of distributed memory parallel systems is the ability to expand both the computational power and memory capacity by adding nodes (i.e. processor/memory pairs). Clearly, ray partitioning schemes cannot exploit this extensibility. Object partitioning removes this limitation by distributing the scene's objects across processors. Each processor is responsible only for the portion of the scene that contains its subset of the objects. When a ray crosses an object partition boundary, a message containing the ray's description now must be sent between the processors that contain the source and destination partitions. Thus, object partitioning permits tracing of larger scenes, but it introduces communication overhead.

An implementation of object partitioning can take many forms; ours has its origins in References 10–12. As in Reference 10, we partition the scene using a tree of boxes, but these boxes need not bound all objects in the scene. The root of the tree represents the containing volume for the entire scene. That volume is partitioned into two smaller volumes such that each contains approximately half the objects. This process is repeated recursively until the number of leaves in the partition tree equals the number of hypercube nodes. In a complete partitioning, each node contains the body of the tree and one leaf; this structure is used to determine the node location of the necessary objects. At each partitioning step, some scene objects intersect the boundaries of the containing volumes; rather than complicate the geometry of the containing volumes, these objects are replicated in each of the partially containing volumes.[5]

To minimize memory use, a scene with N objects on a P processor system would place N/P objects in each node. However, the objects of a scene need not be partitioned disjointly.[6] As an example, Figure 5 shows a tree structure created to represent a teapot and a partition across four processors. Here, each processor retains 75% of the teapot's constituent objects in its local memory. Portions of the scene overlap and many objects are replicated in each processor.

As noted above, each node contains the upper structure of the tree and only its corresponding portion of teapot's objects. A set of partitions ranging from N/P to $N(P-1)/P$ objects per node is possible with this tree structure by varying the number of leaves assigned to each node. By maximizing the percentage of the scene in each node, the

[5] Because scenes typically contain only a few large objects and many smaller objects, the replicated objects are usually a small fraction of the total.

[6] This overlap is not the same as that created as an artifact of the desire to retain simple bounding geometries.

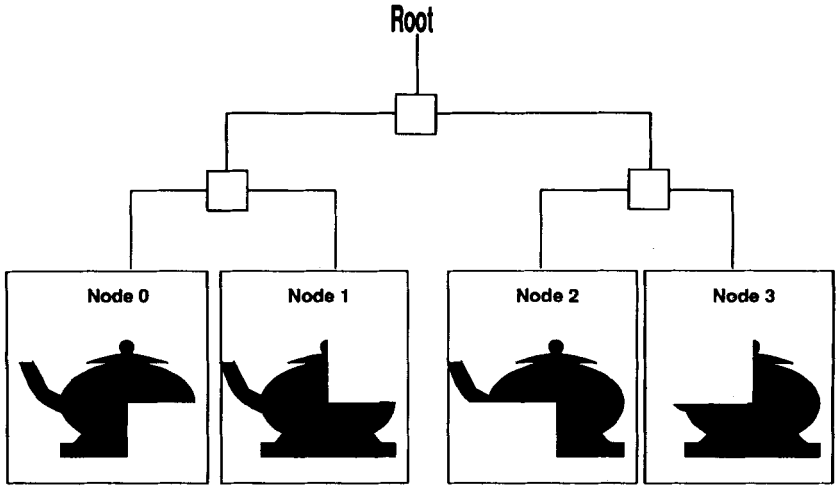

Figure 5. Partial object partitioning organization

amount of internode communication is minimized. At one extreme, a complete partition minimizes total storage requirements. At the other extreme, a complete replication of all objects reduces to ray partitioning—no interprocessor communication is needed.

Although this hierarchical decomposition permits distribution of the scene's objects across processors, it does nothing to reduce the number of calculations needed to determine intersection of rays with objects. Within each processor, a series of synthetic objects are generated that represent bounding boxes. In this scheme, boxes are first built around objects and around boxes containing objects until the last box created contains all the objects of that particular processor. If a ray does not intersect a bounding box, no check on the objects in the box is required.

5. PERFORMANCE EVALUATION

Ray and object partitioning have complementary goals, and when used together they permit parallel ray tracing of large images. However, the number of possible algorithm variations and their associated parameters make evaluation difficult (see Table 1). Object distribution increases communication overhead, and when coupled with ray partitioning may create load imbalances. Performance experiments based on execution times can confirm the efficiency of particular parameter choices, but timing data alone cannot identify the underlying reasons. Consequently, we augmented the timing data with event trace data obtained using our performance instrumentation tools[4,7]. These data are the source of explanations for the performance observed variations.

Using these performance tools, we conducted two sets of experiments. In the first set, we examined the performance of ray partitioning schemes without object partitioning. These simple experiments highlight the importance of load balance and provide a reference point when ray partitioning is combined with object partitioning. Given these data, we examined the performance of object partitioning algorithms, including the benefits of replicated objects, when combined with ray partitioning. In each experiment set

Table 1. Comparison of partitioning schemes

	Ray partitioning schemes			Object partitioning schemes
	Static tiled	Scattered	Bundles	
Implementation Difficulty	Trivial	Simple	Moderate	Difficult
Internode Communication	No	No	No	Yes
System Overhead	None	None	Moderate	Large
Load Balanced	No	Yes	Yes	Potential problems
Memory Limitations	Node	Node	Node	Full hypercube memory

we selected the most interesting test cases, based on timing data, and used the performance instrumentation tools to determine the causes of performance degradations.

As noted earlier, ray tracing's computational complexity is inextricably tied to the structure of the underlying scene. Complex scenes with wide disparities in object density and features generate more rays, via refraction and reflection, and potentially create greater load balancing problems. Thus, understanding the characteristics of the test scenes is crucial; this is the subject of the following section.

5.1. Test scenes

During our performance experiments, four test scenes were used (see Figure 6). One of the test scenes is the University of Utah teapot, containing 3743 objects. The other three test scenes, balls, rings and tree, were generated algorithmically. All scene generators could produce scenes with varying numbers of objects, and unless otherwise noted, the number of objects in each of the other test scenes is approximately equal to that in the teapot test scene.

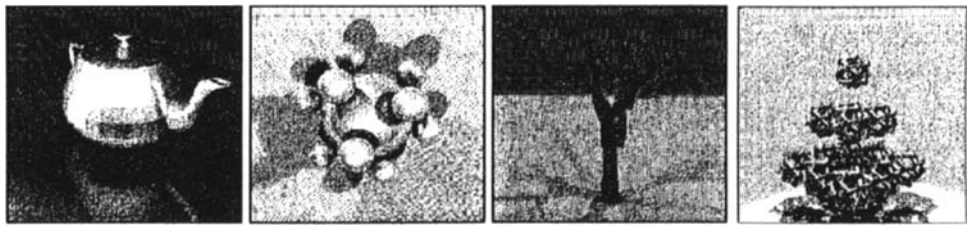

Figure 6. Ray traced test scenes

As Figure 6 suggests, each scene contains different surfaces (e.g. matte and reflective) and object distributions. These differences are manifest as variations in the spatial distribution of ray processing costs. For example, rays that intersect matte backgrounds can be computed quickly, but rays that intersect closely packed, reflective balls generate many recursive rays. Thus, two scenes with equal numbers of objects need not have equal computational complexity.

Figure 7 illustrates the spatial distribution of pixel processing costs for each of the four test scenes.[7] In the Figure, the viewing perspective for the teapot scene is the teapot side. Similarly, the rings' perspective is from above. For the balls and tree scenes, the perspective is the one shown in Figure 6. Pixel cost and object placement are strongly correlated. One can identify the computational cost of specific objects and shadows in each image. As we shall see, this variance of ray processing costs has important implications for ray and object partitioning.

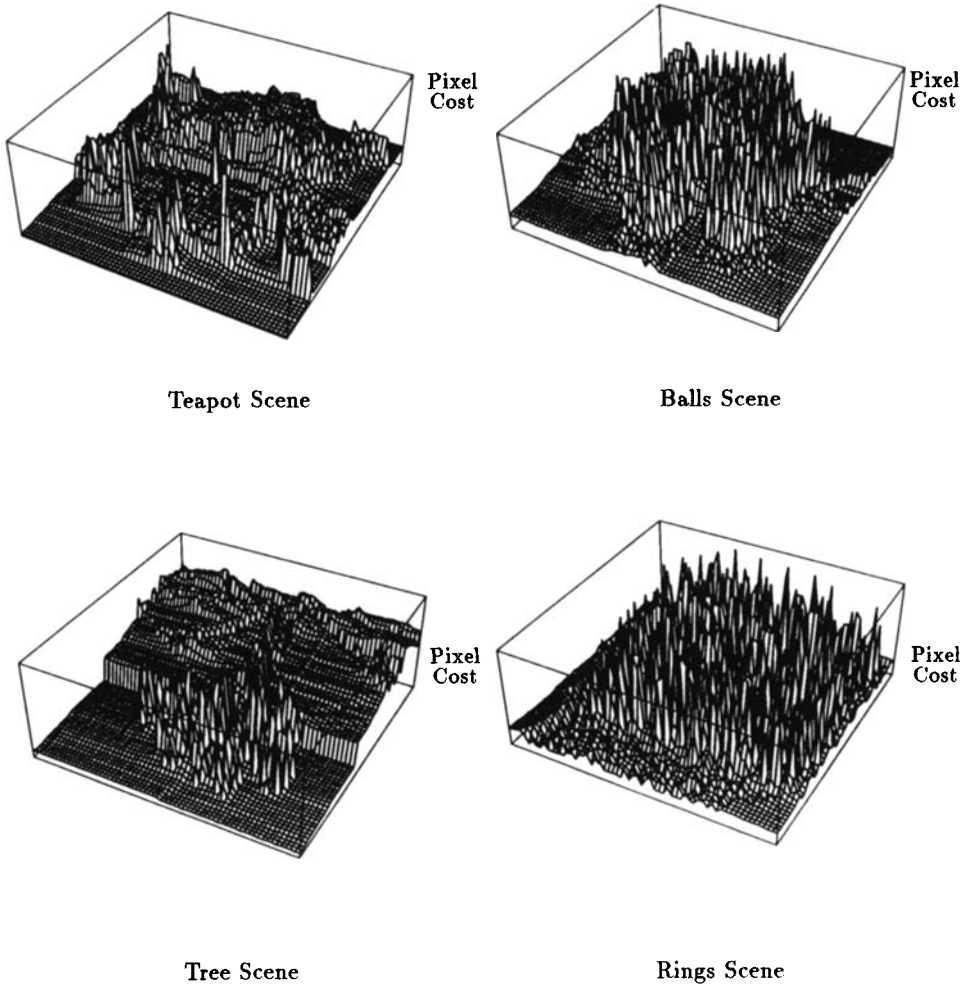

Figure 7. Ray processing costs (64 × 64 pixel images)

[7] In Figure 7, the cost for each pixel is normalized using the maximum pixel cost in each image. Thus, pixel costs should not be compared across scenes.

5.2. Ray partitioning

Our performance study began with the simplest approach to parallelization: ray partitioning with the entire scene replicated in each node. Recall that the three ray partitioning techniques proposed include static tiling, which evenly divides contiguous sections of the image and assigns each section to a node, scattered mapping, which scatters rays by assigning image row i to processor i mod P, and ray bundling, which distributes pixel groups on demand.

Figure 8 shows the execution time for the four test scenes, teapot, balls, rings and trees, using these three ray partitioning schemes. Clearly, the static strategy is inferior to both the scattered and dynamic bundle strategies. To understand the reasons for this inferiority, we instrumented the node computations. Figure 9, derived from performance event traces, shows the execution time for both tiled and scattered ray partitioning schemes. For the tiled partition, the regions of high scene complexity near the center of the image are correlated with high node computation times. In contrast, the scattered partition distributes the processing of high density regions across all processors.

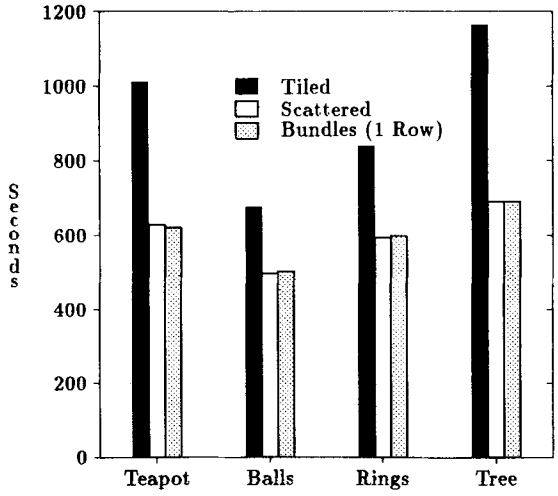

Figure 8. Execution times with ray partitioning (512 × 512 pixel image on 32 nodes)

Perhaps more surprising than the superiority of scattered mapping is the seemingly mediocre performance of the ray bundle strategy. Unlike the static strategies, ray bundles have an associated size (i.e. the number of pixels in a bundle) and a selection algorithm. The latter determines how pixels are selected from the image (e.g. as contiguous, scattered or random groups). Together, the bundle size and selection algorithm define a continuum of strategies that includes both static tiling and scattered mapping. Why? If the pixels of a bundle are contiguous, and the number of bundles equals the number of nodes, the partition is a static tiling. Similarly, if the number of bundles equals the number of nodes, and the pixels are selected via a modulo mapping, the scattered mapping results. At the other extreme, single pixels can be distributed, albeit at great communication cost. Indeed, one reason for the apparent anomaly in Figure 8 is the high cost of communication

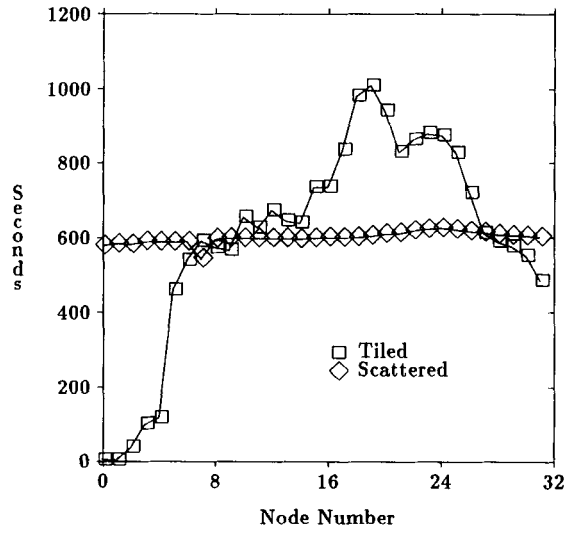

Figure 9. Node execution times (512 × 512 pixel teapot scene)

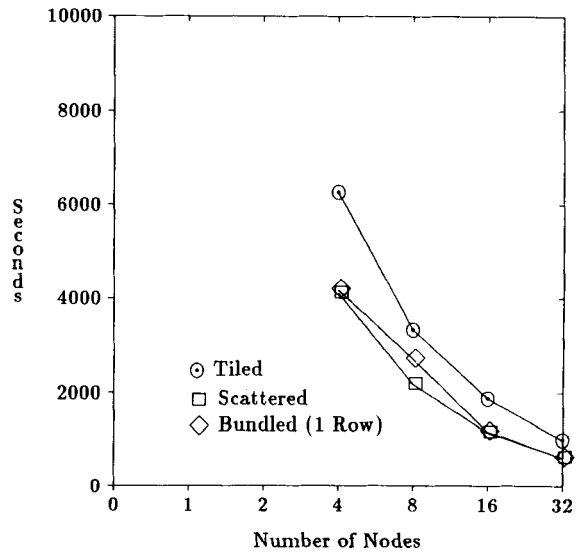

Figure 10. Execution times with ray partitioning (512 × 512 pixel teapot scene)

between the Unix system host and the nodes. Performance instrumentation reveals that the communication daemon on the host creates a much higher communication latency than exists for internode communication. As a consequence, the performance of ray bundling with groups of 512 pixels is comparable to that for the static scattered mapping; Figure 10 shows that this is true across a range of system sizes when the size of a bundle is small.

To investigate the interplay of bundle size and dynamic load balance, we conducted a series of experiments with a range of bundle sizes. Figure 11 shows the result of these experiments as a function of both image and bundle size. Based on the Figure, one might speculate that bundles smaller than one image row would provide even better load balance, further reducing the execution time. Although such bundles might provide better load balance, this is offset by greater communication overhead.[8] Conversely, larger bundles reduce communication overhead and create greater load imbalance. Thus, there exists a critical region of bundle sizes where communication costs are balanced against load distribution; the location of this region depends on both the algorithm and the underlying communication latency. Bundles much smaller than one image row are likely to be ineffective.

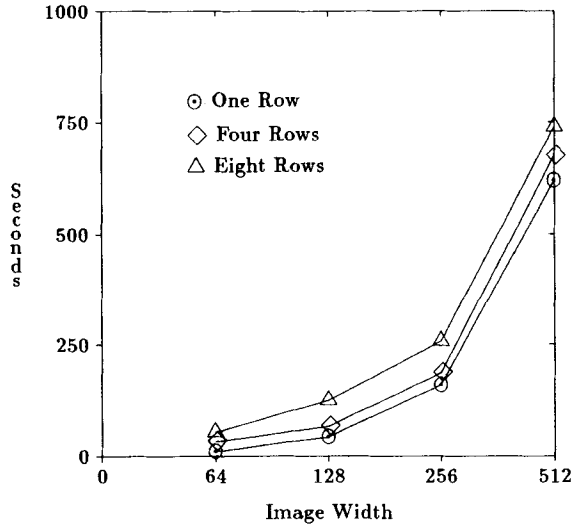

Figure 11. Execution times with bundled ray partitioning (Teapot scene on 32 nodes (or maximum possible))

Figure 12, obtained using our performance instrumentation, shows the importance of identifying the critical bundle size. In the Figure, the time spent in user computation, the idle time, and the system overhead are shown for two bundle sizes on an eight-node system. When the bundle size is one image row, the amount of system overhead, including message passing, is negligible compared to the time for ray calculation,[9] and the system is well balanced. Larger ray bundles create a load imbalance and cannot significantly decrease the already small message passing overhead. Based on our experiments, the performance of ray bundling does not significantly exceed that of a good static scattered partition.

[8] Recall that large messages are needed to offset the high latency of the Intel iPSC/2 communication system.

[9] Indeed, the system overhead is too small to display in Figure 12.

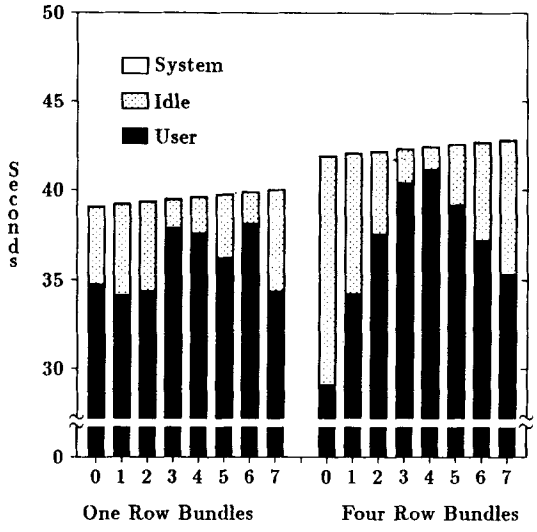

Figure 12. Distribution of execution time with ray bundling (64 × 64 pixel teapot scene)

5.3. Object partitioning

Previously, we considered ray partitions when all objects were replicated in the memory of each node. The two major effects of object partitioning are to increase the number of objects in the scene and to force the exchange of messages to disseminate ray information. Because object partitioning distributes the scene's objects among the nodes, ray information must be passed between regions of the scene (i.e. between nodes).

Recall that maximal partitioning occurs when the objects are evenly distributed across the nodes. Partial partitioning is possible when the memory requirements of the scene are less than the total memory of the system but exceed the memory in one node. As Figure 5 suggests, it is possible to place 75% of the scene's objects in each of four nodes. Thus, object partitioning represents a continuum, with delimiting points at maximal partitioning (i.e. no redundant representation) and no partitioning (i.e. fully redundant representation). Intuitively, partial object partitioning reduces the number of message transmissions and reduces processor stalls due to load imbalance. In the remainder of this Section we investigate the performance of object partitioning, including the potential performance benefits of object replication. We begin, however, by considering the deleterious effects of regular confinement geometries.

As noted in Section 4.2, object partitioning increases the total number of objects because those objects on the boundary of the spatially disjoint volumes are replicated; this permits regular confinement geometries. Figure 13 illustrates this increase for the teapot test scene when maximally partitioned. An eight fold partition increases the number of objects by roughly 30%. Fortunately, image generation time and the number of objects in a scene are not linearly related. Figure 14 shows that a 1000-fold increase in the number of objects results in less than a ten-fold increase in the execution time. Thus, the object replication overhead for regular geometries is negligible.

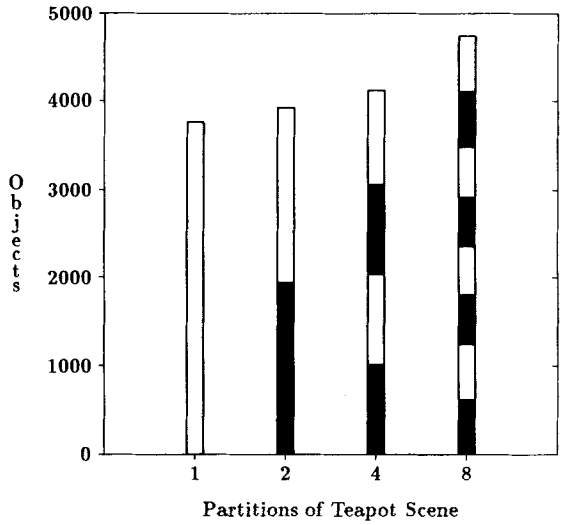

Figure 13. Increase in number of objects from partitioning

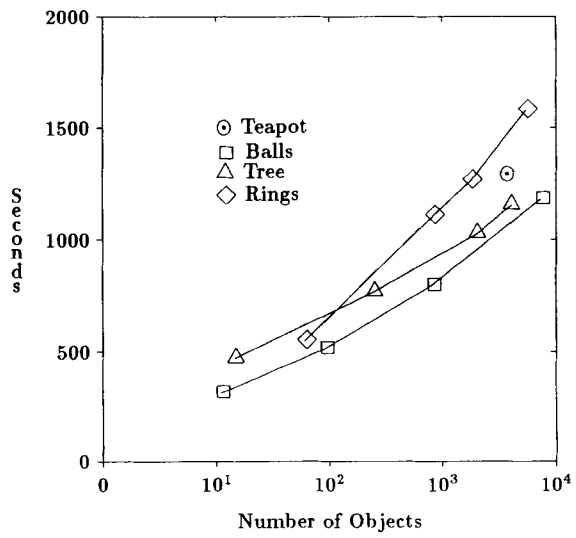

Figure 14. Effects of scene complexity with scattered ray partitioning (512 × 512 pixel image on 16 nodes)

To study the effects of intentional object partitioning, we varied the fraction of the scene replicated in each node. In all cases, ray bundling was used to dynamically distribute rays.[10] Figure 15 shows the execution times for our four test scenes as a function of object duplication. Clearly, object partitioning does increase the overall execution time, but the

[10] In this variation of ray bundling, rays of each bundle were selected via a modulo mapping function.

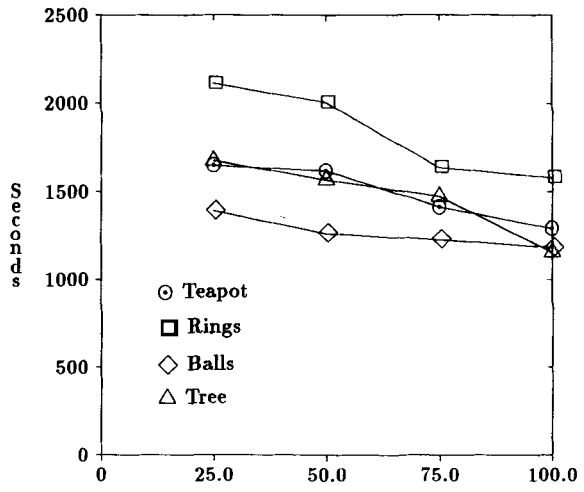

Figure 15. Effects of scene replication (512 × 512 pixel image on 16 nodes)

increase is not linear with respect to the percentage of duplicated objects. In the worst case, the rings scene, the lack of even a few objects in each node increases execution time. This increase could be attributed to any of three phenomena: internode communication, computation for redundant objects intersection checks, or load imbalances from the distribution of the objects in the scene. The data of Figure 15 are not sufficient to identify the primary cause. However, the additional data made possible by detailed event traces do reveal the performance bottleneck.

Although object partitioning does require message passing to distribute ray information among the partitions of the scene, the performance data in Figure 16 show that the primary overhead is load imbalance. Why? There are two partitions, rays and objects. With a dynamic, modulo mapping of pixels (rays) to nodes, no knowledge of the objects' spatial distribution or of the object characteristics is used. When most objects are replicated, the penalty for this decoupling is small. As the object partitioning increases, more rays must be sent to other nodes for processing. When the net influx of rays at a node is positive, its load increases; conversely, when the net influx is negative, the load decreases. Figure 17 shows the effects of this message transfer on processor utilization as a function of time.[11] Early in the computation, nodes receive ray bundles from the host on demand. After the host's ray supply is exhausted, internode ray transfer is the only load balancing mechanism; this creates imbalances near the end of the computation. In Figure 17, each node contains 75% of the objects. As this value decreases (i.e. as the object partitioning approaches maximality), the load imbalances increase. This suggests that ray and object partitions should occur in concert. We plan to investigate this approach in the future.

[11] The Figure represents the reduction of many megabytes of event trace data obtained via our performance analysis tools.

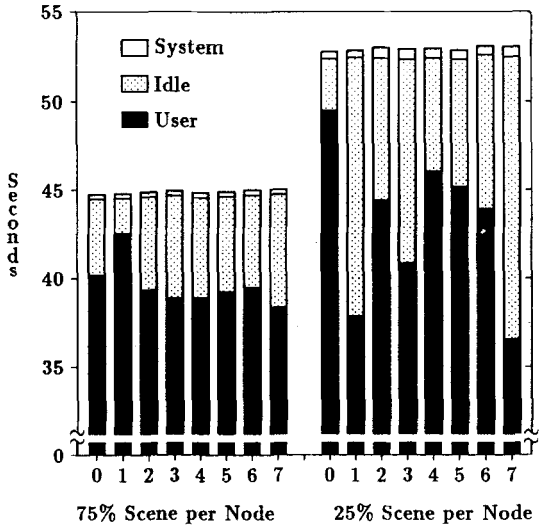

Figure 16. Scene replication comparison (64 × 64 pixel teapot scene)

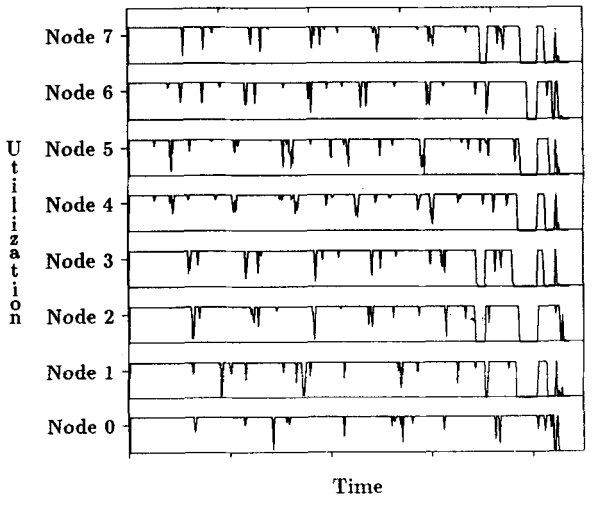

Figure 17. Processor utilization with 75% replicated objects (64 × 64 pixel teapot image on 8 nodes)

Regardless of the approach to ray and object partitions, rays must be exchanged by nodes. A naive implementation sends a message for each ray that crosses a partition boundary. This generates many small messages and is ill-matched to the Intel iPSC/2's high latency communication. To offset this latency, our implementation of object partitioning sends several rays in each message. Figure 18 shows the effects of grouping rays in larger messages. Combining rays reduces the number of system calls and the time spent in system code. This, in turn, changes the time spent both computing and idle.

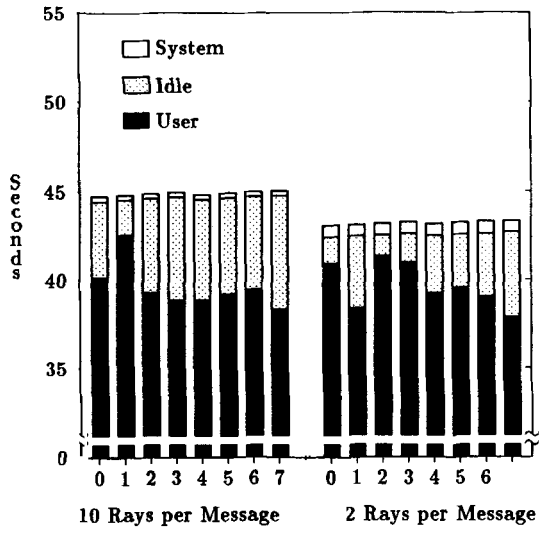

Figure 18. Ray packaging with 75% objects/node (64 × 64 pixel teapot scene)

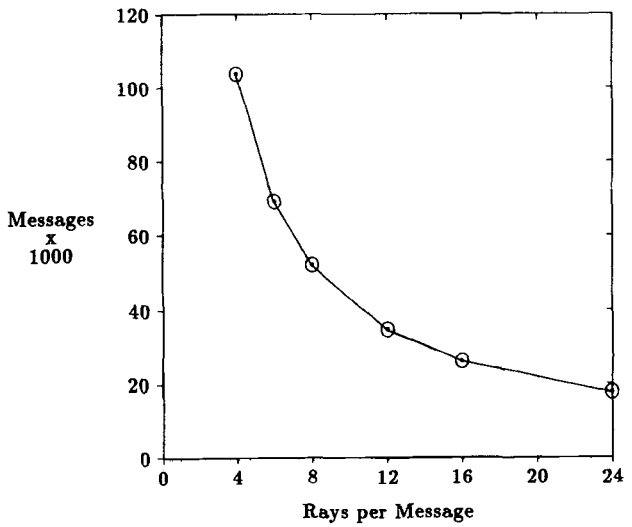

Figure 19. Object partitioning message exchange (512 × 512 pixel teapot scene)

Although the number of messages is reduced by combining rays, the decrease is not linear as one might expect (see Figure 19). As the ray tracing nears completion, each node's queue of outstanding rays begins to empty. To prevent deadlock, each node with no outstanding rays must transmit its remaining ray state information. This reduces the average number of rays per message and creates the effect in Figure 19. When the number of rays in each group is larger, this 'fringe' effect increases—fewer total messages are sent.

6. SUMMARY

Ray tracing's large computation requirements, coupled with its inherently parallel nature, make ray tracing algorithms attractive candidates for implementation on distributed memory parallel systems. In this paper we have considered several such implementations on the Intel iPSC/2, including ones based on ray and object partitioning.

For scenes of modest complexity, substantial parallelism can be attained via simple ray partitioning. Of the many approaches to ray partitioning, simple static partitions yield acceptable performance if each node is assigned rays from disjoint regions of the scene. Dynamic ray partitions are competitive if the number of rays distributed in response to requests is large enough to offset communication latencies.

For complex scenes, the limited memory of each node is not sufficient to store all the scene's objects. To process scenes that exceed the memory of a single node, the objects must be distributed across the nodes. Performance penalties accrue because ray information must be shared by the nodes. The associated communication costs and redundant computation are substantial but not unacceptable. In our experiments, the execution time was no more than twice that without object partitioning.

We have argued, by example, that these algorithm insights are most easily obtained via an integrated performance instrumentation system. Although many of our results might have been inferred from careful timing measurements and algorithm analysis, the ability to quickly and automatically capture and display detailed performance data replaces inference with observation. Our practical experience has been that even after modifying and porting the ray tracing code, we were surprised in several instances by what the performance visualization revealed. However, had it been necessary to manually instrument the ray tracing code to capture event traces, we would not have undertaken this study. This feeling is typical; most users will sacrifice measurement detail for ease of use.

In general, the greater the number and scope of required modifications to instrument an application program, the less likely the instrumentation system is to be used. Similarly, if the performance instrumentation environment does not allow users to choose the appropriate data analysis idiom, either statistical analysis, gestalt views (e.g. timelines) or dynamic graphics, it will not be used, regardless of its potential analytic power. Only when parallel systems software includes flexible performance analysis and data display tools will the systematic evaluation of parallel algorithm variants become commonplace and accepted by most users.

ACKNOWLEDGEMENTS

The first author was supported by IBM during an educational assignment under the resident study program. The second author was supported in part by the National Science Foundation under grants NSF CCR86-57696, NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by grants from AT&T and the Digital Equipment Corporation External Research Program.

Much of the port of the ray tracing code to the Intel iPSC/2 was conducted as part of a class project with Manish Gupta and Andreas Hauenstein. Their reviews, development and debugging efforts are greatly appreciated. The performance

instrumentation environment is the work of the *Picasso* research group: Ruth Aydt, James Arendt, Dominique Grabas, Allen Malony, David Rudolph and Brian Totty. Finally, we thank Tony Reeves and the Cornell Theory Center for time on their 32-node Intel iPSC/2.

REFERENCES

1. R. Arlauskas, 'iPSC/2 system: a second generation hypercube', In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, Pasadena, CA, Jan. 1988, Association For Computing Machinery, pp. 38–42.
2. P. Close, 'The iPSC/2 node architecture', In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, Pasadena, CA, Jan. 1988, Association For Computing Machinery, pp. 43–50.
3. D.K. Bradley, 'First and second generation hypercube performance', Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, Sept. 1988.
4. A.D. Malony, D.A. Reed, J.W. Arendt, R.A. Aydt, D. Grabas and B.K. Totty, 'An integrated performance data collection, analysis, and visualization system', In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989, Association For Computing Machinery.
5. D.A. Reed and D.C. Rudolph, 'Experiences with hypercube operating system instrumentation', *International Journal of High-Speed Computing*, **1**(4), 517–542 (1989).
6. R.M. Stallman, 'Using and porting GNU CC', Tech. Rep., Free Software Foundation, Inc., Cambridge, MA, Dec. 1988.
7. D.C. Rudolph, 'Performance instrumentation for the Intel iPSC/2', Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, July 1989.
8. R.W. Scheifler and J. Gettys, 'The X window system', *ACM Transactions on Graphics*, **5**(2), 79–109 (1986).
9. T.J. Whitted, 'An improved illumination model for shaded display', *Commun. ACM*, **23**(6), 343–349 (1980).
10. J. Salmon and J. Goldsmith, 'A hypercube ray tracer', In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume II*, Pasadena, CA, Jan. 1988, Association For Computing Machinery, pp. 1194–1206.
11. J.G. Cleary, 'Multiprocessor ray tracing', Tech. Rep. Number 83/128/17, University of Calgary, Oct. 1983.
12. A.S. Glassner, 'Spacetime ray tracing for animation', *IEEE Computer Graphics and Applications*, **8**(2), 62–72 (1988).