Parallel Visualization Algorithms: Performance and Architectural Implications

Jaswinder Pal Singh, Anoop Gupta and Marc Levoy

Computer Systems Laboratory
Stanford University

1 Introduction

Several recent algorithms have substantially sped up complex and time-consuming visualization tasks. In particular, novel algorithms for radiosity computation [1] and volume rendering [2][3] have demonstrated performance far superior to earlier methods. Despite these advances, visualization of complex scenes or data sets remains computationally expensive. Rendering a 256-by-256-by-256 voxel volume data set takes about 5 seconds per frame on a 100 MHz Silicon Graphics Indigo workstation using the ray-casting algorithm in [2], and about a second per frame using a new shear-warp algorithm [3]. These times are much larger than the 0.03 seconds per frame required for real-time rendering or the 0.1 seconds per frame required for interactive rendering. Realistic radiosity and ray tracing computations are much more time-consuming.

Multiprocessing provides an attractive solution to this computational bottleneck. It is well known that ray casting algorithms afford substantial parallelism, and we show that the same is true for the radiosity and shear-warp methods as well. However, all these visualization algorithms have highly irregular and unpredictable data access patterns. This makes data distribution and communication management very difficult in the explicit message-passing programming paradigm supported by most scalable multiprocessors (e.g. the Intel iPSC/860 and Paragon or the Thinking Machines CM-5), since these tasks have to be performed explicitly by the programmer. The need for explicit communication management leads to complicated parallel algorithms that look very little like their sequential counterparts, and to substantial performance inefficiencies.

Recently, a new class of scalable, shared-address-space multiprocessors has emerged. Like message-passing machines, these multiprocessors also have a distributed interconnection network and physically distributed main memory. However, they provide hardware support for efficient implicit communication through a shared address space, and they automatically exploit temporal locality by caching both local and remote data in a processor's hardware cache. In this paper, we show that these architectural characteristics make it much easier to obtain very good speedups on the best known visualization algorithms. Simple and natural parallelizations work very well, the sequential implementations do not have to be fundamentally restructured, and the applications have a high enough degree of temporal locality that there is no need for explicit data distribution and communication management. We demonstrate our claims through parallel versions of three state-of-the-art algorithms: a recent hierarchical radiosity algorithm by Hanrahan et al [1], a ray-casting volume renderer by Levoy (parallelized in [2]), and an optimized ray-tracer [12]. We also briefly discuss the parallelization of a new, shear-warp volume rendering algorithm [3] which results in what is to our knowledge the first demonstration of interactive frame rates for a 256-by-256-by-256 voxel data set on a general-purpose multiprocessor.

The rest of the paper is organized as follows. The next section describes cache-coherent shared address space multiprocessors in general and the specific machines that we use. We then describe the sequential algorithms and their parallelizations for these machines. We demonstrate excellent speedups on the Stanford DASH multiprocessor, an experimental cache-coherent shared address space machines with 48 processors, as well as on a commercially available, 16-processor Silicon Graphics Challenge machine. Finally, we contrast the simplicity of the shared address space parallelizations with the complexity and resulting performance losses of explicit message-passing implementations.

2 Cache-coherent Shared Address Space Multiprocessors

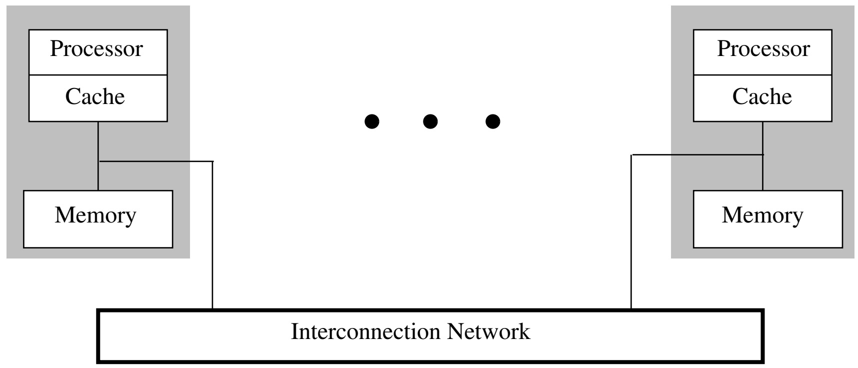


Figure 1: A shared address space multiprocessor with physically distributed memory.

[NOTE TO EDITOR: This section can be put in a side-bar.]

Figure 1 shows the generic shared address space multiprocessor that we assume in our parallelization. The multiprocessor consists of a number of processing nodes, connected by a general interconnection network. Every node contains a processor, a cache and a portion of the total physical (main) memory on the machine. The address space is shared, so that any processor can reference any variable regardless of where it resides. When a processor reads or writes a word, that word is brought into the processor's cache. Modifying locally cached shared data introduces the cache coherence problem, which is solved by using a distributed directory-based protocol supported in hardware [4].

The two important goals in parallelizing an algorithm to run on such a machine are balancing the workload across the cooperating processors and preserving locality of data referencing. Locality is important because although memory is uniformly addressable, it is not uniformly accessible: The cost of accessing a data item increases with the distance the access has to travel from the issuing processor to be satisfied. Load balancing and data locality are often at odds with each other, and must be traded off for good performance.

The generalized multiprocessor shown in Figure 1 affords locality at three levels of the memory and interconnection hierarchy:

- Cache Locality: This includes both the temporal locality exploited by reusing data that a processor brings into its cache (whether from its own local memory unit or from across the network), as well as the spatial locality provided by multiword cache lines.
- Memory Locality: If references miss in the cache, one would like to satisfy them in the local memory unit rather than have to communicate across the network. Memory locality can be provided by distributing data appropriately across physical memory units—either statically or dynamically—or by replicating data in main memory as well as in the caches. Both data distribution and replication in main memory require user intervention on most systems, and make the programming task more difficult.
- Network Locality: If references have to go across the network to be satisfied, one would like them to be satisfied as close as possible to the issuing processor in the network topology.

As we shall see, neither memory nor network locality are important in the visualization algorithms we examine. The temporal cache locality that obviates these falls naturally out of the spatial coherence in the applications exploited by simple partitioning schemes designed to reduce communication.

The specific machine that we use in most of our experiments is the Stanford DASH multiprocessor, an experimental cache-coherent machine built at Stanford University [4]. The machine we use has 48 processors organized in 12 clusters.[†] A cluster consists of four 33MHz MIPS R3000 processors connected by a shared bus, and clusters are connected together in a two-dimensional mesh network. Every processor has a 64KB first-level cache and a 256KB second-level cache, and every cluster has an equal fraction of the 256MB of physical memory on the machine. To demonstrate performance on a commercially available machine with faster processors, we also use a Silicon Graphics Challenge multiprocessor. This has sixteen 150 MHz MIPS R4400 processors connected by a 1.2 GB/sec bus to one another and to a centralized shared memory. The centralized shared memory on the Challenge implies that locality in main memory is not an issue there.

[†]The prototype actually has 64 processors in 16 clusters, but is broken up into two separate machines in usual operation.

Let us now examine the parallelization of the visualization algorithms on these machines. For each application, we describe the sequential algorithm, discuss the parallelization, and present performance results and analyses. We begin with the hierarchical radiosity algorithm, and then proceed to the volume renderer and ray tracer.

3 Hierarchical Radiosity

The radiosity method computes the global illumination in a scene containing diffusely reflecting surfaces. It is a view-independent visualization method, which means that the radiosity does not have to be recomputed when the viewpoint is changed. In traditional radiosity approaches, the large polygonal surfaces that describe the scene (such as walls or desktops) are first subdivided into small enough elements or patches that the radiosity of a patch can be approximated as being uniform. The radiosity of a patch i can be expressed as a linear combination of the radiosities of all other patches j, leading to a linear system of equations. The coefficients in the linear combination are the ‘‘form factors’’ between the patches, where the form factor F_{ji} from patch j to patch i is the fraction of light energy leaving j which arrives at i. The inherent form factor depends on the shape of each patch i and j, the angle the patches make with each other, and the distance between them. However, this must be modified by the presence of any intervening patches that occlude the visibility between i and j. Particularly given the need to test all patch pairs i and j for inter-visibility, the computation of form factors is the most time-consuming part of a radiosity algorithm. The number of form factors among all pairs of n patches is $O(n^2)$, which makes traditional radiosity methods (including progressive radiosity [7]) very expensive.

A new hierarchical method [1] dramatically reduces the complexity of computing radiosities. The method is inspired by recent advances in using hierarchical methods to solve the N-body problem. A scene is initially modeled as comprising a number, say k, of large input polygons (each representing a desktop or a wall in a room scene, for example). Light transport interactions are computed among these polygons, and polygons are hierarchically subdivided as necessary to improve accuracy. Each subdivision results in four subpatches, leading to a quadtree per input polygon. If the resulting final number of undivided subpatches is n, the number of interactions or form factors computed by this algorithm is $O(n+k^2)$. A brief description of the algorithm follows. Details can be found in [1][11].

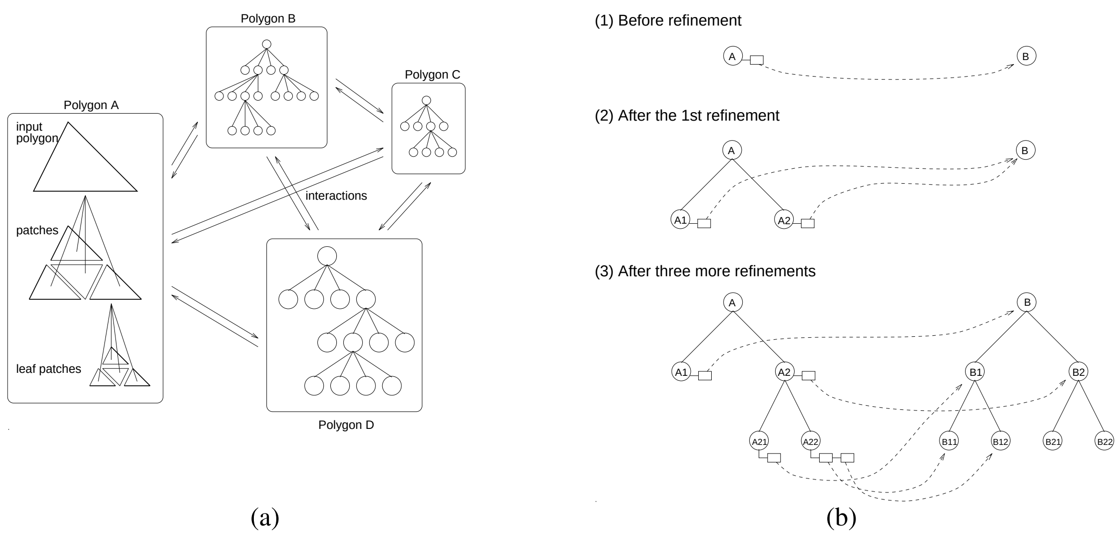


Figure 2: Refinements and interactions in the hierarchical radiosity application. Binary trees are instead of quadtrees for simplicity in (b), and only one polygon’s interaction lists are shown.

3.1 Sequential Algorithm

The input polygons that comprise the scene are first inserted into a binary space partitioning (BSP) tree to facilitate efficient visibility computation between pairs of patches. Every input polygon is given an *interaction list* of other input polygons which are potentially visible from it, and with which it must therefore compute interactions. Then, polygon radiosities are computed by the following iterative algorithm:

1. For every polygon, compute its radiosity due to all polygons on its interaction list, subdividing it or other polygons

hierarchically as necessary. Subdivided patches acquire their own interactions lists, and are processed recursively (see Figure 2).

2. Add all the area-weighted polygon radiosities together to obtain the total radiosity of the scene, and compare it with that of the previous iteration to check for convergence. If the radiosity has not converged to within a user-specified tolerance, return to step 1. Otherwise, go to step 3.

3. Smooth the solution for display by computing the radiosities at the vertices of the leaf-level elements. Since this phase is performed only once at the end of the algorithm, and since it is a very simple phase from the viewpoint of parallelism, we do not discuss it further.

Most of the time in an iteration is spent in step 1. In every iteration, each of the quadtrees is traversed depth-first, starting from the root. At every quadtree node visited in this traversal, interactions of the patch[‡] at that node (patch *i*, say) are computed with all other patches, *j*, in its interaction list. An interaction may cause one of the interacting patches to be subdivided, and children to be created for the subdivided patch if they don't already exist. If patch *i* (the patch being visited) is subdivided, patch *j* is removed from *i*'s interaction list and added to each of *i*'s children's interaction lists. If patch *j* is subdivided, it is replace by its children on patch *i*'s interaction list. Figure 2(b) shows an example of this hierarchical refinement of interactions. Patch *i*'s interaction list is completely processed in this manner before visiting its children in the tree traversal.

At the beginning of an iteration, the interaction list of a patch in any quadtree is exactly as it was left at the end of the previous iteration: containing the patches with which its interaction did not cause a subdivision.

3.2 Exploiting Parallelism

Parallelism is available at three levels in this application: across input polygons, across the patches that a polygon is subdivided into, and across the interactions computed for a patch. Since the patch quadtrees are constructed as the application proceeds, all three levels of parallelism involve communication and synchronization among processors. For example, a processor must lock a patch to ensure that it has exclusive access before subdividing the patch.

Statically assigning polygons or polygon pairs to processors leads to severe load imbalances, since the workload distribution across polygon pairs is highly nonuniform. As in all the applications we consider, dynamic task stealing is needed for load balancing. We obtain the best performance by defining a task to be either a patch and all its interactions or a single patch-patch interaction, depending on the size of the problem and the number of processors (the difference is usually small).

The parallel implementation provides every processor with its own task queue. A processor's task queue is initialized with a subset of the initial polygon-polygon interactions. When a patch is subdivided, new tasks involving the subpatches are enqueued on the task queue of the processor that did the subdivision. A processor consumes tasks from its task queue until there are no tasks left. It then steals tasks from other processors' queues, which it can directly access in the shared address space. While this task stealing provides load balancing, it can compromise data locality. However, locality is preserved as follows [11]. A processor inserts tasks at the head of its queue. It dequeues tasks from the head of its own queue (to yield a depth-first search of quadtrees and hence reuse portions of the BSP tree efficiently across visibility testing interactions) but steals from the tail of another processor's task queue (increasing the likelihood of stealing a large patch, within which locality can be exploited).

3.3 Results and Discussion

This simple parallelization is both conceptually natural and also very easy to implement in a shared address space. As seen in Figure 4(a), it also yields very good speedups on the DASH multiprocessor. This is despite the fact that no attempt was made to distribute (or replicate) data in main memory, which is fortunate since appropriate data distribution at the page granularity would have been very difficult given the irregular, dynamic data structures and fine-grained data sharing patterns of the algorithm. Good speedups are also obtained on the Challenge, although data distribution is not an issue there given its centralized shared memory. The speedups shown are for the relatively small room scene (174 input polygons, see Figure 3(a)) used by Hanrahan et al in [1], which is why they scale more slowly after 32 processors on DASH. We expect to obtain even better speedups with larger input scenes, and that these results about the effectiveness of shared address space multiprocessors

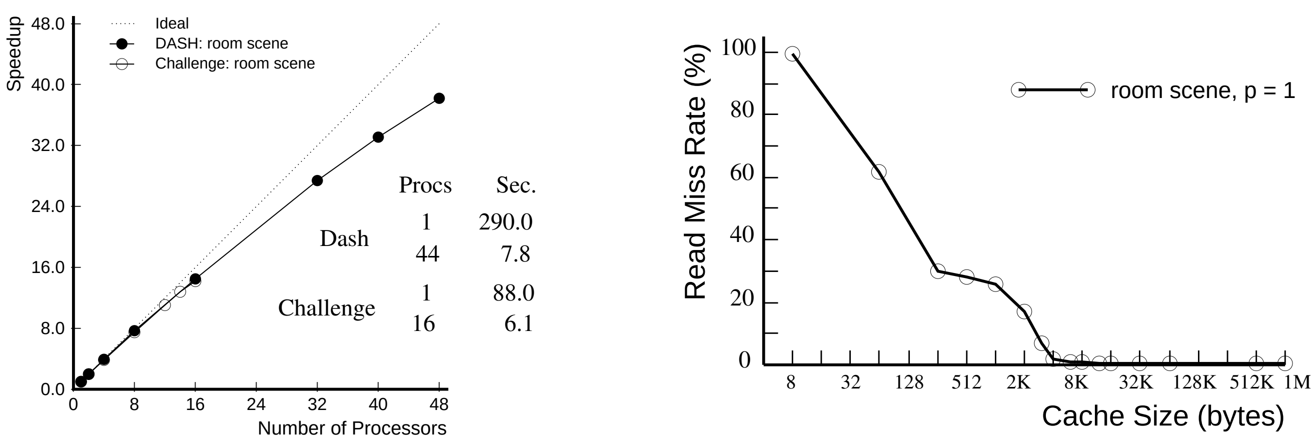
[‡] A "patch" here can mean an input polygon, an intermediate patch, or a leaf element.

will extend to other radiosity algorithms, such as hierarchical radiosity with glossy surfaces, zonal radiosity, and even importance-driven radiosity (since there appears to be no need for data redistribution even if the viewpoint changes).

We now show that the reason we obtain good performance without attention to locality in main memory is the high degree of temporal locality in the application and the effectiveness of automatic caching in exploiting this locality transparently. To analyze temporal locality, we measure the size and impact of the important per-processors working sets of the applications. We measure working sets by using a simulated multiprocessor with fully associative caches to plot the read miss rate versus the cache size used, following the methodology described in [6].

Figure 4(b) indicates a very high degree of temporal locality: The important working set is about 4KB of data for this input, and reduces the miss rate to a negligible quantity. The algorithm spends the vast majority of its time computing the visibility between interacting patches (say *i* and *j*). Visibility for an interaction is computed by firing a number of “random” rays from *i* to *j*, and measuring the fraction of these rays that reach *j* without being occluded. The algorithm therefore repeatedly traverses the relevant portion of the BSP tree between the input polygons that are the ancestors of patches *i* and *j*. The next visibility interaction that the processor computes will likely be between patch *i* and a child of patch *j*, say, and will thus reuse the same portion of the BSP tree. As a result, the important working set for a processor is a fraction of the BSP tree. The BSP tree is very small compared to the entire data set of quadtrees. The size of the working set (BSP tree) grows as the logarithm of the number of input polygons, and is independent of the number of processors used. Given the multi-megabyte cache sizes that people are building on shared-address-space machines today, there is little chance of encountering problems whose working sets will overflow these caches.

The use of hierarchy allows this algorithm to exploit temporal locality better than traditional radiosity algorithms, which sweep through all patches as they shoot radiosity to them. The hierarchical algorithm’ s use of gathering rather than shooting also results in better communication behavior—since only a processor that owns a patch writes the radiosity of that patch—and avoids the tradeoff between concurrency and preserving the sorted order of patches in a shooting approach [11]. In fact, gathering has been observed to work better than shooting in parallel even for traditional radiosity algorithms on message-passing machines [8]. Let us now turn our attention now to volume rendering.



(a) Speedups

(b) Working Sets

Figure 4: Speedups and working sets for the radiosity application. The BF refinement threshold was set to 0.02, and the area refinement threshold to 2000 units [1]. The algorithm took 7 iterations to converge on a uniprocessor.

4 Volume Rendering

Volume rendering techniques are of key importance in the analysis and understanding of multidimensional sampled data, such as those generated in various scientific disciplines. The first parallel algorithm we use, developed in [2], renders volumes using optimized ray casting techniques. Until very recently, the sequential algorithm was one of the fastest known algorithms for volume rendering. In Section 4.4, we also examine a new shear-warp algorithm that is much faster, and by parallelizing which we are able to obtain interactive frame rates for a rotation sequence of a 256-by-256-by-256 voxel data set on a Silicon Graphics Challenge multiprocessor.

Color pictures will appear here.

Figure 3: Images rendered by the three applications: (a) hierarchical radiosity image, (b) radiosity image showing the patches created, (c) volume rendering, (d) ray tracing. Figs. (a) and (b) courtesy of Pat Hanrahan, Fig. (c) courtesy of North Carolina Memorial Hospital, and Figure (d) courtesy of Hewlett-Packard.

4.1 Sequential Ray-casting Algorithm

The volume to be rendered is represented by a cube of voxels (or volume elements). For each voxel, a color and a partial opacity have been computed during a prior shading operation. The outermost loop of the computation is over a sequence of viewing frames. In a typical sequence, successive frames correspond to changing the angle between the viewer and the volume being rendered. For each frame, rays are cast from the viewing position into the volume data through pixels in the image plane that corresponds to that frame. Colors and opacities are computed for a set of evenly spaced sample locations along each ray, by trilinearly interpolating from the colors and opacities of surrounding voxels. These samples are blended together using digital compositing techniques to yield a single color for the ray and hence for the corresponding pixel. Rays in a volume renderer typically are not reflected, but pass straight through the volume unless they encounter too much opacity and are terminated early.

The algorithm uses three optimizations: (i) the early ray termination mentioned above, controlled by a user-defined opacity threshold, (ii) the use of an octree representation of space to avoid unnecessary sampling in transparent regions of the volume, and (iii) adaptive image sampling. Adaptive sampling introduces some synchronization at partition boundaries [2], and we present parallel performance both with and without it.

4.2 Exploiting Parallelism

In a shared address space, every processor can directly reference any voxel in the data set. Only one copy of the voxel data set is maintained, and it is distributed round-robin at the granularity of pages among the local memories of processors. No attempt is made at smart data distribution, since this is very difficult at page granularity and since it is in any case impossible to determine a good static distribution given that the viewpoint and hence the data affinity of processors changes across frames. The voxel data set is read-only. It is therefore very easy to exploit the most natural parallelism, which is across rays (or pixels in the image plane). Since an equal partitioning of the image plane among processors is not necessarily load balanced, owing to the nonuniformity of the volume data, task stealing is once again required for load balancing.

Given p processors, the image plane is partitioned into p rectangular blocks of comparable size [2]. Every image block or partition is further subdivided into fixed sized square image *tiles*, which are the units of task granularity and stealing. These tile tasks are initially inserted into the task queue of the processor that is assigned that block (a distributed task-queue system is used, as in the radiosity application). A processor ray traces the tiles in its block in scan-line order. When it is done with its block, it steals tile tasks from other processors that are still busy. Giving a processor a contiguous set of pixels preserves locality by exploiting the spatial coherence in the algorithm: Successive rays that a processor casts go through contiguous pixels and will tend to access much of the same voxel data. Figure 5 shows a four-processor example of the image plane partitioning.

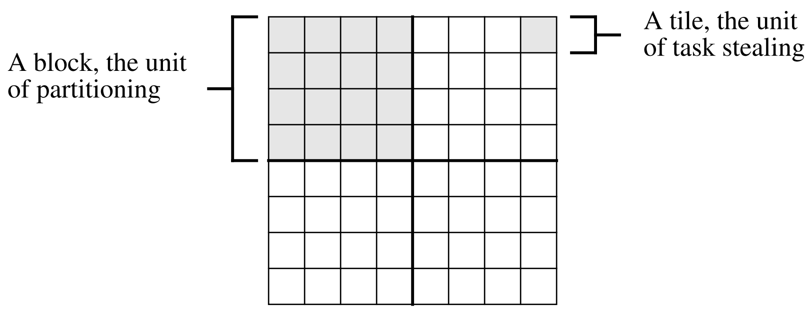
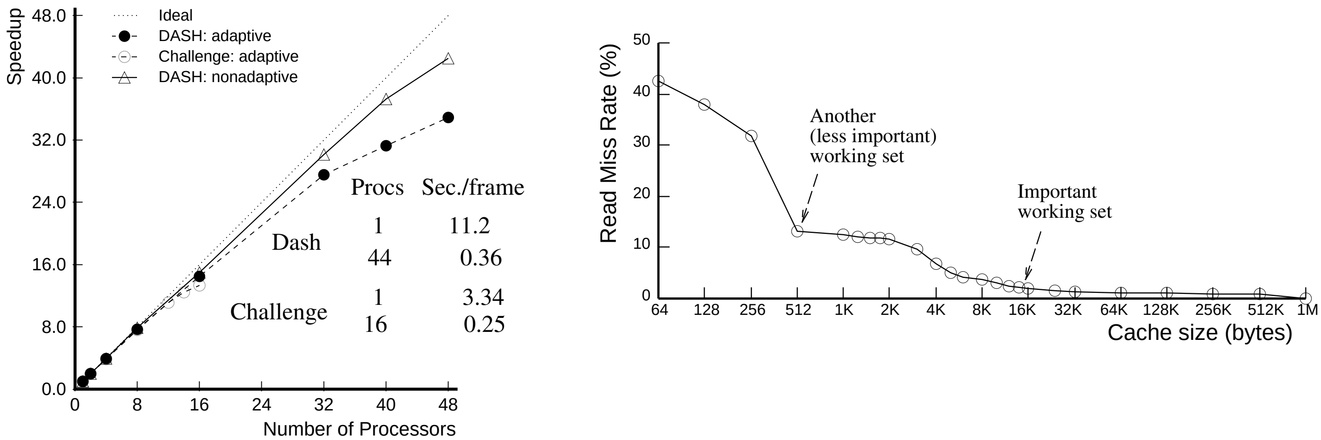


Figure 5: Image plane partitioning in the volume renderer for four processors.

4.3 Results

Figure 6(a) shows speedups on DASH for both adaptive and non-adaptive rendering, and on the Challenge for non-adaptive rendering. The results measure rendering time only, and do not include the time to load in the data set, compute opacities and build the octree, or transfer the rendered image to the frame buffer. A 256-by-256-by-226 voxel data set showing a computed tomography rendering of a human head is used, the resulting image of which is shown in Figure 3(b). The image measures approximately 415-by-415 pixels, and the total data set size is about 29 megabytes. A tile size of 8-by-8 pixels is used as the unit of task stealing.

Clearly, the parallel volume renderer yields very good speedups on both machines. Owing to the need for pixel sharing and additional synchronization with adaptive sampling, the speedups in this case are somewhat worse than with non-adaptive sampling. We are able to reach within a factor of 3 of interactive-time rendering on a 48-processor DASH or a 16-processor Challenge. As in the radiosity algorithm, the observed speedups on DASH are very good despite the fact that we simply distribute data round-robin among physical memories. Figure 6(b) shows that the reason in this case also is the high degree of temporal locality on both private and shared data accesses. The important working set in this case is the amount of read-only voxel and octree data used in sampling a ray that is typically reused by the next ray. The reuse owes itself to spatial coherence resulting from the contiguity of partitions in the image plane: Successive rays cast by a processor pass through adjacent pixels and tend to reference many of the same voxels in the volume. The important working set for the 30-megabyte data set we use (too large to be rendered at interactive rates) is only 16 kilobytes in size. The working set size is independent of the number of processors in this application as well, and is proportional to the number of voxels along a single dimension of the data set (along a ray);



(a) Speedups (b) Working Sets

Figure 6: Speedups and working sets for the ray-casting volume renderer. The threshold opacity value for early ray termination is set to 0.95, on a scale from 0 to 1.0. About 173,000 rays (618,000 samples) are traced in the case without adaptive sampling, and about 22,000 rays (245,000 samples) with adaptive sampling.

i.e. to the cube root of the data set size. Since the push in volume rendering is also toward real-time rendering rather than toward rapidly increasing data set sizes, the important working set for this algorithm is therefore likely to remain small for some time to come.

4.4 Interactive Frame Rates with the Parallel Shear-Warp Method

A new shear-warp algorithm has recently been developed that can render a 256-cube voxel data set in one second on a Silicon Graphics Indigo workstation [3]. We have parallelized this algorithm both on DASH and on the Challenge.

The shear-warp algorithm proceeds in two phases. It first factors the viewing transformation into a three-dimensional shear parallel to the data slices, and projects the data to form a distorted intermediate (composited) image. Then, it performs a two-dimensional warp on the composited image to produce a final undistorted image. Unlike the image-order ray-casting algorithm, this is an object-order algorithm which streams through slices of the volume data set in front-to-back order and splats voxels on to the composited image. The shearing of the volume has the attractive property of exploiting spatial cache locality (with multiword cache lines) in both the object and image data, unlike the ray-casting approach which does not exploit spatial cache locality in the object data. The algorithm uses run-length encoding, min-max pyramids and multi-dimensional summed area tables to achieve its efficiency without sacrificing image quality. Its phases are depicted pictorially in Figure 7.

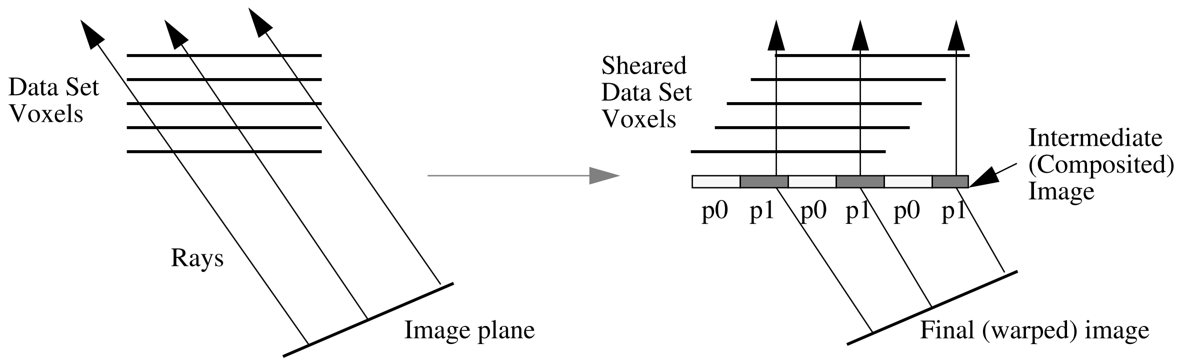


Figure 7: The shear-warp volume rendering method.

We parallelize the first (compositing) phase by partitioning the intermediate or composited image among processors. This ensures that only one processor writes a given pixel in the composited image. If the original voxel data set were partitioned among processors, different processors would write the same pixels (due to the shearing of the voxel data set) and synchronization would be required both to ensure mutual exclusion when updating pixels as well as to preserve dependences between processing slices of the data set. The composited image is divided into groups of scanlines (the optimal group size depends on the size of the problem and the cache line size on the machine), and the groups are assigned to processors in an interleaved manner (see Figure 7, which shows the partitioning for two processors). Instead of streaming through a full slice of the voxel data set before going to the slice behind it, as in the serial implementation, a processor now streams through the voxels in a *sheared* slice that correspond to one group of image scanlines that it is assigned, then proceeds to the similar group in the next slice, and so on. When it has gone through all the slices for one group of image scanlines, it processes the other groups that it is assigned, and finally steals groups from other processors if it is idle. The two-dimensional warp is also partitioned in groups of scanlines, by partitioning the final warped image among processors this time.

This parallelization achieves very good speedups, and allows us to obtain rendering rates of 12 frames a second for a rotation sequence on a 256-cube voxel human head data set. These speeds were obtained on a 16-processor Challenge machine (a single processor takes about 1 second per frame), and to our knowledge represent the first time that interactive frame rates have been achieved without sacrificing image quality on a 256-cube data set on general-purpose hardware. Thus, both image-order and object-order algorithms can be parallelized effectively on cache-coherent multiprocessors.

5 Ray Tracing

Our final application is an optimized ray tracer. The ray tracer was originally developed in [12] for a message-passing machine, with duplication of the entire scene data set on every processing node, and was later adapted to the current implementation on a shared address space machine without data set duplication.

5.1 Sequential Algorithm

As in the ray-casting volume renderer, primary rays are fired from a viewpoint, through the pixels in an image plane, and into a space that contains the objects to be rendered. At the first object that a ray encounters, it is reflected toward every light source to determine whether it is in shadow from that light source and to compute the contribution of the light source otherwise. The ray is also reflected from and refracted through the object as appropriate, spawning new rays. The same operations are performed recursively on the new rays at every object that they encounter. Thus, each primary ray generates a tree of rays, the rays being terminated when they leave the volume enclosing the scene or by some user-defined criterion (such as the maximum number of levels allowed in a ray tree). A hierarchical uniform grid (similar to an octree but with not necessarily binary subdivisions) is used to traverse scene data efficiently [12], and early ray tracing and adaptive sampling are implemented.

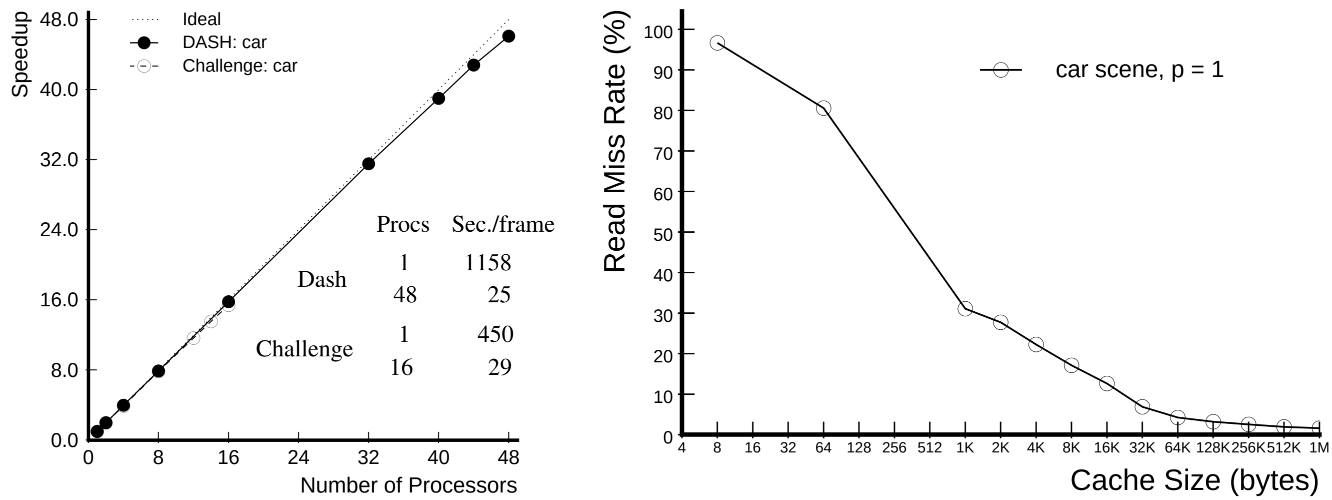
5.2 Exploiting Parallelism

Like the ray-casting volume renderer, the ray tracing algorithm affords substantial parallelism across rays, and the scene data are read-only. Here again, only a single copy of the scene database is maintained in shared space, and it is physically distributed round-robin at page granularity among the memories. The partitioning scheme used is almost identical to the one used for the ray-casting volume renderer, with a similar distributed task queue system for load balancing.

5.3 Results

Figure 8 shows the speedups for the parallel ray tracer. The scene being rendered is a car on a checker-boarded floor, as shown in Figure 3(c), and the image has 512-by-512 pixels. The data set size is about 10 megabytes. No anti-aliasing is used in these measurements. Excellent speedups are obtained, without attention to data distribution.

Like volume rendering, the important working set in raytracing consists of the data encountered in processing one primary ray (and the tree of rays it generates) that are reused in processing primary rays cast through neighboring pixels. The difference is that the working set is larger in ray tracing, and not so well-defined owing to the unpredictability of reflections (Figure 4). The working set size is once again independent of the number of processors. It's size depends on the hierarchical grid parameters discussed above, the reflectivity of the scene, and the number of levels allowed in the ray tree. Modern second-level caches should continue to keep the miss rate low enough to provide good performance.



(a) Speedups (b) Working Sets

Figure 8: Speedups and working sets for the ray tracing application. The hierarchical uniform grid is subdivided to a maxi mum of 3 levels, with 5-way subdivision in each dimension and a maximum of 60 primitive objects per leaf cell of the hier archy. The size of a tile (the unit of task stealing) is 8-by-8 pixels.

On machines that require main memory to be managed at the granularity of pages and under software control[1], several characteristics of these applications would make it very difficult to manage data distribution and replication for locality in main memory. These include: (i) dynamic data structures (the quadtrees) in radiosity and changing viewpoints in the other applications, which make it extremely difficult to determine which processors access which data most often, (ii) fine-grained data sharing, which makes pages an inappropriate granularity for locality management, and (iii) dynamic task stealing. Thus, it is fortunate that caches work well. These same characteristics make it very difficult to program these visualization algorithms for effective parallel performance on message-passing machines that do not support a shared address space, as we shall now see.

6 Cache-Coherent Shared Address Space versus Message-Passing

There are three primary aspects of communication management that distinguish the communication abstractions of a shared address space and message passing between private address spaces: (i) the *naming* of logically shared data, (ii) exploiting *temporal locality* on logically shared data, which includes both managing data replication and renaming as well as maintaining the coherence of replicated data, and (iii) the *granularity and overhead* of communication.

In a shared address space abstraction, any datum—local or nonlocal—can be referenced by any processor using the virtual address (name) of that datum in the shared address space. In the message-passing abstraction, on the other hand, a processor can directly reference only those data that are allocated in its private address space (local memory). A processor must therefore know or determine which processor's address space a datum resides in, and send a message to that processor requesting the datum if it is nonlocal.

As we have seen, temporal locality on both local and nonlocal data is handled automatically in shared address space machines that cache shared data—if the caches are large enough—and machines like DASH automatically keep the cached shared data coherent as well. On message-passing machines, nonlocal data must be replicated explicitly by the user and kept coherent by explicit communication of messages in the application program. The replicated data are thus explicitly renamed in message-passing programs, while hardware transparently takes care of renaming in the cache-coherent approach.

Finally, while hardware-coherent shared address space machines support communication efficiently at the fine granularity of cache lines, the overhead of initiating and receiving communication is much larger on message-passing machines

[1]Note that this is not a necessary feature of cache-coherent machines. Some recent architectures such as the Kendall Square Research KSR-1 manage main memory as a hardware-controlled cache at cache line granularity as well, and are thus able to handle much larger working sets.

(owing to software involvement) and it is therefore important to make messages large to amortize this overhead. Note that a coherent shared address space abstraction can be provided in software on a machine that does not provide any hardware support for it (such as an Intel iPSC/860 or Paragon message-passing machine); however, this is typically too inefficient for complex programs with fine-grained communication needs.

The disadvantage of cache-coherent machines is their cost and design complexity. However, recent efforts to build these machines has shown that the costs are quite small. In fact, the cost of the extra main memory which we find is often needed on message-passing machines for explicit replication of operating system and application code and data often dominates the hardware cost of cache coherence. And we argue that the cost of providing hardware support for a cache-coherent shared address space is more than justified by the ease of programming and performance it affords.

Managing communication explicitly is not very difficult for applications with regular, predictable behavior (such as those that solve systems of equations on regular grids). However, this is not true of visualization applications. We now use the ray tracing and radiosity applications to discuss the difficulties of message-passing implementations for these irregular applications. The issues in volume rendering are similar to those in ray tracing.

6.1 Ray Tracing

The main problems for message-passing in the raytracer are (i) managing the naming, replication and fine-grained communicaton overhead issues in sharing the read-only scene data, and (ii) managing load balancing. A third problem arises in managing synchronization when adaptive sampling is used to reduce computation time.

Naming: Any processor may need to access any part of the scene data set with fairly unstructured access patterns. Replicating the entire data set on all nodes is not an acceptable solution, since it severely limits the size of problem that can be solved and is not scalable. A reasonable distribution for message-passing is to assign every processor (memory) a contiguous subvolume of the scene space, so that a processor P can determine which processor Q's partition a ray goes to when it leaves P's partition. Processor P then has two choices: it can send the ray to Q, which will then continue to trace the ray, or it can communicate with Q to obtain the volume data the ray needs, and continue to process the ray itself. Both approaches have been tried in the literature [12][13]. Managing the naming and naturally fine-grained communication in both approches is complex and inefficient compared to a hardware-supported shared address space.

Replication: We have seen that replication of communicated scene data is very important to good performance. This is in fact accentuated on message-passing machines, where the overheads of communication are much larger. One approach to managing replication is to replicate every remote data structure that is touched and hold it locally for the duration of a frame, replacing data between frames. However, this can lead to large storage overheads without any benefits in complexity. The best approach for managing replication on a message-passing machine, used by Green and Paddon [13], is to emulate a fixed-size hardware cache for nonlocal data in the application program itself. Since this approach essentially amounts to implementing a restricted form of a shared address space with caches in the application program, it itself makes the argument for a shared address space machine (particularly since we have seen that realistic hardware caches are large enough to yield very good performance in such a machine). In fact, implementing this method of managing replication in software on a message-passing machine has significant overheads, since it introduces explicit renaming and in unstructured applications necessitates a check in software on every reference to volume data (to determine whether the referenced item is locally allocated, remotely allocated but in the local cache structure, or remote). None of this is required in a cache-coherent machine.

Communication Overhead and Granularity: All of the above approaches naturally generate fine-grained communication, which is very inefficient given the high message overhead on message-passing machines. Coalescing messages to make them larger requires substantial implementation overhead in such an unstructured application. While communication is fine-grained even in the cache-coherent shared address space approach, those machines support fine-grained communication efficiently in hardware.

Task Stealing and Load Balancing: In the shared address space implementation, the load balancing problem was resolved very simply by task stealing. All that was required to implement stealing were a lock per task queue and simple termination detection. On message-passing machines, task stealing must be done through explicit messages, which must be handled by the application program while it is performing the main computation. Task stealing is therefore much more complex and incurs greater overheads on message-passing machines. In a survey of message-passing implementations, Green and Paddon [13] mention several attempts to address the load balancing problem but not one of them uses task stealing. Instead, they try to pre-partition the image and object space intelligently to improve load balancing over a uniform decomposition (see [14], for example). The approaches are quite complex and input- as well as view-dependent, and the best ones often require profiling low-resolution runs to determine a desirable partitioning.

Finally, optimizations such as adaptive sampling (as used in the volume renderer) further complicate message-passing implementations, by requiring that the necessary synchronization for corner pixel values (see Section 4.2) be performed through explicit messages while the processes are in the midst of the main computation.

6.2 Radiosity

The hierarchical radiosity algorithm is much more complex to implement with explicit message passing. In addition to the irregular, unpredictable data accesses and the need for task stealing that it shares with the raytracer and volume renderer, it has two other complicating properties: (i) the main data structures (quadtrees of patches) are dynamically changing, since they are built as the computation proceeds, and (ii) these data structures are not read-only but are actively read and written by different processors in the same computational phase, which complicates coherence management. We have had graduate students implement message-passing versions on an Intel iPSC/860 machine. However, it has been an exercise in frustration, and only yielded 11-fold speedups on 32 processors before the project was abandoned as not being worthwhile. We briefly describe some of the main problems here. Detailed descriptions and explanations can be found in [11].

Naming: Given the dynamic data structures, we solve the naming problem by giving every patch a unique identifier of the form *quadtree.patch*, where *quadtree* is the number of the quadtree or polygon which that patch is a part of, and *patch* is the (globally consistent) number of the patch within that quadtree. Thus, we essentially implement an application-specific shared address space in software.

Replication and Coherence: We have experimented with two approaches to manage replication and coherence. In the first aproach, processors start a time-step with local copies of all the data corresponding to their patches and interaction lists. They modify these data (subdivide patches, etc.) locally as needed in an iteration, and communicate the modifications to other interested processors only at iteration boundaries. Coherence is thus maintained at a very coarse temporal granularity (an entire iteration), stale local information is often used or extrapolated from, and the amount of replication is typically very large. Special data structures have to be maintained dynamically to keep track of which patches are interested in updates made to a given patch. This is similar to maintaining an application-specific directory for cache coherence.

The second approach is to emulate a shared address space and caches in the application program. A single "master" copy of the forest of quadtrees is maintained in distributed form and manipulated dynamically through the passing of messages. This approach leads to much finer-grained communication, and local/cached/remote checks at every reference to quadtree data.

Task Stealing and Load Balancing: The complexity of maintaining coherence is greatly increased by the need for task stealing, particularly in the local quadtrees approach. When a patch is stolen, it must be decided whether the ownership of the patch remains with the old processor or is passed on to the stealer, both of which complicate coherence and communication management. Although stealing does help load balancing, its communication and bookkeeping overheads are so large in our current implementation that it improves speedups from 10 to only 11 with 32 processors on an Intel iPSC/860 machine.

The control and timing issues in handling messages for data, control, coherence, synchronization and load balancing while performing the computation are very difficult to program and debug in message-passing hierarchical radiosity, particularly given our results that cache-coherent shared address space machines solve this problem so well.

7 Summary and Conclusions

We have shown that general-purpose multiprocessors that efficiently support a shared address space and cache shared data are very effective vehicles for speeding up state-of-the-art visualization and image synthesis algorithms. Excellent parallel speedups were demonstrated on some of the most efficient known sequential algorithms, including hierarchical radiosity, ray-casting and shear-warp volume rendering, and ray tracing. We have shown that a shared address space allows us to easily implement very natural parallelizations, and that transparent coherent caching suffices to exploit enough temporal locality to yield excellent parallel performance. On the other hand, the dynamic nature and unstructured access patterns of all the algorithms make it much more difficult to program them effectively in an explicit message-passing paradigm. We therefore believe that scalable multiprocessors should provide efficient support for a cache-coherent shared address space if they target computer graphics and visualization as one of their application domains. We believe that such general-purpose machines will be very effective at realizing real-time or interactive-time visualization of interesting data sets in the future. We have shown that they can already do this for volume rendering using the new shear-warp algorithm, which is a very encouraging result since it does not rely on special-purpose hardware.

Acknowledgments

We would like to thank several people for implementing and helping with implementations of the parallel versions: Takashi Totsuka, Jim Christy, Jason Nieh, Philippe Lacroute, Maneesh Agrawala, and David Ofelt. This research was funded by ARPA under Contract No. N00039-91-C-0138. Anoop Gupta is also supported by an NSF Presidential Young Investigator Award.

References

[1] Pat. Hanrahan, D. Salzman and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. *Proc. SIGGRAPH*, 1991.

[2] Jason Nieh and Marc Levoy. Volume Rendering on Scalable Shared Memory MIMD Architectures. *Proc. Boston Workshop on Volume Visualization*, October 1992.

[3] Phillipe Lacroute and Marc Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Proc. SIGGRAPH*, 1994.

[4] Daniel E. Lenoski et al. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th Annual International Symposium on Computer Architecture*, pages 148-159, 1990.

[5] Helen Davis, Stephen Goldschmidt and John L. Hennessy. Multiprocessor Simulation and Tracing using Tango. *Proc. Intl. Conf. on Parallel Processing*, August 1991.

[6] Edward Rothberg, Jaswinder Pal Singh and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity for Large-Scale Multiprocessors. In *Proc. 20th Annual International Symposium on Computer Architecture*, 1993.

[7] Michael Cohen et al. A progressive refinement approach to fast radiosity image generation. *Proc. SIGGRAPH*, 1983.

[8] Alan G. Chalmers and Derek J. Paddon. Parallel processing of progressive refinement radiosity methods. *Proc. Second Eurographics Workshop on Rendering*, Barcelona, 1991.

[9] Jaswinder Pal Singh. Parallel Hierarchical N-body Methods and their Implications for Multiprocessors. Ph.D. thesis, Stanford University, Technical Report No. CSL-TR-93-563, February 1993. See also “Load Balancing and Data Locality in Hierarchical N-body Methods”, Technical Report CSL-TR-92-505, Stanford University, to appear in *Journal of Parallel and Distributed Computing*.

[10] Susan Spach and Ronald Pulleyblank. Parallel Raytraced Image Generation. *Hewlett-Packard Journal, vol.* 43, no. 3, pages 76-83, June 1992.

[11] Stuart A. Green and Derek J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, vol. 6, 1990, pp. 62-73.

[12] H. Kobayashi et. al. Load balancing strategies for a parallel ray tracing system based on constant subdivision. *The Visual Computer*, vol. 4, no. 4, pp. 197-209.