Parallel Visualization Algorithms: Performance and Architectural Implications

Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy
Stanford University

Shared-address-space multiprocessors are effective vehicles for speeding up visualization and image synthesis algorithms. This article demonstrates excellent parallel speedups on some well-known sequential algorithms.

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 × 256 × 256-voxel volume data set takes about 5 seconds per frame on a 100-MHz Silicon Graphics Indigo workstation using Levoy's ray-casting algorithm[2] and about a second per frame using a new shear-warp algorithm.[3] These times are much larger than the 0.03 second per frame required for real-time rendering or the 0.1 second 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. 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 (Intel's iPSC/860 and Paragon or Thinking Machines' CM-5, for example), since these tasks have to be performed explicitly by the programmer. The need for explicit communication management leads (1) to complicated parallel algorithms that look very little like their sequential counterparts and (2) to substantial performance inefficiencies.

Recently, a new class of scalable, shared-address-space multiprocessors has emerged. Like message-passing machines, these multiprocessors have a distributed intercon-

0018-9162/94/ $4.00 © 1994 IEEE

nection 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 article, 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 high degree of temporal locality obviates the 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 parallelized ray-casting volume renderer by Levoy,[2] and an optimized ray-tracer by Spach and Pulleyblank.[4] We also discuss a new shear-warp volume rendering algorithm[3] that provides what is to our knowledge the first demonstration of interactive frame rates for a 256 × 256 × 256 voxel data set on a general-purpose multiprocessor. The images used in these demonstrations are shown in Figure 1.

Cache-coherent shared-address-space multiprocessors

Figure A 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.[1]

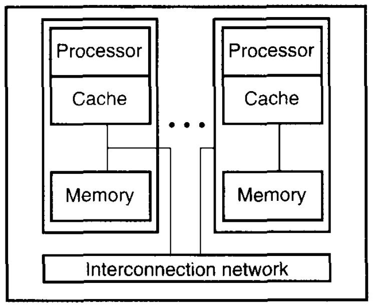


Figure A. The algorithms were parallelized to run on a shared-address-space multiprocessor with physically distributed memory.

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 even though memory is uniformly addressable, it is not uniformly accessible: The cost of accessing a data item increases with the distance the access must travel from the issuing processor. 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 A 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) and the spatial locality provided by multiword cache lines.
- *Memory locality*: If references miss in the cache, we would like to satisfy them in local memory rather than having to communicate across the network. Memory locality can be provided by distributing data appropriately across physical memory units — 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, we want them to be satisfied as close as possible to the issuing processor in the network topology.

Neither memory nor network locality is important in the visualization algorithms we examine in this article. The temporal cache locality that obviates these problems falls naturally out of the spatial coherence in the applications exploited by simple partitioning schemes designed to reduce communication.

In most of our experiments, we use the Dash multiprocessor, an experimental cache-coherent machine built at Stanford University.[1] It has 48 processors organized in 12 clusters, each consisting of four 33-MHz MIPS R3000 processors connected by a shared bus. Clusters are connected in a 2D mesh network. Every processor has a 64-Kbyte first-level cache and a 256-Kbyte second-level cache, and every cluster has an equal fraction of the 256 Mbytes 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 with 16 150-MHz MIPS R4400 processors connected by a 1.2-gigabits-per-second bus to one another and to a centralized shared memory. With centralized shared memory, locality in main memory is not an issue.

Reference

1. D.E. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 2047, 1990, pp. 148-159.

Figure 1. Images rendered by the three applications: (a-b) hierarchical radiosity image and image showing the patches created, (c) volume rendering, and (d) ray tracing. Figures (a) and (b) courtesy of Pat Hanrahan, (c) courtesy of North Carolina Memorial Hospital, and (d) courtesy of Hewlett-Packard.

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 changes.

In traditional radiosity approaches, the large polygonal surfaces that describe a scene (such as walls or a desktop) are first subdivided into small elements or patches whose radiosity is approximately uniform over their surfaces. 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 that 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.

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 $O(n^2)$, and each of these pairs has to be tested for intervisibility, making traditional radiosity methods (including progressive radiosity[5]) 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. Light-transport interactions are computed among these polygons, and polygons are hierarchically subdivided as necessary to improve accuracy. Each

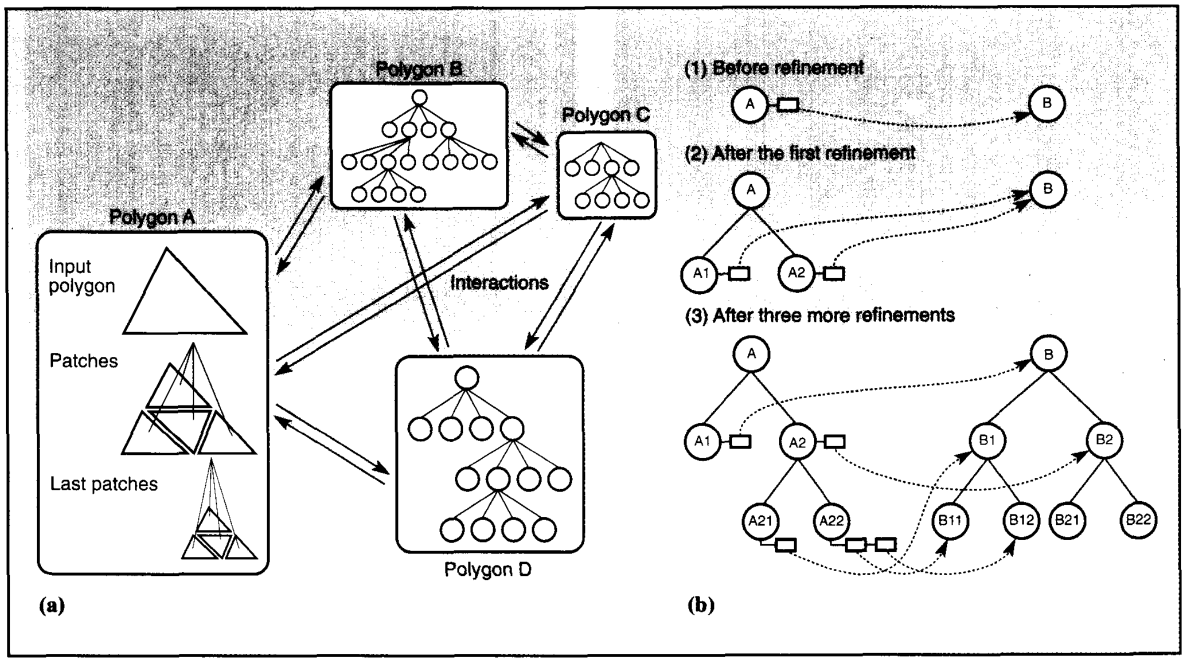


Figure 2. Refinements and interactions in the hierarchical radiosity application. In 2b, we show binary trees instead of quadtrees and only one polygon's interaction lists for simplicity.

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 Hanrahan et al.[1] and in Singh.[6])

Sequential algorithm. The input polygons that constitute 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 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.

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, we compute a patch's (patch i, say) interactions 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 2b 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, a patch's interaction list in any quadtree is exactly as it was at the end of the previous iteration: It contains the patches with which its interaction did not cause a subdivision.

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 and unpredictable. Our parallelization technique therefore uses dynamic task stealing 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.

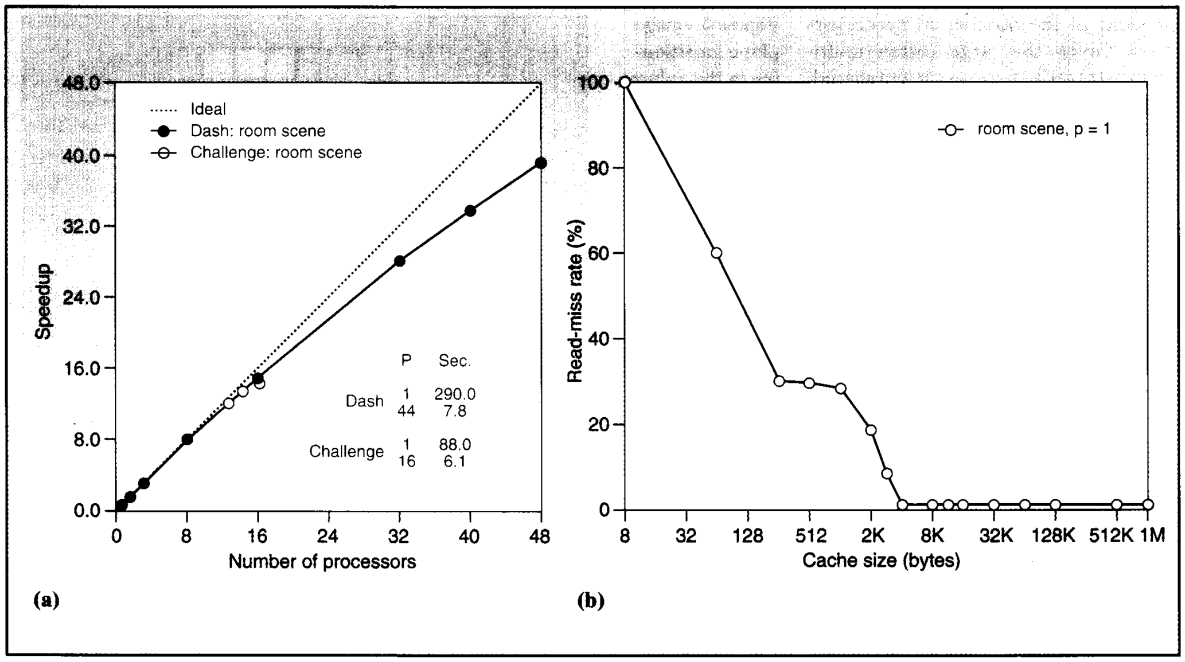


Figure 3. Speedups (a) and working sets (b) for the radiosity application. The BF refinement threshold was set to 0.02 and the area refinement threshold to 2,000 units.[1] The algorithm took seven iterations to converge on a uniprocessor.

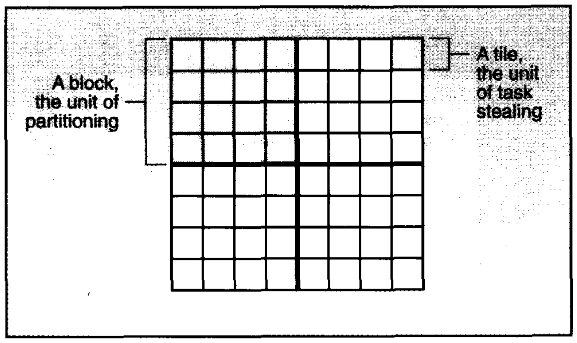
A processor's task queue is initialized with a subset of the initial polygon-polygon interactions. When a patch is subdivided, new tasks involving the subpatches are enqueued on the task queue of the processor that did the subdivision. A processor consumes tasks from its task queue until there are no tasks left. It then steals tasks from other processors' queues, which it can directly access in the shared address space.

While task stealing provides load balancing, it can also compromise data locality. We try to preserve locality as follows.[6] 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).

Results and discussion. This simple parallelization is both conceptually natural and easy to implement in a shared address space. As seen in Figure 3a, it also yields very good speedups on the Dash multiprocessor, even though no attempt was made to distribute (or replicate) data in main memory. (Appropriate data distribution at 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 (data distribution is not an issue here given its centralized shared memory). Because we used the relatively small (174 input polygons) room scene in Hanrahan et al.[1] (Figure 1a), speedups scale more slowly beyond 32 processors on Dash. We expect even better speedups with larger input scenes and that the effectiveness of shared-address-space multiprocessors 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 application's high degree of temporal locality on shared, as well as private, data and the effectiveness of automatic caching in exploiting this locality transparently. To analyze temporal locality, we measure the size and impact of the application's important per-processor working sets. We measure working sets by using a simulated multiprocessor with fully associative caches to plot the read-miss rate versus cache size, following the methodology described by Rothberg et al.[7] Figure 3b indicates a very high degree of temporal locality, given that a 4-Kbyte cache reduces the miss rate to a negligible quantity. We can explain this significant temporal locality as follows.

The algorithm spends most 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. Since occlusion is determined using the BSP tree, the algorithm repeatedly traverses the relevant portion of the BSP tree between the input polygons that are the ancestors of patches *i* and *j*. The processor's next visibility interaction will likely be between patch *i* and a child of patch *j* 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, which is very small compared with 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 large (often multi-megabyte) cache sizes on current shared-address-space machines, there is little chance of encountering problems whose working sets will overflow these caches.

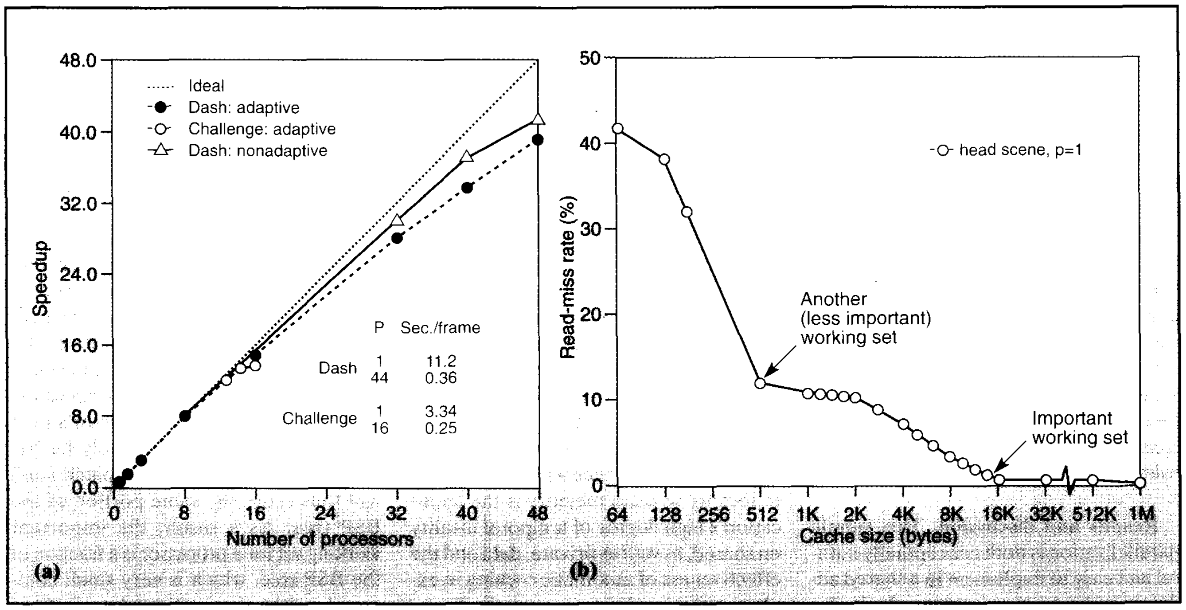

A block, the unit of partitioning
A tile, the unit of task stealing
Figure 4. Image plane partitioning in the volume renderer for four processors.

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 trade-off between concurrency and the shooting approach's need to preserve the sorted order of patches.[6] In fact, gathering has been observed to work better than shooting in parallel even for traditional radiosity algorithms on message-passing machines.[8]

Volume rendering

Volume rendering techniques are very important in the analysis and understanding of multidimensional sampled data, such as those generated in various scientific disciplines. The first parallel algorithm we use, developed by Nieh and Levoy,[2] renders volumes using optimized ray-casting techniques. Until very recently, the sequential algorithm was one of the fastest algorithms known for volume rendering. We then examine a much faster shear-warp algorithm that, when parallelized, can produce interactive frame rates for a rotation sequence of a 256 x 256 x 256-voxel data set on a general-purpose multiprocessor.

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 using digital compositing tech-



	P	Sec./frame
Dash	1	11.2
	44	0.36
Challenge	1	3.34
	16	0.25

Figure 5. Speedups (a) and working sets (b) 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 22,000 rays (245,000 samples) were traced in the case with adaptive sampling and 173,000 rays (618,000 samples) with nonadaptive sampling.

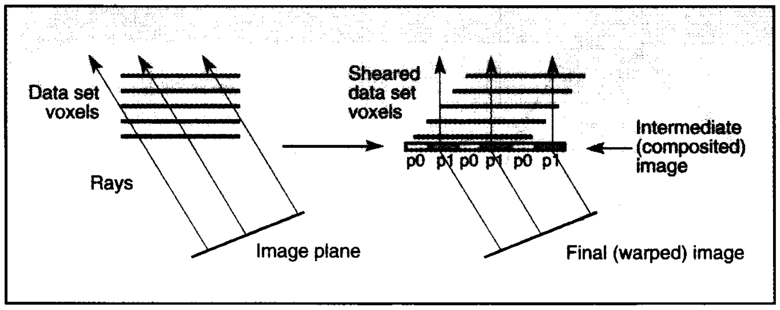


Figure 6. The recently developed shear-warp volume rendering method.

niques 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: (1) the early ray termination mentioned above, controlled by a user-defined opacity threshold; (2) an octree representation of space to avoid unnecessary sampling in transparent regions of the volume; and (3) adaptive image sampling.

Exploiting parallelism. In a shared address space, each 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, both because this is very difficult at page granularity and because it is 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). However, owing to the nonuniformity of the volume data, an equal partitioning of the image plane among processors is not necessarily load balanced, and task stealing is once again required.

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-size 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 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. Figure 4 shows a four-processor example of the image plane partitioning.

Results. Figure 5a shows speedups on Dash for both adaptive and nonadaptive sampling, and on the Challenge for nonadaptive sampling. The results measure rendering time only and do not include the time to load in the data set, compute opacities and build the octree, and transfer the rendered image to the frame buffer. We use a $256 \times 256 \times 256$-voxel data set showing a computed tomography rendering of a human head; the resulting image is shown in Figure 1c. The image measures approximately 415×415 pixels, and the total data set size is about 30 megabytes. A tile size of 8×8 pixels is 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 at partition boundaries with adaptive sampling,[2] the speedups in this case are somewhat less than with nonadaptive sampling. On a 48-processor Dash or a 16-processor Challenge, we are able to come within a factor of three of interactive rendering.

As in the radiosity algorithm, the observed speedups on Dash are very good even though we simply distribute data round-robin among physical memories. Figure 5b shows once more that the speedups result from the high degree of temporal locality on private as well as 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 is possible because of the 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) — that is, to the cube root of the data set size. In addition, the push in volume rendering is toward real-time rendering rather than rapidly increasing data set sizes. The important working set for this algorithm is therefore likely to remain small for some time to come.

Interactive frame rates with the parallel shear-warp method. A new shear-warp algorithm has recently been developed that can render a 256-cubed-voxel data set in one second on a Silicon Graphics Indigo workstation.[3] We have parallelized this algorithm on Dash and the Challenge.

The shear-warp algorithm proceeds in two phases. It first factors the viewing transformation into a 3D shear parallel to the data slices and projects the data to form a distorted intermediate (composited) image. Then it performs a 2D warp on the composited image to produce a final undistorted image. Unlike the image-order ray-casting algorithm, this is an object-order algorithm that streams through slices of the sheared volume data set in front-to-back order and splats voxels onto the corresponding pixels in the composited image. In contrast to the ray-casting approach, volume shearing has the attractive property of exploiting spatial cache locality (with multiword cache lines) in both the object and image data. The algorithm uses run-length encoding, min-max pyramids, and multidimensional summed area tables to achieve its efficiency without sacrificing image quality. Its phases are depicted in Figure 6.

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

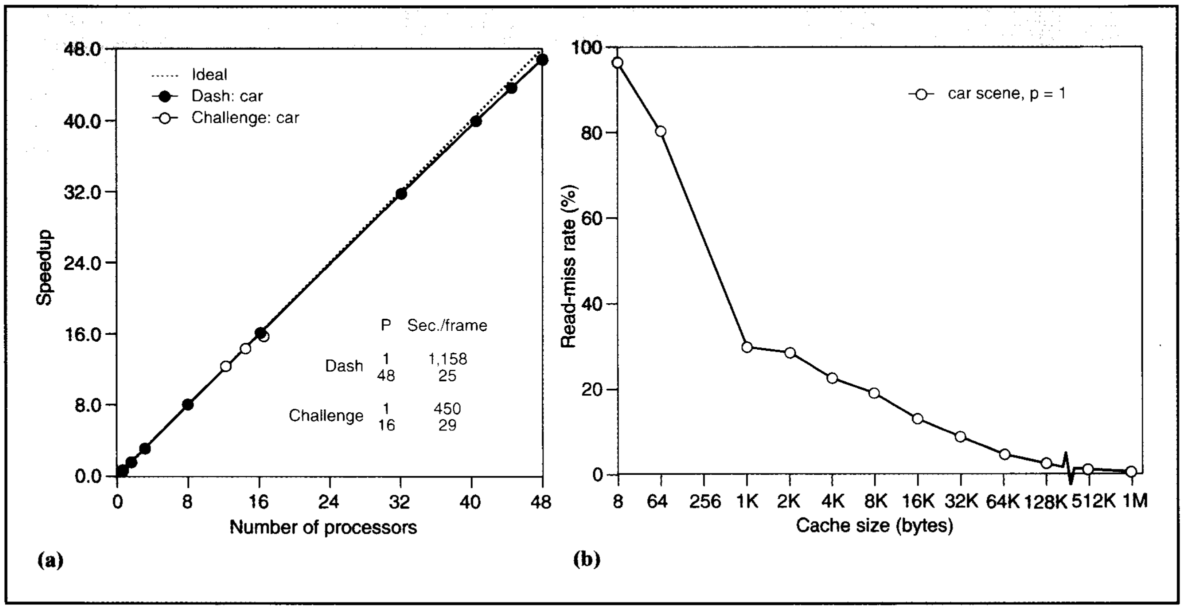


Figure 7. Speedups (a) and working sets (b) for the ray tracing application. The hierarchical uniform grid is subdivided to a maximum of three levels, with five-way subdivision in each dimension and a maximum of 60 primitive objects per leaf cell of the hierarchy. The size of a tile (the unit of task stealing) is 8 × 8 pixels.

(due to the shearing of the voxel data set) and synchronization would be required to ensure mutual exclusion when updating pixels and to preserve dependencies between processing slices of the data set. The composited image is divided into groups of scan lines (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 (Figure 6 shows the partitioning for two processors). Instead of streaming through a full 2D 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 that correspond to one group of image scan lines 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 scan lines, it processes the other groups it is assigned similarly, and finally steals groups from other processors. The 2D warp is also partitioned in groups of scan lines, by partitioning the final warped image among processors this time.

This parallelization achieves good speedups and lets us obtain interactive-rendering rates of 12 frames a second for a rotation sequence on a 256-cubed-voxel human-head data set. These speeds were obtained on a general-purpose, 16-processor Challenge machine (a single processor takes about 1 second per frame). Thus, both image-order and object-order volume-rendering algorithms can be parallelized effectively on cache-coherent multiprocessors.

Ray tracing

Our final application is an optimized ray tracer. The ray tracer was originally developed by Spach and Pulleyblank[4] 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.

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. When it encounters an object, the ray is reflected toward each light source to determine whether it is shielded from that light source and, if not, to compute the contribution of the light source. 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 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 not necessarily with binary subdivisions) is used to traverse scene data efficiently,[4] and early ray tracing and adaptive sampling are implemented.

Exploiting parallelism. Like the ray-casting volume renderer, the ray-tracing algorithm affords substantial parallelism across rays, and the scene data is 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 is almost identical to the one used for the ray-casting volume renderer, with a similar distributed task-queue system for load balancing.

Results. Figure 7 shows the speedups for the parallel ray tracer. The scene being rendered is a car on a checkerboard floor, as shown in Figure 1d, and the image has 512 × 512 pixels. The data set size is about 10 megabytes. No antialiasing is used in these measurements. We obtain excellent speedups without any attention to data distribution.

As in volume rendering, the important working set in ray tracing consists of the data encountered in processing one primary ray (and the tree of rays it generates) that can be reused in processing primary rays cast through neighboring pixels. The difference is that the working set is larger and not so well defined (as compared to that for the ray-casting algorithm shown in Figure 5), owing to the unpredictability of reflections. The working-set size is once again independent of the number of processors. Its 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.

On machines that require main memory to be managed at the granularity of pages and under software control, several characteristics of these applications make it very difficult to manage data distribution and replication in main memory. These include (1) 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; (2) fine-grained data sharing, which makes pages an inappropriate granularity for locality management; and (3) dynamic task stealing. Thus, it is fortunate that caches work well. The 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.

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: (1) the naming of logically shared data, (2) 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 (3) 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.

The cost of hardware support for a cache-coherent shared address space is justified by the ease of programming and performance it affords.

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 is 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 (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 the cost and design complexity of cache coherence. However, recent efforts to build these machines have shown that the costs are quite small. In fact, the cost of the extra main memory needed on message-passing machines for explicit replication of operating system code, application code, and data often dominates the hardware cost of cache coherence. In any case, 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. Below, we 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.)

Ray tracing. The main problems for message passing in the ray tracer are (1) managing the naming, replication, and fine-grained communication overhead issues in sharing the read-only scene data, and (2) 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 parts 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 the problems that can be solved and is not scalable. A reasonable data 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 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.[4,9] Managing the naming and naturally fine-grained communication in both approaches is complex and inefficient compared with using 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,[9] 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 supports 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 irregular applications necessitates a check in software on every reference to scene 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.

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 was 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 to program and incurs greater overheads on message-passing machines. In a survey of message-passing implementations, Green and Paddon[9] mention several attempts to address the load-balancing problem, but not one of them uses task stealing. Instead, they try to prepartition the image and object space intelligently to improve load balancing over a uniform decomposition (see Kobayashi et al.,[10] for example).

We believe that general-purpose multiprocessors will be very effective at realizing real-time or interactive-time visualization.

These complex approaches are input dependent 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 be performed through explicit messages while the processes are in the midst of the main computation.

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 ray tracer and volume renderer, it has two other complicating properties: (1) the main data structures (quadtrees of patches) are dynamically changing, since they are built as the computation proceeds; and (2) 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. Implementations of message-passing versions by graduate students on an Intel iPSC/860 machine have been exercises in frustration and only yielded elevenfold 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 Singh.[6])

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 of which that patch is a part, 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 approach, processors start a time step with local copies of all the data corresponding to their patches and interaction lists. They modify these data by subdividing their and others' patches locally as needed in an iteration, and they 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 extra memory overhead is typically very large. Special data structures also 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 once again 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 message passing. This approach leads to much finer grained communication and to 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, we must decide whether the patch's ownership remains with the old processor or is passed on to the stealer; either possibility complicates 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 only ten- to elevenfold 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 main computation are very difficult to program and debug in message-passing hierarchical radiosity. On the other hand, we have shown that cache-coherent shared-address-space machines solve this problem very well.

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 sequential algorithms, including hierarchical radiosity, ray-casting and shear-warp volume rendering, and ray tracing.

A shared address space allows us to easily implement very natural parallelizations, and 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 harder 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 among their application domains. 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. ■

Acknowledgments

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

References

1. P. Hanrahan, D. Salzman, and L. Aupperle, "A Rapid Hierarchical Radiosity Algorithm," *Computer Graphics* (Proc. Siggraph), Vol. 25, No. 4, July 1991, pp. 197-206.

2. J. Nieh and M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures," *Proc. Boston Workshop on Volume Visualization*, ACM Press, New York, 1992, pp. 17-24.

3. P. Lacroute and M. Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," to be published in *Proc. Siggraph*, 1994.

4. Susan Spach and Ronald Pulleyblank, "Parallel Ray-Traced Image Generation," *Hewlett-Packard J.*, Vol. 43, No. 3, June 1992, pp. 76-83.

5. M. Cohen et al., "A Progressive Refinement Approach to Fast Radiosity Image Generation," *Computer Graphics* (Proc. Siggraph), Vol. 22 No.4, Aug. 1988, pp. 75-84.

6. J.P. Singh, *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*, doctoral dissertation (Tech. Report No. CSL-TR-93-563), Stanford Univ., Stanford Calif., Feb. 1993.

7. E. Rothberg, J.P. Singh, and A. Gupta, "Working Sets, Cache Sizes, and Node Granularity for Large-Scale Multiprocessors," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., Order No. 3810-02T, 1993, pp. 14-25.

8. A.G. Chalmers and D.J. Paddon, "Parallel Processing of Progressive Refinement Radiosity Methods," *Proc. Second Eurographics Workshop on Rendering*, North-Holland, 1991.

9. S.A. Green and D.J. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing," *The Visual Computer*, Vol. 6, No. 2, 1990, pp. 62-73.

10. H. Kobayashi et al., "Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision," *The Visual Computer*, Vol. 4, No. 4, Oct. 1988, pp. 197-209.

Jaswinder Pal Singh is a research associate at Stanford University. His research interests are parallel applications and parallel computer systems. He received his BSE from Princeton University in 1987, and his MS and PhD degrees from Stanford University in 1989 and 1993, respectively.

Anoop Gupta is an associate professor of computer science and electrical engineering at Stanford University. Before joining Stanford, he was on the research faculty of Carnegie Mellon University, where he received his PhD in 1986. His research interests are the design of hardware and software for large-scale multiprocessors. Gupta received the NSF Presidential Young Investigator Award in 1990 and holds the Robert Noyce faculty scholar chair in the School of Engineering at Stanford.

Marc Levoy is an assistant professor of computer science and electrical engineering at Stanford University. His research interests include visualization of multidimensional sampled data, digitization of 3D objects using active sensing technologies, and the design of languages and user interfaces for data visualization.

Levoy received a B. Architecture and an MS from Cornell University in 1976 and 1978, respectively, and a PhD in computer science from the University of North Carolina at Chapel Hill. He was principal developer of the Hanna-Barbera computer animation system, and he received the NSF Presidential Young Investigator Award in 1991.

Readers can contact the authors at the Computer Systems Laboratory, Stanford University, Stanford, CA 94305. Their e-mail addresses are jps@samay.stanford.edu, {gupta,levoy}@cs.stanford.edu.