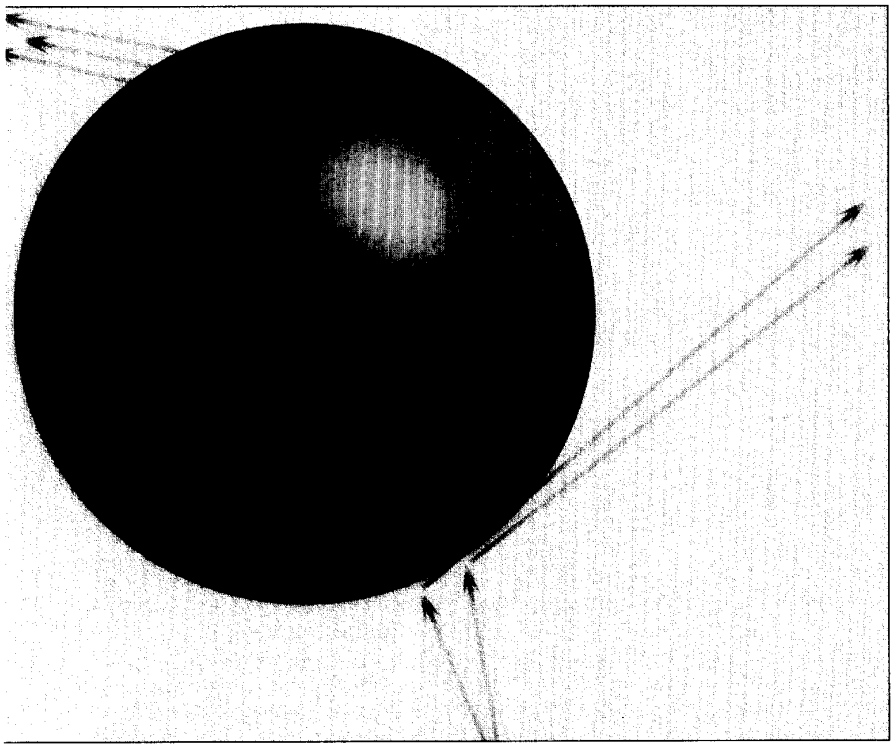
Distributing Data and Control for Ray Tracing in Parallel



Didier Badouel
Ecole des Mines de Nantes

Kadi Bouatouch
University of Rennes and IRISA

Thierry Priol
IRISA/INRIA

Distributed-memory parallel computers offer high performance for ray tracing when supplied with shared virtual memory, which avoid complex message-passing techniques.

Ray tracing a realistic image requires evaluating several million light contributions to a scene described by several hundred thousand objects. This large number of ray-object intersections makes ray tracing very expensive. Attempts to reduce this number have been based on object-access data structures that allow a fast search for objects along a ray path. The data structures are based either on a tree of bounding boxes or on space subdivision.

Ray tracing is intrinsically parallel because each pixel is evaluated independently. But to exploit this parallelism, we must ensure that the load is balanced and that the database is distributed evenly among the processors' memories. The parallelization of a ray-tracing algorithm raises a classical problem in distributed parallel computing: How do we ensure both data distribution and load balancing when there is no obvious relation between computation and data? A schematic ray-tracing algorithm illustrates this problem:

for $i = 1, xpix$ **do**
  for $j = 1, ypix$ **do**
    $pixel[i, j] = \Sigma\ (contrib(..., space[f_x(...), f_y(...), f_z(...)],...))$
  done
done

The computation of one *pixel* is the accumulation of various light contributions *contrib*() depending on the lighting model. Evaluating pixels requires access to a database that models the scene to be rendered. In the ray-tracing algorithm, the database *space* consists of both an object-access data structure (space subdivision) and objects. The data accesses require evaluation of functions f_x, f_y, and f_z. These functions become known only during execution of the ray-tracing algorithm and depend on the ray paths. Hence, before execution, relationships between computation and data are unknown.

In this article, we first briefly describe the methodology of programming ray-tracing algorithms on distributed-memory parallel computers, or DMPCs, and review previous efforts to overcome the problems of data distribution and load balancing. Then we present two algorithms designed for DMPCs and implemented on an Intel iPSC/2. We also compare the results of our experiments with them. The first algorithm, a data-oriented parallel implementation based on message passing, demonstrates how complex designing a parallel ray-tracing algorithm can be. The second algorithm shows how we can eliminate some complexity using a control-oriented parallel approach and a shared virtual memory.

0272-17-16/94/$4.00 © 1994 IEEE

Speeding ray tracing with a DMPC

A DMPC is a multiple instruction, multiple data (MIMD) high-performance computer in which each processor has a local memory to store its own code and data. A network connects all processors. The Intel iPSC, Paragon XP/S, TMC CM-5, Meiko CS-2, and transputer-based architectures are DMPCs. To program these parallel computers, we subdivide the problem into communicating tasks. The lack of general-purpose automatic parallelization tools makes this work difficult. However, there are several programming methodologies we can apply to sequential algorithms.

For *data-oriented parallelization*, we partition the algorithm's data domain into subdomains, each associated with a processor. We assign a computation to the processor that owns the data involved in the computation. If a computation requires other data not located at the processor, the processor sends it to the relevant processors by messages.

The physical distribution of processing elements makes data-oriented parallelization a natural way to use DMPCs. However, we do not know the relationship between computation and data accesses in a ray-tracing algorithm, so finding a data-domain decomposition appropriate for load balancing is difficult.

In *control-oriented parallelization*, we focus on loops. After analyzing loops to discover dependencies, we create a set of tasks representing a subset of iterations. Each task must access a shared database, but DMPC operating systems do not offer shared memory services. Therefore, we must add communication routines that let tasks access remote data. To avoid this step, we can emulate a shared memory on a DMPC to create a *shared virtual memory*.

So far, few experiments comparing these approaches have been conducted. Before analyzing our own experiments, we survey previous work with data-oriented and control-oriented schemes.

Data-oriented parallelization

The two principal techniques for distributing a database among processors depend on the object-access data structure chosen: One approach partitions the scene according to a tree of extents,[1,2] while the other subdivides the scene extent into 3D regions (or voxels).[3-7] In the space subdivision case, rays are communicated from a processor as soon as they leave the region associated with it. We can say these data-oriented techniques *process with ray dataflow*.

Tree of extents

In the distribution of a tree of extents, a technique first proposed by Goldsmith and Salmon,[2] we create a hierarchy of rectangular extents used to compute intersections. Only the upper tree (named the forest by Goldsmith and Salmon) is replicated in each processor. The lowest levels of the forest are pointers to subtrees of the entire hierarchy: A pointer provides access to the processor storing the appropriate part of the database. Thus, each processor controls a subset of pixels together with the forest and a subtree of extents corresponding to a portion of the database.

To compute a pixel's color, a processor shoots a primary ray and follows it through the forest down to the lowest levels, the pointers. If this traversal leads to a pointer to this processor, it performs the necessary computation. Otherwise, it sends the ray to the concerned processor.

A drawback of this approach is that load balancing is based only on the distribution of the pixel array (this problem has been addressed elsewhere[1]). Algorithms based on a tree of extents require less memory than those based on a spatial subdivision, but they require more execution time and numerous floating-point operations for traversal of the hierarchy (the forest and subtrees associated with the database portions).

Spatial subdivision

To use spatial subdivision as the object data structure, we assign each processor one or more regions, with each region containing a part of the whole database and a data structure describing a spatial subdivision[3-8] (for example, an octree or a grid). Processors exchange rays via messages.

Several strategies have been proposed for load balancing.[4,7] It can be accomplished dynamically: Redistribution messages move the boundaries of the regions allocated to processors. Besides the numerous floating-point operations and increased message traffic this approach involves, it raises difficult issues. Dynamic load balancing proceeds by moving a corner or a face. Which corner or face should be selected first? How does the algorithm behave when all processors adjust their loads at once? What is the periodicity of the load redistribution, which might cause oscillations? How do we avoid deadlock caused by heavy message traffic?

Static load-balancing strategies[6,8] try to avoid such drawbacks by allocating neighboring 3D regions to nonadjacent processors. The computational load in a scene extent tends to concentrate in a local space. We can split this space into several regions, distribute them among nonadjacent processors according to a mapping function, and use a 2D torus architecture.

Control-oriented parallelization

The simplest way to parallelize ray tracing is to *process without dataflow:* to duplicate the entire object data structure in the local memory of each processor.[9-11] A sequential process running in each processor is in charge of computing a subset of pixels dynamically determined according to the processor's load. This sequential process is almost the same as one that would run on a sequential computer. Because of its simplicity, many DMPC demonstration ray-tracing algorithms use this technique. However, the limited size of each processor's local memory prohibits its use for rendering complex scenes. We cannot use efficient algorithms such as space tracing because the object-access data structure would require a very large memory.

An alternative approach—emulating a shared memory—uses algorithms that *process with object dataflow*. Green and Paddon's algorithm[12,13] is embedded in an architecture organized as a tree of transputers. Each transputer uses its local memory as a cache

Table 1. Classification of parallel ray-tracing algorithms.

Type of parallelism	Communication	Data Structure
Control	Without dataflow[10]	Tree of extents[9,11]
	Object dataflow	Space subdivision[12,13,15] Bounding volumes[14]
Data	Ray dataflow	Space subdivision[3,4,6-8,16,17] Tree of extents[1,2]

to reduce communication with the other processors (ancestors, parents, and children). The root processor contains the whole database representing the scene. It consists of two parts: an octree description and object descriptions. Each node processor has a copy of the octree description and maintains descriptions of some objects in a local database much smaller than the whole database.

The important feature of this algorithm is that processors must exchange portions of the database to accomplish the ray-scene intersections. Requests for objects pass between processors to ensure a dynamic allocation of objects. Each node processor maintains a stack of rays to be treated, passes requests for work to its parent processor when the stack is empty, and pushes onto the stack the rays spawned from the last popped ray. This approach suits load balancing well.

Green and Paddon[12] also advocated a more general approach. Like the algorithm we propose later in this article, it uses a virtual memory to store the database and a cache mechanism to reduce network contention. In a preprocessing step, the algorithm statically partitions each processor's local memory into two parts: The first part is resident for the local database, while the other acts as a cache. (Our method differs in its dynamic partitioning of the local memories without a preprocessing step.) A similar algorithm has been implemented on the Pixel Machine.[14]

Green and Paddon's algorithm and the Pixel Machine approach locate the virtual memory in the host. In our opinion, this increases the communication cost between the host and the nodes, and can reduce efficiency. In DMPCs it is more efficient to distribute the virtual memory among all the nodes for parallel access. The algorithm we present later emulates a read-only shared memory and distributes it among all the nodes.

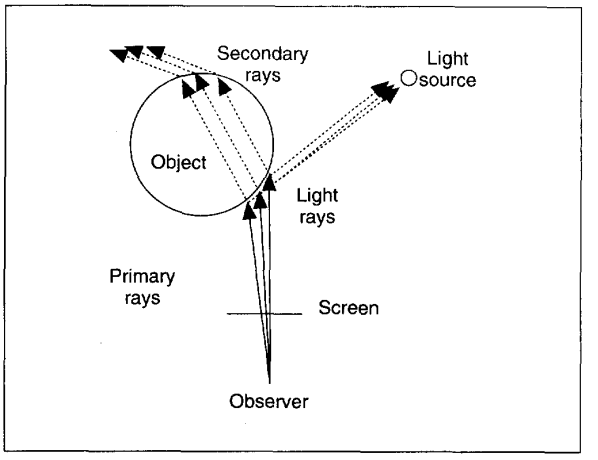
Classification

Table 1 classifies parallel ray-tracing algorithms according to their parallel-programming methodology. In most cases, parallel algorithms have been simulated in sequential environments that do not take into account important features such as communication overhead and synchronization. Moreover, the scenes used in these simulations differ, so we cannot compare the different strategies' performances.

Experiments with data-oriented parallelization

We evaluated a parallel ray-tracing algorithm based on processing with ray dataflow. Because it exchanges rays between processors via messages, it suits DMPCs using a standard message-passing programming methodology. This algorithm resembles one described elsewhere;[17] it differs only in the way it distributes computation and data for load balancing.

Figure 1. Ray-coherence property.



Static load balancing

Cleary's idea[3] of distributing the objects among the processors is elegant, but the computation distribution he proposes is difficult to implement. The load associated with each processor depends on the size of the corresponding subregion and hence on several parameters:

- the relative location of a subregion with respect to the scene extent;
- the number, location, and size of the objects the subregion contains;
- the physical properties of the objects it contains; and
- the number of rays entering it.

Modifying a subregion's size requires changes in these parameters and consequently in the corresponding load. We cannot precisely predict the synthesis time for one region because of the number of parameters involved in the computation. Dippé and Swensen[4] and Nemoto and Omachi[7] suggested balancing the load dynamically during computation, but this technique has never been implemented. If a static load-balancing technique fails, a dynamic approach might be the last chance to improve the performance of a parallel algorithm. However, in this section we present a static load-balancing technique for data-oriented parallelization that avoids the use of a costly dynamic scheme.

Using the ray-coherence property, we can estimate the load associated with a 3D region. As Figure 1 shows, two rays shot from the observer through two adjacent pixels have a high probability of intersecting the same objects. This property is also true for all rays spawned from the two primary rays. Consequently, the time needed to compute a pixel is close to that of its neighboring pixels. With this ray-coherence property, we can subdivide the scene extent into equal-load 3D regions in three steps:

1. Subdivide the scene extent into cells using a 2D grid, as shown in Figure 2a (next page).
2. Subsample the image so only a subset of pixels is computed. Associate a counter with each cell as an estimate of the time needed to treat all the rays that enter the cell.

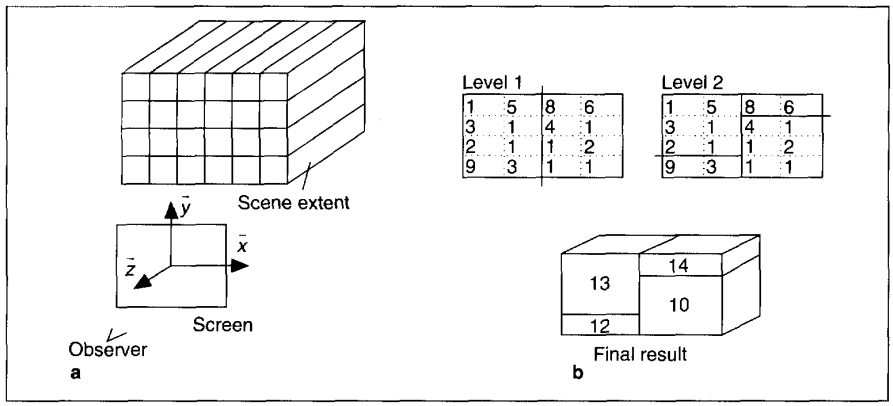


Figure 2. Subdivision of (a) the scene extent and (b) clustering cells.

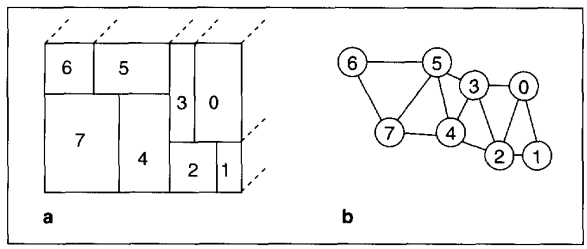


Figure 3. Communication: (a) 3D adjacent regions, (b) connectivity graph.

3. Cluster cells according to the counters' values to get equally loaded 3D regions (see Figure 2b). There should be as many regions as processors.

For clustering, binary space partitioning[18] recursively subdivides the scene extent using median planes perpendicular to the x and y axes of the screen coordinate system. The technique is convenient because it provides 2^N regions, which is the number of processors in a hypercube topology. Figure 2b shows a partitioning of depth 2. The subdivision provides a set of adjacent regions.

We can model these regions' adjacency (or connectivity) with a connectivity graph whose vertices are 3D regions and whose edges represent the adjacency relation. Since each region is associated with a process performing a ray-tracing operation, the connectivity graph is a graph of processes. In Figure 3, a vertex represents a process and an edge the communication between processes. We map this connectivity graph on a real architecture, perform subdivision to produce a number of processes equal to the number of processors, and place communicating processes on neighboring processors.

Algorithm overview

The algorithm consists of two processes. The *host process* achieves static load balancing by subdividing the scene extent into 3D regions. Then, according to a mapping strategy, it assigns to each processor a region and a portion of the database, as well as a part of the pixel array. Then all the processors subdivide the region for which they are responsible. When they finish, they notify the host, which sends them a message to start the synthesis phase. Finally, the host waits for pixel-intensity-contribution messages from the nodes and updates the frame buffer when they arrive.

A *node process* performs two tasks: synthesis and communication. During synthesis, the node subdivides into 3D cells the region provided by the host. Then it shoots the primary rays through the portion of the pixel array it is responsible for. These rays can give rise to secondary rays, which the node passes to other nodes by putting messages into a sending queue. To avoid queue saturation, this synthesis task treats, by priority, the rays sent from other processors, which are stored in a receiving queue. When the synthesis task wants to put a ray message in the sending queue, it calls the communication task, which sends the oldest ray message to avoid saturation. After it computes the fraction of pixel intensity, the synthesis task sends it to the host, which stores it in the frame buffer.

Deadlock

Two kinds of deadlock may occur during the execution of a distributed algorithm. The well-known problem of a process waiting for a message that will never be transmitted does not occur with our algorithm because it uses only asynchronous communication routines. Each process knows whether a message is pending in its receiving queue before it executes the reception routine that could block the process until it receives the message. If there is no pending message, a process makes its own local computation or executes a distributed termination detection algorithm.[17]

The second kind of deadlock occurs with asynchronous communication: Message queues saturate memory resources. The operating system in each processor manages message sending and reception. It temporally stores messages in a queue until they are read by the local process or sent through the network to another processor. Figure 4 shows how a deadlock can develop. If the queue of processor P_1 is full, then P_1 cannot receive messages because they might create new rays that P_1 could not save in its queue. Suppose processor P_0 sends shadow ray messages to P_1 by saving them in its queue. After a while, P_0's queue becomes full because P_1 cannot receive these messages. Then we have a deadlock.

To avoid deadlocks, we use an "optimistic" strategy. When a processor's queue is full, it sends ray messages to the host instead of the concerned processors. Then the host processor saves these messages in its own memory and forwards them to the concerned processors as soon as possible. Since the host processor has a large (virtual) memory, it can receive many messages without saturating its queue.

We call this strategy optimistic because we think that generally processors' message queues do not become saturated most of the time. Hence, the overhead in message passing between processors and host is low. Our experiments confirm this: For

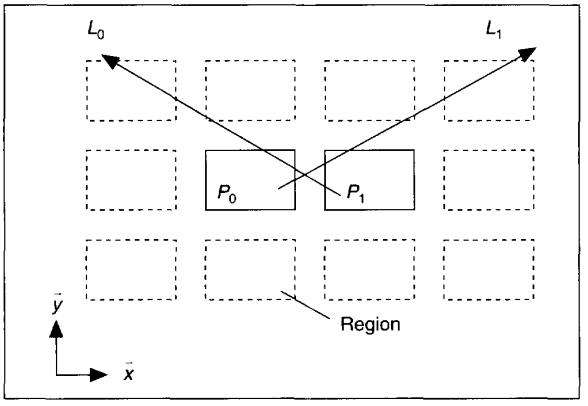


Figure 4. A deadlock situation.

some test images the host received only about 0.5 percent of all the exchanged messages.

Results with data-oriented parallelization

We tested our parallel algorithms using scenes in Eric Haines' Standard Procedural Databases. To evaluate performance, we used these definitions: *Speedup* is the ratio of a processor's running time to the running time obtained with p processors. This quantity represents the number of processors effectively used during the algorithm's parallel execution. *Efficiency* is the ratio of the speedup to the number of processors. It represents the average utilization of the processors.

Figure 5 shows the algorithm's speedup and efficiency for different scenes and different numbers of processors. With two- and four-processor configurations, we could not compute the efficiency for the scene Rings4 because the local memory of each processor is not large enough for the ray-tracing code and its database.

The data-oriented algorithm's efficiency decreases with increased processors for two reasons:

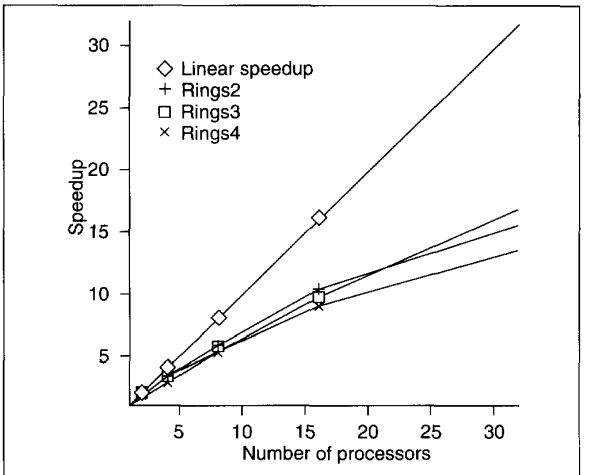
1. *Increased computation with more regions.* When a processor receives a ray message, it must determine the entry cell using floating-point computations. The cost of these computations increases with the number of regions.
2. *Increased communication-to-computation ratio.* Increasing the number of processors decreases the size of regions, reducing synthesis time but dramatically increasing message traffic.

For the Rings3 results, we achieved static load balancing by using a subsampling rate of one pixel in a grid of 8×8 pixels. This represents 1.5 percent of the total image. With this load-balancing technique, the average load of the processors was about 82.5 percent. The average load differs from efficiency because of communication overhead.

With a technique that subdivides the scene extent into equal-size regions and assigns one region to each processor, the average load was less than 20 percent. Hence, the first method is more efficient. Nevertheless, we found three drawbacks in both approaches:

1. Our algorithm's behavior depends on empirical choices such as the size of the subsampling grid and the 2D grid for the partitioning into equally loaded regions.

Figure 5. Speedup curves and efficiency in percent for Rings images with data-oriented parallelization.



PEs	1	2	4	8	16	32
Rings2	100	90	86	74	65	49
Rings3	100	96	85	69	61	53
Rings4	100	—	—	64	53	43

2. Several regions resulting from a spatial subdivision may share the same object, implying repeated intersections with this object when the ray moves from region to region. A solution to this problem of shared objects uses a data structure called "mail box."[19] We could embed this technique in our algorithm, but it would add overhead for each ray-message exchange.
3. When a light source lies in a region assigned to a processor, uniform load distribution becomes difficult. Since database partitioning is performed statically, this processor receives a lot of shadow ray messages; network contention might result.

Partitioning the database statically for a parallel ray-tracing algorithm is possible but complex. Hence we propose another parallel algorithm without the drawbacks of the data-oriented algorithm.

Experiments with control-oriented parallelization

The complexity and inherent limitations in message-passing implementations like the data-oriented algorithm led us to experiment with a shared virtual memory. In fact, this paradoxical approach can ensure efficient data distribution and access by all nodes to the entire database. Then the algorithm can balance the load dynamically during execution with low overhead.

Distributing computation

The distribution of computation must ensure that all processors do roughly the same amount of work. Two ways to distribute the pixels among all the processors are *static scheduling* and *dynamic scheduling*. Static scheduling subdivides the screen

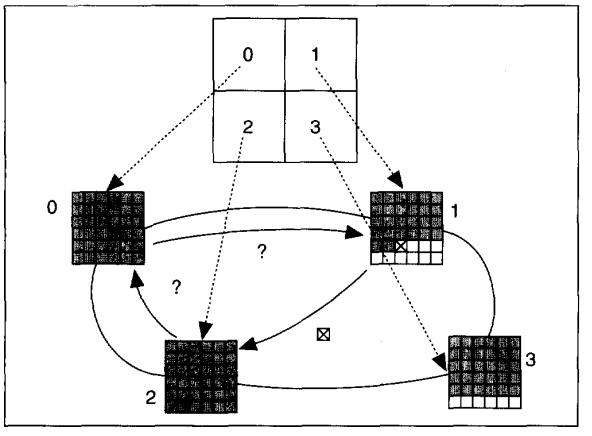


Figure 6. Pixel-map distribution and dynamic load balancing.

into as many pixel areas as processors. Each processor is responsible for computing all pixels belonging to its pixel area. This approach is not effective because pixels can have different computation times, resulting in unequally loaded processors.

Dynamic scheduling dynamically assigns pixels to idle processors. As soon as a processor finishes computing a pixel, it asks for a new pixel. This solution ensures a balanced load but cannot exploit ray coherence to reduce the number of remote data accesses. Because dynamic scheduling does not ensure that the same processor treats neighboring pixels, we combined the two scheduling techniques: Static scheduling reduces the number of remote data accesses, while dynamic scheduling balances the load.

Figure 6 shows four nodes, each with a part of the pixel map. If we use the four nodes to compute an image with a 512×512-pixel resolution, each node manages a 256×256-pixel submap. We use a square (or nearly square) pixel submap to exploit as much as possible the ray-coherence property. When a node finishes computing its pixel submap, it sends a request for a *work item* (a set of pixels) from a node still working on its own pixel submap. This request moves along a ring. If it returns without satisfaction, the node concludes that the image has been computed. This local termination detection is sufficient for our application.

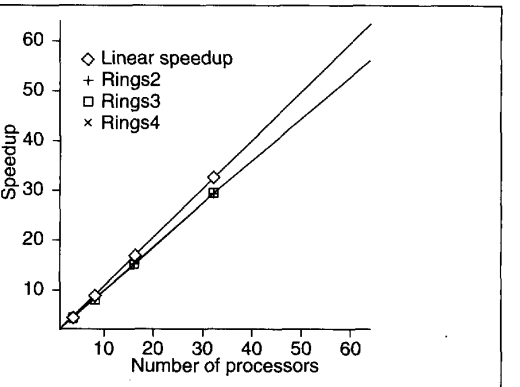
To ensure a balanced load, the only parameter we must determine is the size of a work item. If the size is one pixel, then we have the best load balance we can obtain, assuming that the computation of one pixel is indivisible over the set of nodes. However, minimal work-item size increases communication cost. Experimental results showed that a work item of about 3×3 pixels is a good compromise and achieves load balancing without excess communication.

Distributing data

We used the local memories of a MIMD computer's nodes to distribute a shared virtual memory over all the processors. Each node can virtually access data in a shared memory whose address space is far larger than that of a local memory. If the data a node must access are in its local memory, then the data access is local; otherwise, it is remote. To reduce the expense of remote access, we implemented a cache mechanism.

The database contains the physical and geometric parameters of the scene's primitive objects (polygons in our implementation), together with the object-access data structure (a 3D grid) and the textures. Shared by all the processors, this database is stored in the virtual memory. To manage the shared virtual

Figure 7. Speedup curves and efficiency in percent for Rings images with control-oriented parallelization.



PEs	1	2	4	8	16	32	64
Rings2	100	95	93	91	91	90	89
Rings3	100	95	92	91	90	89	88
Rings4	100	95	93	92	91	90	88

memory, we use object paging, in which an *object* (a polygon, a voxel of the grid, or a texture) is an item of a *page*, the unit of data exchanged between local memories. The paging mechanism does not depend on the types of objects stored in this memory. An object belongs to one and only one page. The shared virtual memory is then a set of pages containing polygons, voxels, or texture elements.

At the beginning of execution, the database is evenly distributed over the nodes without any particular strategy. All nodes' memories contain about the same number of pages. Each local memory is divided into three parts: one containing the code, another to store a part of the database, and a free space for a cache memory to store temporary pages received from other processors. The last two parts are divided into pages managed by shared-memory routines.

During synthesis, the application can potentially access the entire database through a memory-management routine. When a node detects a cache miss (because the page is not in its local database or cache memory), it sends a request to the node responsible for the page. When the node receives the page, it stores the page in its cache according to a least recently used policy (LRU). The search for the page to be overwritten is done during the communication of the new page, at no extra cost.

The classical mechanisms of paging and caching have two advantages. They dynamically exploit data locality, reducing the number of remote data accesses. Moreover, the portable environment they provide simplifies the parallel code and offers a basis for parallelizing other algorithms.

Results with control-oriented parallelization

Figure 7 shows the speedup and efficiency we achieved with the control-oriented algorithm for the Rings databases. If we compare these results with those of the data-oriented algorithm, we see clear improvements.

Table 2. Database characteristics and rendering results.

Database	No. of Polygons	No. of Rays	Shared Memory Size (Kbytes)	Average Pages per Pixel
Teapot	3,754	1,397,473	793	58.19
Mountain	9,920	1,722,415	2,031	61.94
Rings4	18,002	1,872,991	4,632	125.91
Tetra9	262,144	303,239	36,189	17.45
Tetra10	1,048,576	300,962	138,851	33.00

Table 3. Synthesis times for a 512 × 512-pixel image (hours:minutes:seconds).

	Nodes						
Database	1	2	4	8	16	32	64
Teapot	3:12:02	1:39:57	0:51:32	0:26:05	0:13:06	0:06:35	0:03:20
Mountain	4:45:03	2:30:38	1:17:06	0:39:10	0:19:45	0:10:00	0:05:07
Rings4		4:58:46	2:33:24	1:17:35	0:38:56	0:19:41	0:10:04
Tetra9					0:04:55	0:02:26	0:01:18
Tetra10							0:03:46

Tables 2 and 3 show other databases and their synthesis times. We also performed some executions on an iPSC/860: The ratio between the iPSC/2 and the iPSC/860 was about 7.

The control-oriented algorithm's distributed database lets it render scenes like Tetra10, which requires storage far beyond a single node's memory capacity. However, it is difficult to analyze the algorithm's behavior for such large databases because they cannot be handled by a small number of nodes. We have to estimate execution on one node by using other nodes only for memory management. With up to 64 nodes, we almost always achieve an efficiency greater than 78 percent for the databases in Table 2. The table also shows that the average number of pages needed to compute a pixel is small compared with the size of the database.

These encouraging results led us to further analyze our algorithm's behavior. We artificially reduced the cache memory to cause a large cache miss ratio. This increased page communications and decreased algorithm performance. On the next page, Figure 8 shows the performance variation, Figure 9 its correlation with the miss ratio, and Figure 10 its correlation with the page communication rate for various cache sizes. Limiting the cache to only a few pages per node (the left sides of the plots) severely reduced the efficiency of the parallel execution because page traffic on the network increased.

The curves show how performance quickly increases with cache memory size. The more processors can access local data, the quicker they reach maximum efficiency. The two databases in this experiment represent extreme data-locality behaviors: Tetra9 uses a small average number of pages per pixel (see Table 2) and thus can be efficiently rendered by nodes with small caches. Rings4, which uses the greatest average number of pages per pixel, requires a larger cache memory to keep above the efficiency threshold of 50 percent.

The algorithm's behavior depends on the way it exploits data locality. Our results illustrate some representative behaviors depending on two features: large databases and complex rendering effects. When rendering scenes with large databases and few rendering effects, the algorithm efficiently exploits data locality. For databases with complex rendering effects, it does not benefit as much from locality.

The miss-ratio curves in Figure 9 show that using the capacity of all the local memories (about 3.2 Mbytes per node for the shared virtual memory management) prevents network saturation and poor performance. With database Rings4, we get network saturation for a cache of only 204 pages (the peak in Figure 10). The fastest communication rate on the iPSC/2 is 0.9 Mbytes per second when the message size is 1 Kbyte, which is our page size. Note that the iPSC/2's absolute peak communication rate is 2.75 Mbytes per second. The network saturation measured in our application corresponds to about 70 percent of the network peak performance for our message size, or around 23 percent for the absolute peak communication rate. If we further reduce cache size, the communication performance decreases: This network congestion is similar to the "hot spot" that appears in multistage networks using shared-memory architectures.

Conclusion

The parallel ray-tracing algorithms we present here distribute the scene database among the local memories of a DMPC's nodes to render complex scenes. Although so far we have been unable to evaluate their complexity theoretically, we can summarize their advantages and drawbacks.

With the data-oriented message-passing algorithm, load balancing is difficult and expensive. The data-domain decomposition is not obvious when the aim is to balance the workload. Furthermore, with spatial subdivision as the object-access data structure, increasing the number of processors increases the likelihood that several processors will share objects and repeat ray-object intersections, decreasing algorithm efficiency.

With the control-oriented algorithm and a shared virtual memory, load balancing is easy because pixels are dynamically

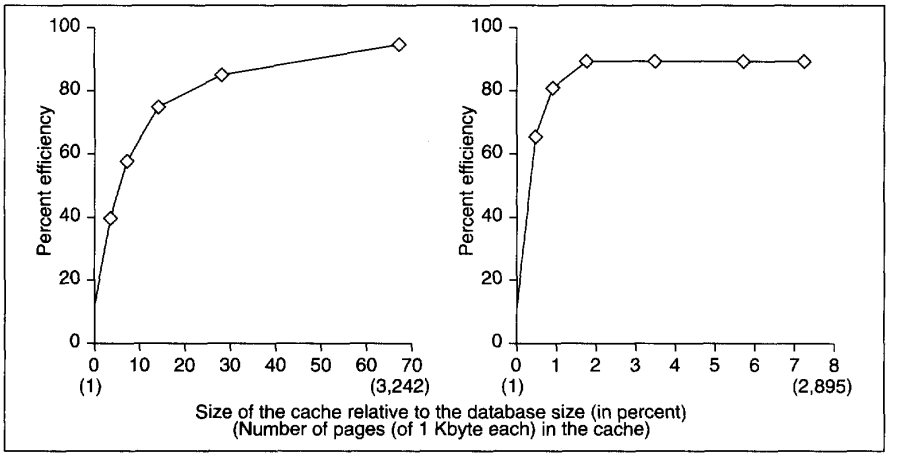


Figure 8. Efficiency and cache size. The plots are for experiments using 32 nodes with 4 Mbytes of memory each. The part of local memory used for the software cache at each node is given as a percentage of the entire database. In raw numbers, the pages in a local cache vary from 1 to 3,242 for the Rings4 database (left) and from 1 to 2,895 for Tetra9 (right). The page size is 1 Kbyte.

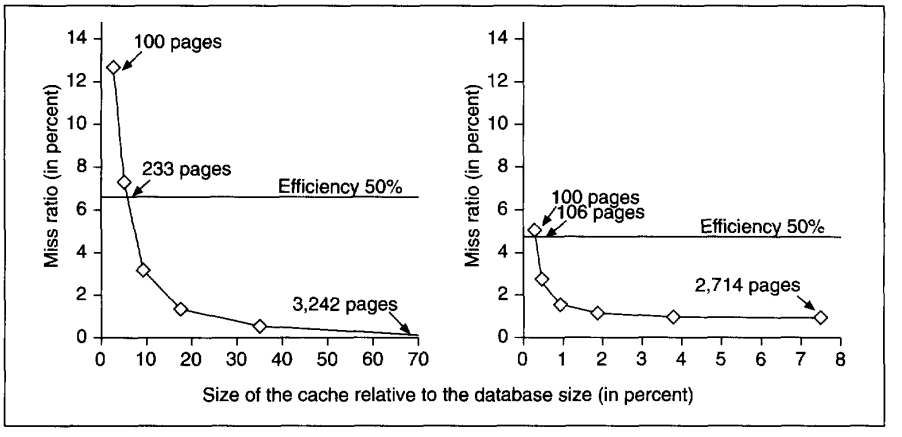


Figure 9. Miss ratio as a function of the cache size for Rings4 (left) and Tetra9 (right).

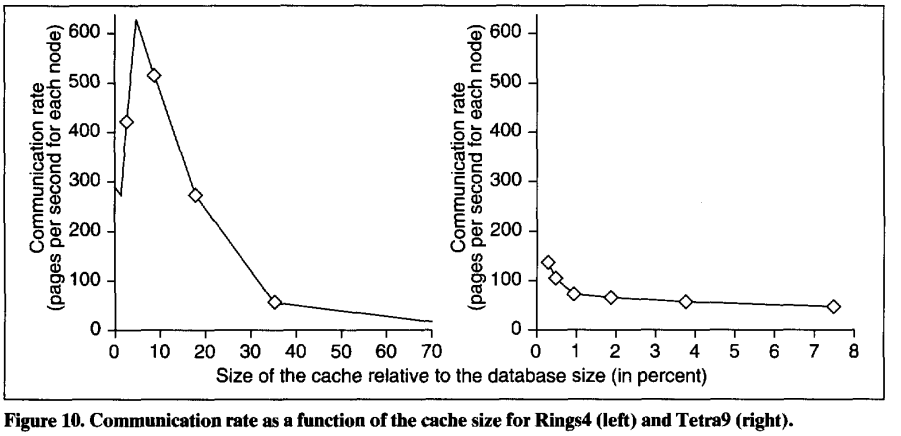


Figure 10. Communication rate as a function of the cache size for Rings4 (left) and Tetra9 (right).

distributed among the processors. All processors access the whole database through the shared virtual memory, and all available memory is used, since the free memory space acts as a cache. The cache exploits the ray-coherence property, reducing remote data accesses. In addition, a light source in a scene is no longer a problem: A processor will frequently use a region containing a light source and therefore store it in cache memory with a low probability of overwriting it.

The implementation of this parallel algorithm is very simple, since each processor runs the sequential ray-tracing algorithm with only a few modifications (pixel assignments). A node computes a pixel's intensity locally without involving other processors.

For small databases fitting entirely in each local memory, a shared virtual memory emulates duplication of these databases in all the local memories. The small extra cost is negligible compared with the computation time. With databases of moderate complexity, a shared virtual memory makes ray tracing easier to implement and faster than with the message-passing approach, since caching reduces the number of exchanged messages. However, with very large databases the cache memory is too small, the number of messages passed among processors increases dramatically, and thus the computing time increases, too. A solution is to increase either the memory available to each node or the number of nodes.

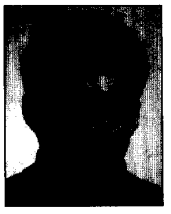
The latest developments in the design of massively parallel computers show shared virtual memory can be integrated either by software (Paragon XP/S) or by specific hardware devices (Kendall Square Research, Convex Exemplar, and Scalable Coherent Interface (SCI) based parallel architecture). Therefore, we could use shared virtual memory and a control-oriented approach to parallelize other computer graphics algorithms such as radiosity. ❑

Acknowledgments

This work has been partly supported by the French Coordinated Research Program C³ and by the CCETT (Centre Commun d'Etudes de Télédiffusion et Télécommunications) under contract 86ME46.

References

1. E. Caspary and I. Scherson, "A Self Balanced Parallel Ray Tracing Algorithm," in *Parallel Processing for Computer Vision and Display*, University of Leeds, UK, 1988.
2. J. Salmon and J. Goldsmith, "A Hypercube Ray-Tracer," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, Vol. 2, Applications, ACM Press, New York, 1988, pp. 1194-1206.
3. J. Cleary et al., "Multiprocessor Ray Tracing," *Computer Graphics Forum*, Vol. 5, No. 1, Mar. 1986, pp. 3-12.
4. M. Dippé and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics* (Proc. Siggraph), Vol. 18, No. 3, July 1994, pp. 149-157.
5. D. Jevans, "Optimistic Multiprocessor Ray Tracing," *Proc. Computer Graphics Int'l*, Springer-Verlag, Tokyo, 1989, pp. 507-522.
6. H. Kobayashi, T. Nakamura, and Y. Shigei, "A Strategy for Mapping Parallel Ray-Tracing into a Hypercube Multiprocessor System," *Proc. Computer Graphics Int'l*, Springer-Verlag, Berlin, 1988, pp. 160-169.
7. K. Nemoto and T. Omachi, "An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing," *Proc. Graphics Interface*, Computer Graphic Soc., Montreal, 1986, pp. 43-48.
8. R. Caubet, Y. Duthen, and V. Gaildrat-Inguimbert, "Voxar: A Tridimensional Architecture for Fast Realistic Image Synthesis," *Proc. Computer Graphics Int'l*, Springer-Verlag, Berlin, 1988, pp. 135-149.
9. C. Bouville et al., "Generating High-Quality Pictures by Ray Tracing," *Computer Graphics Forum*, Vol. 4, June 1985, pp. 87-93.
10. H. Nishimura et al., "Links-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," *Proc. 10th Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1983, pp. 387-394.
11. T. Naruse et al., "Sight: A Dedicated Computer Graphics Machine," *Computer Graphics Forum*, Vol. 6, No. 4, Dec. 1987, pp. 327-334.
12. S. Green and D. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing," *The Visual Computer*, Vol. 6, No. 2, Mar. 1990, pp. 62-73.
13. S. Green, *Parallel Processing for Computer Graphics*, MIT Press, Cambridge, Mass., 1991.
14. M. Potmesil and E. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics* (Proc. Siggraph), Vol. 23, No. 3, 1989, pp. 69-78.
15. D. Badouel and T. Priol, "An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures," *Proc. Eurographics Hardware Workshop*, Springer-Verlag, Berlin, 1989, pp. 93-106.
16. V. Isler, C. Aykanat, and B. Ozguc, "Subdivision of 3D Space Based on the Graph Partitioning for Parallel Ray Tracing," *Proc. Second Eurographics Workshop on Rendering*, Univ. of Catalonia, Barcelona, 1991.
17. T. Priol and K. Bouatouch, "Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube," *The Visual Computer*, Vol. 5, No. 12, Mar. 1989, pp. 109-119.
18. H. Fuchs, Z.M. Kedem, and B.F. Naylor, "On Visible Surface Generation by a priori Tree Structure," *Computer Graphics* (Proc. Siggraph), Vol. 14, No. 3, July 1980, pp. 124-133.
19. B. Arnaldi, T. Priol, and K. Bouatouch, "A New Space Subdivision for Ray Tracing CSG Modelled Scenes," *The Visual Computer*, Vol. 3, No. 2, Aug. 1987, pp. 98-108.



Didier Badouel is an assistant professor at the Ecole des Mines, Nantes. His research interests include computer graphics, performance modeling, and analysis for parallel architectures. Badouel received his diploma in computer science in 1987 from the University of Rennes, France, and his doctoral degree in computer science in 1990 from the Institut de Formation Supérieure en Informatique et Communication (IFSIC), Rennes.

Kadi Bouatouch is a professor at the University of Rennes and a researcher at IRISA (Institut de Recherche en Informatique et Systemes Aléatoires). His research interests include computer graphics and global illumination models. He is an electronics and automatic systems engineer (Ecole Nationale Supérieure d' Electricité et de Mécanique, 1974). He received a doctorate in engineering in 1977 and a higher doctorate in computer science in 1989. He is a member of Eurographics, ACM, and IEEE.

Thierry Priol is a senior research scientist at INRIA (Institut National de Recherche en Informatique et en Automatique). He is designing a parallel programming environment for massively parallel architectures and parallel algorithms for computer graphics. He received a PhD in computer science from IFSIC in 1989 for work in computer graphics and parallel processing. Priol chairs the Intel European Supercomputer Users' Group.

Badouel can be reached at Ecole des Mines de Nantes, 3 rue Marcel Sembat, 64042 Nantes Cedex, France. Bouatouch and Priol can be contacted at IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France. Priol's e-mail address is priol@irisa.fr.