A Data Parallel Algorithm for Raytracing of Heterogeneous Databases

Peter Schröder* **Steven M. Drucker†**
Thinking Machines Corporation
245 First St.
Cambridge, MA 02142-1214

Abstract

We describe a new data parallel algorithm for raytracing. Load balancing is achieved through the use of *processor allocation*, which continually remaps available resources. In this manner heterogeneous data bases are handled without the usual problems of low resource usage. The proposed approach adapts well to both extremes: a small number of rays and a large database; a large number of rays and a small database. The algorithm scales linearly—over a wide range—in the number of rays and available processors. We present an implementation on the Connection Machine CM2 system and provide timings.

Résumé

Cet article présente un nouvel algorithme parallèle pour le lancer de rayons. L'*allocation des processeurs*, qui distribue la tâche aux resources disponibles, permet de garder une charge bien répartie. Ainsi évitons nous les problèmes usuels dus aux ressources de bas niveau tout en manipulant des structures de données hétérogènes. Notre approche s'applique aussi bien à un faible nombre de rayons et des données importantes qu'à un nombre important de lancer de rayons sur des données de taille limitée. L'algorithme fonctionne en temps linéaire en le nombre de rayons et de processeurs disponibles. Nous présentons une implémentation sur le système Connection Machine CM2 et donnons des temps de calcul.

Keywords: Massively parallel, SIMD, raytracing.

Introduction

With the advent of massively parallel computers, many algorithms which require large computing resources have been developed to take advantage of these new machines [2; 20]. While the use of parallel computers, like the Connection Machine CM2 system, has become almost common place in computational fluid dynamics and QCD theories [6], only relatively few graphics algorithms for general purpose parallel computers have been published. We hypothesize that this is due in large measure to the extraordinary success and availability of cheap special purpose graphics hardware.

Most of the work on this hardware (see for example [14; 25; 1; 22; 27]) has focused on polygon scanline rendering. At the same time, the development in rendering algorithms has been towards more sophisticated global shading models. Unfortunately, these can only make limited use of commonly available graphics hardware (e.g. the use of z-buffers for radiosity and raytracing acceleration [9; 31]). There has also been some interest in the use of distributed processing and multi-processing, to accelerate radiosity computations [28; 3], and ray tracing [16; 26; 12]. As the computational needs of sophisticated global illumination models continue to increase and the size of data sets continues to grow (e.g. volume rendering) the importance of algorithms for scalable, general purpose computers will continue to increase.

Applications such as volume rendering, ray tracing, and radiosity have computational needs large enough to take advantage of highly parallel architectures. While, for example, ray tracing appears to parallelize trivially, the published research shows otherwise (see below). Many of the published algorithms struggle with the load balancing issue which becomes worse with increasing numbers of processors. There are also examples—from volume raytracing—where two different algorithms, which are equivalent on serial machines, can have radically different run times on parallel architectures [30]. The Achilles heel of most graphics algorithms on parallel architectures is their high demand for general, versus regular, communication. As the following review of previous work shows, the main focus in this area of research is on minimizing general communication.

In the next section we discuss in detail previous work in parallel raytracing. Following that we describe our new algorithm and present timing results, finishing off with conclusions. The details of the implementation are described in the appendix.

Previous work in parallel ray tracing

Most previous work concerns the mapping of ray tracing onto MIMD hypercubes. Carter and Teague [8] discuss a simple scheme of replicating the entire database at every

*Author's current address: Department of Computer Science, 35 Olden St., Princeton University, Princeton, NJ 08544-2087, e-mail: ps@cs.princeton.edu

†Author's current address: Media Laboratory, 20 Ames St., Massachusetts Institute of Technology, Cambridge, MA 02142, e-mail: smd@media.mit.edu

node of an iPSC/2 computer. The speedup factor gained by the use of many processors is almost exactly the number of processors in the system. The only issue is the assignment of pixels to processors. For this Carter and Teague use the obvious "pixel mod number of processors" *comb* assignment (this same approach was also chosen by [24] for a raytracer on the Pixel machine). Salmon and Goldsmith [29] consider this scheme as well. They go on to consider the more general case of databases that are too large to be replicated at every node. They use a bounding hierarchy data structure for their database and replicate only the top n levels of this structure at every node. In this way the more frequent intersection tests against the top of the hierarchy can be performed locally. The sub trees of level $k > n$ are distributed across the nodes. Considering the statistics of intersection of rays against subtree nodes of the bounding hierarchy, they develop a procedure to determine the height of the subtrees which are deposited across processors.

Another decomposition scheme using a bounding hierarchy was explored in Carter and Teague [7]. The entire bounding hierarchy tree, except for the leaf nodes, is replicated in all processors. The leaf nodes are distributed across the processors. An LRU cache is used to maintain a set of primitives at each processor. Using a blocking scheme to distribute pixels across processors they naturally take advantage of the implied spatial coherence. Load balancing is achieved by a master process which allocates pixel blocks to idle processors.

A different set of algorithms uses SIMD architectures, such as the Connection Machine CM2 system, for *data parallel* ray tracing. While some of the design issues are similar, for example keeping global communication to a minimum, others are quite different. In data parallel algorithms, there is only one thread of execution, and load balancing takes on a new meaning. Instead of distributing possibly different parts of the algorithm across processors, load balancing aims to keep the set of active processors as large as possible for every part of the algorithm.

Delany [11] considers the case of a bounding hierarchy and its efficient traversal. The tree structure is mapped onto processors using the numbering of a preorder traversal of the tree [4]. Assuming that the bounding volumes are the same as the objects to be ray traced, nodes and leaves are of the same type. As is the case, for example, in sphere databases. The algorithm assigns each ray to a processor and at every iteration uses general communication to fetch the next bounding volume as the rays traverse the tree. Using the Euler tour numbering it becomes simple for each ray to find the processor address of the next tree node to intersect with.

Delany has also described a raytracer based on a space subdivision scheme [10]. In this algorithm all objects and rays receive a tag, indicating which leaf node in a uniform octree partitioning of space they currently reside in. The least significant bit of the tag is used to differentiate between rays and objects. The algorithm proceeds by sorting rays and objects based on their tags. As a consequence, rays end up immediately adjacent (in processor space) to objects that they need to intersect with. Using the *scan communication* primitives[1] object data can be efficiently propagated to all rays which occupy the same voxel as the given object. By doing backward and forward scans on the bits of the tags in the sorted list of rays and objects, it is possible for a ray to compute the next non-empty voxel along its direction to within a power of 2. In this way rays are advanced to the next parent voxel in the octree that contains another object and the closest intersection can be found in logarithmic time in the length of the ray.

Discussion

The MIMD hypercube algorithms are generally characterized by their difficulties with load balancing. For example Carter and Teague [8] replicate the entire database at every node. This leaves the distribution of rays to even out the load. For large numbers of rays—small numbers of processors in the system—the simple *comb* assignment does well. As the number of rays per processor decreases, though, the load imbalance goes up markedly. Furthermore we do not consider the replication of the entire database at every node feasible as this limits the size of databases and leads to ever increasing waste of memory as the number of processors increases. Salmon and Goldsmith [29] distribute their database across the processors and use a static analysis to determine how best to achieve this. The actual performance numbers they give indicate even higher load imbalance for increasing numbers of processors (almost 40%). This is not surprising since the work estimates, on which the decomposition is based, exhibit large variance for small numbers of rays per processor; as would be the case with increasing numbers of processors. These concerns also apply to the different distribution scheme of Carter and Teague in [7]. Since their distribution approach adjusts dynamically, through the use of an LRU cache, it exhibits somewhat better load balancing behavior. Both approaches assume that communication is *very* expensive and explicitly maintain multiple processes at each node to mask the idle time associated with communication. Since these algorithms exist in a MIMD message passing context they easily deal with databases containing arbitrary mixtures of object types.

The concerns of the SIMD algorithms are typically centered around the fact that all processors need to execute in lockstep. While this simplifies program development greatly, it leaves these algorithms very vulnerable to low resource usage in the face of multiple object types. In particular, Delany's algorithm [11] is even more restrictive in that the bounding objects must be of the same type as the primitive objects themselves. Some flexibility in the object types is afforded by using a class of objects for which ray/object intersections can be ex-

[1] Briefly, *scan* operations execute an operator such as *sum* across an ordered set of elements. These could be in a vector with a vector processor executing the instruction along the length of the vector, or an ordered set of processors with each element in its own processor. In the latter case the *scan* instructions execute in logarithmic time in the length of the set [4].

pressed in terms of a common meta type (e.g. quadric primitives, with planes as degenerate quadrics, etc.). In this algorithm the top of the tree is, at least initially, a communications bottleneck. During the initial steps of the algorithm all rays attempt to fetch data from the processor which holds the root node. This causes high congestion in the router and results in slow execution of the general communication step. As the algorithm progresses this problem is alleviated since the rays start distributing themselves across all nodes of the hierarchy and new rays are added at different times. Notice that in algorithms of this kind coherence can actually be disadvantageous as it implies high congestion in general communication patterns. A more serious problem with Delany's algorithm is the fact that regardless of the object types, terminal and non-terminal intersections require different treatment. Some processors, for example, may need to evaluate the shading model, while others need to execute further intersection tests with the hierarchy. Delany addresses this with the use of a finite state automaton which always goes to the state with the most processors waiting. Klietz [23] describes the use of these ideas in a framework allowing multiple object types—all of which are sub types of a common meta type—and the automatic object hierarchy construction as described by Goldsmith and Salmon [17]. Klietz reports linear scaling of the algorithm in the number of objects in the hierarchy up to approximately 10000 (on a 32k processor CM2 system), beyond which the performance of the general communication falls off markedly.

The other algorithm proposed by Delany [10] exploits the ability of the CM2 to sort very fast [5]. It however also requires a finite state automaton for load balancing the states, e.g. *need to intersect*, *need to advance*, etc. At each step the algorithm enters into that state which has the most processors waiting. For multiple object types the number of states and the potential for each state having only a few active processors in it increases, potentially resulting in low resource usage overall.

The above analysis shows that it is imperative to consider load balancing from the onset and assure that it scales with increasing numbers of processors. The goal of our work was to develop an algorithm which scales with the available resources—number of processors—as well as the problem size. In particular given that there are typically at least hundreds of thousands of rays it should be possible to take advantage of very large numbers of processors. All the MIMD algorithms discussed above assume small numbers of processors ($\ll 1024$). While this is not explicit in the design, it can be seen from the analyses. As the number of processors goes up the load imbalance increases. The data parallel ray tracing algorithms scale much better with the number of processors, but suffer—as do the MIMD—algorithms from unwieldy load balancing procedures. Even though load balancing is addressed load imbalance can still be very high [29]. These issues are furthermore exacerbated in the SIMD case with the use of different object primitives.

In our algorithm we address the load balancing issue, as well as the issues connected with varying object types in the same database through the use of *processor allocation*. The basic observation is that even for small numbers of objects—say, a few triangles among many spheres in the same database—there are typically more rays *potentially* intersecting these few objects, than there are processors in the machine. Hence we can keep the available resources busy if we simply loop over the object types. This requires the algorithm to continually remap the resource usage. To be sure, this remapping requires general communication, but if the amount of computation between remapping steps is significant, the cost of the general communication steps can be amortized. In the performance section below we will argue that the current algorithm achieves this.

A new data parallel ray tracing algorithm

A simple and straightforward method of data parallel ray/object intersection would intersect every object with every ray in a single intersection step and then extract the closest intersection. This is not only very wasteful but also impractical since it amounts to computing the full cross product of all rays and objects. For a database of 2^{13} objects, and 256x256 pixels/rays we would require 2^{29} processors each executing one ray/object intersection in parallel. The algorithm we propose takes advantage of the fact that this cross product can be made sparse by observing that most rays can only *potentially* intersect a small fraction of the entire object database. The raytracing literature is full of algorithms which use, for example, bounding hierarchies, to exploit this sparsity.

The main task then is to develop an effective strategy to derive small lists of candidate objects for each ray and only intersect each ray with the objects on the respective list. Ideally this list contains only those objects which are actually along the ray. In order to find the list of candidate objects for every ray we use a coarse space subdivision of the object database. Specifically we divide the world into equal sized voxels which are coordinate axes aligned. In a preprocessing phase [15], we find all objects that overlap a given voxel and maintain a list of these for each voxel. During the ray casting step all voxels that a given ray pierces are enumerated and used to access the per-voxel lists of objects. All objects retrieved in this way are candidates and are intersected in parallel with the given ray. A *downward min scan* on the computed intersection distance gives the closest object intersected by a given ray.

Our technique of generating candidate sets is not unlike the shaft culling technique introduced recently by Haines and Wallace [19]. They consider ray casting in the context of radiosity form factor estimation. Candidate lists are comprised of all objects overlapping the the convex hull of an origin and destination object. This technique requires that for each ray an origin and destination object is known, as is the case in radiosity. For ray tracing this does not generally hold. Our technique, while coarser, does not depend on knowing such objects and thus also works for general ray tracing. Since we only use flat lists for each voxel our case corresponds to their *open always* strategy.

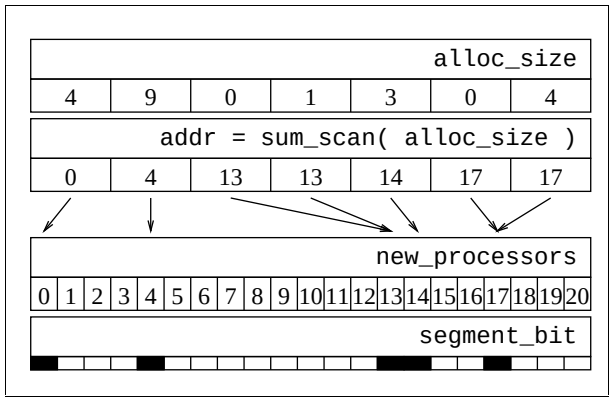


Figure 1: Each box corresponds to a processor in a 1 dimensional processor set. The parallel variable (pvar) `alloc_size` holds the number of processors to allocate. The address of each allocated segment is given by the *sum-scan* of `alloc_size`. The `segment_bit` pvar delineates the actual segments.

The details of the SIMD implementation of the algorithm can be found in Appendix . However, since processor allocation is at the heart of most steps in the algorithm we will give a brief description of this data parallel programming idiom (see also [4]). It is through the use of processor allocation that we achieve the desired load balancing.

Processor allocation

Just as serial algorithms use memory allocation to manage dynamically changing demands, data parallel algorithms use processor allocation for the same ends. Consider, for example, a ray which is to be intersected against a candidate list of objects. Each ray, which is stored in its own processor, typically needs to be intersected against different numbers of candidate objects. In order to exploit ray/object parallelism each ray-processor allocates a number of object-processors. This is accomplished by allocating a new processor set with enough processors to hold the sum total of requested processors. This new processor set is segmented so that each segment consists of as many processors as the associated requesting processor required (see figure 1). The allocating processors receive a pointer (processor address) to the segment allocated to them, which can be used to move data between the allocating and allocated processors. The `segment_bit` can be used in *segmented-scan* operations to execute instructions on a per segment basis. For example, propagating (*segmented-copy-scan*) ray data to all objects which need to be intersected with the given ray. Another typical use in our framework is the *segmented-downward-min-scan* which finds the minimum element of a given pvar on a per-segment basis. The result of this scan is left in the first processor of each segment, whence it can be retrieved easily.

The processor allocation paradigm provides a general way to implement algorithms which dynamically require new resources of uneven length. Another attendant advantage of this approach is the implied load balancing. Since each processor allocates as many new processors as it needs there are no idle processors in the new processor set [2].

Load balancing

When ray tracing in parallel, load balancing is important in two places. First the amount of work necessary to intersect a given ray with the database varies from ray to ray and second the recursion depth of a given ray tree varies based on geometry—rays leaving the scene—and object properties—highly specular versus purely diffuse objects. We use processor allocation as described above to address both of these requirements. For a given set of rays we allocate enough processors per ray to hold the candidate objects from the voxels along the ray. In this way rays that, for example, leave the database will consume less resources than those which are going through an area with many objects. Similarly for each recursion level only those rays which need to be followed further allocate child rays.

This approach is only feasible however, if the router performance is high enough to support the continuous remapping of resources during allocation. On vector computers this corresponds to their ability to perform scatter/gather operations efficiently.

Performance

We use processor allocation extensively throughout the algorithm. Since this involves general communication we need to be sensitive to the underlying communication patterns. Semi-regularity in the routing pattern can lead to very high congestion rates since some form of regularity often implies that many messages need to go through only a few nodes in the communication network. In practice, it has been observed that some of the slowest general communication patterns are very regular, while random patterns tend to yield high throughput. In our case this implies that coherence, which is usually welcome and exploited, can be very disadvantageous. The loading of scene descriptions illustrates this point well. When reading in a database the ordering in the file often correlates with spatial coherence in the database. We have found in those cases that placing the objects into a linear processor set in the order read in, yielded a running time higher then when assigning objects to bit reversed processor addresses. This effect depends on the actual database loaded. For the sphere flake with 7381 spheres and 1 quadrilateral (see [18]) we found the following representative times at a voxel scale of 80^3:

Mach. size	linear load	bit rev. load
16k	208.4 S	170.8 S
32k	118.7 S	98.3 S

The granularity of the database voxelization exerts a much larger influence on the runtime. Two forces need to

[2] There can of course be idle processors if the total number of processors requested is not some integer multiple of the number of physical processors.

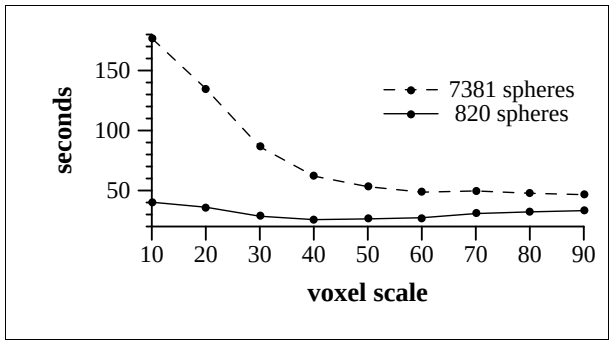


Figure 2: Timings for various voxelization scales. A bounding box for the entire data base is cut into 10^3 to 90^3 voxels. Machine size was 16k processors with an image size of 128^2 pixels.

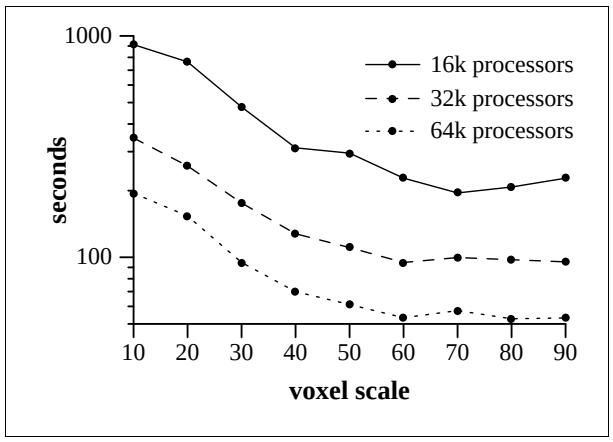


Figure 3: Timings for different machine sizes. All databases are 7381 spheres at an image size of 256^2.

be balanced here. If the voxelization of the world is very fine the candidate lists per ray are fairly small since only those objects overlapping a small neighborhood of the ray will be listed in the object fetch lists. On the other hand a fine voxelization of the world drives up the memory consumption of the system and forces us to subdivide any given ray into many and much smaller pieces. Figure 2 shows timings for two different sphere flake data bases (see [18]) with 820 and 7381 spheres respectively. These databases are good test cases since they contain both very small and very large objects. They also contain a single quadrilateral. The timings include 3 recursive reflection levels and shading via 3 light sources (shadow feelers to all light sources at all intersections). All tests were timed on a CM2 running at 7MHz.

The observed behavior with respect to voxelization scale remains for different machine sizes as well. In figure 3 we see data for the 7381 sphere database for an image size of 256^2 pixels for different machine sizes. When considering larger images with all other parameters the same the runtime scales linearly in the number of rays.

Considerable speedups can be gained from subdividing coarsely along the view direction and finely in the viewing plane. The following table gives one such example for the same scene as above on a 32k processor machine at an image resolution of 256^2

Voxel scale	Time in S	Voxel scale	Time in S
160^2 x 30	95.6	180^2 x 20	101.6
160^2 x 20	90.6	160^2 x 20	90.6
160^2 x 10	97.7	140^2 x 20	95.7

Using our algorithm on various scenes we have found that there always existed an optimal (in the sense of runtime) subdivision granularity. Due to memory constraints this subdivision can not always be attained. Since in most scenes the large majority of rays are view rays it is advisable to transform the database such that the view plane is orthogonal to one of the coordinate axes. By subdividing along that coordinate axis coarsely the incidence of a ray being intersected multiple times with the same object is reduced. At the same time a tight fit—and a small candidate object list—is achieved through the fine subdivision in the image plane. This holds across different machine sizes as well (see figure 4).

Conclusions and future work

We have developed a data parallel ray/object intersection algorithm which scales in the number of processors as well as in the number of rays and objects over several binary orders of magnitude. Through the use of processor allocation every stage of the algorithm is efficiently mapped onto the available resources without requiring explicit load balancing steps. Since the algorithm uses a large amount of general communication, parameters such as subdivision size and ordering of objects in processor space need to be tuned to achieve maximum performance. In particular we observe that the voxelization should be coarse along the view rays and fine parallel to the image plane. The algorithm has been incorporated successfully into a massively parallel radiosity algorithm [13].

In the current implementation the performance is limited, since we do not take advantage of any coherence along the

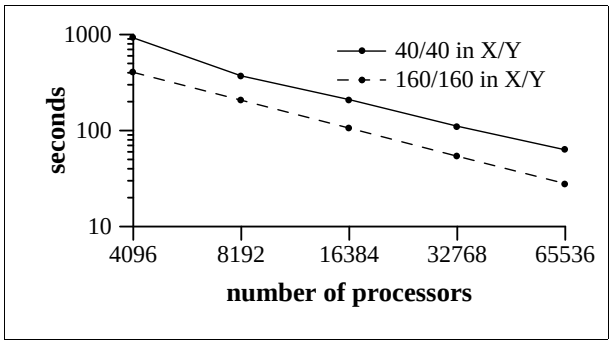


Figure 4: Runtimes for different, weighted voxelization scales. The image size is 256^2 and subdivision in Z is 20 for all runs while the subdivisions in X and Y are 40/40 and 160/160 respectively. The database was the same as in the other examples.

ray. When an object overlaps several voxels along a given ray, the ray will be intersected with it multiple times. Many unnecessary intersection tests also occur because rays are intersected in parallel with all the objects along the ray's length. This implies that there is no notion of stopping as soon as the first intersection is found. Customary acceleration techniques like shadow hit caches, are not currently incorporated into our approach. These shortcomings cannot trivially be addressed in a strict SIMD framework as it is provided by the CM2 Connection Machine System but would be straightforward in a data parallel MIMD framework as given by the CM5 computer, or other MIMD machines. In particular the intersection tests should progress through the candidate lists of each individual voxel in parametric order along the ray and finish as soon as an intersection is found. The problem of a ray intersecting itself multiple times with the same object is not as easy to address. The usual technique of depositing a tag with the object consumes too much memory in our case since all rays are traced in parallel and a possibly very large number of rays can attempt to deposit a tag.

The absolute performance of the current implementation is comparable to the best algorithms on the fastest workstations. This is mostly due to the lack of optimizations in our current implementation. As mentioned above some of these could be added, while others would require the more flexible framework of a MIMD computer. The algorithm also requires a large amount of general communication during load balancing. Current communications networks do not provide enough bandwidth to keep this part of the algorithm from consuming disproportional amounts of time. We expect this to change as communication networks become faster relative to CPU speed. The main feature of our algorithm remains its ability to scale over a wide range of resources and demands.

Acknowledgments

The authors would like to thank Thinking Machines for providing the framework and resources for this work. In particular we would like to acknowledge Lew Tucker, Gary Oberbrunner, Matt Fitzgibbon, Karl Sims, and Jim Salem for many discussions about algorithm design for massively parallel SIMD architectures. Some of the timings in this paper were made on the Los Alamos Advanced Computing Laboratory 64k Connection Machine CM2 computer with the help of Robert Kares.

References

[1] Akely, K., and Jermoluk, T. High-Performance Polygon Rendering. *Computer Graphics 22*, 4 (August 1988), 239–246.

[2] Bailey, J. Implementing Fine-grained Scientific Algorithms on the Connection Machine Supercomputer. Technical Report TR89-1, Thinking Machines Corporation, Cambridge, MA 02142, USA, 1990.

[3] Baum, D. R., and Winget, J. M. Real Time Radiosity through Parallel Processing and Hardware Acceleration. *Computer Graphics 24*, 2 (March 1990), 67–75.

[4] Blelloch, G. *Vector Models for Data Parallel Computing*. Artificial Intelligence Series. MIT Press, Cambridge, MA, 1990.

[5] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings Symposium on Parallel Algorithms and Architectures* (Hilton Head, SC, July 1991), pp. 3–16.

[6] Boghosian, B. M. Computational Physics on the Connection Machine. *Computers in Physics 4*, 1 (January/February 1990), 14–33.

[7] Carter, M. B., and Teague, K. A. Distributed Object Database Ray Tracing on the Intel iPSC/2 Hypercube. In *Proceedings of the 5th Distributed Memory Computing Conference* (1990), D. W. Walker and Q. F. Stout, Eds., vol. 1, IEEE, pp. 217–222.

[8] Carter, M. B., and Teague, K. A. The Hypercube Ray Tracer. In *Proceedings of the 5th Distributed Memory Computing Conference* (1990), D. W. Walker and Q. F. Stout, Eds., vol. 1, IEEE, pp. 212–216.

[9] Cohen, M. F., and Greenberg, D. P. The Hemi Cube: A Radiosity Solution for Complex Environments. *Computer Graphics 19*, 3 (July 1985), 31–40.

[10] Delany, H. C. Ray Tracing on a Connection Machine. In *Proceedings of the 1988 ACM/INRIA International Conference on Supercomputing* (July 1988), pp. 659–667.

[11] Delany, H. C. A Simple Hierarchical Ray Tracing Program for the Connection Machine System. Tech. Rep. VZ 88-4, Thinking Machines Corporation, Cambridge, MA, December 1988.

[12] Dippe, M., and Swensen, J. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *Computer Graphics 18*, 3 (July 1984), 149–158.

[13] Drucker, S. M., and Schröder, P. A Data Parallel Algorithm for Radiosity. In *Proceedings of Third Eurographics Workshop on Rendering* (May 1992), Eurographics. to appear.

[14] Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics 23*, 3 (July 1989), 79–88.

[15] Fujimoto, A., Tanaka, T., and Iwata, K. ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications 6*, 4 (April 1986), 16–26.

[16] Gaudet, S., Hobson, R., Chilka, P., and Calvert, T. Multiprocessor Experiments for High-

Speed Ray Tracing. *ACM Transactions on Graphics 7*, 3 (July 1988), 151–179.

[17] Goldsmith, J., and Salmon, J. Automatic Creation of Object Hierarchies for Raytracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20.

[18] Haines, E. A Proposal for Standard Graphics Environments. *IEEE Computer Graphics and Applications 7*, 11 (November 1987), 3–5.

[19] Haines, E. A., and Wallace, J. R. Shaft Culling for Efficient Ray-Traced Radiosity. In *Proceedings of Second Eurographics Workshop on Rendering* (Barcelona, Spain, May 1991), Eurographics, Springer Verlag. Also published in Siggraph 91 course notes: Frontiers of Rendering.

[20] Hillis, D. W. *The Connection Machine*. MIT Press, 1985.

[21] Kaufman, A. Efficient Algorithms for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes. *Computer Graphics 21*, 3 (July 1987), 171–179.

[22] Kaufman, A., and Bakalash, R. Memory and Processing Architecture for 3D Voxel-Based Imagery. *IEEE Computer Graphics and Applications 8*, 11 (November 1988), 10–23.

[23] Klietz, A. Personal communication. e-mail address: alan@msc.edu.

[24] Mitchell, D. P. Personal communication.

[25] Nader Gharachorloo, e. a. Subnanosecond Pixel Rendering with Million Transistor Chips. *Computer Graphics 22*, 4 (August 1988), 41–49.

[26] Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I., and Omura, K. LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation. In *Proceedings of the 10th Symposium on Computer Architecture* (New York, 1983), ACM, pp. 387–394.

[27] Potmesil, M., and Hoffert, E. M. The Pixel Machine: A Parallel Image Computer. *Computer Graphics 23*, 3 (July 1989), 69–78.

[28] Recker, R. J., George, D. W., and Greenberg, D. P. Acceleration Techniques for Progressive Refinement Radiosity. *Computer Graphics 24*, 2 (March 1990), 59–66.

[29] Salmon, J., and Goldsmith, J. A Hypercube Raytracer. In *Proceedings of HCCA3* (March 1988), pp. 1194–1206.

[30] Schröder, P., and Salem, J. B. Fast Rotation of Volume Data on Data Parallel Architectures. In *Proceedings of Visualization 91* (October 1991), IEEE, IEEE Computer Society Press, pp. 50–57.

[31] Weghorst, H., Hooper, G., and Greenberg, D. P. Improved Computational Methods for Ray Tracing. *ACM Transaction on Graphics 3*, 1 (January 1984), 52–59.

Implementation details

Building the voxel overlap lists

When a database is initially loaded into the system the first step is to build the lists which, for a given voxel, hold all the names of objects intersecting that voxel. For each object type in turn each object allocates enough processors to hold all the voxels that its bounding box overlaps (see figure 5). Call these `cand_voxels`. Using

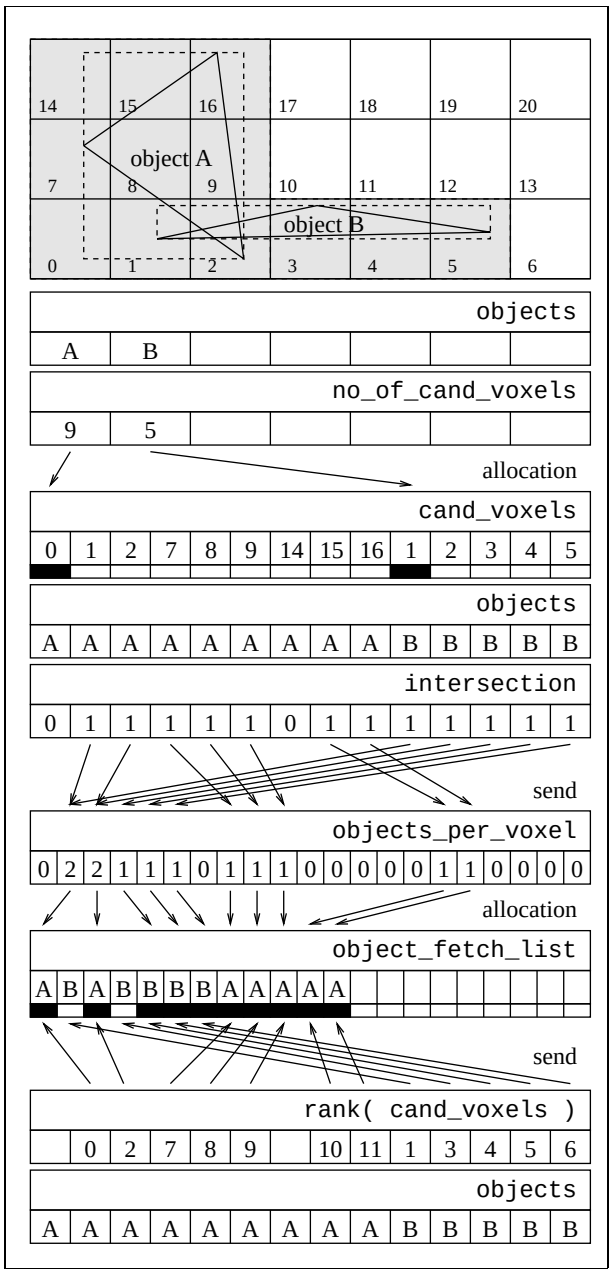


Figure 5: A 2D example of objects on a voxel grid and the voxels their bounding boxes overlap. These voxels are tested against the objects themselves. Only those that actually do intersect with the object, generate an entry in the `object_fetch_list`.

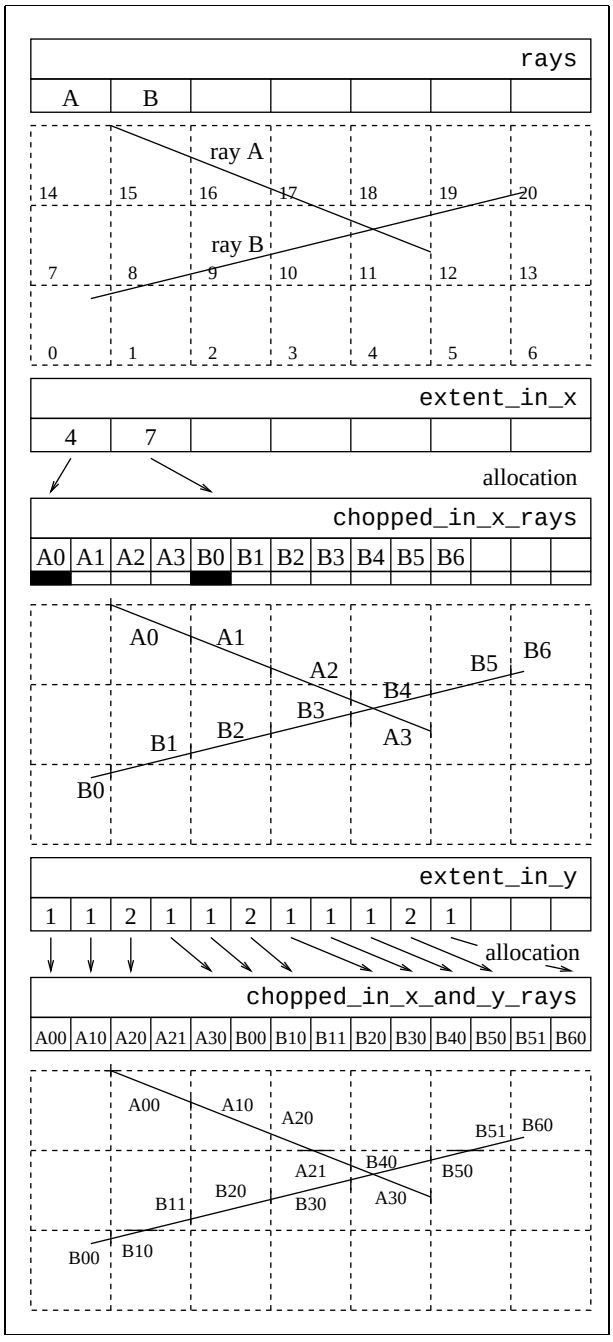


Figure 6: A 2D example of "chopping" lines to the voxel grid along the two dimensions in turn.

the forward pointers, the object data is sent to the beginning processor in each candidate segment. With a *segmented-copy-scan* this data can be propagated to every processor in its segment. At this point we properly intersect the candidate voxels with the respective object. Every processor that finds an actual intersection sends a "1" to the `objects_per_voxel` accumulator. After the intersection step each voxel holds the length of its object list in `objects_per_voxel`. In a second processor allocation step these numbers are used to allocate processors for each voxel to hold the actual lists of object ids. Call this the `object_fetch_list`. Next we find a ranking of the voxel names in `cand_voxels`. Notice that only those processors for which `intersection == TRUE` participate in this ranking. Using the rank as an address we can send the object tags to the proper places in the `object_fetch_list`, which concludes the setup. The `object_fetch_list` consists of segments, each of which corresponds to a particular voxel, that contain ids for those objects which actually intersect a given voxel. Each voxel now holds the address of the beginning of its segment in the `object_fetch_list` as well as the length of that segment. We will use these two pieces of data to build the candidate lists for each ray.

Generating ray/object candidate lists

Given a set of rays we can now build the candidate object lists. In order to do this we need to generate a list of voxels that every ray pierces. Once we have this list, it is an easy matter to use the `object_fetch_list` to get the objects themselves for the ray/object intersection test. One way to generate the voxel lists for each ray, would be to use 3D voxel conversion of the rays with 26 neighbor connectivity (see [21]). This method however is not well suited since the rays are in general of different lengths, leaving many processors idle, while the longest rays are still being voxel converted. Instead we use a nested set of 3 processor allocations, in turn "chopping" the rays to voxel boundaries along each of the 3 axes (see figure 6 for the 2D case). Notice that all children of rays A and B are contiguous and ordered along the ray in the final pvar (`chopped_in_x_and_y_rays`). When this step is completed each chopped ray simply considers its endpoints to find the address of the voxel it crosses

chopped_in_x_and_y_rays													
A00	A10	A20	A21	A30	B00	B10	B11	B20	B30	B40	B50	B51	B60

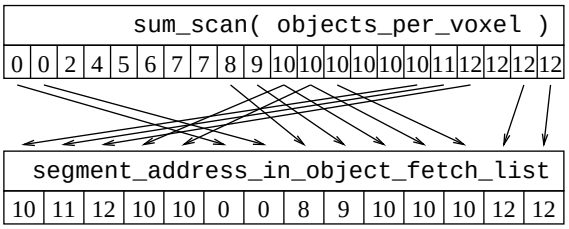
voxels_crossed													
15	16	17	10	11	0	1	8	9	10	11	12	19	20

Using the address in `voxels_crossed` we can now retrieve the number of objects per voxel into `objects_in_voxel_crossed`

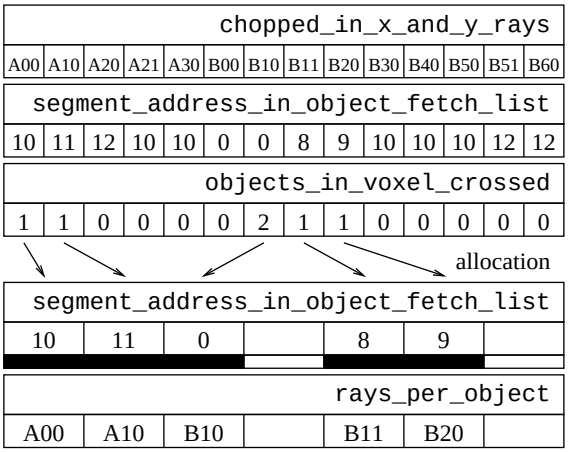
objects_per_voxel																			
0	2	2	1	1	1	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0

objects_in_voxel_crossed													
1	1	0	0	0	0	2	1	1	0	0	0	0	0

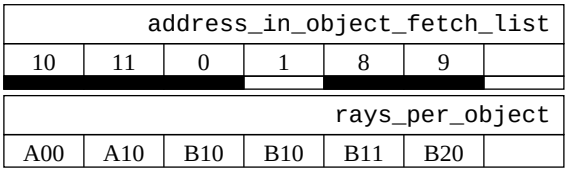
and also where these can be found in the `object_fetch_list`



`objects_in_voxel_crossed` indicates how many intersection processors need to be allocated. After this allocation, the segment addresses and the ray names are propagated into the allocated segments

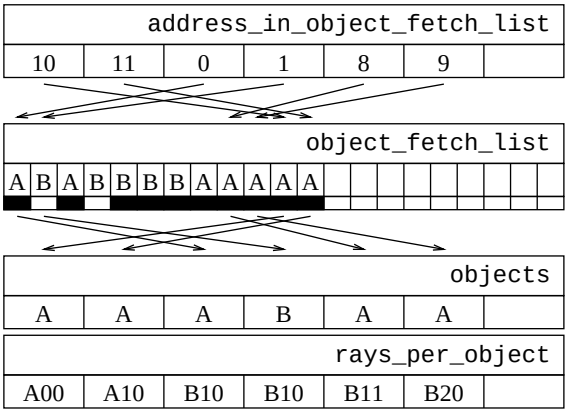


Using the segment bit the addresses of all the objects in a given segment can be generated by incrementing the segment addresses through the length of each segment and the ray data is filled in with a *segmented-copy-scan*

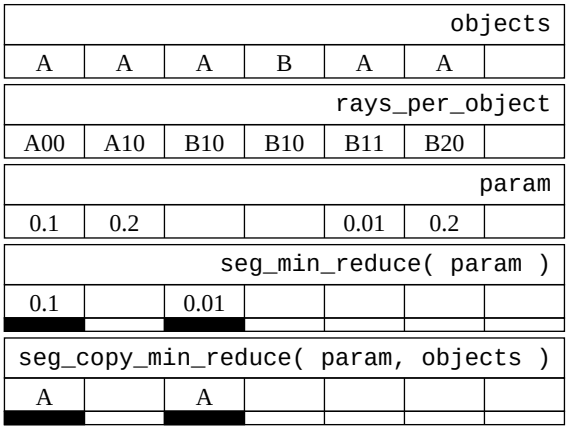


Ray/object intersection

With all the variables from the generation of the candidate list in hand we now only need an indirection through the `object_fetch_list` to pair up each ray with its candidate objects



Since the computed ray/object intersections are ordered within their respective segments in the parameter value along the ray we can execute a *downward-min-reduce* on the computed parameter value of the actual intersections to find the closest (if any) intersection



At this point the algorithm returns with the intersected object and the parameter along the ray where the intersection occurred.