A Hypercube Ray-tracer

John Salmon*, Jeff Goldsmith†

* Mail Code 356-48, California Institute of Technology, Pasadena CA 91125
† Mail Stop 510-264, Jet Propulsion Laboratory, Pasadena CA, 91109

ABSTRACT

We describe a hypercube ray-tracing program for rendering computer graphics. For small models, which fit in the memory of a single processor, the ray-tracer uses a scattered decomposition of pixels to balance the load, and achieves a very high efficiency. The more interesting case of large models, which cannot be stored in a single processor, requires a decomposition of the model data as well as the pixels. We present algorithms for constructing a decomposition based upon information about the frequency with which different elements of the model are accessed. The resulting decomposition is approximately optimized to minimize communication and achieve load balance.

1. Introduction

Generating computer images involves up to three steps. The first step, modeling, translates an idea from the artist's or designer's mind into a form that the computer can understand, a model. The second step, rendering, converts the model, which is described geometrically, into an image, a set of pixels to be displayed on a video screen or in a photograph. The third stage, articulation or animation, introduces time dependence into the model so that objects appear to change or move. In a still image, of course, the articulation stage is not always necessary. It is only required for computer generated animation or simulations.

Ray tracing is an algorithm for rendering geometric models into arrays of pixel values. It simulates the interaction of light rays with the elements of the model and uses classical geometric optics to determine shading. In the simplest version of the ray tracing algorithm, each pixel in the image is evaluated by projecting a primary ray from an imaginary camera, or eye-point, through the point corresponding to the pixel on the plane representing the surface of the display device. The ray is analyzed to determine which, if any, of the objects in the model it hits. If it misses them all, the pixel is colored to show the background of the scene. If the ray hits one or more objects, then only the intersection closest to the origin of the ray is considered further. In this way, primary rays are used to implement a rather costly hidden surface removal algorithm. The color of the pixel is determined by projecting rays from the point of intersection to the light sources in the model. If these rays reach the light sources without hitting any obstructing objects, then surface properties of the object, and the properties of the light contribute to the color of the pixel according to standard shading models. [1] Otherwise, the point is in shadow with respect to the light. In addition, if the object is reflective or transparent, additional rays are also projected in the direction indicated by Snell's law, and/or in the direction indicated by Hero's law (angle of incidence is equal to angle of reflection).

These secondary rays are evaluated just as primary rays, and contribute linearly to the color of the pixel. They may in turn generate further secondary rays until the contribution to the original pixel has fallen to below some minimum threshold.

This optical approach to rendering yields some very desirable effects. Accurate rendition of refracted images, reflected images, and shadows is handled implicitly, effects which are extremely difficult to obtain with other commonly used rendering schemes. More complex optical effects, including motion blur of moving objects, depth of field, and penumbras of shadows are also computable via ray tracing,[2] and are generally not done with other renderers. These effects add greatly to the realism in an image, providing ray tracing with its popularity, despite its substantial computational costs.

Since each pixel is computed independently, ray tracing is an extremely costly rendering algorithm. Other algorithms make use, in one way or another, of the fact that pixels near to one another in a scene have similar properties. When ray tracing was first developed for use in computer graphics,[3] the general consensus was that it was too slow to use except in highly specialized applications. Since then, many techniques have been found to speed up the algorithm, and computers have become much faster. Still, ray tracing uses huge amounts of expensive computer time on some of the worlds fastest computers.

Whitted [1] determined that 75% of the execution time of the simple ray tracing algorithm was spent in testing objects for intersection with rays. Since then, two general approaches have been found for greatly reducing the number of intersection calculations needed for each ray. They are tree-based methods ("tree-tracing") [4,5,6] and space-based ("space-tracing") [7,8,9,10,11] methods. Recently, the two approaches have been combined into a hybrid method that potentially retains the advantages of each.[12] In the space-based methods, the whole model is bounded by a rectangular prism. This prism is then broken into smaller and smaller prisms or cubes until the cubes contain only a few objects. This operation is done as a pre-processing step before any images are calculated. Intersection testing is done by testing each cube along the given ray's path. If such a cube is hit and it contains objects, then those objects are tested for intersection with the ray. This method has the desirable feature that objects are tested approximately in order of distance from the origin of the ray, so once an intersection has been found, most of the remaining cubes need not be tested. It does, however, spend a large amount of memory on empty cubes. Dippé and Swensen [13] analyzed how to break up these cubes so that a parallel machine could use a space-tracing method and found severe load balance problems. Their solution was a dynamic load balance scheme that changed the shape of the cubes, making them much harder to test for intersection. This defeated many of the advantages of the original sequential algorithm.

The other general class of improved ray tracing methods are tree-based. In these methods, each object is surrounded by a tightly fitting, but geometrically simple volume. Before testing the object itself for intersection (which might be expensive), the volume is tested. These volumes are chosen to be very simple objects so that rays which miss the volume completely, and hence miss the enclosed object, can be rejected relatively cheaply. It is natural for sets of bounding volumes to be enclosed by still larger volumes. In this way, intersection with large parts of the model can be ruled out by a simple test with a high-level bounding volume. A tree of bounding volumes is constructed by placing bounding volumes around bounding volumes until one bounding volume, the root of the tree, surrounds the entire model. In these methods, intersection testing is done by a depth-first traversal of the tree, abandoning a branch whenever a bounding volume is missed entirely. Since this method was suggested by Rubin and Whitted,[4] many improvements to it have been found. Kay and Kajiya[5] suggest a new type of bounding volume as well as varying the order in which the tree is traversed, sorting untested nodes in a heap, so that the closest volume is tested first, recovering some of the advantages of space-tracing. Weghorst, et. al. [6] suggest varying the bounding volume types within a scene to better suit the primitives. Goldsmith and Salmon [14] suggest ways to build the tree of bounding volumes so that the sets and subsets are chosen well. Both space-tracing and tree-tracing techniques are still improving, but due to lower cost in space and the desire to avoid the problems Dippé and Swenson encountered, we chose to investigate parallel implementations of tree-tracing methods.

Because of the independence of the subtasks of the ray tracing process, it is a natural candidate for parallel computation. Some work has been done in the past in this regard. Dippé and Swensen[13] discuss the distribution of three-dimensional grids and methods of load balancing them on theoretical concurrent machines. Nishimura et. al. [15] show how a pipelined multimicroprocessor with programmable

interconnects can be used to take advantage of the separate steps in the algorithm. Ullner [16] describes a VLSI implementation of the same sort of similar pipelined hardware.

The technology to design and build parallel hardware has far outpaced the development of system software in recent years. Developing a parallel ray tracer has led us to some insights into the requirements of a truly parallel operating system for parallel hardware systems. In fact, in many cases we had to design and implement operating system software before proceeding with the ray tracer. The areas in which the ray tracer's demands were not met by available systems were the role of the host processor, multitasking, and the availability of common operating system software to aid debugging, I/O, and other frequent computing tasks.

2. Decomposition

The central problem in the design of software for parallel computers is the identification and utilization of parallelism in the algorithms. Indeed, the choice of algorithm can make this task either fairly straightforward or essentially impossible. In the case of ray tracing, one source of parallelism is obvious. Each pixel constitutes a separate and independent problem. Distinct pixels can easily be computed on different processors, requiring only the communication to assemble the result into a single image. This parallelism is of sufficient magnitude to fully utilize any large grain-size parallel processor that will be built in the near future, since even relatively modest images contain upwards of 250,000 pixels. Exploiting parallelism at this level alone, however, leads to problems with distributed memory processors. State-of-the-art graphical models can contain millions of primitive objects, requiring hundreds of megabytes of storage. Even modest models would not fit into the memory of a single NCUBE or iPSC processor. A priori, it is not possible to know which parts of the model will be needed by any given pixel. Hence, every processor would potentially need access to the entire model, which unfortunately, is out of the question due to the limited memory of each processor. To render large models, we will need to distribute the model data as well as the computation. The importance of this can be seen vividly in the case of the NCUBE machines. Each processor has 512 kilobytes of memory, enough to store a complete simple model, but not enough for a large one. A fully configured machine, with 1024 nodes, has 512 megabytes of memory. Few applications will run out of memory on such a machine. An important consideration is that the distribution of the model data over the nodes of the machine strongly affects how well balanced the computational loads are on the different processors, and ultimately determines whether the machine is efficiently used. A careless distribution will lead to severe load imbalance and poor overall performance, so it is important to carefully minimize the load imbalance in the distributed data.

In the context of parallel computing, a decomposition is an assignment of work to processing elements. Currently, no tools exist to automatically decompose a sequential program so that it exploits parallel, distributed memory hardware. Parallelism must be "designed into" the program from the start. The first step in designing algorithms for use on parallel machines is to decide on a decomposition, i.e. to divide the complete task into smaller sub-tasks that can be completed relatively independently. The ultimate goal of a parallel decomposition is to perform the desired computation in the least time, making as efficient use as possible of the parallelism in the hardware. This goal is achieved by:

1) Minimization of interprocessor communication.

2) Minimization of processor idle time.

3) Use of inherently fast algorithms.

Interprocessor communication is expensive on today's hypercubes. For the machines we have used, the communication of a word of data has about the same cost as one to ten floating point operations. Thus, minimization of communication takes on an importance similar to the elimination of unnecessary floating point operations. Clearly, it is not the only criterion by which to judge an algorithm, but it provides a very useful rule-of-thumb. We will find that the ray-tracing of simple scenes, i.e. those for which the entire model fits in a single node, requires little or no communication. Algorithms to minimize interprocessor communication become important in the parallel ray tracer when the model is too large to fit in a single node.

The second issue in decomposition is minimization of idle time. This is often referred to as "load-balancing" because idle time is normally reduced when the computational load is distributed evenly

amongst the available processors. Simply balancing the load is necessary, but not sufficient, to minimize idle time. One must also ensure that no processor waits unnecessarily for another to compute an intermediate result. Load balancing will be important in all aspects of the parallel ray tracer.

2.1. Image Decomposition

Ray tracing is an ideal algorithm for a parallel computer. Indeed, the fundamental source of its slowness, the fact that information about the scene is not retained from one pixel to the next, makes part of the decomposition extremely easy.

The algorithm is decomposed by assigning different pixels to different processors. If we assume, for the moment, that each processor has access to the entire model, then the processors may compute their respective pixels with absolutely no communication. Interprocessor communication is obviously minimized by assigning a fixed number of pixels to each processor at the beginning of the computation. Then, there is no need for communication at all during the bulk of the computation. The processors may need to communicate again at the end of the computation when accessing a frame buffer or mass storage device. Even this may not be necessary if suitable parallel I/O hardware is available.

The simplest assignment of pixels to processors is probably the *tiled* decomposition shown in Figure 1.

1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4
5	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8
5	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8
5	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8
5	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8

Figure 1. Tiled Decomposition

Under this scheme, each processor computes a contiguous rectangular subset of the entire image. Output is particularly simple because most standard image file formats allow for "tiling" in exactly this way. Despite the fact that each processor computes an equal number of pixels, this decomposition suffers badly from load imbalance. The root of the problem is that there is a large variation in the cost of computing a single pixel. If these variations were uncorrelated, the load imbalance would be acceptable, (as we shall show below) but in real images, expensive pixels tend to be near one another in the image, especially near the center of the image. Assigning exclusively central pixels to some processors, while assigning exclusively peripheral pixels to others is very likely to lead to significant load imbalance. It is easy to produce images for which the load is balanced, but most interesting images will suffer severe load imbalance with the tiled decomposition.

The *scattered* decomposition of Figure 2 is a partial solution to the problem of load imbalance due to coherence in pixel complexity. In this decomposition, each processor is responsible for pixels which are scattered across the entire image. The worst case behavior of the scattered decomposition is identical to that of the tiled decomposition. In principle, one processor could still be assigned all the work, by an unfortunate distribution of complexity in the image. The advantage of the scattered decompositions is that such behavior is extremely unlikely. Visually interesting images are very unlikely to have coherence on exactly the same scale as the scattering, and hence are unlikely to suffer from extreme load imbalance.

The scattered decomposition still suffers from load imbalance due to the statistical nature of the time to compute a pixel. It is possible to estimate the magnitude of this residual load imbalance.[17] If the time to compute each of the pixels assigned to a processor is an uncorrelated random variable with mean t_{pix} and root mean square deviation σ_{pix}, then straightforward application of the central limit theorem demonstrates

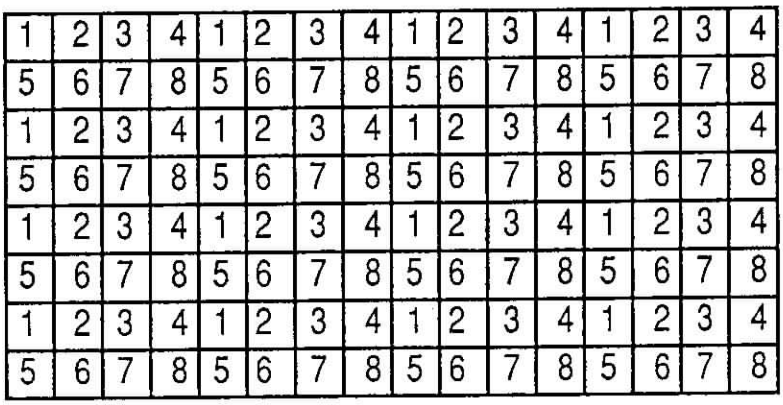

Figure 2. The Scattered Decomposition

that the load imbalance will be less than some value l_c with confidence c, where

$$l_c \propto n^{-1/2}\left(\frac{\sigma_{pix}}{t_{pix}}\right)$$

$$n = \frac{N_{pix}}{N_{proc}}$$

and the constant of proportionality is of order 1.

Our experience with ray-traced pixels is that σ_{pix} and t_{pix} are comparable. The distribution of pixel times is a fairly rapidly decreasing function of time, whose width is approximately equal to its mean. The load imbalance in such cases is thus typically a few times the reciprocal square root of the number of pixels assigned to each processor. Table 1 shows the observed value of the load imbalance for several images.

name	pixels per processor	Load Imbalance number of processors			$n^{-1/2}\left(\frac{\sigma_{pix}}{t_{pix}}\right)$
	n	4	16	64	
solids	1024	0.6%	2.0%	2.7%	3.9%
hcube	1024	1.8%	6.7%	7.5%	3.3%
room	1024	0.5%	0.9%	1.0%	1.8%

Table 1. Values of load imbalance.

In order to achieve better load balance, the decomposition must be modified as the computation progresses, so that processors which find themselves too busy can shed load onto less busy neighbors. In principle, load shedding can lead to almost perfect load balance, but there is communication overhead because the processors must communicate to inform one another, or a central dispatcher, of their relative busyness. Since the load imbalance is typically less than 10% for interesting images on currently available machines, the additional communication overhead could easily consume all that was saved by better load balance. We will defer the discussion of run-time load balancing until after the discussion of data decompositions, because the potential benefits will be much greater at that point. For models which do not require distribution to fit in the memory of the parallel processor, we consider the load imbalance achieved by scattering to be perfectly adequate, and do not pursue additional balance at the expense of additional software complexity.

2.2 Object-Based Decompositions

As discussed in the introduction, the basic data structure of sequential tree-tracing methods is a single tree of bounding volumes. The leaves of the tree contain all the information about the model and its internal structure is of fundamental importance in minimizing the work required to identify ray-object intersections. We now consider the problem of distributing the tree in order to spread the model data across the memory of multiple processors. For well constructed trees, a significant fraction of the intersection testing calculation takes place with bounding volumes that are near the root of the tree. Figure 3.

name	polys	spheres	cyls	lights	size(bytes)	t_{pix} (msec)	σ_{pix} (msec)
solids	64	1	1	2	11380	209 †	264
hcube	1	72	108	1	41436	386 †	410
room	116	0	114	4	63972	655†	373
balls	1	7381	0	3	1.25M	145‡	*
mountain	8192	4	0	1	1.70M	232‡	*

* times for individual pixels could not be determined because of multi-tasking
† nine primary rays were used per pixel for anti-aliasing
‡ values taken from 16-processor timings. See also Table 4.

Table 2. Model Statistics

shows the frequency with which bounding volumes are intersected as a function of depth in the tree for the "room" model. The bounding volumes near the root of the tree generally do not contain objects, so they require only about 30% as much space as a typical leaf node containing a simple object. Since the information in these nodes is used very often, it is important that they are readily available to each node. Since there are relatively few such nodes, they can be placed in each node of the hypercube without unacceptable memory cost. On the other hand, individual objects (stored in the leaf nodes of the tree) are needed only rarely, so these lesser-used objects can be stored only once, and accessed by message passing or remote procedure call without unacceptable communication overhead. Thus, we have chosen to include the root bounding volume of the model tree in each processor, along with some of its closer descendants. We call this data structure the *forest*, in contrast to the *sub-trees* which are stored in individual processors. The terminal nodes of the forest contain information which can be used to locate sub-trees which reside on other processors. Many of the rays that hit no objects can be completely resolved in the forest, without recourse to model data stored in sub-trees on other processors.

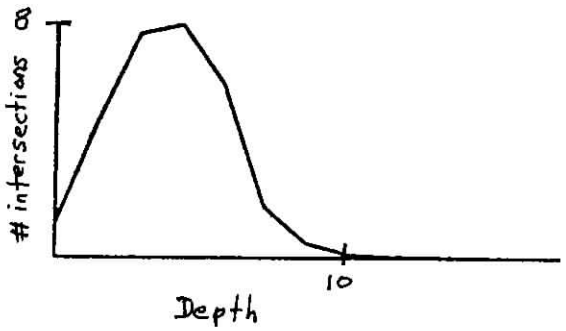

Figure 3. Number of intersections per ray vs. depth for the room model

A decomposition consists of specification of a forest, and a mapping from the terminal nodes of the forest onto the set of processors. In principle, the mapping can be many-to-many, but for simplicity we only consider many-to-one mappings. That is, several sub-trees may be assigned to a single processor, but any given sub-tree will be assigned to exactly one processor. The optimal decomposition, of course, fits within the memory limitations of the processors, balances the computational load and minimizes communication overhead. The memory constraint is absolute, while some tradeoff is allowed between load-balance and communication. We may usefully consider the decomposition a two-step process, in which we first determine the sub-trees, and then assign individual sub-trees to processors.

We cannot hope to achieve a desirable decomposition of the hierarchy of bounding volumes without being able to estimate the memory requirements, computational load and communication communication requirements of a proposed decomposition. That is, we must have some predictive criterion with which we can judge one decomposition as being superior to another. The easiest of the desiderata to quantify is memory. Each node of the original tree which remains in the forest uses a fixed amount of memory in each and every processor. Similarly, each node which is assigned to a sub-tree uses the same amount of

memory, but only on the processor to which it is assigned. It will be convenient to define the quantity S_n which is the size of the entire sub-tree rooted at node n.

The next criterion to quantify is the computational load associated with each of the sub-trees. First, we assign to each node in the tree a probability. This probability measures the fraction rays generated in a "typical" scene that intersect with this bounding volume. We have found that to a very crude approximation, one can estimate that the probability that a volume is hit is proportional to the volume's surface area.[14] This estimate is, of course, very easy to compute. Unfortunately, it is subject to very large fluctuations, and it is inadequate for use in the decomposition algorithm. We have found that a good estimate of the probability is obtained by ray-tracing the image at very low resolution, i.e. 32x32 pixel resolution. Although this estimate is also subject to fluctuations, they do not effect the final decomposition as adversely as those from the area estimate. The cost of such a low resolution rendering is small, even though it is done as a pre-processing step on a sequential processor. In comparison with an anti-aliased 512x512 resolution image, the pre-processing stage constitutes about one two-thousandth of the total work.

Given the probability that a node is hit, one can compute the work that is entailed by the presence of the node on a processor. Once a node is hit, an intersection test must be performed with each of its children. If the node is a terminal node of the tree, then an intersection test must be performed with a primitive object in the model. If we take the unit of work to be the cost of an intersection calculation with a bounding volume, then the work, w_n associated with node n is given by:

$$w_n = p_n c(n) \quad \text{if n is not a leaf}$$

$$w_n = p_n w_{prim}(n) \quad \text{if n is a leaf}$$

where $c(n)$ is the number of children of node n, p_n is the probability of hitting node n, and $w_{prim}(n)$ is the amount of work required to perform an intersection with the primitive object associated with the leaf node n. Of course, the type of bounding volume, the type of leaf and the details of both the hardware and software implementations determine the values of $w_{prim}(n)$. We used rectangular prisms with faces perpendicular to the coordinate axes as bounding volumes and spheres, cylinders and polygons as primitive objects in the models. In our NCUBE implementation, the costs were approximately 2.5, 6.0 and 12.0 bounding box intersections per sphere, cylinder and polygon intersection, respectively. In analogy with S_n, we define the quantity W_n, which is the sum of the w_i for all nodes i which are descendants of node n. If node n is a terminal node of the forest, then W_n is an estimate of the amount of work which will be performed on the processor to which the corresponding sub-tree is assigned.

The third factor to be quantified is the interprocess communication. The relative importance of communication depends on the hardware and software environment in use. Each time a request is made to compute the intersection of a ray with a sub-tree, a certain amount of data must be transmitted to the processor that stores the sub-tree. This data consists of seven floating point numbers (origin (3), direction (3) and distance to nearest previously determined intersection (1)). On the machines we have used, and within a software environment that allows for arbitrary, asynchronous message forwarding, the time to transmit such a message is approximately equal to the time to perform several bounding volume intersections. The exact cost depends on whether the recipient of the message undergoes a context switch, and the distance the message must travel. It is only weakly dependent on the size of the message, because the time to send such short messages is dominated by "message latency" rather than "bandwidth". This suggest that minimization of the total number of interprocessor communications is crucial to a high-performance implementation.

Communication occurs each and every time the bounding box of terminal node in the forest is intersected by a ray. A description of the ray must be sent to the processor to which the sub-tree is assigned, and the result of the ensuing calculation must be sent back. Thus, each terminal node of the forest incurs a communication cost proportional to the number of rays that intersect that node's bounding volume. We can estimate the communication cost, C_n, of cutting the tree at node n as

$$C_n \approx p_n$$

where p_n is, as before, the probability of a ray hitting the bounding volume of node n.

The difficulty of the distribution problem is now clear. The communication cost associated with making a cut in the tree decreases as one descends the tree, because the bounding volumes become smaller, and are less likely to be hit. On the other hand, cuts near the leaves of the tree are less effective in decreasing the size of the forest and conserving memory, and more cuts must be made to satisfy the memory constraint. Finding the set of cuts that minimizes the total communication, while satisfying the memory constraint is extremely hard.

We now return to the basic problem of decomposing the model tree. We divide the problem into two phases. The first is choosing the sub-trees; the second is allocating the sub-trees to specific processors. Since the second problem is already NP-hard, (it is analogous to the weighted bin packing problem)[18] truly optimizing the selection of sub-trees, which depends on the solutions to the allocation problem, is not feasible. Nevertheless, the dependence of the sub-trees selection problem on the allocation problem seems to be weak, so we treat the problems as independent.

2.2.1. Choosing the cut-points

Deciding on a set of sub-trees is equivalent to selecting a set of nodes from the model tree which will be the terminal nodes of the forest. We shall refer to these nodes as *cut-points*, which suggests the fact that the model has been cut at these locations, and that the sub-tree that has been cut off must be allocated to a single processor. According to the analysis of communication above, a cut-point at node n, entails communication proportional to p_n. The problem is to select a set of cut-points so that:

a) no element in the set is an ancestor of any other element

b) the forest and sub-trees fit in the available memory, M_{avail}. At this stage, we insist that the model and sub-trees fit into a fraction, f, of M_{avail} so there is more freedom when we allocate the sub-trees to processors. The decompositions used in this paper were obtained with a value of f of 95%.

c) the sum of C_n for all cut nodes n is minimized.

An approximate solution is obtained by the following procedure:

1) Sort the nodes of the model tree in order of increasing p_n. Remove from consideration any node with S_n larger than a fraction, g of the memory of a single processor. This is to allow room for the largest sub-tree to fit in a processor alongside the forest. We used a value of 20% for g.

2) If the resulting forest and sub-trees satisfy the memory criterion, (b), FINISH.

3) Select the first node, n, from the sorted list and place a cut at that node. If node n is the ancestor of any node currently containing a cut, delete the old cut. In addition, delete n and all its descendants from the sorted list since they are no longer candidates for cut-points.

4) For each ancestor, a, of node n, reduce the value p_a by an amount equal to p_n, and adjust the sorted list accordingly.

5) return to step 2.

The purpose of step 4 is to quantify the fact that it is desirable to place a cut at a node which is the ancestor of other cut-points. When the new cut is made, all the other cuts disappear, and the effective cost of the new cut is its true cost, C_n minus the sum of all cuts previously below it.

To evaluate the termination criterion, a running sum of total memory usage must be maintained as cuts are inserted and deleted. This is straightforward because a cut at node n decreases the size of the forest by by $(N_{proc}-1)S_n$. A forest with no cuts has total size $N_{proc}S_{root}$. If the parallel processor has too little memory, the procedure will exhaust all the nodes in the sorted list without satisfying the termination criterion.

Without knowing the globally optimal solution to the selection problem, it is difficult to precisely measure how well our algorithm is working. For purposes of comparison, in Table 3, we show the amount of predicted communication in the decomposition selected by the algorithm compared with that obtained by placing cut-points at random until the forest is small enough to fit in a processor.

model name	per proc. data size	random decomp.	Number of processors			
			16	32	64	128
balls	120kb	1.683	0.744	0.594	0.540	0.538
mountain	200kb	2.251	0.790	0.675	0.647	0.647

Table 3. Communication in units of expected ray transmissions per ray evaluation.

2.2.2. Distribution of sub-trees

The time it takes the parallel processor to render the entire image is proportional to the work in the forest (i.e. the work associated with the root node minus that associated with all the sub-trees) plus the sub-tree work allocated to the most heavily burdened node. Since the whole ensemble of processors is no faster than its slowest component, it is crucial to balance the load. Obviously, the best possible situation is one in which all processors finish at the same time. Usually, no such solution exists and we must search for a distribution of sub-trees to processors which minimizes the load imbalance. Unfortunately, the problem is equivalent to the "weighted bin-packing problem", which is known to be NP-hard. Nevertheless several $O(N_{tree}(\log(N_{proc}) + \log(N_{tree}))$ time algorithms exist that appear to perform adequately. One such algorithm is known as the "best fit decreasing" algorithm. The sub-trees are first sorted in order of decreasing size, taking time $O(N_{tree}\log(N_{tree}))$. Then, one at a time, they are considered for assignment to processors. Each sub-tree is assigned to the least loaded processor for which the memory constraint is not violated. If the list of processors is managed as a heap, the least loaded processor can be found in constant time, and the heap restructured in time $O(\log(N_{proc}))$. Sometimes, no processor remains with enough space for a tree. Unless memory is extremely tight, sub-trees selected by the cut selection procedure of Section 2.2.1 will be small, and will always fit in the available memory. In the event that they do not, the cutting algorithm should be re-applied, with a smaller value of f. A solution to the packing problem can then usually be found.

3. Object Decomposition: Implementation

Here, we present in somewhat more detail the steps involved in computing the color of ray when the model tree is split into a forest and sub-trees. As in the sequential case, the ray is tested against the bounding volumes that constitute the forest in depth-first order. If the ray fails to intersect some node, further processing on the descendents of that node is abandoned. Whenever a ray reaches a terminal node of the forest, and intersects with the bounding volume at that node, a message must be sent to another processor requesting additional processing. This message contains the origin and direction of the ray, and the distance from the origin to the closest "hit" found so far. Eventually, a message is returned, indicating whether the current "hit" was improved on the remote node, and if so, additional information describing the object that was hit.

While computations are being performed by another processor, several options are open to the requesting processor: it can wait for a reply, it can work on another pixel, or it can continue the traversal of the forest with the current ray. Clearly, the simplest thing to do is to wait for the other processor to reply. Unfortunately, this also leads to a huge amount of wasted processor time. Continuing the traversal of the tree is undesirable because if the remote request returns with information about a new hit, that information can be used to great advantage in the subsequent traversal of the rest of the tree. Traversing the tree without that information may entail significantly more work than the sequential algorithm. The best option is to work on another pixel. A multi-tasking operating system resident on each processor allows one to effectively begin processing another pixel, while retaining the simplicity (at least in the ray-tracing code) of simply waiting for a reply. It also allows one to conveniently separate the processing of rays in the forest from the processing of rays in sub-trees on behalf of other processors.

Once the remote processor has replied, we continue the traversal of the forest. Finally, the closest hit has been found, and we shade the ray according to the usual prescription for ray-tracing. The shading algorithm, of course, may require the analysis of reflection, transmission or illumination rays. These rays are treated exactly like pixel rays. They are tested against the forest in the processor in which they are created, and requests are sent to other processors whenever needed. They are eventually shaded in the processor that created them.

Thus, we have an overall implementation in which pixels are decomposed according to either the tiled or scattered decomposition. All rays associated with a given pixel traverse the forest in the processor responsible for that pixel. Furthermore, all shading calculations for these rays are done in that processor. When a ray leaves the forest, a message is sent to the processor controlling the appropriate sub-tree. The ray then traverses that sub-tree in the remote processor, and a reply is returned to the originating processor, indicating whether an intersection was detected and if so, the relevant photometric parameters of that intersection.

To avoid unnecessary idle time, but retain simplicity in the code, it is important that several pixel evaluation *processes* be active simultaneously on each processor. Multi-tasking is not strictly necessary; multiple active pixels could be managed by the ray-tracing code itself, but interest in irregular problems was growing at Caltech just as the need for some form of multi-tasking in the ray-tracer became clear, so we decided to pursue a multi-tasking approach. With multiple active processes on a processor, when process α on processor A waits for a reply from process β on processor B, processor A will not become idle because another process (γ) will be available to continue work on another pixel. We found that three to six pixel processes per processor are sufficient to make idle time unimportant.

The operating systems available on hypercubes, both commercially and internally at Caltech, did not provide the support we needed to allow multi-tasking of this type. Consequently, part of our task in developing the ray tracer involved operating system development not directly related to computer graphics. To a large extent, the experimental MOOSE operating system for the Mark II, NCUBE and Intel hypercubes was motivated by the needs of the ray tracer. The MOOSE operating system has powerful run-time multi-tasking facilities, and will provide a framework for future developments in dynamic load management on hypercubes and solution of irregular problems. MOOSE is described in detail in two other papers in this proceedings [19,20]

We also made extensive use of the CUBIX I/O system,[21,22] which allowed us to write a program that is portable not only among different brands of hypercubes but also among several sequential machines, including VAX, Sun, Elxsi and the Apple Macintosh II. The version of the code suitable for small geometric models (i.e. it does not distribute the database) can be recompiled without modification on sequential as well as parallel machines. The high quality editors, debuggers, profilers and other tools on the sequential machines were a great help in debugging and optimizing the code in the initial stages.

3.1 Dynamic load balancing revisited

In the parallel scenario so far, both the model and the image have been statically decomposed onto the parallel processor. A dynamic decomposition of the image offers the possibility of improved load balance. Since both pixel tasks and hit tasks are competing for the same CPU resource on each node, we can try to keep the entire machine busy by dynamically balancing only one of the two types. The pixel tasks are the obvious candidate. The simplest thing to do is to allocate blocks of pixels on demand to processors which are idle. This can be done by maintaining a single global list of unfinished pixels. Since a block of pixels can be describe succinctly with only a few words of data, the communication overhead necessary to obtain a block of unfinished pixels is not too large. Once a block of pixels is complete, it can be written directly to the frame buffer, or to a mass storage device.

4. Results

Load balancing results for small models have already been reported in Table 1. For such models, load imbalance constitutes essentially all of the inefficiency of the algorithm. We discount the time to load the model and dump the completed image for two reasons: it is highly dependent on hardware, which is rapidly improving at the moment (parallel I/O systems), and we did not make any effort to optimize the parallel loading of the database. Black and white reproductions of the color images at 512x512 resolution appear in Figure 2. The three models, "hypercube", "room" and "solids" were rendered with nine primary rays per pixel for anti-aliasing on a 256 processor NCUBE system in times: 423, 675 and 219 seconds, respectively.

We used two large models placed in the public domain by Eric Haines[23] to assess the efficiency of the system when the database is distributed. The model called "balls" is a recursive structure of a large sphere surrounded by 9 smaller spheres, each surrounded by 9 smaller spheres, etc. "Mountain" is a fractal

mountain with four crystal balls suspended above it. Statistics about the models can be found in Table 2. Timing results from running the raytracer on an NCUBE system appear in Table 4.

model name	per processor data size	Number of processors 16	32	64	128	Sun 3/160
balls	120kb	145(5%)	144(5%)	143(14%)	144(39%)	55
mountain	200kb	232(7%)	230(6%)	231(13%)	232(37%)	83

Times are in msec per pixel per processor, i.e. $\frac{T_{avg} N_{proc}}{N_{pix}}$.

Numbers in ( ) are the observed load imbalance.

Table 4. Timing results for models with a distributed database.

The times in Table 4 are already scaled to the number of processors, and should be compared directly across a row. The number of pixels per processor, n, is either 1024 or 2048 for each of these images. The fact that the per-pixel performance remains almost exactly constant across each row means that communication and other forms of overhead depend primarily on the number of pixels per processor, and only weakly on the number of processors. Substantial load imbalance occurs when there are many processors because some processors are assigned only one sub-tree, but that sub-tree accounts for more than one processor's share of the total work. The large values of the load imbalance suggest that it may be necessary to dynamically allocate pixels to processors on demand. Then, those processors that have more than their share of sub-tree evaluations will simply compute fewer pixels. An alternative to dynamic load balancing is to assign particularly time-consuming sub-trees to more than one processor, so that the processing of requests directed toward that sub-tree is distributed.

It is difficult to accurately measure the efficiency of the parallel program for large models because, by definition, the image cannot be rendered on a single processor. For purposes of comparison, in addition to hypercube running times, we report the time to render the images on a Sun 3/160, with floating point accelerator. When rendering small models, each processor of our NCUBE system operates at approximately 30-45% the speed of the Sun.

A black and white reproduction of the "hcube" image appears in Figure 4. It represents a six-dimensional hypercube as a three-dimensional cube with a three-dimensional cube at each corner. The image was originally rendered at 24-bits RGB resolution.

5. Acknowledgments

We thank Eric Haines for supplying the "mountain" and "balls" models. This work was supported in part by Department of Energy Grant No. DE-FG03-85ER25009, the Director's Discretionary Fund of the Jet Propulsion Laboratory, the Program Manager of the Joint Tactical Fusion Office and the ESD division of the USAF as well as grants from IBM and SANDIA. In addition, J.S. was partially supported by a Shell Foundation Fellowship.

References

1. T. Whitted, "An Improved Illumination Model for Shaded Display," *Comm. ACM*, vol. 23, pp. 343-349, 1980.
2. Robert L. Cook, Thomas Porter, and Loren Carpenter, "Distributed Ray Tracing," *Computer Graphics (Proc. SIGGRAPH 84)*, vol. 18, July 1984.
3. A. Appel, "Some Techniques for Machine Renderings of Solids," *AFIPS Conf. Proc*, pp. 37-45, AFIPS, Reston, VA, 1968.
4. S. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics (Proc. SIGGRAPH 80)*, vol. 14, pp. 110-116, July 1980.
5. T. Kay and J. Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics (Proc. SIGGRAPH 86)*, vol. 21, pp. 169-178, Aug 1986.

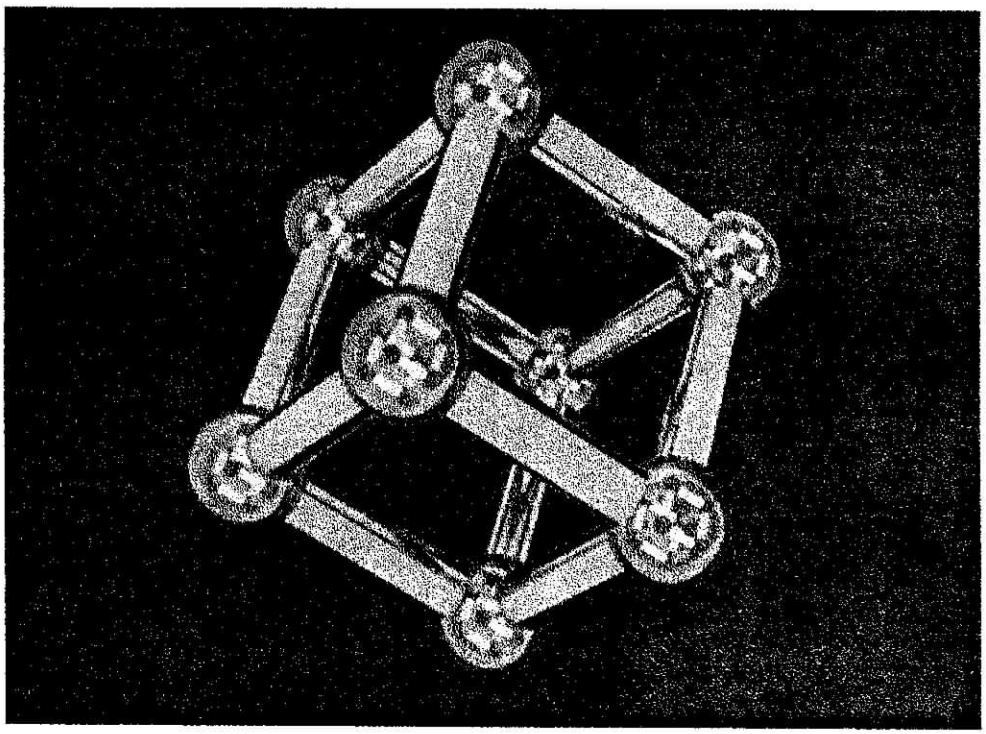

Figure 4. The image entitled "hcube"

6. H. Weghorst, G. Hooper, and D. Greenberg, "Improved Comptational Methods for Ray Tracing," *ACM Trans. on Graphics*, pp. 52-59, 1984.
7. A. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE CG&A*, pp. 15-22, Oct. 1984.
8. A. Fujimoto, "ARTS: Accelerated Ray-Tracing System," *IEEE CG&A*, pp. 16-26.
9. M. Kaplan, "Space-Tracing a Constant Time Ray-Tracer," *Computer Graphics (Proc. SIGGRAPH 85 Tutorial on ray tracing)*, vol. 19, July 1985.
10. L.V. Warren, "Geometric Hashing for Processing Complex Scenes," CS Dept. Memorandum, Univ. of Utah, Salt Lake City, Utah, 1985.
11. J. Arvo and D. Kirk, "Fast Ray Tracing by Ray Classification," *Computer Graphics (Proc. SIGGRAPH 87)*, vol. 21, pp. 55-64, July 9817.
12. A. Barr and J. Snyder, "Ray Tracing Complex Models Containing Surface Tesselations," *Computer Graphics (Proc. SIGGRAPH 87)*, vol. 21, pp. 119-128, Aug 1987.
13. Mark Dippé and John Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics (Proc. SIGGRAPH 84)*, vol. 18, July 1984.
14. J. Goldsmith and J. Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing," *CG&A*, pp. 14-20, May 1987.
15. H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa, and K. Omura, "LINKS-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," in *Proc. 10th Symposium on Computer Architecture (SIGARCH)*, ACM, 1983.
16. M. Ullner, *Parallel Machines for Computer Graphics*, PhD Thesis, California Institute of Technology, 1983.
17. J. Salmon, "A Mathematical Analysis of the Scattered Decomposition," in *Proceedings of the Third Conference on Hypercube Computers and Applications*, ed. G.C. Fox, ACM, New York, 1988.

18. Horowitz and Sahni, *Fundamentals of Computer Algorithms*, CS Press, New York, 1978.

19. J. Salmon, S. Callahan, A. Kolawa, and J. Flower, "MOOSE: A Multi-tasking Operating System For Hypercubes," in *Proceedings of the Third Conference on Hypercube Computers and Applications*, ed. G.C. Fox, ACM, New York, 1988.

20. J. Koller, "Dynamic Load Balancer on the Intel Hypercube," in *Proceedings of the Third Conference on Hypercube Computers and Applications*, ed. G.C. Fox,, ACM, New York, 1988.

21. J. Salmon, "CUBIX: Programming Hypercubes Without Programming Hosts," in *Hypercube Multi-Processors 1987*, ed. M.T. Heath, pp. 3-9, SIAM, Philadelphia, 1987.

22. R. Williams and J. Flower, "Hypercube Programming in Comfort," in *Proceedings of the Third Conference on Hypercube Computers and Applications*, ed. G.C. Fox,, ACM, New York, 1988.

23. Eric Haines, "A Proposal for a Standard Graphics Environment," *IEEE CG&A*, vol. 7, pp. 3-5, Nov 1987.