# Parallel implementation of a ray tracing algorithm for distributed memory parallel computers

TONG-YEE LEE<sup>1</sup>, C. S. RAGHAVENDRA<sup>2</sup> AND JOHN B. NICHOLAS<sup>3</sup>

#### SUMMARY

Ray tracing is a well known technique to generate life-like images. Unfortunately, ray tracing complex scenes can require large amounts of CPU time and memory storage. Distributed memory parallel computers with large memory capacities and high processing speeds are ideal candidates to perform ray tracing. However, the computational cost of rendering pixels and patterns of data access cannot be predicted until runtime. To parallelize such an application efficiently on distributed memory parallel computers, the issues of database distribution, dynamic data management and dynamic load balancing must be addressed. In this paper, we present a parallel implementation of a ray tracing algorithm on the Intel Delta parallel computer. In our database distribution, a small fraction of database is duplicated on each processor, while the remaining part is evenly distributed among groups of processors. In the system, there are multiple copies of the entire database in the memory of groups of processors. Dynamic data management is acheived by an ALRU cache scheme which can exploit image coherence to reduce data movements in ray tracing consecutive pixels. We balance load among processors by distributing subimages to processors in a global fashion based on previous workload requests. The success of our implementation depends crucially on a number of parameters which are experimentally evaluated. ©1997 by John Wiley & Sons, Ltd.

## 1. INTRODUCTION

Ray tracing is a popular rendering technique in the generation of realistic images. It simulates simple optics by tracing light rays (the primary rays) backwards from the eyepoint, through each pixel of the virtual 2D screen space, and onto a 3D object world space[1]. Of all objects which intersect a primary ray, the closest of these will primarily contribute the color intensity of the corresponding pixel. If the closest intersected object is made of a reflective or transparent material, then the secondary rays are traced in the direction of reflection or refraction. If these rays also intersect with other objects, then a contribution of the illumination of the closest objects is added into the intensity of the original pixel. Therefore, ray tracing can embrace many natural phenomena, such as reflections and refractions.

However, one disadvantage of ray tracing is that it is computationally intensive; most of the time is spent in finding intersection points between millions of rays and thousands of objects. There were many sequential algorithms proposed to decrease the computational complexity of ray tracing[2]. However, they were still too slow to make ray tracing practical on a general purpose workstation; it can take several hours to ray trace a 512×512 resolution pixel image which contains thousands of objects on current workstations. The sizes of the

<sup>&</sup>lt;sup>1</sup>Department of Information Management, Nantai Institute of Technology, Tainan County, Taiwan, Republic of China

<sup>&</sup>lt;sup>2</sup>School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164, USA

<sup>&</sup>lt;sup>3</sup>Environmental Molecular Science Laboratory, Pacific Northwest Laboratory Richland, WA 99352, USA

complex scene that are used today ranges from hundreds of megabytes to gigabytes. Such large memory requirements are often too great for current workstations. The use of massively parallel computers can potentially alleviate both the greater computing and memory requirements for ray tracing.

Since the computation of each pixel is independent of the others, ray tracing is highly suitable for parallelization. However, the pixel calculation time can vary significantly and cannot be predicted until runtime. The key to achieving efficient parallel ray tracing is to ensure that the processors have approximately equivalent amounts of computation load to perform. In processing the image, there is no *prior* knowledge to determine which parts of the scene database individual pixels will access. Therefore, each pixel computation could potentially access the entire database. Since the size of the database can be very large and the memory on each processor is limited, the scene database must be distributed. If some data which are necessary for ray tracing the current pixel are not available at a local processor, data must be remotely fetched from other processors in the system. This increases the communication cost required for the pixel computations. We must judiciously choose the data distribution scheme to avoid excessive data movements. This paper describes approaches for parallelizing a ray tracing algorithm on the Intel Delta parallel computer, emphasizing efficient data movement, database distribution, dynamic data management, and load balancing strategies.

In this paper, a load balancing scheme is presented which is based on our earlier work[3]. In this method, balanced load is achieved through global workload servers. Initially, a given image screen is subdivided into as many rectangular subimages as the number of processors and each processor raytraces its assigned subimages. During ray tracing, an idle processor will request extra load from potential candidate workload servers based on previous workload requests. This scheme achieves dynamic load balance through efficient global distribution of pixel calculations.

Our parallel ray tracing implementation is based on the hierarchical bounding volume tree algorithm proposed by Kay *et al.*[4]. The scene represented by this tree structure is decomposed into two levels. The upper level of subtrees, which are frequently used in searching for ray and object intersections, are duplicated on all processors and the lower level subtrees are evenly decomposed among the groups of processors. We form many groups with a given number of processors and thus multiple copies of the entire tree structure exist in the system. Since image coherence is inherent in adjacent pixels, these pixels usually need the same set of data to ray trace them. An efficient cache scheme is very useful to exploit image coherence to reduce the amount of data movement[5–9]. In our implementation, a variable size cache scheme called ALRU is used for this purpose. We determine the best subtree size to be moved around within each group according to the communication overhead and the probability the subtree would be intersected. Many important parameters such as cache size and group size are also examined experimentally. Our results show that there is a critical cache size, optimal extra region size and group size for a scene to be efficiently rendered.

## 2. PREVIOUS WORK

There have been several implementations of ray tracing on parallel machines. Most of these approaches can be classified according to whether they exploit parallelism in *object space* or *image space* [6,10]. In the object space method, each processor is assigned one part of

the whole database. During ray tracing, the paths of rays will likely traverse most of the database and the ray information must be communicated to the processors assigned to the particular portion of the database through which the rays propagate.

In [11–15], the 3D space of the scene was partitioned into many subvolumes. Each processor is assigned one or more subvolumes. In Dippe *et al.*[11], the size and shape of each subvolume is changed dynamically (i.e. objects must be moved) to balance the workload among the neighboring nodes. In this method, when the load is moved by moving any vertex, there are eight regions to be reshaped. This scheme suffers from heavy overhead, and it is hard to determine the correct update frequency. An improvement on this approach was proposed by Nemoto *et al.*[12]. In their method, the shape of rectangular subvolumes is maintained by only sliding the axis-aligned boundaries. This compromise allowed for more efficiency and lower overhead than Dippe's scheme, but at the cost of less effective load balancing.

In another object space scheme, Jevan[15] evenly divided the space into equal size subvolumes and then statically scattered these subvolumes among processors in an attempt to achieve load balancing. This scheme often gives marginal performance. It lacks the flexibility required to handle overloaded or underloaded processors. Boutouch *et al.*[14] adopted a different static load balancing scheme by exploiting *ray coherence*. In this scheme, the load associated with each subvolume can be estimated by subsampling the image (i.e. only a subset of pixels are ray traced) due to the ray coherence property. After this stage, a clustering technique using the median-cut method is used to partition 3D space into two sets of subvolumes with almost the same load. Partitioning is continued recursively until the number of subvolumes is equal to the number of processors. Finally, the subvolumes are mapped onto the processors such that the adjacent subvolumes are assigned to neighboring processors, thereby reducing some of the communication overhead. However, their experimental results showed that this method suffers from an imbalance of load when several processors were used.

In Kobayashi *et al.*[13], the space is divided recursively into octree subvolumes to make sure that each processor is assigned almost the same number of objects. They assumed that the rays will evenly traverse through all objects in order to justify this static load balancing. However, this assumption does not hold in real life examples. For example, zooming in on some part of a scene will lead to very serious load imbalances.

In the image space methods, the screen is split into many subimages which are distributed either statically or dynamically among available processors. In many previous studies[3,9,16–19] each processor was assumed to have access to the whole scene database, so each processor can compute any subimage. Carter *et al.*[9] used a 'pixel region modulo number of processors' scattering decomposition method to achieve static load balancing. However, their method only works well when the number of regions is much greater than the number of processors. Packer[17] and May[18] used a master/slave model in which the size of subimages can be easily changed to control the granularity of tasks. The master processor will become the bottleneck when the number of processors increases. To alleviate this drawback, we proposed two schemes[3]. In local distributed control (LDC) we distribute load imbalances among local neighboring processors only, and in global distributed control (GDC) a mechanism to allow redistribution of load in a global fashion is developed. In both schemes, the task of scheduling is distributed among all processors, and thus the drawback of central scheduling is eliminated. These two schemes are complementary: LDC works well for low computational complexity images and GDC performs well for

high computational complexity images.

The drawback of image space methods discussed above is that the size of the scene database is limited by the size of the local memory of each processor. There have been many schemes proposed to solve this problem. For example, Green *et al.*[5,6] organized the topology of the processors as a tree structure, in which a rooted host processor contains the whole database and the other descending processors use their local memory as a cache memory to reduce the data communication between other processors. When an object is missed locally, a processor will request this object from its parent processor. If this object is also missed at the parent processor, this request message will traverse the tree upward until the missed object is found. A problem pointed out by the author was the potential for the root processor to become a bottleneck, which limits the scalability of the method.

In Badouel's solution [7,8], a shared virtual memory (SVM) invented by [20] was implemented on distributed memory parallel machines to provide a mechanism to have global access to the whole database. In SVM, the whole system memory is divided into many object pages, where an object only belongs to one page. Each page contains a group of objects. All pages are scattered among the processors such that each processor will have about the same number of pages in its local memory. In each processor, the remaining local memory will serve as cache to dynamically capture the most recently used objects. In the case that an object is missed locally, the local processor will send a request to the target processor to bring back an object page that contains the requested object, and then store this page in its local cache memory. Since each object is only located on a single object page stored on a single processor, there is a potential for communication bottlenecks. In addition, the object pages are moved across the whole system, causing additional communication problems. In Carter's approach[9], the bounding hierarchy tree, except for the leaf nodes, is replicated in all processors. The leaf nodes are distributed among the processors. A cache is also used to maintain a set of objects at each processor. Unfortunately, the duplication of all non-leaf nodes cannot allow the rendering of very large databases. In the above three methods, the cache memory is all maintained by the *least recently used* (LRU) policy. The success of this caching scheme is strongly related to the extent of *image coherence* inherent in ray tracing adjacent pixels.

# 3. SEQUENTIAL ALGORITHM

Kay's[4] hierarchical tree is built with a fixed branching ratio using a bottom-up construction. An example of the hierarchical tree is shown in Figure 1.

In this tree, a leaf node is an object and a non-leaf node is a bounding volume, the region of space being occupied by a group of objects or bounding volumes. The bounding-volume—ray intersection is designed to be much faster than a ray—object intersection. If a ray misses a bounding volume, then those objects or bounding volumes contained by it will not be considered in the computation. Thus, the usage of bounding volumes can decrease a major portion of the expensive ray—object intersection calculation.

In our parallel implementation, the finest granularity of workload is a pixel computation. The core of this sequential computation is finding the closest-intersected object determined by the ray traversal algorithm[4]. More than 90% of the pixel computation time is due to the ray traversal computations. A brief description of this algorithm is given below.

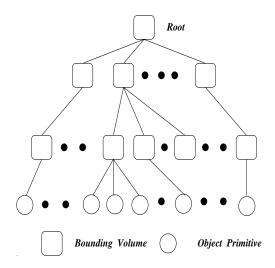


Figure 1. A hierarchical bounding volume tree

## Ray traversal algorithm

```
An incoming ray
t = \infty
p = nil
while the heap is non-empty and the distance to top node < t
      Extract a candidate node from the heap
      if this candidate node is a leaf
             Compute the ray-primitive intersection
            if the ray hits this candidate node and the distance < t
              t = distance
              p = candidate node
             endif
      else
             for each child of the candidate node
                Compute the ray-bounding volume intersection
                if the ray hits the bounding volume
                  Insert the child into the heap
                endif
             endfor
      endif
endwhile
```

For each ray, we start with a ray traversal from the root of the tree and a *priority queue* implemented via a *heap* structure is used to order the bounding volume tests. Only those children nodes intersected with the ray will be considered further. This traversal will continue recursively and will terminate when either no intersections with objects are found, or the closest bounding volume on the top of queue has a larger estimated distance than that of currently found intersected object.

## 4. INTEL TOUCHSTONE DELTA PARALLEL COMPUTER

We performed our implementation on the Intel Touchstone Delta machine. The Delta is a high-speed concurrent multicomputer, consisting of an ensemble of 512 computational

nodes arranged as a  $16 \times 32$  mesh[21]. The nodes are Intel i860 microprocessors, each with its own memory space. Groups of nodes can work on the same problem and communicate with each other by message passing. The Delta architecture has three kinds of message passing methods: synchronous, asynchronous and interrupt handler.

## 5. PARALLEL IMPLEMENTATION OF RAY TRACING ALGORITHM

The problem in our ray tracing parallelization is to find an optimal load and database distribution such that the parallel execution cost for the independent task  $t_0, t_1, \ldots, t_{n-1}$  on N processors is minimum. Each  $t_i$  is the task to calculate the (R,G,B) value at the ith pixel on the image screen. The total execution cost of a task  $t_i$  on the processor is  $C_{t_i} + CC_{t_i}$ , where  $C_{t_i}$  is the processing cost of executing  $t_i$  on a processor and  $CC_{t_i}$  is the communication cost incurred by executing  $t_i$  on that scheduled processor. In our cost model, the communication cost includes the overheads incurred in scheduling  $t_i$  and in fetching remote data required by  $t_i$ . The computational characteristics of  $t_i$ s and their patterns of data access cannot be predicted until runtime. Finding an optimal data distribution is very difficult, and therefore only heuristic approaches can be employed for this class of problems. The main steps of our parallel implementation are the following:

## Parallel implementation of ray tracing

 $step_0$ : Host parses database file and does database partition.

 $step_1$ : Load data onto the memory of the processors.

 $step_2$ : Construct the hierarchical tree extents.

 $step_3$ : Ray trace the assigned pixel regions.

step<sub>4</sub>: Collect rendered results and display image.

The heuristic methods used in our implementation are given in the following subsections.

## 5.1. Database partitioning and distribution

In our implementation, the scene is represented by a hierarchical bounding volume tree structure. To compute the (R,G,B) values for a pixel, the ray traversal algorithm is used to parse this tree structure. It can be seen that bounding volumes near the root node will be tested against an incoming ray more often than nodes at lower levels. As the ray intersection testing moves down the tree, the number of objects or bounding volumes tested against will be smaller. Therefore, it is expected that the major portion of the intersections of ray tracing will occur in the upper part of the tree hierarchy. We also know that the storage requirements increase exponentially as we travel down the tree. The lower portion of the tree is responsible for more than 90% of the total memory requirements. Therefore, we decided to duplicate all bounding volumes of the upper part of the tree in each processor (which will be referenced more frequently) and hold only a small portion of the lower part in each processor. Using this distribution, we remove the cost of fetching the upper part of the tree from  $CC_{t_i}$ . Later, we describe the cache scheme to decrease the cost of accessing the lower part of the tree. Since there is no prior way to decide which part of the lower level tree is used most frequently, the best way is to distribute these lower level subtrees evenly among the processors. To achieve this initial data distribution, three steps are used.

In the first step, the whole database is sorted and partitioned into  $2^d$  sets, where each

set of objects are near one another in 3D world space. A median-cut scheme is used for this purpose. The idea is to simply cut sorted objects (in one dimension) according to one co-ordinate axis, say the X-co-ordinate, into two sets. Next, these two sets are cut into two sets along the Y-axis. This process of cutting the objects into subsets will be repeated along a selected axis based on the range of that co-ordinate. This is repeated until we have  $2^d$  sets and each set has N objects ( $N \leq N_{limit}$ ).  $N_{limit}$  is the maximum number of objects that can be stored in the memory of each processor. If the system has  $2^D$  processors, it will be divided into  $2^{D-d}$  groups, where each group can have the entire database.

Our implementation provides multiple copies of the lower part subtrees (i.e.  $2^{D-d}$  copies) in the system. This implementation differs from those of [5,7–9]. There are many advantages of having multiple copies of the scene data in the system. Firstly, with multiple copies, we can potentially reduce the probability of having a data server bottleneck. Secondly, we require the non-local subtree access be limited within the same group of processors to avoid global data communication. A global-wide remote data access is believed to require more communication links than a remote data access occurring in the neighboring processors.

In our implementation, a Sun host is used to partition the database and to distribute multiple copies of the entire database among groups of processors. Let  $P_{ij}$  denote the jth processor of the ith group, where  $0 \le i \le 2^{D-d}-1$  and  $0 \le j \le 2^d-1$ . By using the median-cut method, the database is partitioned into  $2^d$  sets denoted by  $X_0, X_1 \dots X_{2^d-1}$ . The number of objects in each partition  $X_j$  is almost the same (the difference in the number of objects in any two partitions is just 1).

In the second step, we receive these data partitions from the host through processor  $P_{00}$ . The initial data distribution is accomplished as follows:

# The initial data distribution

```
 \begin{cases} \textbf{for } j = 1 \text{ to } 2^d - 1 \textbf{ do} \\ P_{00} \text{ receives } X_j \text{ from the host;} \\ P_{00} \text{ sends } X_j \text{ to } P_{0j}; \\ P_{0j} \text{ multicasts to } P_{ij} \text{ for all } i; \setminus \star \text{ sends to corresponding processor in other groups. } \star \setminus \textbf{end;} \\ P_{00} \text{ receives } X_0 \text{ from the host;} \\ P_{00} \text{ multicasts } X_0 \text{ to } P_{i0} \text{ for all } i; \end{cases}
```

After receiving each block,  $P_{00}$  sends it to  $P_{0j}$  via a 'forced' type message<sup>1</sup> while the multicasting from  $P_{0j}$  to other groups is done through an 'unforced' type message to keep  $P_{00}$  free to receive data from the host. This way we can pipeline moving data from  $P_{00}$  to  $P_{0j}$  and then to the other groups.

In the third step, each processor builds its local lower subtrees on its own assigned objects. The subtrees are built with a branching ratio of 8 using a bottom-up construction. Let i be the level number at which we want to break the tree into upper and lower parts. Each processor builds its local subtrees up to level i, also determined by the size of the available memory. After this initial construction, those branching bounding volumes on

 $<sup>^1</sup>$ In Delta, the message passing bandwidth is about 8–10 MB/s using a force-type message and about 5–7 MB/s using synchronous unforced message types.

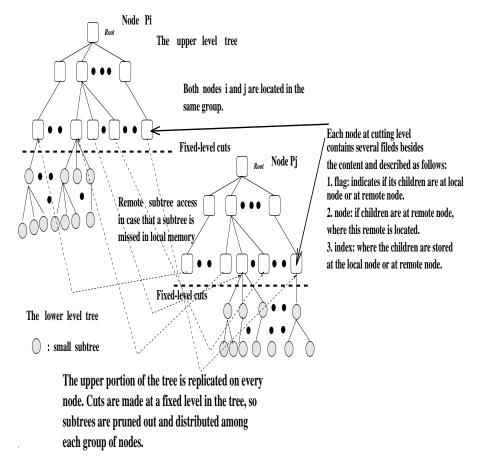


Figure 2. Each group of processors keeps a copy of the two-level tree

level i are in turn multicasted to each processor within each group. This way, all shared branchings are local in all processors and can be used to build the upper level of the tree hierarchy. After this step, we complete Kay's hierarchical bounding volume tree, where the upper level is duplicated on each processor and the lower level is evenly distributed among the processors in each group (see Figure 2). For example, in the  $4 \times 8$  partition of the Delta, we divide 32 processors into eight groups. Each group has four processors and holds a copy of entire two-level tree structure (see Figure 3).

## 5.2. Dynamic data management and cache scheme

When a task  $t_i$  is executed on processor  $P_j$ ,  $CC_{t_i}$  is large if the data required by  $t_i$  is not placed on  $P_j$ 's local memory at initial data distribution time. The degree of affinity between  $t_i$  and  $P_j$  determines the total execution cost of  $t_i$ . Within any image, the similarities can be seen everywhere in continuous pixels or blocks of image. Ray tracing these continuous pixels always requires the same subset of the lower part of the tree. This similarity is called *image coherence* in computer graphics and its degree highly depends on the scene rendered.

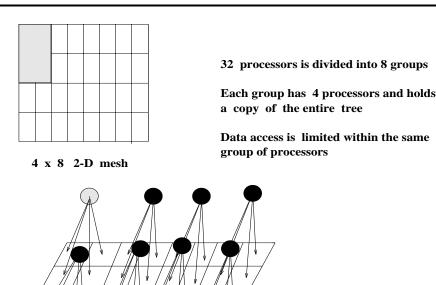


Figure 3. Multiple copies of two-level tree exist in the system

The distribution of  $t_i$  must exploit image coherence in decreasing  $CC_{t_i}$ . This issue will be addressed later. Here, we assume that our load distribution method can capture image coherence and we propose a ALRU cache scheme to reduce the amount of data movement in continuous  $t_is$  (i.e. increase the degree of affinity between  $t_is$  and  $P_i$ ).

Any processor may need to access part of the lower level tree with very unpredictable access patterns. Each processor knows the stored location of each lower part subtree, because the roots of all subtrees are known after our initial data distribution. Therefore, a processor  $P_i$  can determine which processor  $P_j$  to ask for missing data. In several previous papers [5,7–9], a fixed size cache memory is emulated for the non-local data. When a non-local data reference is required, a group of objects are fetched and stored in a single data page. Dynamic data management is maintained in cache memory by the LRU policy. To implement the LRU policy, sorting is necessary to maintain the order of the data page reference in cache. In case of a data miss, the cost is  $O(N \log N)$ , where N is the size of the cache. For large values of N this cost can be significant.

In our parallel implementation, the data are organized as a tree structure where the size of the subtree is also variable. Therefore, a fixed size data page is not suitable. We adopt a different approach to cache implementation. A fixed size continuous memory is allocated and called the cache pool. A double-linked list is used to maintain the order of data reference in the cache pool (see Figure 4). In the case that a subtree is non-local, we will first check if there is enough space in the cache pool. If there is space, we allocate a block for the incoming subtree. Then, the double-linked list will maintain the usage of this allocated block using the ALRU strategy (LRU with variable cache size). ALRU requires the update of the double-linked list after each reference. When a resident block is referenced or a new block is allocated, it will be placed at the head of the list. For replacement we choose a

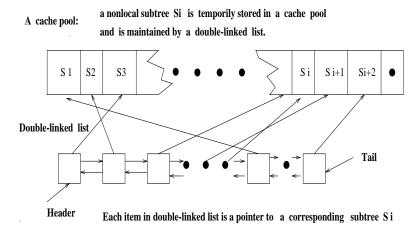


Figure 4. Dynamic data management for non-local subtrees

cache block starting from the tail of the list. Therefore, using ALRU, blocks in the cache list are always in the most to least recently referenced order, without the exact sorting overhead incurred in LRU (see Figure 5). However, ALRU needs to adjust the pointers when a reference is made to each cache block.

The ALRU cache scheme is adopted to capture the most recently used non-local subtrees. It can greatly reduce  $CC_{t_i}$ s for executing continuous  $t_i$ s tasks with a stronger degree of image coherence. However, the degree of image coherence depends on many factors, such as the scene data and view point. It is difficult to model this effect formally; therefore, it is hard to find the best cache size automatically for any scene. However, the key to selecting

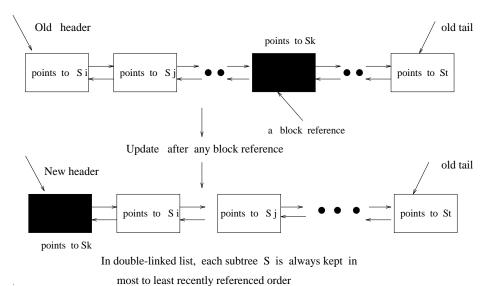


Figure 5. ALRU policy

a good cache is to make sure that the cache is large enough to hold the lower part of the subtrees for a given ray that is assigned to each processor. This will at least avoid thrashing in the cache system. In Kay's ray traversal algorithm in the worst case, a ray will travel through all lower subtrees. Fortunately, this is very rare in real data. We will show the effect of cache size on performance in our discussion of the experimental performance evaluation section.

## 5.3. Efficient data movement strategy

Next, we need to decide the most efficient size of the subtree to be transferred at one time. It is not wise to transfer the subtree whole at one time. This can exhaust the cache pool very quickly and may result in unnecessary cache block swapping. In the ray traversal algorithm, if a bounding volume is hit, the ray will test against all of its children (eight children in our implementation). In a message passing system such as the Intel Delta, there is a higher cost in communication setup time, plus a smaller cost for each byte transferred. In order to decrease the overall cost of message passing, long messages are preferred. Therefore, it makes sense to transfer all the children at one time, rather than one by one. We need to know the optimal message size, and to this end we measured the time necessary for transferring data from one node to another in the Delta. In our implementation, a processor uses asynchronous send/receive to request data servers, which use message handler routines to transfer the requested data. We measured this round trip time experimentally by the following steps:

# The measure of round trip time

 $Node_i$ : starts the clock; sends a request to  $Node_i$   $(i \neq j)$ .

 $Node_j$ : as the request message arrives, a message handler routine sends reply message back.

 $Node_i$ : stops the clock after receiving the reply.

We tested different sizes of reply message varying from 1K to 512K bytes and averaged over 10,000 runs. The round trip time cost (in  $\mu$ s) can be modeled by:

$$Cost(l) = 286.4 + 0.133l$$
 (1)

In Equation (1),  $286.4 \mu s$  is the setup time for this round trip and its cost is a factor 0.133 of the message size l. Each non-leaf bounding volume needs 100 bytes of storage in our implementation. Hence, we let:

- $T(1) = Cost(8 \times 100)$  and is the time to transfer eight children of one bounding volume at one time (i.e.  $8^1$ ). This single level subtree is called S(1).
- $T(2) = Cost(72 \times 100)$  and is the time to transfer a 2-level bounding volume subtree, which is called S(2) at one time (i.e.  $8^1 + 8^2$ ).
- T(n) is the time to transfer an n-level bounding volume subtree, which is called S(n), at one time.

We find the condition when it is efficient to transfer messages by S(2) rather than multiple S(1) messages by the equation:

$$T(2) \le T(1) + k \times T(1) \tag{2}$$

We find  $k \ge 2.163$ , which means that if more than two of eight bounding volumes on one level are hit, we should use S(2). The probability that Equation (2) is true is high, so we adopt S(2). To transfer subtree S( $n \ge 3$ ) as a message, the storage requirements increases exponentially and it is hard to predict their future usage. Thus, we consider the S(2) subtree to be the most reasonable size. Therefore, in our tree representation, each subtree of the lower part of the tree hierarchy is decomposed into many S(2) subtrees. Each S(2) subtree is stored in continuous storage and is ready to be fetched. Traveling S(2) by the depth-first or breadth-first order, S(2) can be stored easily in continuous storage. Each element of S(2) contains the index of each child by which it can find a specific child in this storage.

## 5.4. Load distribution and load balancing

Our load distribution has two steps: a static load initialization where image coherence is considered, and a run time load redistribution which smoothes out load imbalances incurred in the first step. In the first step, given N processors, the image screen is partitioned into N rectangular subimages of comparable size. Each subimage is further subdivided into fixed square regions which are the units of load redistribution. The first step attempts to give a roughly balanced load among processors. More importantly, in each subimage, pixels are continuous or regions are close in 2D. Therefore, better image coherence is maintained and the possibility of non-local data movement is reduced. For load redistribution, square pixel regions are preferred in that they have more potential for exploiting image coherence than the long and skinny rectangular regions [10]. After the database and initial load distribution, each processor begins to ray trace its allocated subimage. At run time, a load balancing scheme is used to achieve a global balanced load.

# Screen space

0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

Figure 6. Snake mapping assignment

We logically organize the N processors of the Intel Delta in a ring topology (see an example in Figure 6). Two neighboring processors will be assigned two neighboring subimages, also due to image coherence. When a particular node becomes idle, say  $P_i$ , it requests extra work from successive nodes on the ring,  $P_{i+1}, \ldots, P_N, P_1, \ldots, P_{i-1}$ , until it finds a node that is busy. This busy node, say  $P_j$ , dispatches a square pixel region  $(a \times a)$  to this request. The node  $P_i$  will remember that node  $P_j$  was the last node from which a request was made. The next time it is idle, it will start requesting work from node  $P_j$ , bypassing nodes between  $P_i$  and  $P_j$ . An additional feature of this strategy is that if, in the meantime, node  $P_j$  becomes idle and  $P_j$  remembers that it received work from node  $P_k$ , node  $P_i$  will

jump from  $P_j$  directly to  $P_k$  without asking for work from nodes between  $P_j$  and  $P_k$ . In this strategy, node  $P_i$  stops its search for work if it ends up at itself or at some node that it has already visited or skipped in a search step. This ensures that the search will end when all nodes are idle.

In our implementation, the initial subimage is stored at the task queue of the processor allocated that subimage. Each processor will fetch an  $a \times a$  (a bundle of  $t_i$ s pixels) region for execution at one time. When it becomes idle, it will try to find an extra  $a \times a$  region from other busy processors along the ring. The size of the  $a \times a$  region is used to control the granularity of the load distribution. This size depends greatly on the scene rendered and other factors, and it is hard to find the optimal value for all images. From our experience with test scenes, the optimal size ranges from  $4 \times 4$  to  $8 \times 8$ .

#### 6. EXPERIMENTAL RESULTS AND DISCUSSIONS

We used a set of standard scenes from Eric Haines's database to perform our experimental evaluation[22]. Table 1 shows the geometric characteristics of three scenes. These test scenes have been used in many previous studies and are believed to be a good representation of real data[6,7].

Table 1. Database characteristics

Scene	Number of triangles	Number of light sources
Ball	157441	3
Mountain	33536	1
Tree	57122	7

To evaluate our scheme, we make the assumption that the maximum available memory size to store the database is 4 MB. This size is about 0.25 times the available memory size of each processor on the Delta. This allows us to experiment with distributing the database. We divide this 4 MB into two parts: 2.5 MB (called PM) is used to store both the entire upper level tree and part of the lower level subtrees, while 1.5 MB (CM) is used as cache memory. In this configuration, the maximum number of primitives is approximately 6000 triangles and the number of bounding volumes on the cutting level is less than 4096 ( $8^4 = 4096$ ). Table 2 shows how we store three test scenes. In this table, the memory requirement is the size of the whole hierarchical tree, N-way is the number of processors required to store the whole tree, and the ratio is CM over the memory requirement. Of the three scenes, the Balls scenes is the largest, and the available cache memory is small. The small cache can potentially lead to more data communication overhead in the processing of rays.

Table 2. Data distribution for three test scenes

Scenes	Balls	Tree	Mountain
Memory requirement, bytes $N$ -way Ratio of $CM$ to total memory requirement, %	48404088	17074408	10124216
	32	16	8
	3.25	9.2	15.5

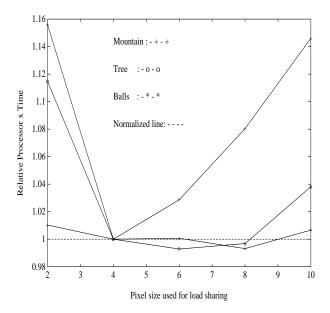


Figure 7. Relative time for 3 scenes for different square region sizes (pixel size  $\times$  pixel size).

In our experimental evaluation, several key parameters affect the performance of our parallel ray tracer. These parameters include the size of the square region used in load sharing among processors, the type of load balancing scheme employed, the size of software cache in each processor, and the number of copies of the entire database (group size). In order to study scalability, we ran our experiments on the Delta machine using 16 to 512 nodes. All rendering timings are for an image size of  $512 \times 512$  with no anti-aliasing. Unless stated otherwise, the following values are used in the experimentation: 2.5 MB for PM, 1.5 MB for CM and  $4 \times 4$  for a region size. If we do not need the entire 2.5 MB for PM, then the remaining memory is also allocated to CM.

The results of our experiments regarding pixel region size used in load sharing are shown in Figure 7. For each region size selected, we ran experiments with the number of processors varying from 16 to 512. We calculated the total processor time product (PT) as  $\sum_{i=4}^{9} {_{or}} {_{5}} 2^{i} T_{2^{i}}$ , where  $T_{2^{i}}$  is the time taken for parallel rendering with  $2^{i}$  processors. The total processor time product with a 4 × 4 pixel region is used to normalize the results of other region sizes. As evidenced by Figure 7, the parallel rendering time was best for all three scenes with a small size square region for load sharing; a region size in the range 4 × 4 to 8 × 8 gives the best overall performance for the three scenes studied.

In the next three Figures (Figures 8–10) we present our comparative evaluation of load balancing schemes. For these experiments, we considered four different load balancing policies; our scheme, which is called GDC; a master–slave (MS) scheme[9]; scatter interleaved (SC)[9]; and guided self-scheduling (GSS)[23]. Again we used the three standard scenes and used a  $4 \times 4$  region size for load sharing between processors. Figures 8–10 show plots of the number of processors versus speedup relative to 16 processors for both tree and mountain scenes and speedup relative to 32 processors for the balls scenes. For all three scenes, the GDC scheme performs best and is quite scalable. In the case of the mountain

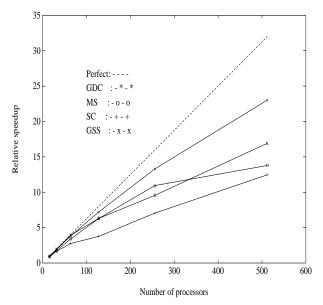


Figure 8. Relative speedup for different load balancing strategies (the tree scene)

scene, the significantly larger amount of computation and smaller memory requirements minimize the amount of data movement. Thus, the MS scheme also performs well.

In our implementation, each processor has a certain amount of memory set aside as software cache. The cache size is the same in all processors. To evaluate the effect of

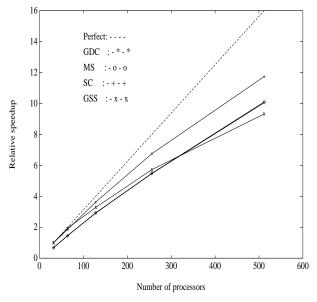


Figure 9. Relative speedup for different load balancing strategies (the balls scene)

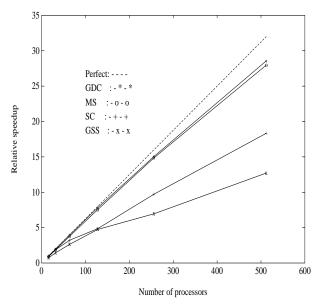


Figure 10. Relative speedup for different load balancing strategies (the mountain scene)

cache size on performance, we let the size vary from 1 to 10% of the size of the scene database. For this set of experiments, we fix the number of processors to 64 and do not allow the remaining portion of PM to be used as extra CM. In general, we can expect improved performance with increasing cache size. In our experiments, we found that beyond a certain critical cache size the improvement in performance is insignificant. In Figure 11, we plot cache size versus relative efficiency, where relative efficiency is defined as the ratio of

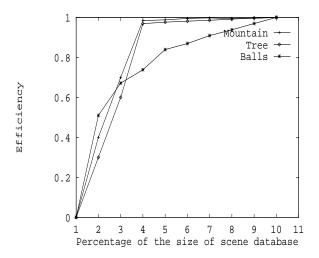


Figure 11. Variable cache sizes for three scenes using 64 processors

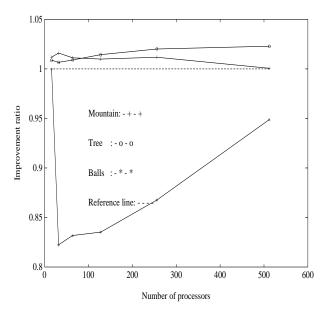


Figure 12. Relative improvement for three scenes using smaller group size

rendering time with a given percentage of cache to that of rendering time with 10% cache. As evidenced by Figure 11, for the mountain and tree scenes there is very little improvement in performance beyond the 3% cache size. However, the balls scene is more sensitive to cache size.

In our parallel implementation, a scene database is partitioned and stored in the memory of a number of processors. With a group size of eight, we will have the entire database in the memory of eight nodes. Thus, if we increase the amount of memory used for the database and reduce the cache size, we will have more copies of the database in the system. With a larger group size, we spread the database among more processors and so we will have increased cache size. To evaluate the effect of group size on performance, we increase the size of PM to 3.5 MB and (CM size is 0.5 MB). With this new arrangement, the group size of the mountain scene will be four instead of the default size, the group size for the tree scene will be eight instead of 16 and the group size for the balls scene will be 16 instead of 32. The new ratios of cache size to the whole database for the three scenes are 5.2% for the mountain scene, 3.2% for the tree scene and 1.08% for the balls scene. Figure 12 shows the results for having the group size change with respect to the default size. The plot shows performance ratio versus number of processors for two different group sizes. Reducing the group size from the default one gives a small improvement for the mountain and tree scene. However, for the balls scene the default size performs better. In Figure 12, we showed that performance of the mountain and tree scenes degrades when the cache size is less than 3% of the database. Both new cache ratios are now beyond 3%, so the performance improves by decreasing the group size. In contrast, the new ratio for the balls scene is too small and thereby degrades the performance. This experiment shows that the group size does effect the performance and needs to be carefully selected depending on the scene.

#### 7. CONCLUSIONS

In this paper, we presented experimental results for a load balanced, distributed memory parallel ray tracer on the Intel Delta. Several important parameters affect the performance of parallel ray tracing with a distributed database. Our implementation uses a hierarchical tree decomposition approach to store the scene database in a group of processors. We developed the software cache scheme to take advantage of locality in pixel computations, and thus improve the performance. Experiments were conducted extensively to investigate the scalability of various parameters on the performance of parallel ray tracer. Our conclusions are that there is a critical cache size which depends on the scene being rendered, and that there is an optimal group size and region size in load sharing for a given scene. In our earlier work, we developed the GDC load balancing policy for ray tracing with each node having a copy of the database[3]. Here, we showed that GDC with some modification also performs well and is scalable for a distributed database.

## **ACKNOWLEDGEMENTS**

This research was performed in part using the Intel Touchstone Delta System operated by California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest Laboratory, a multipurpose national laboratory which is operated for the US Department of Energy by Battelle Memorial Institute under contract DE-AC06-76RLO 1830. This research is supported by the Boeing Centennial Chair Professor funds.

## REFERENCES

- 1. Turner, 'An improved illumination model for shaded display', Commun. ACM, 23, (6), (1980).
- 2. An Introduction to Ray Tracing, A. S. Glassner (Ed.), Academic Press, 1989.
- 3. T. Lee, C. S. Raghavendra and J. B. Nicholas, 'Experimental evaluation of load balancing strategies for ray tracing on parallel processors', in *International Conference on Parallel Processing*, 1994.
- 4. T. L. Kay and J. T. Kajiya, 'Ray tracing complex scenes', ACM SIGGRAPH 1986, August 1986.
- S. Green et al., 'Exploiting coherence for multiprocessor ray tracing', IEEE Comput. Graph. Appl., 9, (6), (1989).
- 6. S. Green, Parallel Processing for Computer Graphics, MIT Press, 1991.
- 7. Badouel et al., 'Ray tracing on distributed memory parallel computers', SIGGRAPH 1990.
- 8. Badouel *et al.*, 'Distributed data and control for ray tracing in parallel', *IEEE Comput. Graph. Appl.*, July 1994.
- 9. M. B. Carter and K. A. Teague, 'The hypercube ray tracer', in *Proc. of the 5th Distributed Memory Computing Conference*, 1990.
- S. Whitman, Multiprocessor Methods for Computer Graphics Rendering, Jones and Bartlett, 1992.
- 11. M. Dippe and J. Swensen, 'An adaptive subdivision algorithm and parallel architecture for realistic image synthesis', *Comput. Graphics*, (July 1984).
- 12. K. Nemoto and T. Omachi, 'An adaptive subdivision by sliding boundary surfaces for fast ray tracing', in *Proceedings of Graphics Interface* '86, 1986.
- 13. H. Kobayashi *et al.*, 'Parallel processing of an object space for image synthesis using ray tracing', *Vis. Comput.*, **2**, (1), (1987).
- 14. K. Bouatoch and T. Priol, 'Parallel space tracing: An experience on an iPSC hypercube', in N. Magnenat-Thalmann and D. Thalmann (Eds.), New Trends in Computer Graphics, 1988.
- D. A. J. Jevans, 'Optimistic multi-processor ray tracing', in R. A. Earnshaw and B. Wyvill (Eds.), New Advances in Computer Graphics, Springer Verlag, 1989.

- M. J. Muuss RT and REMRT, 'Shared memory parallel and network distributed ray tracing programs', Proc. USENIX Assoc. Comput. Graph. Workshop, 1987.
- 17. J. Packer, 'Exploiting concurrency: a ray tracing example', in *The Transputer Application Notebook*, Inmos, 1989.
- 18. D. May, Toward General Purpose Parallel Computers, MIT Press, Cambridge, 1989.
- 19. M. Potmesil *et al.*, 'A parallel image computer with distributed frame buffer: system architecture and programming', in *Eurographics 89*, North Holland, Hamburg, 1989.
- K. Li, Shared Virtual Memory on Loosely Couple Multiprocessor, PhD dissertation, Yale Univ., 1986.
- 21. Touchstone Delta User's Guide, Intel Corporation, October 1991.
- 22. E. Haines, 'A proposal for standard graphics environments', *IEEE Comput. Graph. Appl.*, **7**, (4), (1987).
- 23. C. Polychronopoulos *et al.*, 'Guided self-scheduling: A practical scheduling scheme for parallel computers', *IEEE Trans. Comput.*, (12), (1987).