ELSEVIER

Parallel Computing 23 (1997) 1993–2015

PARALLEL COMPUTING

Practical aspects and experiences

Exploitation of image parallelism for ray tracing 3D scenes on 2D mesh multicomputers

Tong-Yee Lee *

Department of Computer Science and Information Engineering, National Cheng-Kung University, Tainan, Taiwan, ROC

Received 15 June 1996; revised 23 May 1997

Abstract

Ray tracing is a well known technique to generate life-like images based on models of light shading, reflection, and refraction. The massive computation and memory demands of ray tracing complex scenes have long motivated researchers to use parallel processing in reducing the ray tracing time. This paper gives a study of parallel implementation of a ray tracing algorithm on a distributed memory parallel computer. The computational cost of rendering pixels and patterns of data access can not be predicted until runtime. To efficiently parallelize such an application, the issues of database partition, data management and load balancing must be addressed. In this paper, we discuss the ways of database partition and propose a dynamic data management scheme which can exploit image coherence to reduce data communication time. A global load balancing mechanism is presented to ensure a good load balance among processors during ray tracing time. The success of our implementation depends crucially on a number of parameters which are experimentally evaluated. © 1997 Published by Elsevier Science B.V.

Keywords: Ray tracing; Communication; Distributed-memory multiprocessor; Load balancing; Scalability

1. Introduction

Ray tracing is an object space rendering technique for image synthesis [1]. It can produce very high quality 2D pictures of a 3D scene. Ray tracing has been a major area

* E-mail: tonylee@mail.ncku.edu.tw.

PII S0167-8191(97)00099-9

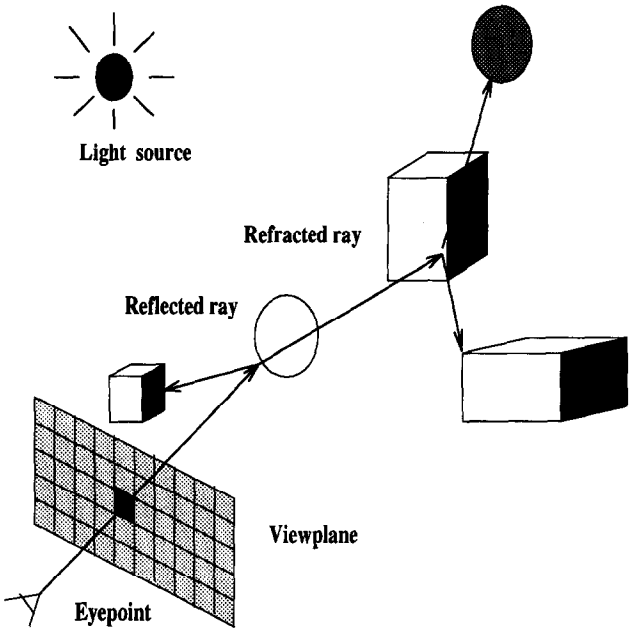

Fig. 1. The process of ray tracing.

of research for two reasons. First, it can determine visible surfaces, compute shadows, model reflection and transmission of lights and create other special effects. Second, it is very simple and elegant in both concept and algorithm. Ray tracing is based on simple models of light shading, reflection, and refraction [2]. As shown in Fig. 1, the main rays are sent from the eye through pixels on an imaginary view-plane and are traced as they are reflected and transmitted by objects. The reflected and transmitted rays are in turn traced until a maximum tracing depth is reached or the rays no longer hit any object. For each pixel, the ray tracing algorithm calculates the (R, G, B) value at the intersection points between rays and objects. Each pixel must be examined and typically it requires hundreds of thousands to millions of rays to be traced. The amount of computation time required for each pixel depends primarily on the number of intersections between eye-rays and reflected rays and the objects in the scene. Therefore, ray tracing is very computationally expensive; it can take several hours to ray trace an image on current workstations.

In the past, research has been focused on minimizing the cost of intersection calculations. Space partitioning structures, bounding volumes, shadow buffers and ray coherence techniques have all been proposed to reduce this cost. Many techniques have exploited different data structures to speed up the search for a closest intersection on a ray. These data structures such as Octrees, BSP trees, K-D trees, and nested bounding volumes, allow us to search a small percentage of the scene to determine the closest intersection. A detailed discussion of the above mentioned techniques is beyond the

scope of this paper. For more details, see James Arvo and David Kirk's excellent article [2] on the survey of ray tracing acceleration techniques.

There are growing needs in ray tracing increasingly more complex scenes and in requiring shorter rendering time. Acceleration techniques mentioned above are still too slow to make ray tracing efficient on a sequential machine. On the other hand, many efforts have been pursued to exploit either more hardware processing power or a supercomputer for ray tracing problems with an attempt to reduce the total ray tracing time. Such methods use different image subdivision schemes and ways to efficiently schedule these small image regions. These approaches potentially provide a reasonable rendering time for ray tracing complex scenes.

In this paper, we consider the parallel ray tracing problem with the database partitioned among the processors on the distributed memory parallel computers (DMPC). This problem is required when the size of database is too large to be duplicated on each processor. Our parallel techniques belong to the image space parallelization class of algorithms.

We form many groups with a given number of processors and allow multiple copies of the entire database to exist in the system. The database is decomposed into two parts decided by the available memory on each processor. One part, which is frequently used to speed up the search for a closest intersection point, is duplicated on each processor. The other part, which is the main contributor of the memory requirement, is evenly decomposed among the groups of processors.

Our data partition scheme is suitable for parallelizing any ray tracing acceleration scheme. The amount of data to redistribute is determined by the data size, the number of processors, and a specific scene. The redistribution cost grows as the data size and the number of processors increase. To reduce the data size requirement, we adopt the hierarchical bounding volume tree algorithm proposed by Kay et al. [3] in our implementation. This scheme in general needs $O(n \log n)$ memory requirement, where n is the number of objects in the scene. Dynamic data management is achieved by a variable size cache scheme to exploit image coherence to reduce data movements in ray tracing consecutive pixels. The ALRU (another least recently used) strategy is designed for cache replacement policy.

Many important parameters such as the cache size and the group size are also experimentally examined. Our results show that there is a critical cache size, optimal extra region size and group size for a scene to be efficiently rendered. In our earlier work [4], we described load balancing techniques for parallel ray tracing. These balancing techniques are scalable for both high and low computational complexity images when the entire dataset is duplicated on each processor. In this paper, we will show that our scheme with some modification is still scalable in the case that the dataset is too large to store on the memory of each processor.

We implement our parallel raytracer on the Intel Delta DMPC at Caltech, consisting of an ensemble of 512 computational nodes arranged as a 16×32 mesh. 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 handling.

2. Parallel ray tracing problems on the DMPC system

Since the computation of each pixel is independent of the others, ray tracing is potentially highly suitable for parallel processing. However, obtaining efficient scalable parallel rendering with ray tracing is non-trivial. A naive approach to parallelizing this problem is to assign equal number of pixels to each processor. However, because pixel calculation time can vary significantly, some processors will have much more computation to perform than others. The key to achieving efficient parallel ray tracing is to ensure that the processors have closely equivalent amounts of computation to perform.

In ray tracing the scene, there is no *prior* knowledge to determine which parts of the scene database individual pixels will access. 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 may have to be distributed on the DMPC. If some of the data which is necessary for ray tracing the current pixel is not available at a local processor, data must be remotely accessed from other processors in the system. This increases communication cost required for the pixel computations. We must judiciously choose the data distribution scheme to avoid excessive data movements.

2.1. Parallel ray tracing problem formulation on DMPCs

The scheduling of ray tracing problem on DMPCs is a subset of the general scheduling problem, which has been extensively studied by researchers in a theoretical context [5,6]. Scheduling in general is an NP complete problem [5].

Let I be the image space domain, which can be decomposed into L disjoint subregions, $\{r_1, r_2, \ldots, r_L\}$, which can be independently computed by a ray tracing algorithm. Let D be the database for rendered scene which can be partitioned into many disjoint subsets, $\{d_1, d_2, \ldots, d_M\}$. Constrained by the memory size of each processor, these subsets can be either distributed or duplicated among the processors completely or partially. Further, let us say we are using a DMPC that has P processors. The computation time for each r_l can vary significantly and can not be predicted until run time. The number of d_ms required for ray tracing each r_l is also unknown a priori.

The parallel ray tracing problem is to find an optimal solution in scheduling both r_l and d_m such that the total execution time on a DMPC is minimized. That is,

$$\min\left\{\max_{1\le k\le P}\left\{\sum_{l\in I} S_{lk}(T_{lk}+C_{lk})\right\}\right\} \tag{1}$$

where T_{lk} is the total ray tracing time for region r_l on processor P_k plus the overhead for scheduling r_l on P_k, C_{lk} is the communication overhead incurred by accessing the required data d_ms for ray tracing r_l on P_k, and S_{lk} is a zero-one function which indicates if r_l is scheduled on P_k.

To find an optimal solution for the above problem is NP-complete. To complete ray tracing on parallel machines in a reasonable amount of time, heuristics are used to find a suboptimal solution. In the following section, related work on parallel ray tracing will be reviewed.

3. Previous work in parallel ray tracing

There have been several implementations of ray tracing on parallel machines. In [8–10], the 3D space of the scene is partitioned into many sub-volumes. Each processor is assigned one or more sub-volumes. In Dippe et al. [8], the size and shape of each sub-volume is dynamically changed (i.e., objects must be moved) to balance the workload among the neighboring nodes. On the other hand, Jevans [9] evenly divides the space into equal size sub-volumes and then statically scatters these sub-volumes among processors in an attempt to achieve load balancing. Bouatouch et al. [10] adopt a different static load balancing scheme by exploiting *ray coherence*. In this scheme, the load associated with each sub-volume can be estimated by sub-sampling 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 sub-volumes with almost the same load. Partitioning is recursively continued until the number of sub-volumes is equal to the number of processors.

Goldsmith et al. [11] and Caspary et al. [12] independently propose similar hierarchical tree database partition schemes. Considering the statistics of hit ratio between rays and subtrees, they duplicate only the top n levels of tree structure, and evenly distribute the lower level of subtrees among all processors. During ray tracing, no database movement is involved. The main difference in the two schemes is as follows: Goldsmith et al. achieve load balancing by statically scattering the image regions among the processors. On the other hand, Caspary et al. allocate two processes to each processor in an attempt to better distribute the load. One process dynamically requests sub-images from the host to compute the intersections between rays and the upper trees. The other process is statically responsible for the intersections of rays with the lower trees which are stored on that processor.

Lefer [13] proposes a computation-driven approach which is very similar to Caspary's scheme. In this approach, two tasks are allocated in each processor: one is to determine the candidate object list for a ray within a region, and the other is to calculate the intersection points and photo-realistic effects. The database is organized as a hierarchical SEADS structure where non-leaf voxels are duplicated on each processor and the object primitives are evenly distributed among the processors. All ray messages must be scheduled by a server processor.

In [7,14,15], the screen is split into many sub-images which are distributed either statically or dynamically among available processors to balance workload. Green et al. [7,14] organize 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.

In Badouel's solution [15], a *shared virtual memory* (SVM) invented in [16] is 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.

4. Database partitioning

4.1. Interleaved partitioning

Since data access patterns for rendering image regions is unknown until run time, it is very difficult to find the best way to partition the whole database among the processors. Usually, the database is evenly partitioned and distributed among the processors in either a continuous or in an interleaved fashion. Continuous data distribution can lead to imbalances in both workload and data access; bottlenecks can occur which degrade the network performance. Interleaved partitioning potentially solves the above problems because it randomizes the data such that data is uniformly accessed. Therefore, network communication can be uniformly distributed.

In Eq. (1), C_{lk} is contributed from the data access (i.e. data redistribution). For simplicity, we assume that each processor needs $1/P$ of the total database as in [17], D, to ray trace all scheduled r_l. If the database is distributed in an interleaved way, on the average, each processor holds $1/P$ of those needed for all scheduled r_l. Therefore, the data redistribution cost for each P_i is:

$$\left(D/P - D/P^2\right) \times U_{\text{comm}} \tag{2}$$

where U_{comm} is the communication cost to redistribute an object. The cost incurred in C_{lk} increases as the number of the processors increases.

Two-level Partitioning In Eq. (2), all data is assumed to be ray traced with equal probability. Many ray tracing techniques exploit hierarchical structure to accelerate the ray tracing. The hierarchical structure can influence the data partitioning. In our ray tracing, a hierarchical bounding volume tree is used. In this structure, any bounding volume gets hit by a ray then its children nodes must be tested by that ray. To determine the closest intersections, in general, the bounding volumes near the root are frequently tested against, and in contrast, the nodes approaching the leaf nodes are referenced less often. Clearly, the most referenced nodes should be readily available at each processor. Experimental results show that the number of intersection tests occurring in the upper part of the tree contribute to the major portions of the ray intersection tests.

When a bounding volume object or a primitive is missed locally, then communication is needed to access that data. The communication cost is proportional to the number of intersections against the data which is missed. Two level partitioning, where the upper level of tree is duplicated and the remaining subtrees are evenly distributed among the processors, can reduce the data redistribution size. With two level partitioning, Eq. (2) becomes

$$\left(1 - H(h)\right) \times \left(D'/P - D'/P^2\right) \times U_{\text{comm}} \tag{3}$$

where $H(h)$ is the ratio of the number of intersection tests occurring in the upper

level over the total number in the tree and D' is the number of data located in the lower level of the tree. $H(h)$ is a function of the depth of the tree and h is the level at which to partition the tree into two parts. After extensively experimenting many real scenes, we found that $H(h)$ ranges from 50% to 80% after an appropriate h depth. Therefore, to reduce data communication, we prefer to keep the upper level subtrees in each processor consisting of data from as many levels as possible.

4.2. Data replication

For a specific scene, a fixed amount of memory is needed. Even distribution of data can be impractical without considering the scalability of system memory size as the number of processors increases. In Eqs. (2) and (3), both data redistribution sizes increase as the larger system is used. Data replication can lower redistribution size to some extent. With R copies of the entire lower part subtrees, the data redistribution cost is:

$$(1 - H(h)) \times (D'/P - (R \times D')/P^2) \times U'_{\text{comm}} \tag{4}$$

Under this memory arrangement, the system forms R groups of processors where each group has P/R processors and owns a copy of the entire lower level subtrees. Non-local subtree access is limited to within the same group of processors to avoid global data communication. Localized communication within group, where can exploit nearest-neighbor and adjacent-neighbor communication to reduce the number of link contention, provides lower communication cost U'_{comm}. The ratio of U'_{comm} over U_{comm} depends on the network load, message size, system size and group size.

4.3. Dynamic data redistribution

So far, we have considered the data redistribution cost for different schemes. However, we ignore the fact that part of D'/P can be reused during the ray tracing. For ray tracing neighboring pixels, the data to be tested against is similar due to image coherence. Therefore, it is necessary to keep these data temporarily stored in the local memory to further reduce data communication. A cache scheme is an ideal mechanism for this purpose.

To quantify the image coherence in ray tracing is an outstanding problem in computer graphics due to unpredictability of special effects such as shadows, reflections and refractions. Therefore, it is also very difficult to determine the best cache size for the image coherence. However, at least, we must maintain enough cache space to hold the most important subtrees in the cache all the time. This size is proportional to the size of the subtree in the lower part, and that is to D'/B^{h-1}, where B is the number of branches if the tree is cut at depth h. For simplicity, we quantify the cache effect on D'/P by $C(s, cs)$, where s is the rendered scene and cs is the cache size. Thus, the redistribution cost becomes:

$$(1 - H(h)) \times ((C(s, cs) \times D')/P - (R \times D')/P^2) \times U'_{\text{comm}} \tag{5}$$

In the following section, we will experimentally evaluate the impact of cache size for

different scenes. Eqs. (2)–(5) represent data redistribution cost. To achieve data redistribution efficiently, data redistribution block size must be determined carefully. It is not a good idea to transfer an entire subtree at one time. This can exhaust the cache memory very quickly and may result in unnecessary cache block swapping.

In Kay's ray traversal algorithm [3], if a bounding volume is hit, the ray will test against all of its children (8 children in our implementation). In a message-passing system such as the Intel Delta, which we are working on, 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. The question is, what is the optimal message size? To answer this question, 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 experimentally measured this round trip time by the following steps:

Measure of round trip time

P_i: starts the clock, send a request to P_j $(i \neq j)$, which j is randomly chosen.

P_j: as the request message arrives, a message handler routine will send a reply message back.

P_i: stops the clock after receiving the reply.

We tested different size of reply message varying from 1 K to 512 K bytes and averaged over 10000 runs. The round trip time cost (in microseconds) can be modeled by:

$$\text{Cost}(l) = 286.4 + 0.133 \times l \tag{6}$$

where 286.4 microseconds is setup time for this round trip and its cost is a factor 0.133 proportional to message size l. Each non-leaf bounding volume needs 100 bytes of storage in our implementation. Hence, we let:

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

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

$$T(2) \leq T(1) + k \times T(1) \tag{7}$$

We find $k \geq 2.163$, which means that if more than 2 of 8 bounding volumes on 1 level are hit, we should use $S(2)$. The probability that Eq. (7) is true is high (2 of 8), so, we adopt $S(2)$. For transferring subtree $S(n \geq 3)$ as a message, the storage requirements increase exponentially and it is hard to predict their future usage. Thus, we consider $S(2)$ 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.

4.4. Implementation

In our implementation, the memory in each processor is divided into two parts: one part is called permanent memory (PM) which is used to store both the entire upper level tree and part of the lower level subtrees, while the other part is used as cache memory called (CM). To achieve initial data distribution, three steps are used as described below:

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 coordinate axis, say the X-coordinate, 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 coordinate. This is repeated until we have 2^d sets and each set has N objects ($N \leq N_{\text{limit}}$). N_{limit} is the maximum number of objects that can be stored in the PM 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.

To achieve the first phase, a Sun host is used to partition a given database and to distribute multiple copies of the entire database among groups of processors. Let P_{ij} denote the jth processor of ith group, where $0 \leq i \leq 2^{D-d} - 1$ and $0 \leq j \leq 2^d - 1$. By using the median-cut method, the database is partitioned into 2^d sets denoted by X_0, X_1, ..., X_{2^d-1}. The number of objects in each partition X_j is almost the same.

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

Initial data distribution
{
for $j = 1$ to $2^d - 1$ **do**
P_{00} receives X_j from the host;
P_{00} sends X_j to P_{0j};
P_{0j} multi-casts to P_{ij} for all i;
\ * sends to corresponding processor in other groups. * \
end;
P_{00} receives X_0 from the host;
P_{00} multi-casts X_0 to P_{i0} for all i;
}

After receiving each block, P_{00} sends it to P_{0j} via a 'forced' type message [1] while the multi-casting from P_{0j} to other groups are done through an 'un-forced' type message to keep P_{00} free to receive data from the host. This way we can pipeline the data movement 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 construc-

[1] In the Delta, there are two types of communication protocols available, namely 'forced' and 'unforced' messages. The former provides the low transfer cost, but also has high startup. The latter performs in the reverse manner.

tion. 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 available PM. After this initial construction, those branching bounding volumes on i level are in turn multi-casted 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.

5. Dynamic data management

Generally, direct and set-associative mapping are used in cache memory management. In either case, the size of each slot or block is fixed. Using a fixed block will waste memory space in our implementation, since the size of each $S(2)$ can be different. One major disadvantage of direct mapping is that replacement will occur even when the cache is not full. This can be serious, particularly when many frequently used data are mapped onto the same slot. In the case of set-associative mapping, we need to pay more overhead cost to place data into cache.

In our design, we implement a software cache memory which is a portion of the local memory. This portion of memory forms an available cache memory pool and each block can hold a variable size of data. When a cache block is needed, we first check if there is enough cache memory left to store this block. If there is space, we assign the block to this $S(2)$ and maintain it by a double-linked cache index list (see Fig. 2). Otherwise, a suitable block will be searched through this cache index list using a replacement policy.

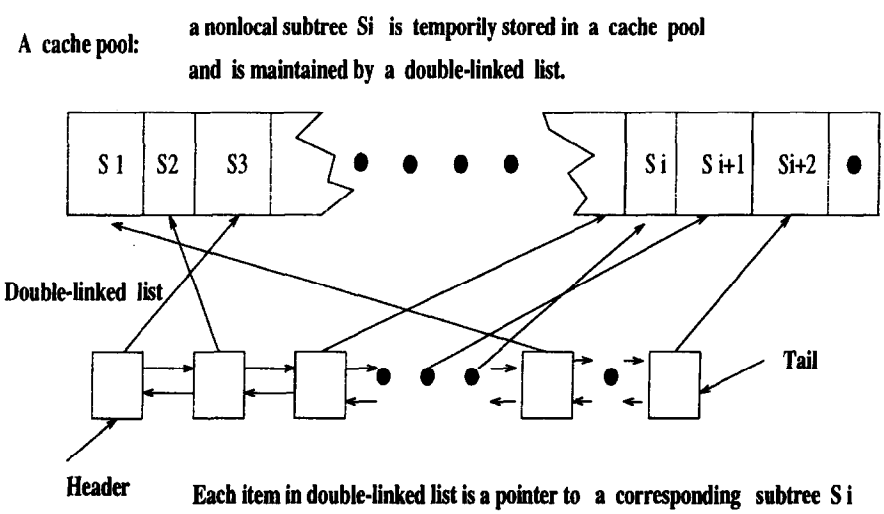

Fig. 2. Dynamic data management for nonlocal subtrees.

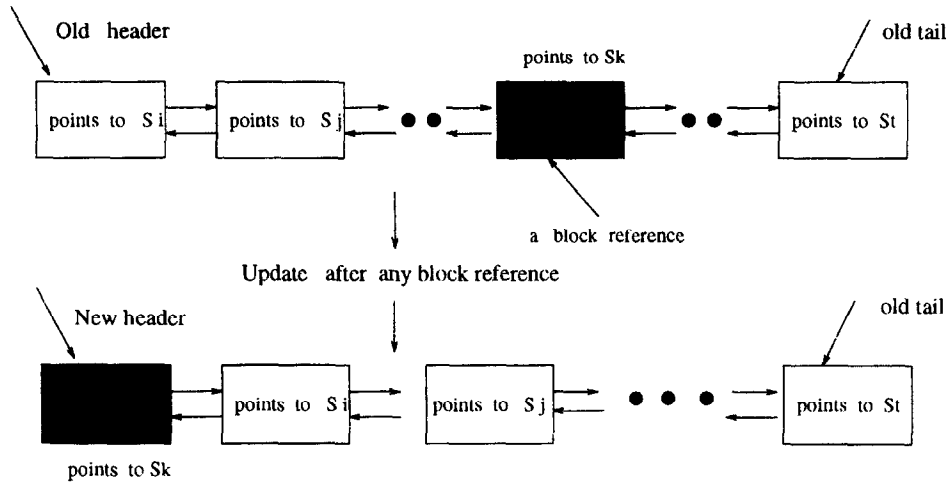

Fig. 3. ALRU policy.

A replacement policy is used to determine which cache block will be swapped out when the cache becomes full. As discussed previously, we need to keep the most frequently used data or most recently used data in the cache to exploit the image coherence inherent in ray tracing. In previous work [14,15], a least recently used (LRU) strategy was used for this purpose. To implement this policy, somewhat sorting is necessary to maintain the order of data reference in cache in case of data miss, $O(N \log N)$ cost is required, where N is the number of items in the cache. For large values of N, this cost can be significant. Thus, we consider a different strategy for our replacement policy described as follows:

· ALRU (another least recently used): The double-linked list will be updated after each reference. When a block is referenced, it will be placed at the head of the list. Thus, the newly allocated block is placed at the head of the list (see Fig. 3). For swapping, choose a cache block starting from the tail of the list In the ALRU, those $S(2)$ blocks in the cache list are always in most to least-recently referenced order without the exact sorting overhead incurred in LRU. The ALRU is designed to capture the most recently used $S(2)$, which is helpful for ray tracing the adjacent pixels, and keep the most referenced $S(2)$ in the cache, which is helpful as some $S(2)$ dominates the data access. However, ALRU needs to adjust the pointers every time a reference is made on each $S(2)$ block.

6. 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 sub-images of comparable size. Each sub-image 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 sub-image, pixels are continuous or regions are close in 2D. Therefore, better image coherence is maintained and the possibility of nonlocal data movements 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. After the database and initial load distribution, each processor begins to ray trace its allocated sub-image. At run time, our GDC load balancing scheme is used to achieve global balanced load. The GDC scheme is described as follows:

We logically organize the N processors of the Intel Delta in a ring (snake-like mapping) topology. Two neighboring processors will be assigned two neighboring subimages due to image coherence also. 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.

Using our scheme, there are two ways of maintaining the image coherence. In each subimage (region), pixels are continuous and close, so the rendering of these pixels tends to access the same data. For the same purpose, two neighboring subimages are assigned to two neighboring processors in the ring and each processor searches extra region starting from its neighboring node. In our GDC, when a lightly loaded processor P_i becomes idle again, it will first check processor P_j in which P_i's previous request was made. Rendering two square pixel-region (from two consecutive load requests) $(a \times a)$ from the same subimage potentially needs the same data. Therefore, remote data fetches can be avoided or be reduced in case that multiple $a \times a$ regions are from the same processor (the same subimage).

In this paper, the region size (parameter a) is used to control the granularity of load distribution in the run time. This size depends highly on the scene rendered and other factors, and it is hard to find the optimal value for all images. From our experience, the optimal size ranges from 4×4 to 8×8 for our test scenes.

6.1. Finding a good region size

For simplicity, we assume that the communication cost incurred in each processor is proportional to the number of square regions obtained from other processors. This assumption is not quite accurate; however, it is reasonable. If R is the resolution of the image (512×512 in our case), A is the size of extra square region $(a \times a)$ and N is the

number of processors in use. Using our load balancing method, the average number of extra regions is bounded by $R/(A \times N)$, since the total number of an extra regions in the system is less than R/A. Therefore, the communication cost, Cost(A), is $(k_0 \times R)/N + (k_1 \times R)/(A \times N)$, where k_0 is the average communication cost incurred in a single pixel and k_1 is the average communication cost incurred in searching for an A. As for the imbalances among processors, Im(A), it is bounded by T_{A_i}, where T_{A_i} is the execution time needed to process A_i. The imbalance on the average is bounded by $k_2 \times A$, where k_2 is the average pixel execution time. Now, we define a function, $F(A) = \text{Im}(A) + \text{Cost}(A)$. In this function, if A is larger, Im(A) among the processors becomes bigger and the communication cost becomes smaller, and vice versa. To tradeoff the communication cost and imbalances, we need to optimize $F(A)$ (i.e., try to find minimum of $((k_0 \times R)/N + (k_1 \times R)/(A \times N) + k_2 \times A)$. We find that $F(A)$ is minimum when A is equal to $((R \times k_1)/(N \times k_2))^{0.5}$. Therefore, $\lceil (A^{0.5}) \rceil$ is our suggested size for a. Using our heuristic to find the suggested region sizes $(a \times a)$ for our test scenes, they are 3×3 for the 'Mountain' scene, 4×4 for the 'Tree' scene and 7×7 for the 'Balls' scene under our default configuration.

The values of both k_1 and k_2 depend highly on the scene rendered, machine architecture and so on. Since a low resolution image can be seen as the miniature of its high resolution image, we decide to find the estimated values for both k_1 and k_2 by ray tracing low resolution image (32×32). Ray tracing a low resolution image, we approximate k_1 by the average communication cost incurred in searching an extra pixel and k_2 by the average pixel execution time. The overhead of this preprocessing cost is not significant. More accurate estimations can be done when animation sequences of ray traced images are required. We can use the information of the previous frame to estimate both k_1 and k_2 for the current frame.

To raytrace an image, the number of times a load request is made by each processor is between 1 (the best case) and $((N-1) \times R)/(A \times N)$ (the worst case). The best condition occurs when the total execution time of all sub-images are roughly the same. The worst case occurs when all processors except one are overloaded by their first square regions. Both cases are rare in the load distribution of real data.

7. Experimental results and discussions

We use a set of standard scenes from Eric Haines's SPD database [18] to perform our experimental evaluation. The SPD database has been used for many previous work [7,13–15]. We tessellate three test scenes from the SPD database into higher resolution to create larger database. Table 1 shows the geometric characteristics of three scenes. The rendered images of three test scenes are shown in Figs. 4–6.

Table 1
Database characteristics

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

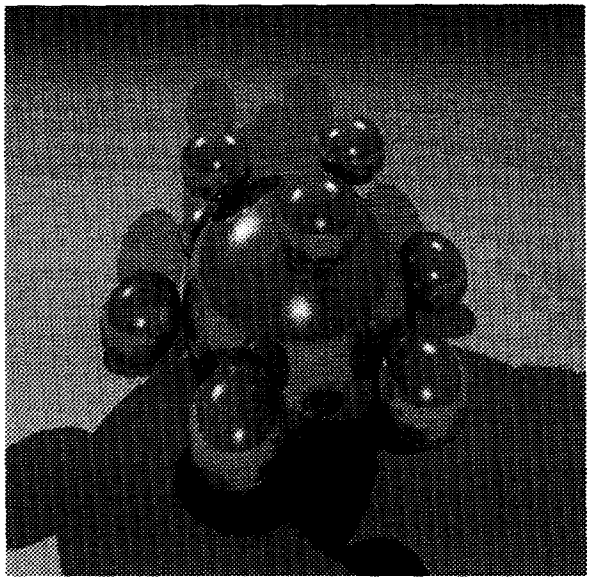

Fig. 4. The 'Balls' scene.

To evaluate our scheme, we make the assumption that the maximum available memory size per processor 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

Fig. 5. The 'Tree' scene.

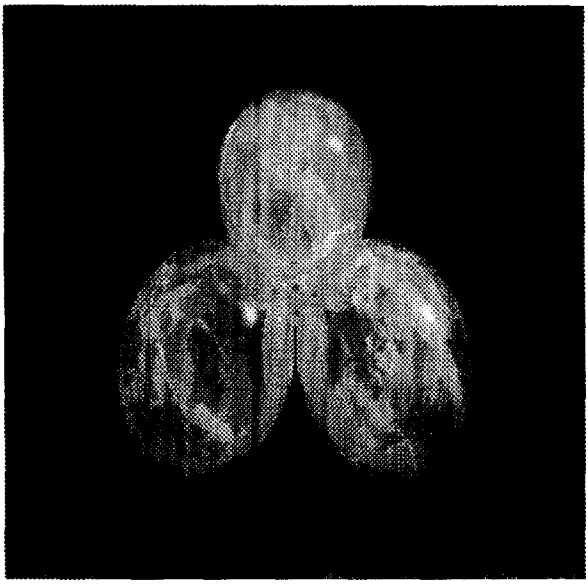

Fig. 6. The 'Mountain' scene.

with distributing the database. In this 4 MB, PM size is 2.5 MB and CM size is 1.5 MB. This is our default configuration. In this configuration, the maximum number of primitives is approximately 6000 triangles and the number of bounding volumes on a cutting level is less than 4096 ($8^4 = 4096$). Table 2 shows how we store these 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 scene is the largest and the available cache memory becomes comparatively small. The comparatively small cache can potentially lead to more data communication overhead in the processing of rays.

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 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 512×512 with no anti-aliasing. Unless it is stated otherwise, the

Table 2
Data distribution for three test scenes

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

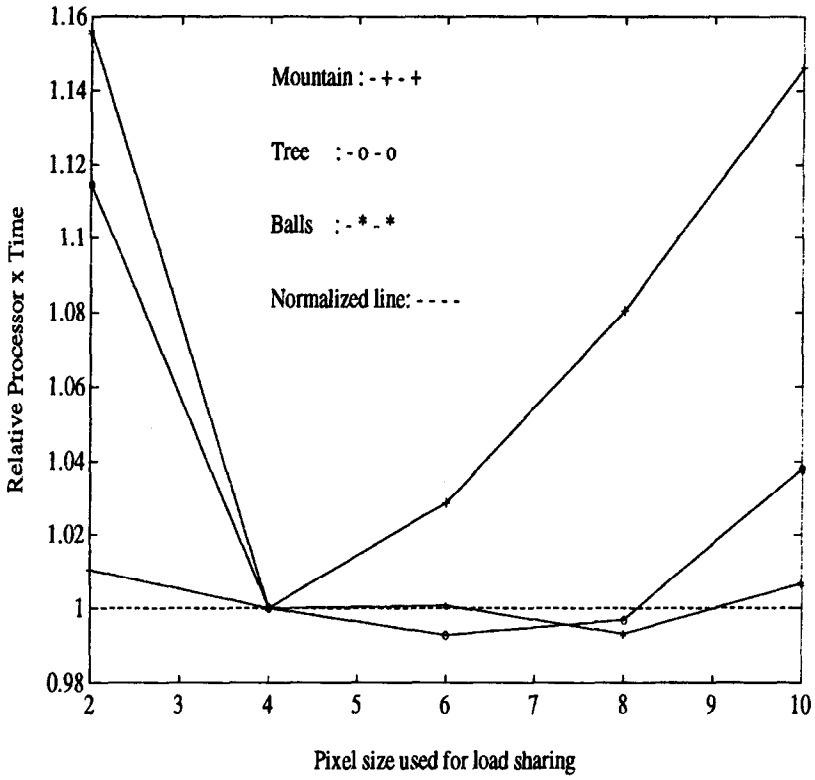

Fig. 7. Relative time for 3 scenes for different square region sizes (pixel size × pixel size).

following values are used in the experimentation: 2.5 MB for PM, 1.5 MB for CM, ALRU for cache replacement policy and 4×4 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 is shown in Fig. 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 $\sum_{i=4 \text{ or } 5}^{9} 2^i T_{2^i}$, 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 Fig. 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. Specific scenes may require different region size in load sharing. For example, the Mountain scene requires more rendering computation but it only needs a small memory size to contain the whole database. Therefore, it tends to use smaller region size (4×4) in load sharing.

In the next three figures (Figs. 8–10), we show plots of the number of processors versus speedup relative to 32 processors for the Balls scene, 16 processors to the Tree

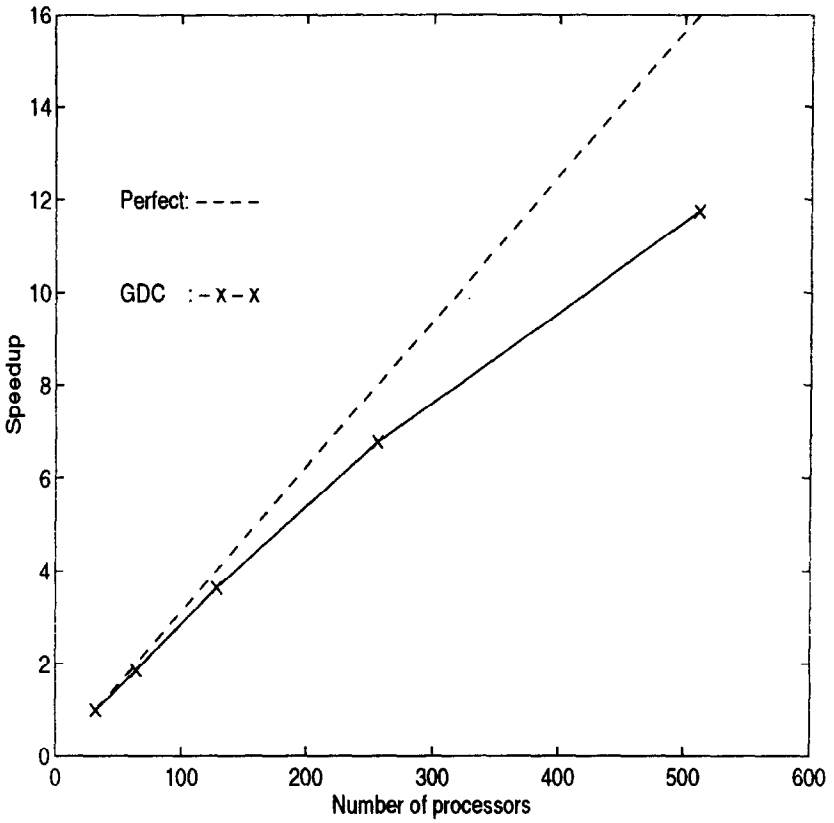

Fig. 8. Relative speedup for 'Balls' scene.

scene and 8 processors to the Mountain scene. These relative speedup curves are based on the timing derived on the minimum configuration that test scenes can fit in (i.e., assuming that PM is 2.5 MB and CM is 1.5 MB per processor). For these plots, the 'perfect' means perfect speedup relative to the minimum processor configuration for the test scene. Our load balancing scheme gives almost linear relative speedup for all three scenes. As expected from our previous discussion, the 'Mountain' scene performs the best and the 'Balls' scene performs the worst. The 'Balls' scene requires larger memory and generates more secondary rays; it leads to more data communication overhead.

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 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 fixed 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 insignifi-

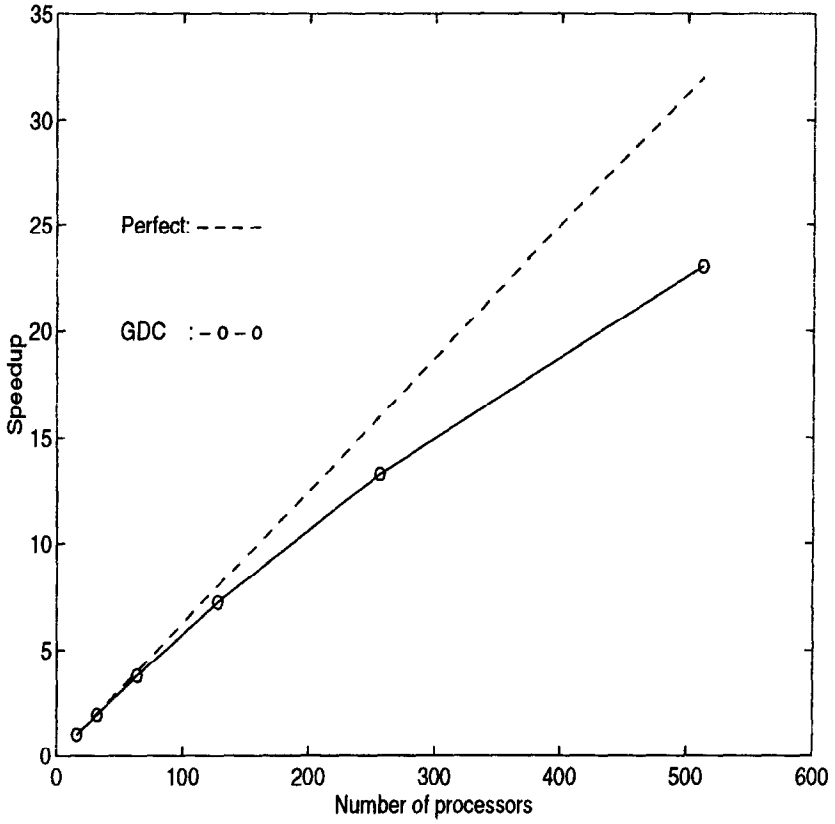

Fig. 9. Relative speedup for 'Tree' scene.

cant. In Fig. 11, we plot cache size versus relative efficiency, where relative efficiency is defined as the ratio of rendering time with a given percentage of cache to that of rendering time with 10% cache. As evidenced by Fig. 11, for the Mountain and Tree scenes there is very little improvement in performance beyond 3% cache size. There is enough cache space to capture the most recently used subtrees beyond 3% cache size. However, below 3% the performance for both scenes are significantly effected by the cache thrashing. On the other hand, the 'Balls' scene needs at least 10% to allow performance to scale well. Between 10% and 2.5%, its performance is very sensitive to cache size (i.e. data movement is increasing). Below 2.5%, cache thrashing happens and results in poor performance.

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 will spread the database among more

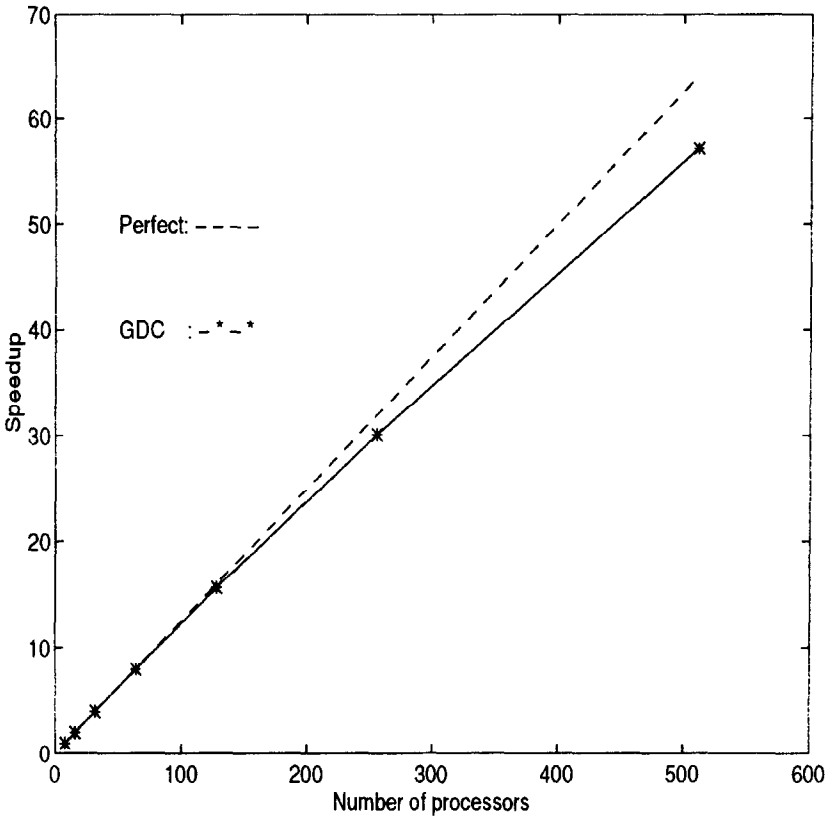

Fig. 10. Relative speedup for 'Mountain' scene.

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 (CM size is 0.5 MB). With this new arrangement, the group size of the Mountain scene will be 4 instead of the default size 8, the group size for the Tree scene will be 8 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. Fig. 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 Fig. 11, 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 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.

8. Comparison with other related work

As indicated in early section, there have been many work proposed to parallelize ray tracing. Though it is very difficult to compare different algorithms running on different architectures, here, we would like to differentiate our proposed scheme from other most related work with emphasis on the characteristics of our implementation. The main characteristics of our implementations are listed as follows:

- To the best of our knowledge, most previous work used small number of processors (less than 128 nodes) and obtained either reasonable or poor results. In contrast, our scheme is very scalable ranging from small (less than 8 nodes) to large scale system (512 nodes).
- Our scheme allows multiple copies of the entire database to exist in the system. The potential of communication bottlenecks for target objects or pages can be reduced compared with [7,14,15].
- Our scheme belongs to image space scheme which schedules subimages (coarse-grained parallelization) either statically or dynamically among available processors [7,14,15]. On the other hand, other previous work [8–13] exploit object space methods which schedule ray computations (fine-grained parallelization) among available processors.
- In the proposed load balancing scheme (GDC), the task of scheduling is distributed among all processors, and thus the drawback of central scheduling is eliminated

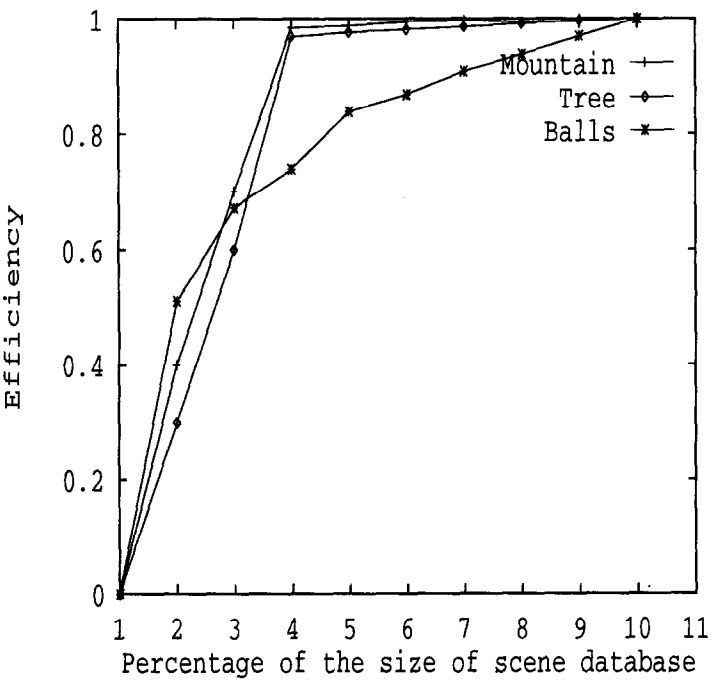

Fig. 11. Variable cache sizes for three scenes using 64 processors.

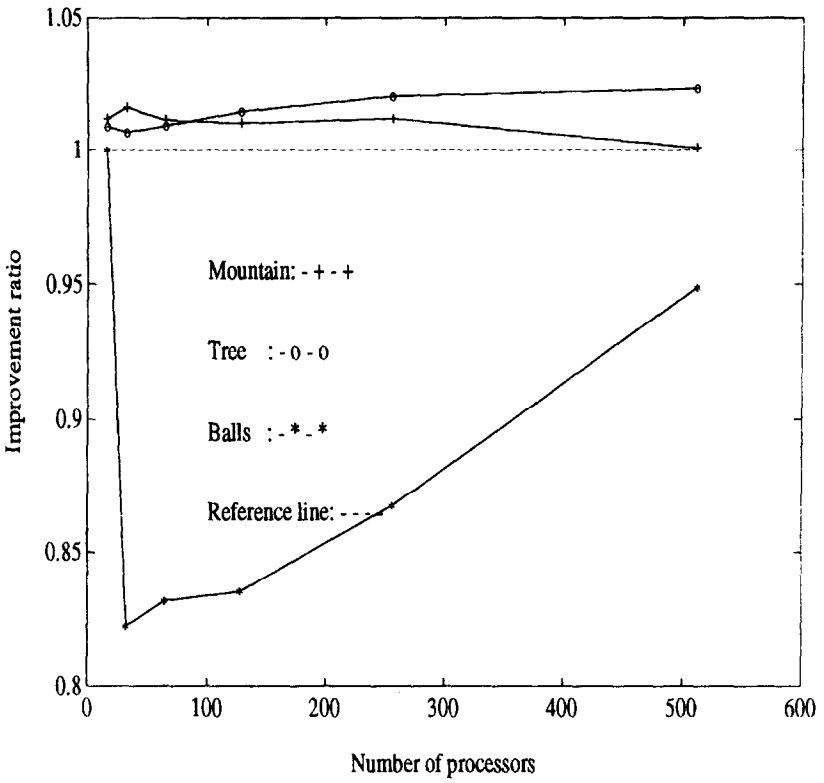

Fig. 12. Relative improvement for three scenes using smaller group size.

[7,13,14]. In addition, the GDC scheme can dynamically adjust workload among the processors in a global manner. In contrast, [9–11] used static methods to schedule ray tracing computation, and hence, their schemes potentially lead to poor performance.

- The redistribution cost grows as the data size increases. To reduce the data size requirement, we adopt the hierarchical bounding volume tree algorithm (generally needs $O(n \log n)$ memory requirement, where n is the number of objects) instead of SEADS structure (generally needs $O(n^3)$ memory requirement, where n is the resolution of uniform space partition) [13,15].
- We manage data objects in a dynamically manner to take image coherence rather than in a static fashion [9–13]. The unit of data movement is a subtree whose size is optimally determined rather than a group or page of objects [7,14,15].

9. Conclusions

In this paper, we present many techniques to reduce the data redistribution cost and experimentally evaluated these techniques on the Intel Delta. Several important paramc-

ters affect the performance of parallel ray tracing when the database is partitioned among the processors. 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 image coherence in pixel computations, and thus improve the performance. Experiments were conducted extensively to investigate the scalability of various parameters on the performance of the parallel ray tracer. Our conclusions are that there is a critical cache size which depends on the scene being rendered, and there are an optimal group size and region size in load sharing for a given scene. In our earlier work [4], we developed the GDC load balancing policy for ray tracing with each node having a copy of database. 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 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 and in part by the National Science Council of Taiwan, ROC, project No. NSC-86-2213-E-218-011.
We would like to give special thanks to Prof. C.S. Raghavendra's careful proofreading of this paper.

References

[1] T. Whitted, An improved illumination model for shaded display, Commun. ACM 23 (6) (1980) 343–349.
[2] A.S. Glassner (Ed.), An Introduction to Ray tracing, Academic Press, 1989.
[3] T.L. Kay, J.T. Kajiya, Ray tracing complex scenes, ACM SIGGRAPH 20 (4) (1986) 269–278.
[4] T.-Y. Lee, C.S. Raghavendra, John B. Nicholas, Experimental Evaluation of Load Balancing Strategies for Ray Tracing on Parallel Processors, in the Proceedings of 1994 International Conference on Parallel Processing, 1994.
[5] E.G. Coffman (Ed.), Computer and Job-shop Scheduling Theory, John Wiley and Sons, New York, 1976.
[6] R.L. Graham, Bounds on Multiprocessor Scheduling Anomalies and Related Packing Algorithms, Proceedings of Spring Joint Computer Conference, 1972, pp. 205–217.
[7] S. Green, Parallel Processing for Computer Graphics, MIT Press, 1991.
[8] M. Dippe, J. Swensen, An adaptive subdivision algorithm and parallel architecture for realistic image synthesis, ACM Comput. Graphics 18 (3) (1984) 149–158.
[9] D.A.J. Jevans, Optimistic Multi-processor Ray Tracing, In: R.A. Earnshaw, B. Wyvill (Eds.), New Advances in Computer Graphics, Springer Verlag, 1989, pp. 507–522.
[10] K. Bouatouch, T. Priol, Parallel Space Tracing: An Experience on an iPSC Hypercube, In: N. Magnenat-Thalmann, D. Thalmann (Eds.), New Trends in Computer Graphics, 1988, pp. 170–187.
[11] J. Salmon, J. Goldsmith, A Hypercube Ray-tracer, Proceedings of the 3rd Conference Hypercube Computers and Applications, vol. 7(5), 1988, pp. 14–20.
[12] E. Caspary, I.D. Scherson, A Self-balanced Parallel Ray Tracing algorithm, in the Parallel Processing for Computer Vision and Display, P.M. Dew, T.R. Heywood, R.A. Earnshaw (Eds.), Addision-Wesley, 1989, pp. 408–419.

[13] W. Lefer, An Efficient Parallel Ray Tracing Scheme for Distributed Memory Parallel Computers, Proceedings of 1993 Parallel Rendering Symposium, 1993, pp. 77–80.
[14] S.A. Green, D.J. Paddon, Exploiting coherence for multiprocessor ray tracing, IEEE Comput. Graphics Appl. 9 (6) (1989) 12–26.
[15] D. Badouel, T. Priol, Distributed data and control for ray tracing in parallel, IEEE Comput. Graphics Appl., July (1994) 69–77.
[16] K. Li, Shared Virtual Memory on Loosely Couple Multiprocessor, Ph.D dissertation, Yale Univ., 1986.
[17] U. Neumann, Communication costs for parallel volume-rendering algorithms, IEEE Comput. Graphics Appl., July (1994) 49–58.
[18] E. Haines, A proposal for standard graphics environments, IEEE Comput. Graphics Appl. 7 (11) (1987) 3–5.