An Efficient Parallel Spatial Subdivision Algorithm for Object-Based Parallel Ray Tracing [*]

Cevdet Aykanat, Veysi İşler, Bülent Özgüç
Department of Computer Engineering and Information Science
Bilkent University
06533 Ankara, Turkey

Abstract

Parallel ray tracing of complex scenes on multicomputers requires distribution of both computations and scene data to the processors. This is carried out during preprocessing and usually consumes too much time and memory. In this paper, we present an efficient parallel subdivision algorithm to decompose a given scene into rectangular regions adaptively and map the resultant regions to the node processors of a multicomputer. The proposed algorithm uses efficient data structures to find out the splitting planes quickly. Furthermore the mapping of the regions and the objects to the node processors is being performed while parallel spatial subdivision proceeds. The proposed algorithm is implemented on an Intel's iPSC/2 hypercube multicomputer and promising results are obtained.

Keywords : Parallel ray tracing, spatial subdivision, multicomputers, hypercube interconnection topology.

Introduction

In recent years, research on ray tracing has been mostly concentrated on speeding up the algorithm by parallelization [1]. There are mainly two approaches to parallelize ray tracing. One of them is image-space subdivision in which the computations related to different rays are distributed to the processors. The other approach is object-space subdivision which should be adopted for parallelization of ray tracing on distributed-memory message-passing architectures (multicomputers). Multicomputers are very promising architectures for massive parallelism due to their nice scalability features. In a multicomputer, there is no global memory, and synchronization and coordination between processors are achieved through message exchange. For an efficient parallelization on a multicomputer (called object-based parallel ray tracing), the object space data (scene description with the auxiliary data structure) as well as computations should be distributed among processors of the multicomputer, since the whole object space data may not fit into the local memory of each processor for complex scenes.

The approach taken in this paper is to subdivide the 3-D space containing the scene into disjoint rectangular subvolumes and assign both computations and the object data within a subvolume to a single processor. The proposed subdivision algorithm recursively bipartitions the rectangular subregions into two rectangular subsubregions starting from a given initial window until P rectangular subregions are obtained where P denotes the total number of processors in the multicomputer. The subdivision and mapping should be performed in such a way that each processor is assigned equal amount of computational load. Furthermore, the neighboring objects should be maintained in the local memories of adjacent node processors to achieve better *data coherence* [1]. The proposed subdivision algorithm also achieves the mapping of the rectangular subvolumes to processors during the decomposition process. The subdivision algorithm has efficient data structures to locate the splitting planes.

The spatial subdivision problem is a preprocessing overhead introduced for the efficient implementation of the object-based parallel ray tracing on the target multicomputer. If the spatial subdivision algorithm is implemented sequentially, this preprocessing can be considered in the serial portion of the parallel ray tracing which limits the maximum parallel efficiency. For a fixed input scene instance, the execution times of the parallel ray tracing and the sequential subdivision programs are expected to decrease and increase, respectively, with increasing number of processors in the target multicomputer. Thus, this preprocessing will begin to constitute a drastic limit on the maximum efficiency of the overall parallelization due to Amdahl's law. Hence, parallelization of the subdivision algorithm on the target multicomputer is a crucial issue for efficient object-based parallel ray tracing. In this work, we propose an efficient parallel spatial subdivision algorithm to utilize the processors of the target multicomputer to be used for object-based parallel ray tracing algorithm. After an initial random distribution of objects to processors, objects intermittently migrate during the execution of the recursive bisection algorithm in accordance with the mapping strategy such that all objects arrive at their *home* processors at the end of the parallel subdivision process. Each object traverses at most $\log_2 P$ processors to reach its home processor.

[*]This work is partially supported by Intel Supercomputer Systems Division grant no. SSD100791-2 and Turkish Scientific and Technical Research Council (TÜBİTAK) grant no. EEEAG-5

Object-Space Decomposition

The decomposition of object space data can be performed by utilizing the techniques that are developed to improve the naive ray tracing algorithm. These techniques are *hierarchy of bounding volumes* [2] and *spatial subdivision* [3, 4] and can be adapted to parallel ray tracing as follows. The first technique forms a hierarchy of clusters consisting of neighboring objects. In the parallel processing case there might be two approaches, namely *static* and *demand-driven*, to accomplish a fair distribution of computations and storage. The former approach performs a static allocation by partitioning the entire hierarchy into a set of clusters each of which is assigned to a node processor. This resembles a graph partitioning process [5]. The latter approach allocates object space data and relevant computations to the node processors on demand. The second technique called spatial subdivision decomposes the 3-D space containing the scene into disjoint rectangular prisms. As in the first technique, the resulting prisms are distributed to the node processors either statically or on demand [1, 5, 6, 7].

In this paper, the second technique, spatial subdivision, is used to decompose the object space data. Spatial subdivision can be performed in several manners that give rise to different rectangular volumes. Regular Subdivision [8], Octree [3] and Binary Space Partitioning (BSP) [4] are widely used spatial subdivision schemes.

Utilizing BSP in Parallel Ray Tracing

Although both Octree and regular subdivision schemes have very nice properties when used in conventional ray tracing algorithm, it is difficult to achieve computational load balance among processors, if some coherence properties such as object, data, and image coherence are to be utilized. A manifestation of coherence called *data coherence* first exploited by Green and Paddon [1] is a very powerful and useful property that might reduce the communication overhead. Communication among the node processors is one of the most time consuming operations in an object-based parallel ray tracing system. Therefore, exploiting data coherence is essential in speeding up object-based parallel ray tracing. In order to exploit data coherence, we propose a variant of BSP - we call it BBSP (Balanced Binary Space Partitioning) since a complete binary tree is generated at the end of the subdivision. The subdivision is carried out on a window defined over a viewing plane onto which the objects in the scene are projected (parallel) (see Figure 1). The subdivision resultantly produces a set of rectangular regions on the window and a set of 3-D volumes obtained by extending the rectangular regions in the viewing direction. By means of this subdivision preprocess, the decomposition of both object space data describing the scene and the image-space computations associated with the pixels on the window are performed. It is assumed that the viewing volume and the produced 3-D volumes have rectangular (parallepiped) shape rather than pyramid shape.

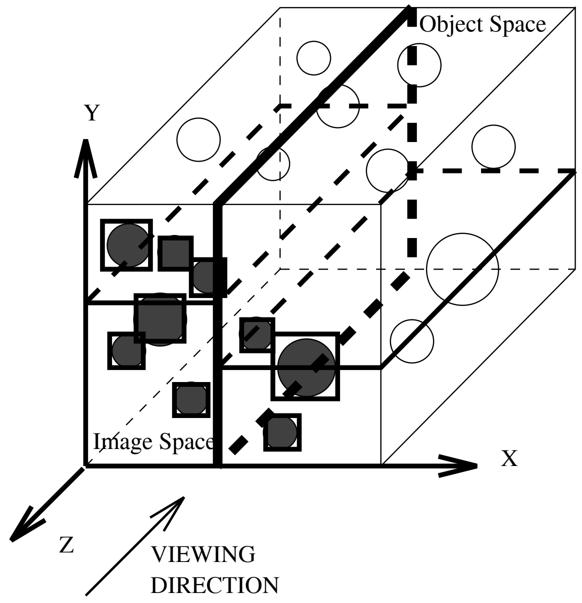


Figure 1: A 4-way subdivision of a scene using BBSP.

The proposed BBSP algorithm starts by projecting all objects in the view volume onto a given window W. The window W is the initial rectangular region for the recursive subdivision process. In the following steps of the algorithm, each generated rectangular subregion is subdivided into two subsubregions by a splitting plane which is parallel to either x-z (horizontal) or y-z (vertical) plane as shown in Figure 1. This recursive subdivision process proceeds in a *breadth-first* manner until the number of generated subregions (at the leaves of the recursion tree) becomes equal to the number of processors. Here, the number of processors is assumed to be a power of two.

The proposed algorithm decomposes both the image space and the object space, and meanwhile maps the resulting image-space subregions and the respective object-space subvolumes to the processors in one phase. Each 3-D subvolume is labelled using the label of the respective 2-D subregion from which the 3-D volume is obtained. Each 3-D volume and the corresponding 2-D region are then assigned to the node processor that has the node number equal to the label of the volume. Finding out the position of the splitting plane (i.e., subdivision) and labelling of the generated regions (i.e., mapping) are key operations in the algorithm.

Finding Out the Optimal Splitting Planes

The subdivision is practically carried out on the screen since the window is mapped to the viewport that is defined on a display device (screen). A splitting plane thus divides a given rectangular region of the screen into two disjoint rectangular subregions consisting of pixels. A rectangular region on the screen can be subdivided into two using either a horizontal splitting plane or a vertical splitting plane. Either a vertical or a horizontal splitting plane with minimum *cost* is chosen among all possible vertical and horizontal splitting planes based on an objective (cost) function. Hence, a splitting plane is characterized by its cost, direction (vertical or horizontal) and its location where the screen is cut.

In BSP trees, the location of the splitting plane is usually chosen along either object median or spatial median. MacDonald and Booth [9] have examined two heuristics for space subdivision using BSP. They pointed out that the probability of intersection of a given ray with an object is proportional to the surface area of the object - called the *surface area heuristic*. Using this heuristic, they have also found out that the optimal splitting plane lies between the object median and the spatial median. This result reduces the required search range to find out the location of the splitting plane. However, it is still an expensive operation to carry out search within the reduced search range. Furthermore, the analysis in [9] neglects the existence of shared objects between the generated subregions.

In this work, we propose an efficient search algorithm for finding optimal splitting planes during recursive space subdivision. The proposed search algorithm uses efficient data structures and requires only integer arithmetic. In the proposed algorithm, the position of an optimal splitting plane is determined by using an objective function that considers both the minimization of the computational load-imbalance and the number of shared objects between the generated subregions. The proposed objective function exploits the surface area heuristic for maintaining the computational load balance between the generated subregions.

Objective Function

The cost of a vertical splitting plane b on a window W consisting of $n \times m$ pixels (resolution) is defined as

$$C_v(b) = \frac{|n \times b \times L_b - n \times (m-b) \times R_b|}{n \times m \times N} + \frac{S_b}{N} \quad (1)$$

for $b = 0, 1, \ldots, m$, where N denotes the total number of objects projected onto the window W under consideration. The objective function for a horizontal splitting plane can easily be obtained by exchanging n with m in Equation 1. Here, L_b and R_b denote the number of objects in the left (below) and right (above) of the vertical (horizontal) splitting plane b, respectively. Furthermore, S_b denotes the number of shared objects straddling across the splitting plane b.

The denominator of the first term in Equation 1 denotes the total computational load associated with the window when only primary rays are considered. Hence, the first term in Equation 1 represents percent load imbalance between the two subregions generated by a particular splitting plane. Similarly, the second term in Equation 1 denotes percent number of shared objects between those two subregions. The shared objects cause several problems. First, the shared objects are duplicated in the local memories of the processors to which these objects are assigned. Second, an intersection test with a shared object might be repeated if the first intersection point is not inside the subvolume that is assigned to the processor performing the test. As in conventional ray tracing, there might be another closer intersection point within the next subvolume along the path of the ray.

The objective function in Equation 1 is computed for all splitting planes in both vertical and horizontal directions. The splitting plane with the smallest cost is chosen as the optimal splitting plane. Hence, the objective function should be efficiently computed. The objective function for vertical splitting planes can be simplified as:

$$C_v(b) = \frac{1}{m \times N}\{|b \times L_b - (m-b) \times R_b| + m \times S_b\} \quad (2)$$

for $b = 0, 1, \ldots, m$. The simplification for horizontal splitting planes can be obtained by replacing m with n in Equation 2. The parameter $1/N$ can be neglected since it is a constant factor common in all cost computations (both vertical and horizontal). Similarly, the parameters $1/m$ and $1/n$ appear as constant factors common in vertical and horizontal splitting plane computations, respectively. Hence, it is sufficient to compute the following functions.

$$C_v(b) = |\{b \times L_b + (m-b) \times R_b\}| + m \times S_b \quad (3)$$

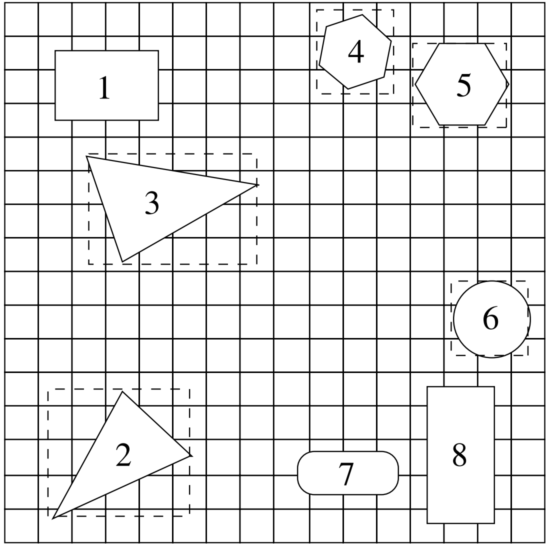
$$C_h(b) = |\{b \times L_b + (n-b) \times R_b)\}| + n \times S_b \quad (4)$$

for $b = 0, 1, \ldots, m$ and $b = 0, 1, \ldots, n$ in order to find the optimal vertical and horizontal splitting planes b_v^{min} and b_h^{min}, respectively. The optimal splitting plane is then chosen among these two splitting planes by comparing $C_v(b_v^{min})/m$ with $C_h(b_h^{min})/n$. This formulation enables the use of only integer arithmetic during the cost computations.

Data Structures

Horizontal and vertical splitting planes subdivide a given rectangular region in x and y dimensions, respectively. Two integer arrays are defined to hold the information related to the distribution of objects along each one of these two dimensions. To form the data structures, the objects are projected onto the viewing plane and the projections of the objects are surrounded by bounding boxes to simplify the computations as seen in Figure 2. After this operation, each object o in the scene has four attributes: $xmin[o], xmax[o], ymin[o]$ and $ymax[o]$. Here, $xmin[o]$ ($ymin[o]$) and $xmax[o]$ ($ymax[o]$) denote the left (bottom) and the right (top) borders of the bounding box of an object o, respectively. Assuming that the window W consists of $n \times m$ pixels (resolution), the arrays for x and y dimensions have sizes of m and n, respectively. The following major data structures are constructed and used for the x dimension: *XMinCntr* and *XMaxCntr*, where *XMinCntr[b]* and *XMaxCntr[b]* contain the number of objects whose $xmin$ and $xmax$ values are equal to b, respectively, for $b = 1, 2, \ldots, m$. The *YMinCntr* and *YMaxCntr* are similar data structures constructed and used for the y dimension.

Having formed these data structures, prefix sum operation is performed on these integer arrays. These



XMinCntr

0	2	3	3	3	3	3	3	4	5	5	5	7	8	8	8

XMaxCntr

0	0	0	0	1	2	2	3	3	3	3	5	5	5	7	8

Figure 2: A sample scene projected onto the viewing plane

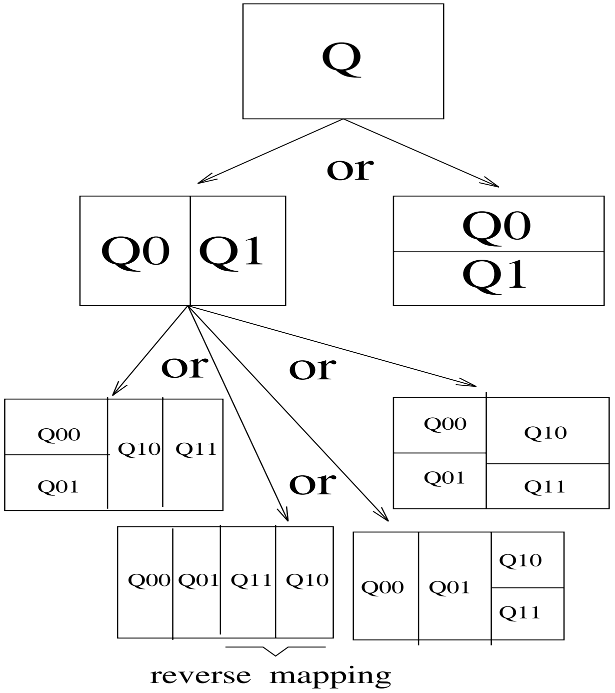


Figure 3: Subregion labeling for a sample case.

integer arrays are then used in the computation of the objective functions in Equation 3 and 4. These equations need the values of R_b, L_b and S_b for each possible splitting position b. After prefix sum operations, *XMinCntr[b]* and *XMaxCntr[b]* contain the number of objects whose $xmin$ and $xmax$ values are equal to or less than b, respectively. Hence, *XMinCntr[b] (YMinCntr[b])* denotes the number L_b of objects in the left (bottom) subregion of the vertical (horizontal) splitting plane. Similarly, *XMaxCntr[b] (YMaxCntr[b])* denotes the number of objects in the left (bottom) subregion which do not straddle across the vertical (horizontal) splitting plane b. Hence, S_b and R_b can easily be computed as

$$S_b = L_b - XMaxCntr[b] \quad (5)$$
$$R_b = (N + S_b) - L_b \quad (6)$$

for a vertical splitting plane b. For a horizontal splitting plane b, S_b and R_b can similarly be computed using these two equations by replacing *XMaxCntr* in Equation 5 with *YmaxCntr*. Note that the values of R_b, L_b and S_b are efficiently computed using only 3 integer additions which will be performed for all possible splitting planes.

Mapping

The proposed algorithm achieves the mapping of the generated subregions during the recursive subdivision process. Each generated subregion is assigned a label that corresponds to the processor-group to which it is assigned. Initially, the window W is assumed to be assigned to all processors in the parallel architecture. While splitting a region into two subregions, the processor-group assigned to that region is also split into two halves and these two halves are assigned those two subregions, respectively. This recursive spatial subdivision of the window proceeds together with the recursive subdivision of the processor interconnection topology. The recursive subdivision and assignment scheme to be adopted for the processor interconnection topology is a crucial factor in achieving the data coherence mentioned earlier.

In this work, we propose a recursive labeling scheme for the generated regions during the recursive subdivision of the window. This labelling scheme emulates the recursive definition of the hypercube interconnection topology as the target architecture for the object-based parallel ray tracing algorithm. However, the proposed labeling can easily be adopted to other parallel architectures implementing symmetric and recursive interconnection topologies (e.g., 2D Mesh and 3D Mesh) with minor modifications.

Here, we will briefly summarize the topological properties of hypercubes exploited in the proposed labeling. A multicomputer implementing the hypercube interconnection topology consists of $P = 2^d$ processors with each processor being directly connected to d other neighbor processors. In a d-dimensional hypercube, each processor can be labeled with a d-bit binary number such that the binary label of each processor differs from its neighbor in exactly one bit. A *channel* c defines the set of $P/2$ links connecting neighbor processors whose binary labels differ only in bit c, for $c = 0, 1, 2, \ldots, d-1$. In the recursive definition of the hypercube topology, a d-dimensional hypercube is constructed by connecting the processors of two

$(d-1)$-dimensional hypercube in a one-to-one manner. Hence, a d-dimensional hypercube can be subdivided into two disjoint $(d-1)$-dimensional hypercubes, called subcubes, by *tearing* the hypercube across a particular channel (e.g., $c=d-1$). Each one of these two $(d-1)$-dimensional subcubes can in turn be divided into two disjoint $(d-2)$-dimensional subcubes by tearing them across another channel (e.g., $c = d-2$). Hence, d such successive tearings along different channels (e.g., $c=d-1, d-2, \ldots, 1, 0$) result in 2^d 0-dimensional subcubes (i.e., processors). An h-dimensional subcube in a d-dimensional hypercube ($0 \leq h \leq d$) can be represented by a d-tuplet containing h *free-coordinates* (x's) and $d-h$ fixed-coordinates (0's and 1's) [10].

In the proposed mapping scheme, the label Q of the initial rectangular region (window W) is initialized to null. Consider the subdivision of a particular subregion labelled as Q by a vertical or horizontal splitting plane. Note that the label Q of this subregion is a q-bit binary number where q denotes the depth of this subregion in the subdivision recursion tree. Hence, subregion Q is already mapped to the $(d-q)$-dimensional subcube Qx...x. The left (below) and right (above) subsubregions generated by a vertical (horizontal) splitting plane are labelled as $Q0$ and $Q1$, respectively. This labelling corresponds to tearing the subcube Qx...x across channel $d-q-1$ and mapping the resulting (d–q–1)-dimensional subsubcubes Q0x...x and Q1x...x to left (below) and right (above) subsubregions, respectively (see Figure 4). However, if two subregions $Q0$ and $Q1$ generated from the same region by a vertical (horizontal) splitting plane are both splitted again by vertical (horizontal) planes, then the subsubcube-to-subsubregion assignment in one of these two subregions is performed in reverse order. The proposed labeling scheme tries to maximize the data coherence by mapping neighboring subregions to neighboring subcubes, as much as possible, during the recursive subdivision process. Figure 3 illustrates the possible labeling combinations in a particular subpath of the recursion tree.

Parallel Spatial Subdivision

For complex scenes, spatial subdivision using the proposed BBSP scheme still may take too much time. For that reason, we can use the node processors of the target multicomputer to speed up the subdivision process. Furthermore, these processors are already idle waiting for the start of the ray-tracing-loop. This approach increases the utilization of the parallel system. Reducing the spatial subdivision time is also studied by other researchers. McNeill et al. [11] have suggested an algorithm for dynamic building of the octree to reduce the data structure generation time. In this work, we propose a parallel subdivision algorithm - a parallel version of BBSP scheme for hypercube multicomputers. The proposed BBSP algorithm is based on divide-and-conquer paradigm. Hence, BBSP algorithm is very suitable for parallelization on hypercubes due to their recursive structures mentioned earlier. The proposed

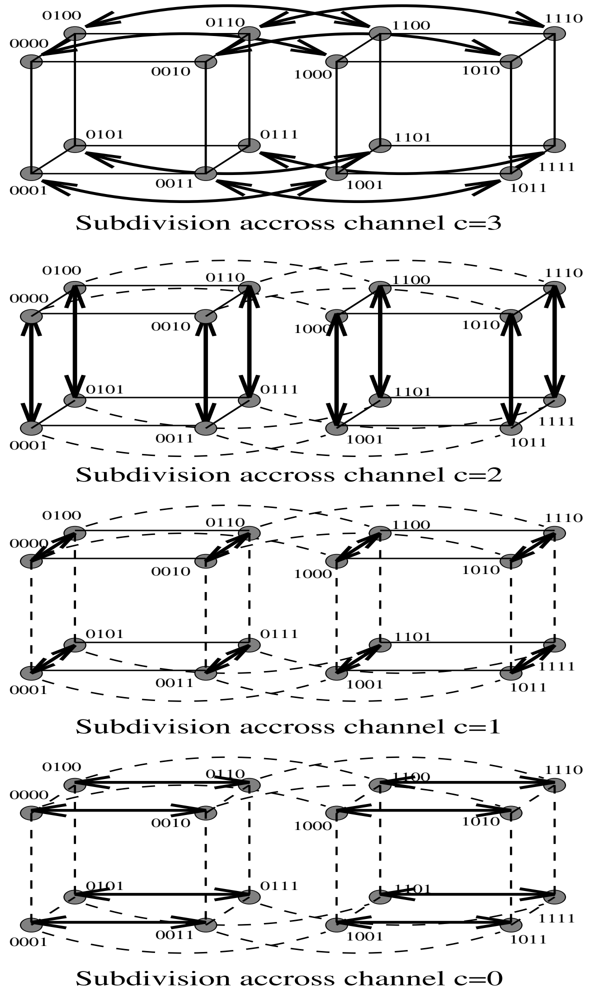

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
Subdivision accross channel c=3
Subdivision accross channel c=2
Subdivision accross channel c=1
Subdivision accross channel c=0

Figure 4: Operation structure of the proposed parallel BBSP algorithm.

parallel BBSP algorithm has a very regular communication structure and requires only concurrent single-hop communications (i.e., communications between neighbor processors) on hypercubes. The proposed parallel BBSP algorithm may also be adopted to other interconnection topologies. However, multi-hop communications may be required in other topologies.

In the proposed scheme, host processor randomly decomposes the object database into P even subsets such that each subset contains either $\lceil N/P \rceil$ or $\lfloor N/P \rfloor$ objects and it sends each subset to a different node processor of the hypercube. Then, the following steps are performed in a divide-and-conquer manner ($d = \log_2 P$ times) for each channel c from $c=d-1$ down to $c=0$.

Step 1 Node processors concurrently construct their local integer arrays corresponding to their local object database.

Step 2 Processors concurrently perform *prefix-sum* operation on their local integer arrays.

Step 3 Processors of each $(c+1)$-dimensional disjoint subcube perform global vector sum operation on their local integer arrays. Note that there exist 2^{d-c-1} global *vector-sum* operations performed concurrently. At the end of this step, processors of each $(c+1)$-dimensional subcube will accumulate the same local copies of the prefix-summed integer-arrays.

Step 4 Replicated integer arrays on x and y dimensions in each subcube are virtually divided into 2^{c+1} even slices and each slice is assigned to a different processor of that subcube. Then, processors perform the cost computations of the splitting planes corresponding to their slices in order to find their local optimal splitting planes.

Step 5 Processors of each subcube perform a global *minimum* operation to locate the optimal splitting plane corresponding to the subregion mapped to that subcube.

Step 6 Processors of each subcube determine their local subsubregion assignment, for the following stage $c-1$, according to the proposed mapping scheme. Then, processors concurrently perform a single pass over their local object database to gather and send the objects which belong to the other subsubregion to their neighbors on channel c. Hence, two subsubcubes of each subcube effectively exchange their subset of local object databases such that each subsubcube collects the object database corresponding to their subsubregion assignment in the following stage $c-1$. Note that 2^{d-c-1} subsubcube pairs perform such exchange operation concurrently.

During Step 6, processor pairs also determine their local shared objects which are not involved in the exchange operation. However, processors update either $xmin$ $(ymin)$ or $xmax$ $(ymax)$ values of their local shared objects according to their subsubregion assignment for a vertical (horizontal) splitting plane. Hence, processors maintain and process disjoint rectangular parts of the bounding boxes corresponding to the shared objects.

Figure 4 illustrates the operation structure of the proposed parallel BBSP algorithm on 4-dimensional hypercube topology. In this figure, links drawn as dashed lines illustrate the idle links in a particular stage of the parallel algorithm. Links drawn as solid lines illustrate the disjoint subcubes working concurrently and independently for the subdivision of their subregions at each stage. That is, processors of each subcube work in cooperation to determine the optimal subdivision of the subregion assigned to that subcube. These links also show the subcubes in which intra-subcube global vector-sum and global minimum operations are performed. In Figure 4, links drawn as solid lines with arrows illustrate the channel over which object-exchange operation takes places. These links also illustrate the subdivision of each subcube into two disjoint subsubcubes at the end of each stage. As is also seen in Figure 4, all objects arrive at their home processors after $log_2 P$ concurrent object-exchange operations. Note that shared objects will have more than one home processors and they will be replicated in those processors.

Experimental Results

The proposed parallel subdivision algorithm is implemented on an Intel‘s iPSC/2 hypercube multicomputer with 16 processors. The performance of the parallel program is experimented on several scenes containing different number of objects.

As is mentioned earlier, computational load balance and communication overhead are two crucial factors that determine the efficiency of a parallel algorithm. The recursive spatial bisection scheme employed in the BBSP algorithm tries to maintain load balance among the disjoint $(c+1)$-dimensional subcubes at each subdivision stage c during the first level of the parallel ray tracing computations. That is, in a particular subdivision stage c, the products of the number of local objects and areas of the rectangular subregions assigned to disjoint subcubes are approximately equal to each other. Note that, at the end of each stage of the parallel subdivision algorithm (Step 6), objects always migrate to their destination subcubes for the following stage. That is, at the beginning of each subdivision stage, each subcube holds only the local objects which belong to its respective local rectangular subregion. However, in the subdivision algorithm, the complexities of local object-based computations (Steps 1 and 6) and computations on local integer arrays (Steps 2, 3 and 4) within a subcube are proportional to the number of local objects and the semi-parameter (height+width), respectively, of the rectangular subregion assigned to that subcube. Hence, the complexities of local computations within a subcube during the parallel ray tracing and the parallel subdivision algorithms depend on the same factors; number of local objects, height and width of the rectangular subregion assigned to that subcube. However, the dependence is multiplicative in the parallel ray tracing, whereas, it is additive in the parallel subdivision. Hence, this deviation in the load balance measures of these two parallel algorithms may introduce load imbalance among subcubes during the parallel subdivision since the proposed parallel subdivision algorithm inherently operates in accordance with the mapping strategy adopted by the recursive spatial bisection scheme which tries to maintain a load balance during parallel ray tracing. This type of load imbalance is referred here as *inter-subcube* imbalance. There exists no load imbalance among the processors of the individual subcubes during the local integer computations at Steps 2, 3 and 4, since each processor of a subcube operates on local integer arrays of the same size. However, processors of the same subcube may hold different number of local objects belonging to the respective subregion during a particular stage of the algorithm. This type of load imbalance, which is referred here as *intra-subcube* imbalance, may introduce imbalance during the concurrent object-based computations (Steps 1 and 6) between the processors of the same subcube. Intra-subcube load imbalance may introduce processor idle time both during the global synchronization at Step 3 (global vector-sum operation) and object exchange synchronization at Step 6 within subcubes.

Initial *random* distribution of objects to processors is an attempt to reduce intra-subcube load imbalances.

The communication overhead of the proposed parallel algorithm involves two components; number and volume of communication. In a medium-to-coarse grain architecture with high communication latency, the number of communications may be a crucial factor affecting the performance of the parallel algorithm. Each one of the intra-subcube global operations at Steps 3 and 5 require $c+1$ concurrent exchange communication steps at stage c. Under perfect load balance conditions, these global communications within different subcubes will be performed concurrently. Hence, the total number of concurrent communications due to these intra-subcube global operations is $d(d+1)$. Thus, the total number of concurrent communications becomes $d(d+2)$ since the object exchange operations (Step 6) require d concurrent communications in total under perfect load balance conditions. Hence, percent overhead due to the number of communications is negligible for sufficiently large granularity (N/P) values.

The volume of concurrent communication during an individual intra-subcube global minimum operation (Step 5) is only $2(c+1)$ integers at stage c. On the other hand, the volume of the concurrent communication during an individual intra-subcube global vector-sum operation (Step 3) is $2(c+1)(n+m)$ integers where $n+m$ denotes the semi-perimeter of the rectangular subregion assigned to that subcube at stage c. That is, the total volume of this type of communications depend on the semi-perimeter of the initial window and d. Hence, percent overhead due to these types of integer communications decreases with increasing scene complexity for a fixed window size. The total volume of communication due to the object migrations is a more crucial factor in the parallel performance of the proposed parallel algorithm. Under average-case conditions, half of the objects can be assumed to migrate at each stage of the algorithm. Hence, if shared objects are ignored, the total volume of communications due to object migrations can be assumed to be $(N/2)\log_2 P$ objects. Experiments on various scenes yield results very close to this average-case behavior.

Under perfect load balance conditions, each processor is expected to hold N/P objects and each processor pair can be assumed to exchange $N/2P$ objects, at each stage. Hence, under these conditions total concurrent volume of communications due to object migrations will be $(N/2P)\log_2 P$ objects. Experiments on various uniform scenes yield results very close to these expectations. However, results slightly deviate from these expectations for non-uniform scenes with objects clustered toward particular positions.

Figure 5 illustrates the efficiency curves for different dimensional hypercubes as function of the scene complexity. Efficiency values on a hypercube with P processors are computed as $E_p = T_1/PT_p$ where T_1 and T_p denote the execution times of the sequential and parallel subdivision programs on 1 and P node processors, respectively. As is seen in Figure 5, efficiency increases with increasing scene complexity and fixed window resolution size. This decrease can be attributed to two factors. The total number of communications stays fixed for a fixed hypercube size. Hence, percent overhead due to the total number of communications decreases with increasing scene complexity. Similarly, the volume of integer communications also stays fixed for fixed hypercube size and window resolution size. Hence, percent overhead due to the volume of integer communications also decrease with increasing scene complexity. As is seen in Figure 5, efficiency values close to 100% are obtained for $P=2$ processors since initial even distribution of objects entirely avoids both intra- and inter-subcube load imbalances during the first stage of the parallel BBSP algorithm. However, for a fixed scene instance, efficiency decreases considerably with increasing number of processors. This decrease is mainly due to the increase in the inter-subcube load imbalances, since each doubling of the number of processors introduces an extra stage to the algorithm. Therefore, load re-balancing algorithms should be developed for larger number of processors.

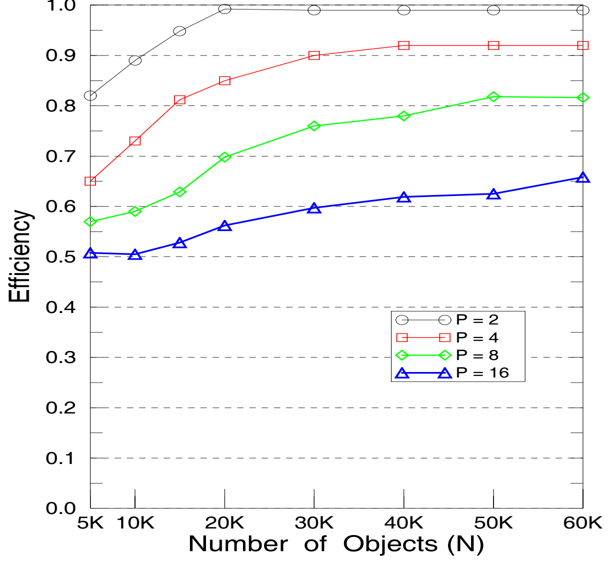


Figure 5: Efficiency curves with respect to the total number of objects in the scene

Conclusion

An efficient subdivision algorithm based on BSP (called BBSP) is proposed for object-based parallel ray tracing. The proposed BBSP algorithm tries to minimize the communication overhead during the object-based parallel ray tracing by exploiting data coherence. The other advantage of the proposed BBSP is that subdivision process does not generate empty boxes. Empty boxes may occupy significantly large space. Besides, rays may spend time while skipping the empty boxes. Subdivision of space into the 3-D grid elements and in the octree fashion suffer from these factors.

The preprocessing due to the subdivision of the 3-D space may be time consuming for complex scenes. An efficient parallel BBSP algorithm is proposed and presented to reduce the preprocessing time. The implementation on an Intel iPSC/2 multicomputer achieved promising results.

References

[1] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE CG&A*, pages 12–26, November 1989.

[2] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE CG&A*, pages 14–20, May 1987.

[3] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE CG&A*, pages 15–22, October 1984.

[4] M. R. Kaplan. The use of spatial coherence in ray tracing. In *Techniques for Computer Graphics*, pages 173–193. Springer-Verlag, 1987.

[5] V. İşler, C. Aykanat, and B. Özgüç. Subdivision of 3d space based on the graph partitioning for parallel ray tracing. In *Proceedings: Second Eurographics Workshop on Rendering*. Eurographics, Spain, May 1991.

[6] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, (4):197–209, 1988.

[7] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing. *The Visual Computer*, (5):109–119, 1989.

[8] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE CG&A*, pages 16–26, April 1985.

[9] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, (6):153–166, 1990.

[10] F. Özgüner and C. Aykanat. A reconfiguration algorithm for fault tolerance in a hypercube multiprocessor. *Information Processing Letters*, 29:247–254, 1988.

[11] M. D. J. McNeill, B. C. Shah, M. P. Hébert, P. F. Lister, and R. L. Grimsdale. Performance of space subdivision techniques in ray tracing. *Computer Graphics Forum*, 11(4):213–220, 1992.