Subdivision of 3D Space Based on the Graph Partitioning for Parallel Ray Tracing

Veysi İşler, Cevdet Aykanat, Bülent Özgüç

Abstract

An approach for parallel ray tracing is to subdivide the 3D space into rectangular volumes and assign the object descriptions with their related computations in each volume to a different processor. The subdivision process is critical in reducing the interprocessor communication overhead, and maintaining the load balance among processors of a multicomputer. In this paper, after a brief overview of parallel ray tracing, a heuristic is proposed to subdivide the 3D space by converting the problem into a graph partitioning problem. The proposed algorithm tries to minimize the communication cost while maintaining a load balance among processors.

1 Introduction

Realistic images are used widely in many computer graphics applications such as computer-aided design (CAD), animation and visualization, simulation, education, robotics, architecture, advertising, medicine, etc. Ray tracing is the most powerful algorithm to produce realistic images by finding the interaction of light sources and the objects in an environment. Although ray tracing is a simple algorithm, it requires excessive floating point operations. The amount of computation mainly depends on two parameters: the total number of pixels in the generated image and the number of objects in the scene. In the naive algorithm, the number of objects has a great effect on the total computation time, since each ray is tested with all objects in the scene to find the first intersection point. Several methods have been developed to reduce the computation time by testing only the objects on the ray's path for intersection. Spatial subdivision and the use of a hierarchy of bounding volumes are two of the well known techniques that aim to generate ray traced images independent of the complexity of the objects in the scene.

Even if we could reduce the total computations for a ray to a constant time, it is still necessary to process all of the pixels independently. Additionally, we may wish to shoot more than one ray for each pixel to increase the accuracy of the image as well as adding diffuse effects. This means that the algorithm has another bottleneck due to the number of rays traveling in the scene. Thus, parallelism is essential in speeding up the ray tracing algorithm. Furthermore, ray tracing is easily amenable to parallelization on a multiprocessor, since each primary ray is traced independently.

In recent years, research on ray tracing has been mostly concentrated on speeding up the algorithm on multiprocessors. There are mainly two approaches to parallelize ray tracing. One of them is an image space subdivision in which the computations related to different rays are distributed to the processors. The other approach studies the parallel ray tracing on a distributed-memory message-passing multiprocessor (multicomputer). 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, the object space data as well as computations should be distributed among processors of the multicomputer. Since the model database (scene description with the auxiliary data structure) may not fit into the local memory of each processor, object space data need to be distributed.

In this paper, our concern is to speed up the ray tracing on a multicomputer for complex scenes that require large amounts of memory. Thus, both computations and the

P. Brunet et al. (ed.), *Photorealistic Rendering in Computer Graphics*
© EUROGRAPHICS The European Association for Computer Graphics 1994

entire database are distributed fairly among the processors. Section 2 contains a short overview of parallel ray tracing algorithms. In Section 3, we propose a subdivision scheme which maintains the load balance among processors and minimizes the interprocessor communication cost. The scheme transforms the subdivision of 3D space problem to a graph partitioning problem with some imposed constraints as discussed in Section 3.1. A Kernighan-Lin-like [11] algorithm is presented as a solution in Section 3.2.

2 Parallel Ray Tracing

A large number of parallel systems have been proposed to exploit the inherent parallelism in the algorithm. Most of these are special-purpose systems that require the construction of custom hardware using VLSI. The recent developments in the VLSI technology have made it feasible to design and implement special-purpose hardware for the ray tracing algorithm [5, 10, 16]. In spite of the gain obtained in this way, these special purpose architectures have several disadvantages. First, there are still studies to improve the algorithm itself. Researchers should thus work on general purpose machines in order not to be restricted by the hardware. Second, special purpose hardware is expensive and often restricts the applications that require other computer graphics algorithms.

The other approach that exploits speedup through the inherent parallelism in ray-tracing investigates the algorithm on a general purpose parallel architecture independent of the hardware configuration [4, 8, 12, 13, 14]. The effective parallelization of the ray tracing algorithm on a multicomputer requires the partitioning and mapping of the ray tracing computations and the object space data. This partitioning and mapping should be performed in a manner that results in low interprocessor communication overhead and low processor idle time. Processor idle time can be minimized by achieving a fair load balance among the processors of the multicomputer. Two basic schemes exists for parallelization. In the first scheme, only ray tracing computations are partitioned among the processors. In the other scheme, both ray tracing computations and object space data are partitioned among the processors.

In the first scheme, the overall pixel domain of the image space to be generated is decomposed into subdomains. Then, each pixel subdomain is assigned to and computed by a different processor of the multicomputer. However, each processor should keep a copy of the entire information about the objects in the scene in order to trace the rays associated with the pixels assigned to itself. Hence, an identical copy of the data structure representing the overall object space is duplicated in the local memory of each processor. This scheme requires no interprocessor communication since the computations associated with each pixel is independent of each other.

The image space subdivision achieves almost a linear speedup. No communication is needed between processors. The only overhead is the communication between the scheduler and the processors of the multicomputer. On the other hand, each processor should have access to the whole scene description, since ray-object intersection tests may be carried out with any object in the scene. This is a big disadvantage. Furthermore, sometimes a large amount of storage is needed to hold the object definitions and other related information. Therefore, processors cannot store the entire information about the objects in the scene.

The subdivision of the object space necessitates interprocessor communication, because each processor owns only a portion of the database. During the execution, a processor may need some portion of the database that exists in the local memory of another processor. In this case, either the needed portion is sent to the requesting processor [2, 10, 9] or the ray with the other relevant information is passed to the processor that has the needed part of the database [3, 4, 5, 13].

In object-based type of algorithms, the load imbalance is the major problem to deal with, since some processors may contain objects that are more likely to be intersected than others. Additionally, it is not easy to achieve a linear speedup as in image space subdivision where object space data is duplicated in the local memory of each processor. The communication overhead between processors might drastically degrade the performance.

3 Subdivision of 3D Space

A new scheme is proposed in order to reduce the negative effects of object-space subdivision techniques mentioned above. The proposed scheme consists of three phases. In the first phase, the problem is converted to a graph partitioning problem. The graph obtained in this phase is a directed graph with weighted nodes and edges. In the second phase, this graph is partitioned into P clusters where P is the number of processors in the multicomputer. The objective in this partitioning is to minimize the weighted sum of inter-cluster edges, subject to the constraint that the cluster weights are balanced to within a specified tolerance. The actual assignment of the clusters to the processors of the multicomputer is done in the third phase. The objective in the assignment of these P clusters to P processors is to minimize the distances of interprocessor communications.

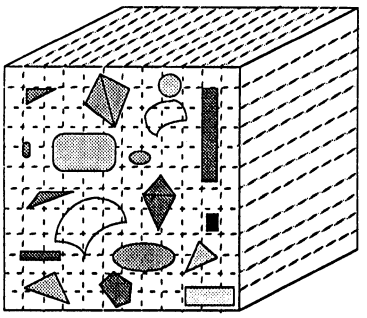

Fig 1. Regular subdivision of 3D scene.

3.1 Graph Generation

We start by subdividing the 3D space into disjoint rectangular volumes by planes perpendicular to x-y plane as shown in Figure 1. Then, a primary ray is shot into the 3D space for each region (called coarse grid) in a very similar way Bouatouch, et. al. proposed [1]. The primary rays are traced until they reach a certain level. During this sub-sampling phase, an account of rays entering and leaving each subvolume is collected. Additionally, the number of rays processed in each subvolume is stored. The subvolumes and adjacency relations correspond to the nodes and edges of the graph, respectively. Figure 2 illustrates the resulting graph for a 12 × 12 coarse grid. Nodes of the graph represent the rectangular subvolume in the 3D coarse grid. Each node u of the graph is associated with a weight $\omega(u)$ where $\omega(u) = \alpha\omega_{CL}(u) + \beta\omega_{MR}(u)$. Here, computational load $\omega_{CL}(u)$ is proportional to the number of rays processed in subvolume u during the subsampling phase. The complexity of the intersection tests for each ray processed in subvolume u is also taken into account during the computation of $\omega_{CL}(u)$. The second term $\omega_{MR}(u)$ represents the storage requirement for the objects resident in subvolume u. The α, β parameters are introduced to achieve a balance between the computational and memory load in assigning weights to the nodes of the graph. The node weights will be used to determine the load (computational + memory) of each partition. The directed edge uv from node u to node v represents the adjacency relation between the associated subvolumes. Each directed edge uv is associated with a weight $d(uv)$ where $d(uv)$ is proportional to the number of rays entered into subvolume v from subvolume u during subsampling phase. The edge weight $d(uv)$ represents the additional number and volume of communications to be performed, due to the rays originating from subvolume u and entering subvolume v, by processor P_i to processor P_j if subvolumes u and v are mapped to different processors P_i and P_j, respectively. Hence, edge weights will be used to determine the interprocessor communication load.

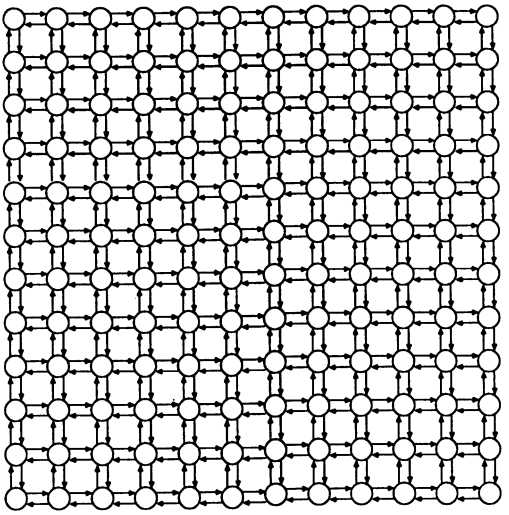

Fig 2. Resulted graph from the first step.

3.2 Graph Partitioning

It is known that partitioning a graph into a number of clusters with predetermined sizes and with minimum number of inter-cluster edges is an NP-complete problem [7]. The algorithm proposed by Kernighan and Lin (KL) is a popular non-trivial heuristic to solve the graph partitioning problem [11]. This heuristic provides hill-climbing ability to strictly local search methods by finding a favorable sequence of swaps of nodes between partitions rather than a single favorable swap of groups of nodes. Fiduccia and Mattheyses [6] improved the KL algorithm by introducing efficient data structures and node moves across clusters instead of swaps between clusters.

The partitioning algorithm proposed in this paper resembles the one proposed by Fiduccia and Mattheyses. However, the application problem that is of interest here imposes a major constraint on the spatial shape of the clusters. The constraint imposed is the convexity requirement for the image subspace associated with each cluster. The reason for this constraint is the problem that arises due to the traversal of the rays in 3D space in order to utilize spatial coherence. For spatial subdivision into non-convex subvolumes, movement to the next subvolume on the ray's path is complex and time consuming; besides the same ray might exit and reenter a subvolume thus causing extra tests and interprocessor communication for intersections and other operations. The convexity constraint for clusters can easily be maintained by restricting a move to a move of a row or a column of nodes at a time instead of a single node move. The simplest way to achieve this is to introduce horizontal and vertical virtual lines to partition the graph into clusters. Assume that the nodes of the graph are to be mapped onto a multicomputer with $P = P_x \times P_y$ processors. The algorithm partitions the graph into P clusters by $P_x - 1$ vertical and $P_y - 1$ horizontal lines. Figure 3 illustrates the partitioning of a 12×12 coarse grid for a $P = 4 \times 4 = 16$ processor multicomputer. In this scheme, a move is considered to be the move of a whole horizontal and a vertical line effectively resulting in the move of groups of nodes across clusters. The resulting clusters will be in rectangular shape thus satisfying the convexity requirement and making the use of simple traversal algorithms feasible. This scheme also ensures the 2D mesh adjacency relation among clusters.

An alternative approach is to move either vertical or horizontal boundary line segments of the clusters instead of whole lines dividing the mesh graph into two mesh subgraphs.

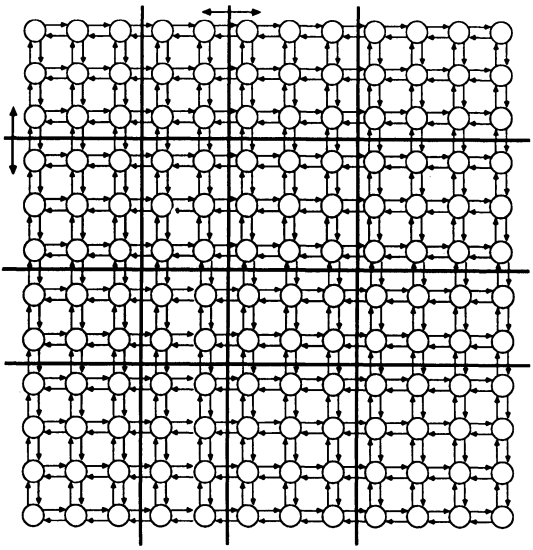

Fig 3. Line moves.

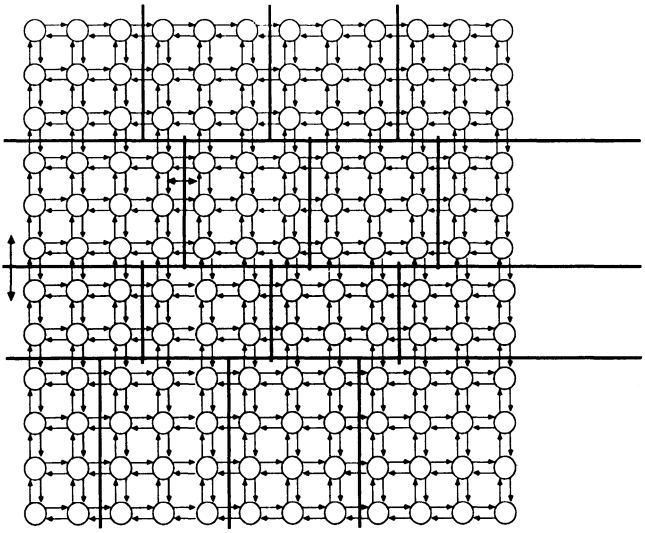

Fig 4. Line segment moves.

That is, either whole vertical lines and horizontal line segments or whole horizontal lines and vertical line segments are allowed to move. Vertical line segment moves are illustrated in Figure 4 for partitioning a 12×12 coarse grid for a $P = 4 \times 4 = 16$ processor multicomputer. This approach increases the search space by increasing the total number of possible moves. In this approach, the convexity of the clusters is ensured by restricting the moves to either horizontal or vertical line segment moves. The only drawback in this approach is that the mesh adjacency relation among the resulting clusters is not ensured. The problem of mapping of the clusters to the processors of the hypercube in this case is not straightforward and necessitates heuristics discussed in [15].

Assume that the nodes of the graph are to be mapped onto a multicomputer with P processors, where $P = P_x \times P_y$. The algorithm initially partitions the graph into P clusters by $(P_x - 1) \times P_y$ vertical line segments and $P_y - 1$ horizontal lines. The vertical and horizontal lines are aligned with the horizontal and vertical coarse grid lines such that each cluster is assigned equal number of coarse grid pixel areas. Each line is associated with two moves. Two moves associated with a horizontal line (vertical line segment) correspond to the action of sliding the horizontal line (vertical line segment) up or down (right or left) by one position on the 2D coarse grid. A line move may increase the load of some clusters by enlarging them spatially and decrease the load of some other clusters by contracting them spatially. The steps of the algorithm are given below.

1. **a)** Compute the weight $W_{C_{ij}} = \sum_{u \epsilon C_{ij}} \omega(u)$ for each cluster C_{ij}.
   b) Find all feasible moves associated with horizontal lines and vertical line segments. A move is feasible if the weight of any cluster affected by the move does not deviate from the average weight within a specified tolerance.
   c) Compute the gain of each feasible move. The gain of a move corresponds to the decrease in the sum of the inter-cluster edge weights if the move is realized.
2. Unlock and unmark each move.
3. Choose the move M_i with maximum gain g_i among the feasible unlocked moves.
   a) Mark move M_i, lock the reverse move of M_i. Reverse move is the other move associated with the same line segment as M_i but in reverse direction.
   b) Recompute the gain of move M_i.
   c) If move M_i is a horizontal line move then:
      i) Update the gains of the moves associated with its $2(P_x - 1)$ adjacent vertical line segments.
      ii) Unlock these adjacent vertical line segment moves (if they were previously locked).
   d) If move M_i is a vertical line segment move then unlock the moves associated with its two adjacent horizontal lines (if they were previously locked).
   e) Update the weights of the clusters that are affected by the move M_i. Then, update the feasibility status of the moves adjacent to these clusters.
4. Step 3 is repeated until no move can be selected, resulting in the marked moves $M_1, ..., M_n$ with gains $g_1, g_2, ..., g_n$.
5. Choose k $(1 \leq k \leq n)$ which maximizes $G = \sum_{i=1}^{k} g_i$. If $G_{max} > 0$, realize the sequence of moves $M_1, M_2, ..., M_k$ and then go to step 1. Otherwise exit.

In the first step of the algorithm, the initializations related to the cluster weights, the feasibility status and the gains of moves are carried out. There are $P_x \times P_y$ clusters which are divided by $P_y - 1$ horizontal lines and $(P_x - 1) \times P_y$ vertical lines. Since each line is associated with two moves, the number of possible moves is $2 \times ((P_x - 1) \times P_y + P_y - 1) = 2 \times (P - 1)$. These initial computed values are updated incrementally during the inner loop at step 3.

The second step of the algorithm is used to clear all flags before the execution of the inner loop. A locked move is not taken into consideration until it is unlocked during the

iteration of the inner loop. On the other hand, a marked move is still considered during the iteration of the inner loop.

The third step picks a move with the highest gain from feasible and unlocked moves. This move is marked at step 3.(a) for a possible realization at step 5. On the other hand, the reverse move associated with the same line as the marked move is locked to prevent oscillations. The move M_i of a horizontal line or a vertical line segment does not prevent the further move of the same line or line segment in the same direction. This new candidate move is automatically relabeled as M_i again, and its gain is recalculated as indicated at step 3.(b).

A horizontal line move alters the nature of the adjacent vertical line segment moves. A vertical line segment and a horizontal line are considered to be adjacent if they form a corner of a cluster. Therefore, there are two successive rows of vertical line segments adjacent to an individual horizontal line. If a horizontal line move is realized, then each adjacent vertical line segment move corresponds to the move of a different group of nodes. That is, either one node is to be added or deleted from the original group of nodes. For this reason, the moves associated with these adjacent vertical line segments should be unlocked if they were locked and their gains should be updated incrementally as indicated at step 3.(c).

Similarly, a vertical line segment move alters the nature of the moves associated with the two successive horizontal lines which are adjacent to that vertical line segment. Hence, these adjacent horizontal lines should be unlocked if they were previously locked as indicated at step 3.(d). However, the gains of the moves associated with these two adjacent horizontal lines are not affected at all.

A vertical line segment move alters the weights of only two adjacent clusters which share this vertical line segment as a right or left boundary. A horizontal line move modifies the weights of the two successive row of clusters sharing this horizontal line as an up or down boundary. Hence, the weights of these clusters should be updated incrementally as indicated at step 3.(e). The change in the weight of a cluster may affect the current feasibility status of the moves associated with the lines or line segments which are adjacent to that cluster. A line or line segment is considered to be adjacent to a cluster if it constitutes a boundary for that cluster. Hence, the feasibility status of the adjacent line or line segment moves should be updated accordingly as indicated at step 3.(e).

Step 3 is iterated until no feasible and unlocked move remains as indicated at step 4. Then, at step 5, the sequence of moves which maximize the overall gain is selected. If this maximum gain $G_{max} > 0$, it means that the sequence of marked line or line segment moves $M_1, M_2, ..., M_k$ result in a further reduction in the overall interprocessor communication overhead. In this case, this sequence of moves is realized and then the outer loop (steps 1-5) is iterated once more. Note that, the selected moves are not really realized at step 3. In fact, the effects of these selected moves are simulated as if these moves are realized. If $G_{max} \leq 0$ then the algorithm is terminated since no further reduction in the overall interprocessor communication can be obtained.

Note that, at step 3, a selected move M_i with maximum gain may have a negative gain. Such a move is still considered for realization since it may result in moves with positive gains in the following cycles of the inner loop. This selection scheme enhances the KL algorithm with hill-climbing ability which is not available in other strictly local search methods.

Figure 5 illustrates a subgraph (two middle successive cluster rows) of a sample graph to be partitioned. Assume that, move M_j (left move of vertical line segment L_x) has a maximum gain and hence, selected at the k^{th} iteration of step 3. Note that move M_j corresponds to the move of group of nodes $u_{5,4}, u_{6,4}, u_{7,4}$ from cluster $C_{y2,x1}$ to $C_{y2,x2}$. Hence, the gain of M_j is $g_j = \sum_{i=5}^{7} D(u_{i,4}, u_{i,5}) - \sum_{i=5}^{7} D(u_{i,3}, u_{i,4})$. Here, $D(u,v)$ is the total weight of the directed edges between node u and v. That is, $D(u,v) = d(u,v) + d(v,u)$. At step 3.(a), move M_j is marked as M_k with gain $g_k \leftarrow g_j$. The reverse move M_{j+1} (right move of vertical line L'_x) is also locked at step 3.(a). The gain of move M_j is recalculated as $g_j = \sum_{i=5}^{7} D(u_{i,3}, u_{i,4}) - \sum_{i=5}^{7} D(u_{i,2}, u_{i,3})$ L_{y-1} are adjacent to L_x. Hence, these lines are unlocked (if they were previously locked) at step 3.(d). The weights of the two clusters $C_{y2,x1}$ and $C_{y2,x2}$ are updated at step 3.(e) as follows: $W_{y2,x1} = W_{y2,x1} - \sum_{i=5}^{7} \omega(u_{i,4})$ and $W_{y2,x2} = W_{y2,x2} + \sum_{i=5}^{7} \omega(u_{i,4})$. The feasibility status of the moves associated with hori-

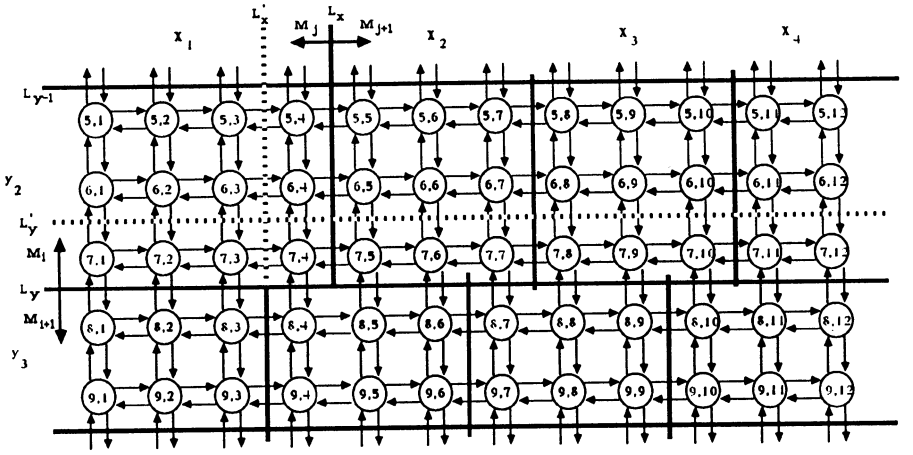

Fig 5. Sample graph to be partitioned.

zontal lines L_{y-1} and L_y and all vertical line segments lying between these two successive horizontal lines are updated, accordingly.

Assume that, move M_i (up move of horizontal line L_y) is selected in a later iteration of step 3. The dotted horizontal line L_y' denotes the new position of L_y if move M_i is realized. Move M_i as marked and the reverse move M_{j+1} is locked similarly at step 3.(a). The gain of move M_i is recalculated at step 3.(b) as $g_i = \sum_{j=1}^{12} D(u_{8,j}, u_{7,j}) - \sum_{j=1}^{12} D(u_{7,j}, u_{6,j})$. All vertical line segments seen in this subgraph are adjacent to the horizontal line L_y. Hence, the gains of the moves associated with all the vertical line segments are updated at step 3.(c). For example, $g_j = g_j - D(u_{7,3}, u_{7,4}) + D(u_{7,2}, u_{7,3})$. Similarly, these moves are unlocked if they were previously locked. For instance, previously locked move M_{j+1} is unlocked at this step. All clusters shown in the subgraph (Fig. 5) are affected by move M_i. Thus, the weights of these clusters are updated accordingly at step.3(e). All moves shown in the subgraph are adjacent to the updated clusters. Hence, the feasibility status of all possible moves shown in the subgraph are also updated at step 3.(e).

4 Conclusion and Future Work

The proposed heuristic algorithm explicitly tries to minimize the overall sum of inter-cluster edge weights while maintaining the load balance within a specified tolerance. The minimization of the sum of inter-cluster edge weights corresponds to the minimization of the interprocessor communication overhead since different clusters will be mapped to different processors.

Since the implementation of the proposed heuristic partitioning algorithm is still in progress, no performance results can now be presented. It seems to converge to a solution as in the original graph partitioning algorithm given by Fiduccia and Mattheyses. However, we are not sure of the ratio between the preprocessing phase time and the observed image generation time. In the near future, the efficiency and speed-up curves for the parallel ray tracing algorithm mapped to the hypercube using this proposed heuristic partitioning algorithm will be obtained.

In order to enlarge the search space for line moves in the given graph, the number of horizontal line moves might be increased as vertical moves. That is, both horizontal and vertical line segments can be allowed to move. However, in this case efficient graph theoretical methods should be developed to restrict the line segment moves in order to maintain the convexity of the resulting clusters.

Acknowledgements

This project is supported by the following grants and funds: Bilkent University Research Funds, The Scientific and Technical Research Council of Turkey (TÜBİTAK) Research Grant MAG917-EEEAG5, Intel Corporation Grant SSD100791-2.

References

[1] K. Bouatouch and T. Priol. Parallel space tracing: An experience on an ipsc hypercube. In N. M. Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics, Proceedings of CG International'88*, pages 170–188. Springer-Verlag, Berlin Heidelberg, 1988.

[2] M. B. Carter and K. A. Teague. Distributed object database ray tracing on the intel ipsc/2 hypercube. Technical report, Dept. of Electrical and Computer Engr., Oklahama State University,USA, 1990.

[3] E. Caspary and I. D. Scherson. A self-balanced parallel processing for computer vision and display. In *International Conference*. University of Leeds, UK, 1988.

[4] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, 5:3–12, 1986.

[5] M. Dippé and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *ACM Computer Graphics*, 8(3):149–158, July 1984.

[6] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th Design Automation Conf.*, pages 175–181, 1982.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York,, 1979.

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

[9] S. A. Green and D. J. Paddon. A highly flexible multiprocessor solution for ray tracing. *The Visual Computer*, (6):62–73, 1990.

[10] S. A. Green, D. J. Paddon, and E. Lewis. A parallel algorithm and tree-based computer architecture for ray-traced computer graphics. In *International Conference*, University of Leeds, UK, 1988.

[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, February 1970.

[12] H. Kobayashi and T. Nakamura. Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, (3):13–22, 1987.

[13] K. Nemoto and T. Omachi. An adaptive subdivision by sliding boundary surfaces. In *Proceedings: Graphics and Vision Interface '86*, pages 43–48. Canadian Information Society, Toronto, 1986.

[14] D. J. Plunkett and M. J. Balley. The vectorization of ray-tracing algorithm for improved execution speed. *IEEE CG&A*, pages 52–60, August 1985.

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

[16] R. Pulleyblank and J. Kapenga. The feasibility of a vlsi chip for ray tracing bicubic patches. *IEEE CG&A*, pages 33–44, March 1987.