A Comparison of Spatial Subdivision Algorithms for Sort-First Rendering*

Tahsin M. Kurç, Hüseyin Kutluca, Cevdet Aykanat, and Bülent Özgüç

Dept. of Computer Engineering and Information Sci.
Bilkent University, 06533 Ankara, TURKEY

Abstract. This paper describes and experimentally evaluates three adaptive spatial subdivision heuristics for sort-first parallel graphics rendering on distributed-memory multicomputers. In sort-first rendering, image-space, or screen, is divided into regions. Each processor is assigned one or multiple regions to render. Primitives in the scene are redistributed among the processors according to region assignments. The total number of primitives in the system may increase due to duplication of primitives crossing multiple regions assigned to different processors. The common characteristic of the algorithms discussed in this paper is that a 2-dimensional mesh is superimposed on the screen and screen is subdivided using the primitive distribution on this mesh. Each processor is assigned a single region to render. The first algorithm uses summed area table for subdividing the screen. The second algorithm uses quadtrees for subdivision. The third algorithm utilizes a graph partitioning approach. The second and the third algorithms are new approaches to adaptive screen subdivision in sort-first rendering. We compare the algorithms in terms of load balancing performance, execution time of the algorithms, and percent increase in the total number of primitives after redistribution step. In this paper, we do not target a specific rendering method such as polygon rendering or volume rendering. Our load balancing criteria is the number of primitives in each processor. We present experimental results on a Parsytec's CC parallel system.

1 Introduction

Computer graphics rendering requires excessive amount of computation time and large memory space as the complexity of scenes to be rendered increases and more computation intensive algorithms are used for higher quality images. In addition, visualization of scientific data has become very popular in recent years. In such applications data sets are generally large and rendering algorithms require long execution times on single processor architectures. Distributed-memory multicomputers can provide necessary computing power and memory space for such applications. Molnar et al. [9] classify and evaluate parallel rendering approaches in three groups as *sort-first*, *sort-middle*, and *sort-last*. This classification is based on where in the rendering pipeline the redistribution of data is

* This work is partially supported by the Commission of the European Communities, Directorate General for Industry under contract ITDC 204-82166, and The Scientific and Technical Research Council of Turkey (TÜBİTAK) under grant EEEAG-160.

carried out. Here, we briefly describe *sort-first* parallel rendering. Initially each processor is assigned a subset of primitives in the scene. A pre-transformation step is applied on the primitives in each processor to find their positions on the screen. This pre-transformation step typically produces screen-space bounding boxes of the primitives. Screen is subdivided into regions and each processor is assigned one or multiple regions of the screen to perform rendering operations. In this paper, a region is referred to as a subset of pixels on the screen. This subset of pixels may form connected or disconnected subregions on the screen. The primitives are then redistributed among the processors using the screen-space bounding boxes so that each processor has the primitives crossing the region assigned to it. After the redistribution step, each processor performs geometry processing and rasterization steps of rendering pipeline on its region independent of the other processors. Primitives crossing more than one region, referred to here as *shared primitives*, are duplicated in the processors assigned those regions. For this reason, the total number of primitives in the system may increase after redistribution. Efficient sort-first parallel rendering on distributed-memory multicomputers requires, first of all, even subdivision of data and computations among processors. Subdivision of the screen can be done adaptively using primitive distribution over the screen for better load balancing. In addition, number of shared primitives should be minimized since such primitives result in inefficient utilization of computing power and overall memory space in the multicomputer due to duplicated computations and duplicated storage.

There are numerous approaches for sort-middle [17, 2, 4], sort-last [1, 14, 7, 8], and for sort-first [10, 16] parallel rendering. Mueller [10] investigates the sort-first rendering approach for interactive applications in polygon rendering. He uses an adaptive subdivision algorithm, referred to as mesh-based adaptive hierarchical decomposition (MAHD) algorithm, based on summed area table [3] generated from 2-dimensional mesh superimposed on the screen. In our earlier work [16], spatial subdivision is investigated for direct volume rendering. In that paper, 1-dimensional data structures, in particular arrays for each dimension of the screen, were used to find load distribution and to subdivide the screen space.

In this work, we present and experimentally evaluate three adaptive spatial subdivision heuristics for sort-first rendering on distributed-memory multicomputers. The common characteristic of these algorithms is that a 2-dimensional mesh is superimposed on the screen and screen is subdivided using the primitive distribution on this mesh. Each processor is assigned a single region to render. The first algorithm is the MAHD and uses summed area table for subdividing the screen. The second algorithm uses quadtree and orthogonal recursive bisection with medians-of-medians (ORB-MM) for subdivision. The third algorithm utilizes a graph partitioning approach. The second and the third algorithms are new approaches to adaptive screen subdivision for sort-first rendering. We compare the algorithms in terms of load balancing performance, execution time of the algorithms, and percent increase in the total number of primitives after redistribution step. In this paper, we do not target a specific rendering method such as polygon rendering or volume rendering. Our load balancing criteria is the number of primitives in each processor. We present preliminary experimental results on a Parsytec's CC parallel system.

2 Subdivision Algorithms

In this section, adaptive subdivision algorithms investigated in this paper are presented. Initially, each processor is assigned a subset of primitives in the scene. In our work, each processor receives an equal number of primitives. Pre-transformation operations are applied to obtain screen-space bounding boxes of the primitives and eliminate trivially invisible primitives (e.g., primitives totally clipped out). Then, a 2-dimensional mesh is superimposed on the screen. The distribution of primitives on this mesh is used to subdivide the screen into P regions, where P is the number of processors in the parallel machine. After the subdivision, primitives are redistributed according to new screen-to-processor assignments.

2.1 Mesh-based Adaptive Hierarchical Decomposition (MAHD)

In this approach [10], after the mesh is superimposed on the screen, primitives are tallied to mesh cells. Some primitives may cross multiple cells. In order to decrease the errors due to counting such primitives many times, Mueller uses a simple heuristic. Each primitive increments the weights of each cell it intersects by a value inversely proportional to the number of cells the primitive crosses. In this heuristic, if we assume that there are no shared primitives between screen regions, the sum of the weights of individual cells forming a region gives a value linearly proportional to the exact number of primitives in that region. However, shared primitives still cause some errors but it can be expected that such errors are less than counting such primitives multiple times while adding cell weights. Mueller also points out in the paper that this heuristic gives good results.

After all primitives in each processor are tallied, each processor sends local meshes to the host processor or to another processor, say processor 0, if there is no host. The receiving processor adds the weights of the received cells to the weights of the corresponding cells in the global mesh to be used in subdivision. This processor then converts the global mesh into a summed area table [3]. The summed area table can be generated by performing a prefix sum on each individual row of the mesh followed by a prefix sum on each individual column. The screen is subdivided into regions at cell boundaries recursively using this summed area table. At each subdivision step longer dimension of the intermediate region is divided. Dividing the longer dimension aims at reducing the length of the perimeter of the final regions as an attempt to reduce the number of shared primitives crossing region boundaries. The structure of the summed area table allows binary search to determine region boundaries. After finding P regions, host processor (or processor 0) broadcasts the region assignments to all processors so that primitives can be redistributed accordingly. In this scheme, resulting regions are rectangular and each region consists of adjacent cells.

2.2 ORB with Medians-of-Medians (ORB-MM)

In this scheme, we generate a quadtree from the mesh superimposed on the screen. Each leaf node of the tree is referred to here as *quad-node*. The quadtree is generated in such a way that each quad-node has approximately equal workload. This quadtree is partitioned using orthogonal recursive bisection with medians-of-medians (ORB-MM) [15, 11]. Pilkington and Baden [11] applies ORB-MM on a rectilinear mesh. Singh et al. [15] uses ORB-MM (referred to as ORB-Final in their paper) on a quadtree for fast multipole method. In their approach, quadtree is already generated by fast multipole method.

In this work, we use a 2-dimensional segment tree [13] to generate the quadtree. The 2-dimensional segment tree data structure has hierarchical and recursive structure. The root of the tree covers all screen and at each level a node is divided into four quadrants, forming child nodes. This division is repeated until the size of a leaf node is equal to one mesh cell. In our work, segment tree is implemented as an array in which children of any node are reached with $4 \times node\_index + i$ for $i = 1, ..., 4$, and $node\_index$ of the root is equal to 0. Each processor creates its own segment tree using local primitives. A bounding box contributes to a tree node if and only if the box overlaps with the node and no ancestor of the node is contained by the bounding box. Our structure is an *augmented* data structure such that each non-leaf node stores the contributions to itself and sum of its children. One bounding box can be partitioned into at most $O(N)$ squares for an $N \times N$ mesh. In order to reduce the errors due to counting primitives contributing multiple nodes, we use a similar method to that of MAHD. We add a value proportional to the fraction of overlapping area of bounding box on the node to total area of the bounding box. After all bounding boxes in each processor are inserted into the segment tree, local segment trees are merged to obtain global segment tree in each processor to generate the quadtree.

Quadtree has a similar structure to segment tree. Each node is also divided into four quadrants, but this time subdivision ends when workload of a node drops under a specified limit. The aim of the quadtree is to divide the space to rectangular subregions such that each subregion has approximately the same amount of work. Quadtree is generated by traversing the segment tree recursively as follows. When a node in segment tree has a value under the specified limit or is a leaf node in segment tree it is added to quad-node list, otherwise it is subdivided into four by traversing the child nodes. When segment tree is traversed completely leaf quad-nodes of the quadtree are inserted into a linked list structure to be used in subdivision.

We selected the limit with experimental results. In our case, limit value is taken as a certain percentage of number of primitives. Larger limit values cause poor load balance, whereas smaller values result in more irregular partitions and more expensive partitioning and redistribution time.

After quad-nodes have been created, subdivision is carried out using ORB-MM. ORB-MM splits the space into two parts each of which have approximately same amount of work and then recurses on each part in alternate dimensions. The mid-points of quad nodes are used to find the median-line. One problem with this scheme is how to assign quad-nodes which are intersected by the median-line. Taking centers of quad-nodes and assigning them according to their centers may cause load imbalance. The idea in ORB-MM is once bisection has been determined, the border nodes that straddle the median-line are identified and repartitioned . In this phase, we sort the border nodes in bisection direction and assign nodes to the one side until it has half of the total cost. Then, we assign remaining nodes to the other side. The ORB-MM may generate screen regions that are not rectangular. However, generated regions are restricted to be composed of adjacent mesh cells.

2.3 Graph Partitioning Based Subdivision (GS)

This partitioning scheme models the spatial subdivision as a graph partitioning problem. Each cell in the mesh is assumed to be connected to its north, south, west and east neighbors. The vertices of the graph are the mesh cells and conceptual connections between mesh cells form the edges of the graph. The weight of a cell represents the number of primitives intersecting this cell. The weight of the edge between two cells represents the number of primitives crossing the boundary between these two cells. The objective in graph partitioning is to minimize the cutsize among the parts while maintaining the balance among the part sizes. Here, cutsize refers to the weighted summation of cut edges which connect more than one part. The size of a part refers to the weighted summation of the vertices in that part. In our case, balanced partitioning corresponds to maintaining computational load balance during rendering. Minimizing cutsize corresponds to minimizing the number of shared primitives. A state-of-the-art graph partitioning tool, Metis [6], is used in this work.

In this subdivision scheme, each processor tallies local primitives and updates corresponding cell and edge weights. Cell weights are updated in the same way as in MAHD. Edge weight update scheme will be described in the next paragraphs. Each local graph is globally merged to obtain the graph representing the distribution of all primitives in the scene. Then, each processor has global graph to be used in the partitioning. The mesh representation of the graph is converted into the representation used by Metis. In the original mesh representation, cell and edge weights are real numbers. These values are converted into integers since Metis operates on integer vertex and edge weights.

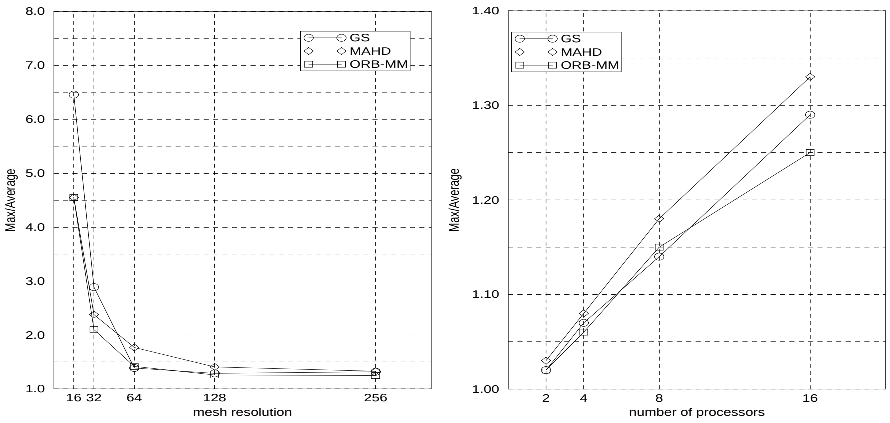
Metis uses multilevel partitioning approach consisting of three phases: coarsening, initial partitioning, and refinement. In the coarsening phase, the graph is coarsened down level-by-level to decrease the number of vertices by combining vertices to form new vertices. The coarsest graph is partitioned in the initial partitioning phase and this partitioning is refined in the refinement phase.

In the coarsening phase, various matching schemes can be used in Metis to combine appropriate vertices. The matching scheme used in this paper is called *heavy edge matching*. In heavy edge matching scheme, at each level of coarsening, an unmatched vertex is combined with another unmatched vertex such that the weight on the edge between two vertices is maximum. When two vertices are matched and combined to form a new vertex, which is used in the next level of coarsening, the weight of a new vertex is the sum of the weights of the cells forming this new vertex. The weight of the edge between two vertices on the same level is equal to the sum of weights of the edges between vertices forming these two new vertices. Since edge weights are directly added, the weight of the edge between two mesh cells should be updated appropriately during the tallying phase to reflect the number of shared primitives between vertices in the coarse graph. In order to decrease errors caused by primitives shared between more than two cells, we adopt the following scheme to update edge weights. First, we classify shared primitives into three categories: vertical primitives, horizontal primitives, and general primitives. Vertical primitives are the ones that cross only the cells in a single column. Similarly, horizontal primitives cross only the cells in a single row. General primitives cross cells in different rows and columns. The weight of the edge between two cells is incremented by a value proportional to the number of vertical or horizontal primitives crossing those two cells. On the other hand, the weight of the edge between two cells is incremented by a value inversely proportional to the number cells a primitive crosses for general primitives. In this way, we try to minimize the errors incurred on the edge weight between two vertices formed by cells in neighboring rows or columns.

The graph partitioning approach subdivides the screen in the most general way. Unlike previous subdivision heuristics, noncontiguous sets of cells may be assigned to a processor. In addition, generated regions may not be rectangular regions.

3 Experimental Results

The algorithms presented in this paper were implemented on a Parsytec's CC system using C language and PVM 3.3 [5, 12] for message passing. Each processing node of the CC system has 32 Mbytes of memory, each I/O node has 64 Mbytes of memory and each node has 133 MHz PowerPC 604 processor. We present experimental results for 2, 4, 8 and 16 processors. Experiments were done using two data sets, referred to as D1 and D2 in this paper. These data sets were generated from *blunt fin* and *post* data sets used in volume rendering applications. All results presented in this paper are the averages of results for two data sets obtained for three different viewing locations for each data set. On the average, the data set D1 contains approximately 243K triangles after pre-transformation and D2 contains approximately 453K triangles after pre-transformation. No actual rendering of data sets were performed since we do not target any specific rendering method, such as polygon rendering or volume rendering, in this paper. We used the number of primitives in each processor to measure load balancing performance and to measure percent increase in the total number of primitives after subdivision. The load balancing values were calculated as $Max/Average$. Here, Max is the maximum of the number of primitives in each processor after subdivision. $Average$ is the average number of primitives and is calculated by dividing the number of primitives in the scene before redistribution to the number of processors. In ORB-MM, limit value was taken as 1/20000 of total number of primitives. In GS, heavy edge matching was used as the matching algorithm, the number of vertices the graph should be coarsened down in the coarsening phase was taken as 100, the refinement algorithm used was Boundary Kernighan-Lin.



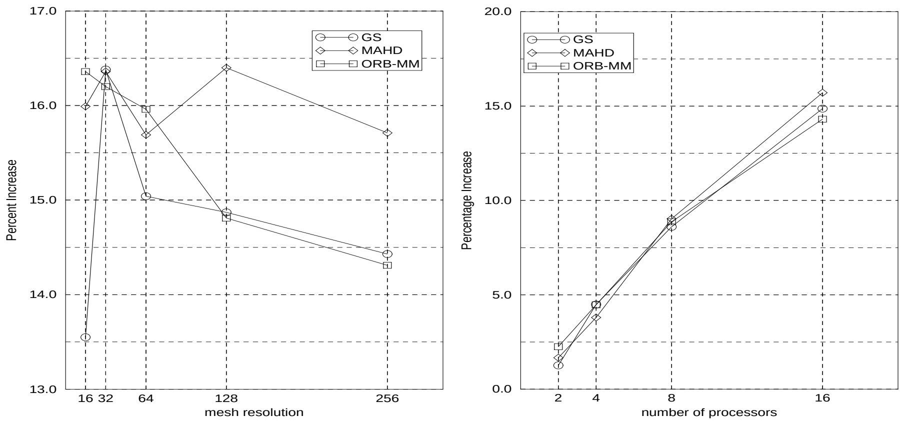
(a) (b)

Fig. 1. Load balancing (a) different mesh resolutions on 16 processors (b) different number of processors.

Figure 1 illustrates the behavior of algorithms for load balancing. As is seen in Fig. 1(a), as the mesh size increases load balancing performance of MAHD and ORB-MM improves. We observed slightly different behavior for GS. It produces best load balance on mesh of resolution 128×128. Note that both MAHD and ORB-MM uses primitive distribution only for load balancing. They try to minimize number of shared primitives implicitly. Graph-based subdivision, on the other hand, uses primitive information for both load balancing and to minimize number of shared primitives. Minimizing number of shared primitives and achieving even work load distribution can be conflicting for some cases. As is seen in Fig. 1(b), load imbalance increases with increasing number of processors. This is expected since the subdivision is done on a discrete space.

We observed that both GS and ORB-MM achieve better load balance than MAHD. MAHD divides the screen into rectangular regions, thus restricting the search space of the algorithm. In addition, a straight division line is likely to produce more load imbalance at a division step. Thus, the load imbalance in MAHD propagates and increases more in further division steps than other algorithms. As medians-of-medians scheme allows ORB-MM to divide the screen in a more balanced way at a division step, the increase in the load imbalance at further division steps is less. The load balance performances of ORB-MM and GS are similar until 8 processors. ORB-MM achieves better load distribution on 16 processors. This is because of the fact that the errors incurred due to inverse area heuristic is less with larger quad nodes. Moreover, the cell and edge weights should be converted to integers to be used by Metis. Thus, additional approximations are introduced in GS scheme.

Figure 2 illustrates the percent increase in the total number of primitives in the system after redistribution. We do not observe a regular pattern in this case when mesh resolution is changed as seen in Fig. 2(a). As is seen in Fig. 2(b), the number of shared primitives increases with increasing number of processors as is expected. The values in the graph are given for the mesh resolution an algorithm achieves the best load balance. Increase in the number of processors increases the



(a) (b)

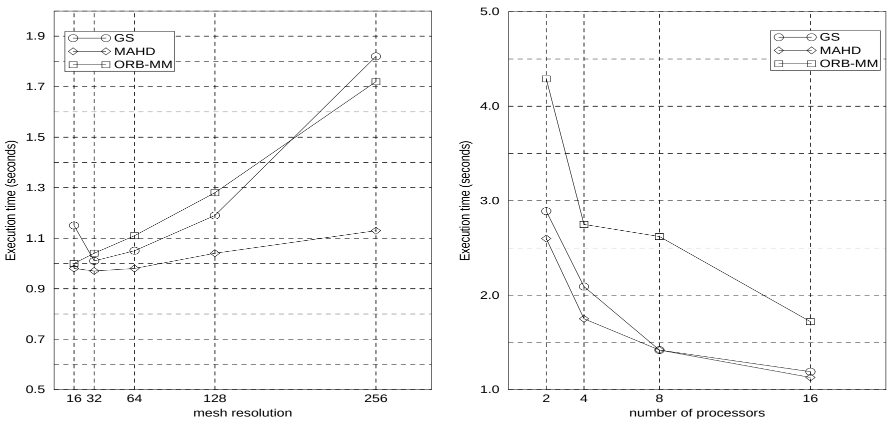
Fig. 2. Percent increase in the number of total primitives after redistribution (a) different mesh resolutions on 16 processors (b) different number of processors

possibility of more primitives crossing multiple regions as expected. We observe that minimum number of shared primitives on 16 processors is achieved by ORB-MM algorithm. The GS algorithm was expected to give better results than other two algorithms. However, we hypothesize that approximations due to inverse area heuristic and conversion into integer values affect the performance of the GS algorithm.

Figure 3 represents the execution time of algorithms. The execution time of each algorithm is the sum of the time required for pre-transformation step, execution time of the division, and redistribution time. Execution times of algorithms increase with increasing mesh resolution as expected (Fig. 3(a)). MAHD is the fastest among the three algorithms. MAHD involves simpler data structures and simpler operations for subdivision. In addition, more volume of communication incurs in GS and ORB-MM during combination of local meshes to form global mesh structure since they involve more complex data structures. In addition, GS has an extra overhead of converting mesh graph to the graph representation of Metis. For these reasons, execution times of GS and ORB-MM are more sensitive to mesh resolution than that of MAHD. Figure 3(b) illustrates change in the execution time when processor number changes. In this figure, each graph represents the variation of the execution time of the respective algorithm for the mesh resolution it finds the best load balance. Execution time of each algorithm decreases as the number of processors increases. However, it starts saturating after 8 processors. This is because of the fact that while percentage of time required for pre-transformation and division in overall execution time decreases as number of processors increase, percentage of redistribution time increases and becomes dominant as is seen in Table 1.

4 Summary and Conclusions

In this paper, we presented and experimentally investigated three adaptive spatial subdivision algorithms for sort-first rendering on distributed-memory multicomputers. The common characteristic of these algorithms is that a 2-dimensional



(a) (b)

Fig. 3. Execution time (in seconds) of subdivision algorithms (a) different mesh resolutions on 16 processors (b) different number of processors.

Table 1. Dissection of execution time of each algorithm for different number of processors.

	GS			MAHD			ORB-MM		
P	box	division	redist.	box	division	redist.	box	division	redist
2	1.43	0.51	0.95	1.43	0.42	0.75	1.43	1.89	0.97
4	0.75	0.54	0.80	0.75	0.38	0.62	0.75	0.94	1.06
8	0.39	0.38	0.65	0.39	0.33	0.70	0.39	1.39	0.84
16	0.21	0.37	0.61	0.21	0.29	0.63	0.21	0.83	0.68

mesh is superimposed on the screen and screen is subdivided using mesh structure and the primitive distribution on this mesh.

Our preliminary results show that ORB-MM algorithm achieves the best load balance and results in minimum number of shared primitives. Note that ORB-MM divides the screen in a more general way than MAHD. Furthermore, ORB-MM operates on quadnodes rather than mesh cells and there are less number of larger quadnodes than mesh cells. Thus, the errors incurred in inverse area heuristic due to shared primitives on quadnodes can be expected to be less. Although GS divides the screen in the most general way, errors because of shared primitives and due to conversion into integer values degrades the performance of the algorithm.

MAHD is the fastest of the three algorithms since it involves simpler data structures and simpler operations for subdivision. In addition, more volume of communication incurs in GS and ORB-MM during combination of local meshes to form global mesh structure since they involve more complex data structures. Moreover, GS has an extra overhead of converting mesh graph to the graph representation of Metis.

Based on our experimental results, we can conclude that MAHD can be used if the cost of processing one primitive is relatively small since MAHD executes faster. On the other hand, if processing cost per primitive is large and memory space is scarce, GS or ORB-MM should be chosen since they produce better load balance and less number of shared primitives.

References

1. M. Cox and P. Hanrahan. Pixel merging for object-parallel rendering: A distributed snooping algorithm. In *Proceedings of the 1993 Parallel Rendering Symposium, IEEE Computer Society Press*, 49–56 (October 1993).
2. T. W. Crockett and T. Orloff. A MIMD rendering algorithm for distributed memory architectures. In *Proceedings of the 1993 Parallel Rendering Symposium, IEEE Computer Society Press*, 35–42 (October 1993).
3. F. C. Crow. Summed-area tables for texture mapping. *Computer Graphics*, **18**(3), 207–212 (1984).
4. D. Ellsworth. A multicomputer polygon rendering algorithm for interactive applications. In *Proceedings of the 1993 Parallel Rendering Symposium, IEEE Computer Society Press*, 43–48 (October 1993).
5. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing, The MIT press*, 1994.
6. G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system, Version 2.0. Dept. of Computer Science, University of Minnesota. `http://www.cs.umn.edu/~karypis`
7. T. M. Kurç, C. Aykanat, and B. Özgüç. Active pixel merging on hypercube multicomputers. In Proc. of *The International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '96)*, 319–326 (1996).
8. S. Molnar, J. Eyles, J. Poulton. PixelFlow: high-speed rendering using image composition. *Computer Graphics*, **26**(2), 231–240 (1992).
9. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics & Applications*, **14**(4), 23–32 (July 1994).
10. C. Mueller, The sort-first rendering architecture for high-performance graphics. In *Proceedings of 1995 Symposium on Interactive 3D Graphics*, 75–84 (1995).
11. J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Trans. on Parallel and Distributed Systems*, **7**(3), 288–299 (March 1996).
12. PowerPVM/EPX for Parsytec CC systems: PowerPVM/EPX User's Guide, *Genias Software GmbH.*
13. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*, Springer-Verlag (1985).
14. R. Scopigno, A. Paoluzzi, S. Guerrini, and G. Rumolo. Parallel depth-merge: A paradigm for hidden surface removal. *Computers & Graphics*, **17**(5), 583–592 (1993).
15. J. P. Singh, C. Holt, J. L. Hennessy, and A. Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing 93*, 54–65 (1993).
16. E. Tanin, T. M. Kurç, C. Aykanat, and B. Özgüç. Comparison of two image-space subdivision algorithms for direct volume rendering on distributed-memory multicomputers. In Proc. of *Workshop on Applied Parallel Computing in Physics, Chemistry, and Engineering Science (PARA95)*, 503–512 (August 1995).
17. S. Whitman. Dynamic Load Balancing for Parallel Polygon Rendering. *IEEE Computer Graphics & Applications*, **14**(4), 41–48 (July 1994).

This article was processed using the LaTeX macro package with LLNCS style