Image-Space-Parallel Direct Volume Rendering on a Cluster of PCs*

B. Barla Cambazoglu and Cevdet Aykanat

Bilkent University, Department of Computer Engineering,
06800, Ankara, Turkey
{berkant,aykanat}@cs.bilkent.edu.tr

Abstract. An image-space-parallel, ray-casting-based direct volume rendering algorithm is developed for rendering of unstructured data grids on distributed-memory parallel architectures. For efficiency in screen workload calculations, a graph-partitioning-based tetrahedral cell clustering technique is used. The main contribution of the work is at the proposed model, which formulates the screen partitioning problem as a hypergraph partitioning problem. It is experimentally verified on a PC cluster that, compared to the previously suggested jagged partitioning approach, the proposed approach results in both better load balancing in local rendering and less communication overhead in data migration phases.

1 Introduction

In many scientific simulations, data is located at the vertices (data points) of a 3D grid that represents the physical environment. Unstructured datasets are a special case of grid-based volumetric datasets in which the data points are irregularly distributed and there is no explicit connectivity information. Direct volume rendering (DVR) is a popular volume visualization technique [1] employed in exploration and analysis of such 3D grids used by scientific simulations.

The main aim in DVR is to map a set of data values defined throughout a 3D volumetric grid to some color values over a 2D image on the screen. DVR algorithms are able to produce high-quality images, but due to the excessive amount of sampling and composition operations performed, they suffer from a considerable speed limitation. Furthermore, memory requirements for some recent datasets are beyond the capacities of today's conventional computers.

These facts bring parallelization of DVR algorithms into consideration [9]. Image-space-parallel (IS-parallel) [5,8] or object-space-parallel (OS-parallel) [6,7] methods can be used for distributed-memory parallelization. OS-parallel methods decompose the 3D data into subvolumes and distribute these subvolumes to processors. Each processor works only on its assigned subvolume and produces a partial image using its local data. IS-parallel methods try to decompose the screen and assign subscreens to processors. Each processor computes only

* This work is partially supported by The Scientific and Technical Research Council of Turkey (TÜBİTAK) under project EEEAG-199E013.

a small but complete portion of the whole image on the screen. In IS-parallel DVR, screen decomposition and subscreen-to-processor mapping is important for successful load balancing and minimization of the data migration overhead.

This work investigates IS-parallel DVR of unstructured grids on distributed-memory architectures, and focuses on load balancing and minimization of data migration overhead. In the following section, we start with a brief description of the underlying sequential DVR algorithm. In Section 3, we present the details of our parallel algorithm together with a description of the proposed screen partitioning model. In Section 4, we present some experimental results that verify the validity of the proposed work. We give our conclusions in Section 5.

2 Sequential Algorithm

The underlying sequential algorithm used in our IS-parallel DVR algorithm is based on Koyamada's ray casting algorithm [4]. The algorithm starts with scan-converting all front-facing external faces. From each pixel found, a ray is shot into the volume and is followed until it exits from a back-facing external face, forming a ray segment (Fig. 1). Along a ray segment, some sampling points are determined. The number and location of the sampling points depend on the sampling technique used. In this work, mid-point sampling technique is used.

At each sampling point, new sampling values are computed by interpolating the actual data values on nearby data points. Passing the sampling values from appropriate transfer functions, RGB color triples and opacity values are obtained. The color and opacity values found along a ray segment are composited in visibility order, using a weighting formula that assigns higher weights to values on points closer to the screen. Consequently, for each ray segment, a final color is generated. The generated color is put in the related pixel buffer to which the ray segment contributes. Due to the concavity of the volume, there may be more than one ray segments generated for the same pixel, and hence more than one color values may be stored in the same pixel buffer (two in the example of Fig. 1). After all ray segments are traced, the colors in pixel buffers are composited in visibility order and the final colors over the screen are generated.

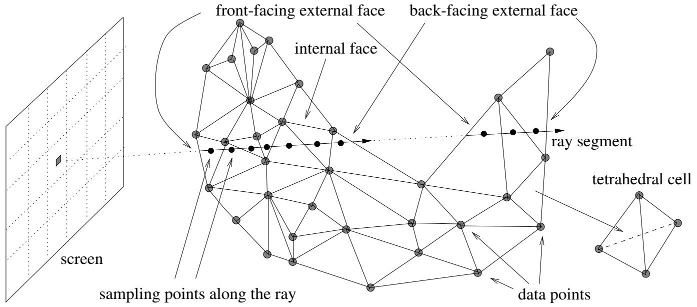

front-facing external face
back-facing external face
internal face
ray segment
tetrahedral cell
screen
sampling points along the ray
data points

Fig. 1. Ray-casting-based rendering of unstructured grids with mid-point sampling

3 IS-Parallel DVR Algorithm

3.1 View-Independent Preprocessing

The view-independent preprocessing phase, which is performed once at the beginning of the whole visualization process (Fig. 2), performs the operations that are independent of the current visualization parameters. These include retrieval of the data from disk, cell clustering and the initial data distribution. Since the data to be visualized is usually produced by simulations performed on the same parallel machine, it is assumed that the data is already partitioned and stored in the local disks of the processors. Hence, processors can read a subset of cells in parallel.

After the data is read, processors concurrently perform a top-down clustering on their local cells. For this purpose, a graph-partitioning-based clustering approach is followed, where the cells and faces respectively form the vertices and edges of the graph representing the unstructured grid. In the implementation, state-of-the-art graph partitioning tool MeTiS [3] is used to partition this graph and create the cell clusters (subvolumes). Since clustering is performed just once, at the beginning of the whole visualization process, independent of the changing visualization parameters, the cost of cell clustering is almost negligible.

The main idea behind cell clustering is forming groups of close tetrahedral cells and obtaining some volumetric chunks within the data such that the total surface area to be scan-converted is smaller. This way, the computational cost of screen workload calculations during the view-dependent partitioning phase is reduced. Furthermore, since the parallel algorithm works on cell clusters instead of individual tetrahedral cells, the housekeeping work is simplified, number of iterations in some loops are reduced, and some data structures are shortened.

After cell clustering, an initial cluster-to-processor mapping is found. All the following view-dependent preprocessing phases utilize this initial cell cluster mapping. Even if a cell cluster may be temporarily replicated in other processors during the local rendering, it is statically owned by only a single processor. This static owner keeps the cluster's data structures throughout the whole visualization process, and is responsible from sending them to the other processors.

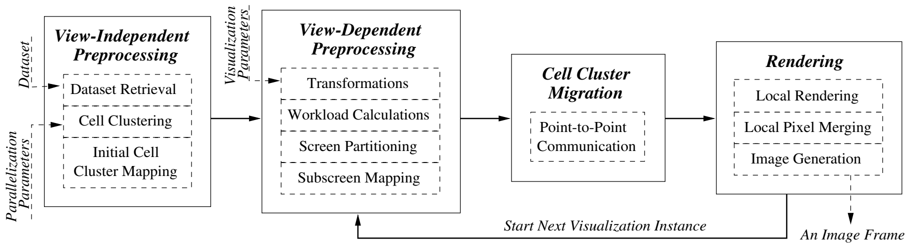


Fig. 2. The proposed IS-parallel DVR algorithm

3.2 View-Dependent Preprocessing

In IS-parallel DVR, the changing visualization parameters at each visualization instance require repartitioning of the screen and assignment of subscreens to processors for a load-balanced rendering. In order to decompose the screen, the rendering-load distribution over the screen pixels should be known beforehand. For this purpose, approximate rendering load of a cell cluster is estimated by summing the areas of its front-facing internal faces. This estimated rendering load for the cell cluster is evenly distributed over the pixels under the projection area of the cell cluster. After this process is repeated for all cell clusters, it becomes possible to know approximately how many samples will be taken along a ray fired from a specific pixel.

Once the workload distribution over the screen is computed, the screen is decomposed into subscreens such that the total load of the pixels in subscreens will be almost equal. Here, the number of subscreens is chosen to be equal to the number of processors, so that each processor is assigned the task of rendering one of the subscreens. In our implementation, for efficiency purposes, an n by n coarse mesh, which forms n^2 screen cells, is imposed over the screen. In this scheme, an individual screen cell corresponds to an atomic task which is assigned to a processor and is completely rendered by that processor. The set of screen cells assigned to the same processor forms a subscreen for the processor. A subscreen is not necessarily composed of geometrically connected screen cells.

We model the the screen partitioning problem as a hypergraph partitioning problem. A hypergraph can be considered as a generalization of a graph, where each hyperedge can connect more than two vertices. In this model, the vertices of the hypergraph correspond to screen cells, that is, a vertex represents the atomic task of rendering a respective screen cell. Each vertex is associated with a weight, which is equal to the total rendering load of the pixels contained in the respective screen cell. Hyperedges of the hypergraph represent the cell clusters. Each hyperedge connects the screen cells which intersect the projection area of the corresponding cell cluster. Each hyperedge is associated with a weight, which is equal to the number of bytes needed in order to store the corresponding cell cluster. By partitioning the hypergraph into equally-weighted parts (subscreens), the model tries to minimize the total replication amount while maintaining a load balance in rendering of subscreens. In the implementation, our hypergraph partitioning tool PaToH [2] is used for partitioning the hypergraph.

Moreover, we formulate the subscreen-to-processor mapping problem as a maximal-weighted bipartite graph matching problem. The K processors in the system and the K subscreens produced by hypergraph partitioning form the partite nodes of a bipartite graph. An edge connects a processor vertex and a subscreen vertex if and only if the processor stores at least one cell cluster of which projection area intersects with the subscreen. Each edge is associated with a weight, which is equal to the sum of migration costs of such cell clusters. By applying maximal-weighted bipartite graph matching algorithm to this graph, the total volume of communication during cell cluster migration is minimized.

3.3 Cell Cluster Migration

After partitioning the screen and mapping subscreens to processors, the processors at which the cell clusters will be replicated are determined. Then, the cell clusters are migrated from their home processors to the new processors through point-to-point communication, according to the replication topology found.

3.4 Rendering

Once the cell cluster migration is complete, processors are ready to render their assigned subscreens in parallel. Most parts of the local rendering is similar to that of the sequential algorithm. The rays are followed throughout the data utilizing the connectivity information, which keeps the neighbor cluster, cell and face ids for the four faces in each cell.

Although it is possible to have non-convex cell clusters as a result of the clustering algorithm, this does not cause an increase in the number of ray segments created. Existence of such non-convexities is eliminated due to replication of the cell clusters. However, the non-convexities in the nature of the dataset still require the use of ray buffers. Since, after the local rendering, all processors have a subimage, an all-to-one communication operation is performed and the whole image frame is generated in one of the processors.

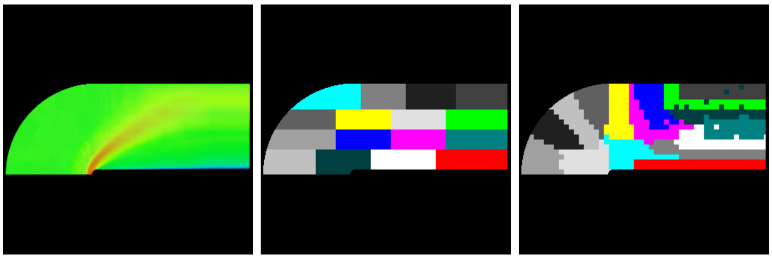
4 Experiments

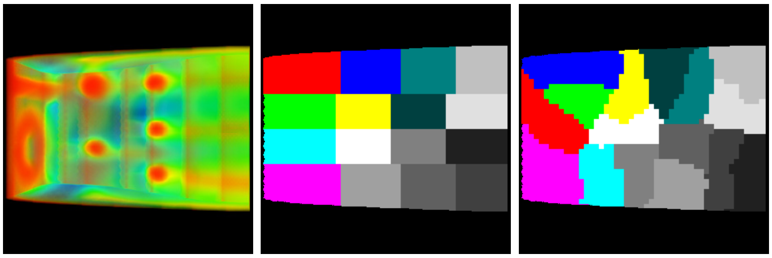
As the parallel rendering platform, a 24-node PC cluster is used. The processing nodes of the cluster are equipped with 64 Mb of RAM, and are interconnected by a Fast Ethernet switch. Each node contains an Intel Pentium II 400 Mhz processor and runs Debian GNU Linux operating system. The DVR algorithm is developed in C, using MPI as the communication interface.

Each experiment is repeated over three different datasets using five different viewing parameter sets and the average values are reported. In the experiments, each processor is assigned 10 cell clusters, which is an empirically found number. As mentioned before, to make the view-dependent preprocessing overhead affordable, coarse meshes of varying size are imposed over the screen. In all experiments, the proposed hypergraph-partitioning-based (HP-based) model is compared with the previously suggested jagged-partitioning-based (JP-based) model. The details of the JP-based model can be found in [5].

Experiments are conducted over *Blunt Fin*, *Combustion Chamber* and *Oxygen Post* datasets obtained from NASA Research Center. Fig. 3 displays our renderings and the subscreen-to-processor mapping for different models. In each figure, the first image represents the actual rendering obtained using the standard viewing directions. The second and third images show the screen partitioning (by associating each subscreen with a different color) produced by the JP-based and HP-based models, respectively.

The experiments which verify the solution quality are conducted at high number of virtual processors by allocating more than one executable instance per node. Fig. 4 displays the total volume of communication with the varying number


a) The Blunt Fin dataset


b) The Combustion Chamber dataset

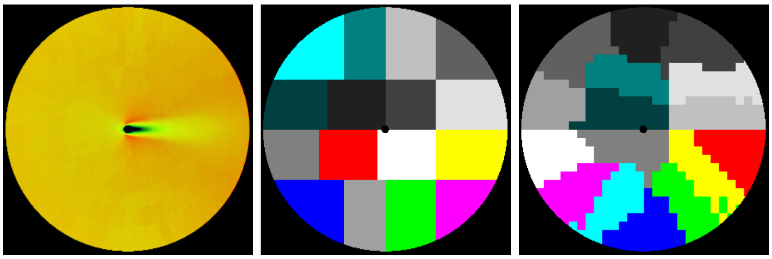

c) The Oxygen Post dataset

Fig. 3. Example renderings and subscreen-to-processor assignments of the datasets

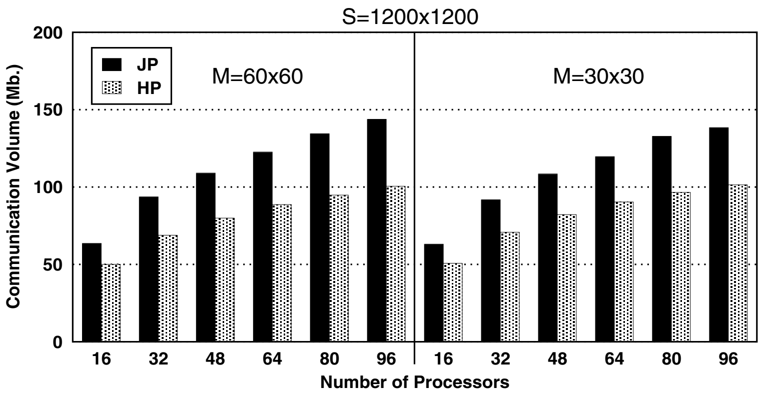


Fig. 4. Total communication volume with the changing number of processors

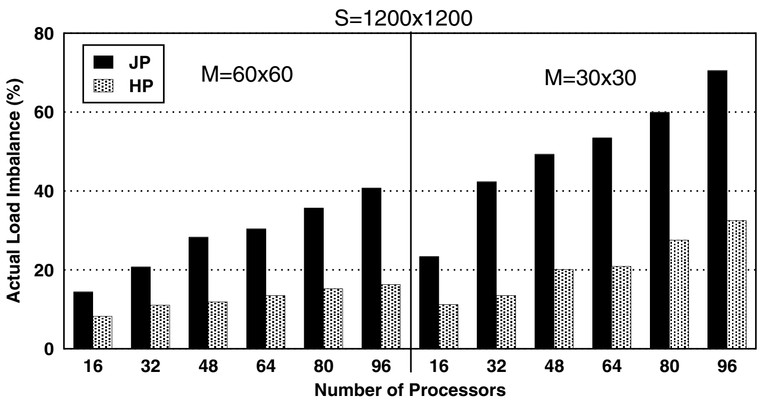


Fig. 5. Load imbalance in local rendering with the changing number of processors

of processors. With a coarse mesh resolution of 60 × 60, using 96 processors, HP-based model results in around %30 less communication overhead than the JP-based model. When coarse mesh resolution is decreased to 30 × 30, there occurs a slight decrease in communication volume. This can be explained with the decrease in lengths of subscreen boundaries, and hence the decrease in the amount of overlaps between cell clusters and subscreen boundaries.

Fig. 5 gives the load imbalance rates in local rendering phase. With a coarse mesh resolution of 60×60, using 96 processors, JP-based model results in a load imbalance of 40.7%. With the same parameters, load imbalance values for the HP-based model is 16.3%. When coarse mesh resolution is decreased to 30×30, both models perform worse in load balancing. This is due to the decrease in the number of screen cells, and hence the reduction in solution space.

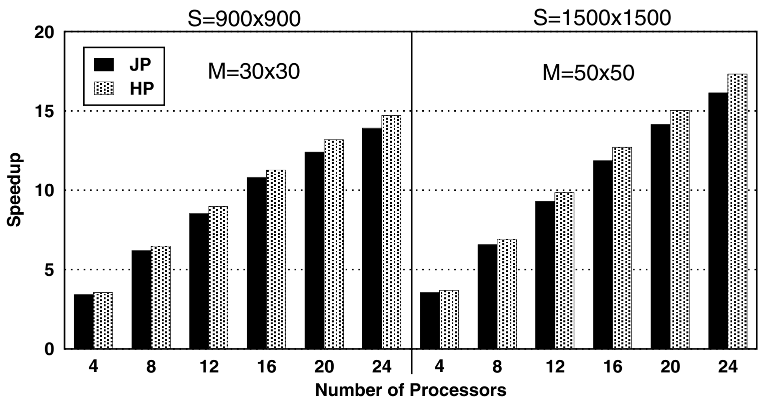


Fig. 6. Speedups at different screen and mesh resolutions

At 900×900 screen resolution, with 24 processors, our parallel algorithm is capable of producing around 12 image frames per minute. Fig. 6 displays the speedups achieved by the two models with the varying number of processors. At 24 processors, with a screen resolution of 1500×1500 and a coarse mesh resolution of 50×50, speedups are 16.1 and 17.3 for the JP-based and HP-based models, respectively. It is observed that the increasing screen resolution and number of processors favor the proposed HP-based model.

5 Conclusions

Compared to the jagged partitioning (JP) model, the proposed hypergraph partitioning (HP) model results in both better load balancing in the rendering phase and less total volume of communication during the data migration phase. The experiments conducted at the available number of processors indicate that HP-based model yields superior speedup values than the JP-based model. We should note that, HP step, which is carried out sequentially on each processor, is the limiting factor on the speedup values. A parallel HP tool, which will probably be implemented in the future, can eliminate this current drawback of our implementation. A nice aspect is that as the new heuristics are developed for HP and the existing HP tools are improved, the solution quality of the proposed model will also improve.

References

1. T. T. Elvins, A survey of algorithms for volume visualization, Computer Graphics (ACM Siggraph Quarterly), 26(3) (1992) pp. 194–201.
2. Ü. V. Çatalyürek and C. Aykanat, PaToH: Partitioning tool for hypergraphs, Technical Report, Department of Computer Engineering, Bilkent University, 1999.
3. G. Karypis and V. Kumar, MeTiS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices, Technical Report, University of Minnesota, 1998.
4. K. Koyamada, Fast traversal of irregular volumes, in: T. L. Kunii (Ed.), Visual Computing, Integrating Computer Graphics with Computer Vision, Springer-Verlag New York, 1992, pp. 295–312.
5. H. Kutluca, T. M. Kurc, and C. Aykanat, Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids, Journal of Supercomputing, 15(1) (2000) pp. 51–93.
6. K.-L. Ma, Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures, in: Proceedings of the IEEE/ACM Parallel Rendering Symposium '95, 1995, pp. 23–30.
7. K.-L. Ma and T. W. Crockett, A scalable cell-projection volume rendering algorithm for unstructured data, in: Proceedings of the IEEE/ACM Parallel Rendering Symposium '97, 1997, pp. 95–104.
8. M. E. Palmer and S. Taylor, Rotation invariant partitioning for concurrent scientific visualization, in: Parallel Computational Fluid Dynamics '94, 1994.
9. C. M. Wittenbrink, Survey of parallel volume rendering algorithms, in: Proceedings of the PDPTA'98 Parallel and Distributed Processing Techniques and Applications, July 1998, pp. 1329–1336.