Clustering for Parallel Volume Visualization

Cemal Köse
Department of Computer Engineering,
Black Sea Technical University,
61080 Trabzon,
TURKEY.
kose@osf02.ktu.edu.tr

Abstract

Volume visualization is a powerful engineering tool. However, the visualization of a three dimensional volume is computationally expensive taking significant amounts of time to produce the images on conventional computers. Parallel processing offers the possibility of rendering the volume in acceptable times. This paper discusses hierarchical and distributed clusteral models with dynamic cluster re-sizing and caching which are used in combination with dynamic task and data management strategies to provide an efficient parallel implementation for volume visualization on a large distributed memory multiprocessor system.

Keywords: Parallel, Volume Visualization, Data management, Task Management, Load balancing, Clustering

1. INTRODUCTION

Volume visualization enables users to “peer” into complex three dimensional volume data sets and extract meaningful information as to their structure and complexity [3,5]. By manipulating a view point the user can examine the volume from any direction. Such a tool is increasingly important as volume data, for example, medical data from CAT or MRI scanners, is used more frequently.

One straightforward method of representing volumetric data sets is by a three-dimensional regular grid of volume elements (known as voxels). As the view point is moved, volume rendering techniques are used to produce an image of the volume from each new viewing position. The computational effort required to render a single image of a complex volume is significant and may take many minutes, even hours, to render on a conventional machine. Parallel processing offers the potential of significantly reducing this rendering time. However, volume data sets exhibit certain characteristics which complicate their visualization on multiprocessor systems:

- The volume data is typically very large - far larger than can be accommodated on a single processor. Thus, a parallel implementation of a volume rendering algorithm must be able to cope with data sets which are distributed amongst many processors. The correct distribution of the data, and the minimization of the communication latency associated with a remote data fetch, are fundamental to any efficient parallel solution to volume visualization [6,12,16].

- In a parallel implementation of volume visualization, the tasks required to render one image may be quite different from that necessary to produce the image from a different view point, and may vary in computational complexity. Therefore, efficient load balancing schemes will be necessary to maximize overall system performance.

If anything approaching an interactive visualization system is to be achieved on a distributed memory multiprocessor system then the issues of data and task management and communication must be effectively addressed. This paper discusses a number of strategies and shows how coherence in task distribution requirements can be exploited to improve significantly the overall performance of the parallel solution.

2. RAY CASTING

A number of techniques have been developed for volume rendering. The “splatting method”, for example, although frequently used for sequential implementations [17], require large overlapping of data portion of each processing elements for large splat sizes which is important for a good quality picture. For an efficient volume visualization with splatting, a large amount of data redistribution is necessary to achieve even load balancing across the parallel system. Ray casting, on the other hand, is a simple technique, well suited for parallel processing [3,7,9,14].

An early ray casting model for volume rendering was presented in [9]. The renderer casts a group of rays from the view point through the image plane to the volume data. Each ray now travels through the volume data. The renderer interpolates this data to generate new sample points at the intersection points along the path of the ray. The path terminates when the volume data is exhausted or the accumulated opacity along the ray equals one. The early termination of the process allows optimization of the ray casting method. Opacity and intensity are accumulated along the rays during these processes. Finally, the volume is shaded according to light transmission and reflection.

The volume data is usually sampled at a regular interval by rays sent into the volume data from the view point passing through each pixel in the image plane, as shown in Figure 1. An interpolation function is used to reconstruct the object from the discrete values at the image space. In a sequential implementation for rendering the discrete volume data, the process proceeds pixel by pixel.

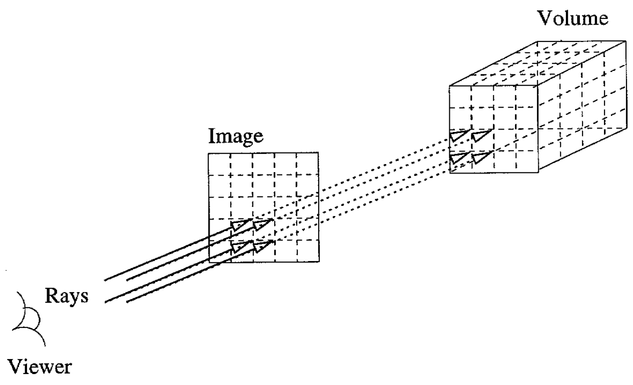

Figure 1: Ray casting

The rendering algorithm traces the rays through the voxels until they hit a surface and then assigns an intensity inversely proportional to the distance from the eye. The radiation transfer equation with single scattering approximations is used to simulate transmission of light through the volume and model reflectance from the layered volume. Opacity and inverse transparency are defined as scalar functions and evaluated at the nearest face of each cell along the ray's path. This path is stepped along until the entire cell has been traversed with evaluations of the scalar field, shading function, opacity, and texture mapping [9].

3. PARALLEL IMPLEMENTATION

Recently, many parallel algorithms for volume rendering have been developed, for example [11,12]. Early parallel approaches targeted volume rendering directly on specialized, and thus expensive, hardware. Here, we consider parallel volume visualization on a general purpose MIMD system; a network of transputers.

A single computational element of a parallel rendering algorithm may be chosen as the calculation of the local color and opacity contribution of an intersection of a voxel of the volume data with a ray cast through a pixel of the image plane. Parallel volume rendering may now be classified as either *image partitioning* or *volume partitioning* depending on how these computational elements are combined as tasks in the parallel implementation [13]. Figure 2 shows the difference between these two approaches in two dimensions. In this figure we assume there are three processing elements and that a third of the volume data is accommodated at each processing element.

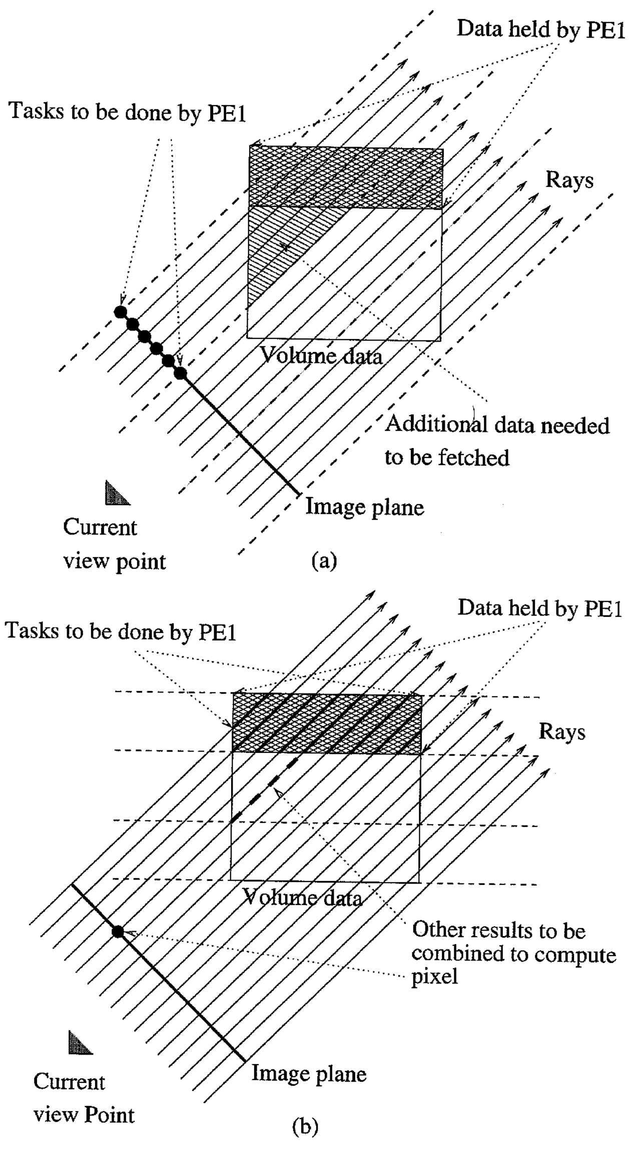

Figure 2: Division of data and tasks (a) Image partitioning (b) Volume partitioning

- Image-partition techniques initially partition the image plane evenly amongst the processors. Each processor calculates the pixel values for its image portion. The work load at each processor is proportional to the number of scan lines of the image plane to be computed. As can be seen in Figure 2 (a), with large distributed volume data sets, image partitioning may require a processing element to fetch data items from other processing elements in order to complete its tasks.

To ensure an even load balance it must be possible to migrate some tasks from those processing elements allocated complex tasks to those whose initial allocation contained scan lines which were computationally easier to compute. As each processing element is responsible for the rendering tasks of its region, there is no need for an additional combination of partial results to produce the final image.

- The volume-partition method performs the reconstruction and re-sampling tasks with the portion of the volume data held at each processing element. Because there is no world model, processing elements may only compute partial results of the tasks from their allocated portion of the volume data. In order to rendering the final image, it is necessary, therefore, to combine the partial results computed by several processing elements, as shown for a single pixel in Figure 2 (b).

  The advantage of this method is, of course, there is no need for a processing element to fetch potentially large amounts of volume data from other processing elements.

In this paper we have used volume partitioning for our parallel implementation in order to reduce the need to communicate data across the system. To exploit the task and data coherence that such an approach offers, clusteral models with dynamic cluster re-sizing and clusteral models with caching have been introduced.

3.1. Task Management

Volume partitioning requires each processing element to perform the local color and opacity calculations for the intersection of the rays with the volume data held at that processing element. This volume data is evenly distributed amongst the processing elements prior to the start of any visualization. This data remains *in situ* at the processing elements throughout the entire volume visualization, however, the nature of the tasks associated with this data will vary according to the selected view point.

Volume partitioning does have a disadvantage. This technique is unable to fully exploit the “early termination” optimization of ray casting. Early termination may occur if an opaque layer hides the rest of the volume from a cast ray or the opacity accumulation exceeds a certain level. A front-to-back opacity accumulation technique is able to determine this situation and thus stop any further computation of tasks on the path of the considered ray. Such early termination may save a substantial amount of computation, especially when considering high density objects [10]. With volume partitioning, processing elements can still take advantage of any early termination within the tasks they are considering, but the significance of this will vary according to the current view point.

Exploiting any early termination and the position of the view point means that tasks have variations in computational complexity. Such variations may result in significant load imbalances within the system unless a load balancing scheme is adopted [2,4,15]. Task management ensures that tasks may migrate from processing elements which are struggling with high complexity tasks to those which have finished all their less difficult tasks. Migration of tasks implies that the data associated with the tasks must also be fetched to the task receiver. Therefore, care must be taken to ensure that tasks migrate to processing elements which are physically “close” to the task's source in order to maintain the benefits of low communication overheads of volume partitioning.

For further improvement of the clusteral model for parallel volume visualization on a distributed memory parallel system, dynamic cluster re-sizing and caching strategies are introduced. Dynamic cluster re-sizing rearranges the task grain size for each sequential frame by re-sizing of each processing elements clusters' portion. Thus, efficiency of parallel volume visualization will be improved gradually. In the course of volume visualization, after a few frames, the system will reach an even load distribution. A second strategy uses caching technique so that a task migration policy ensures that the same task must migrate to the same processing elements. Such a caching strategy may reduce the amount of data redistribution and may improve system performance by making reuse of data previously fetched.

3.2. Data Management

Applications with data requirements which are such that the total number of data items is small enough to be accommodated at each processing element, termed a *world model*, may be solved without recourse to additional fetching of data items. Volumetric data sets are typically represented by a three-dimensional regular grid of voxels. The size of data for even a moderately complex volume data is substantial, precluding a world model. Thus, a parallel implementation of a volume rendering algorithm must be able to operate on data sets which are distributed amongst many processors. The correct distribution of the data, and the minimization of the communication latency associated with a remote data fetch, are thus fundamental to any efficient parallel solution to volume visualization [6,7,8].

Volume partitioning allows the distribution of the data amongst the processing elements to be determined in advance. The data associated with the tasks allocated to a processing element is known and available locally [7,8,18]. Even when tasks migrate to other processing elements in the course of load balancing, the data requirements for these tasks can specified in the migrating task packet. This enables the necessary data items to be prefetched from the task's source. Provided these known data item can be fetched sufficiently quickly, they should thus be available in the local memory of the processing element executing the task when required. The efficiency of the underlying communication system is fundamental to the rapid delivery of data requests and this can be increasingly effective with the correct choice of configuration [2].

3.3. Caching for Clusteral Models

In the data management strategy, the local memory at each processing element, which we term the *local cache*, assumes the röle of the cache memories of conventional processors. The purpose of the cache, which has an access time of up to ten times faster than main memory, is to store those portions of the main memory contents which are in current use by the processor. The use of a cache with careful design can improve the average access time of the memory considerably, and this is directly attributable to the property of locality of reference. Thus, the access time for a data manager to fetch data item from a remote “memory” location will be substantially higher than a fetch from its local cache. Coherence is used to ensure a high “cache-hit” ratio.

The spatial coherence in the problem domain and the temporal coherence together with the preferred biased task allocation provides the data manager with a good estimate of the future data requirements of the tasks being computed at a processing element. The data manager can now use this information to *prefetch* those data items which are *likely* to be used by subsequent tasks being performed at that processing element. If the data manager is always correct with its prediction, then a data item will always be available locally when required by the application process and thus the process is never delayed awaiting a remote fetch.

A cache normally consists of the *cache directory* and the *random access memory*. The cache is divided into a number of *block frames* of equal size which correspond to the blocks which make up the main memory. Information in the cache directory identifies the contents of the cache at any particular time. Two key design parameters characterize a cache memory: the *placement policy*, and the *replacement policy*. For cache management four basic placement policies have been used, namely *direct*, *fully-associative*, *set-associative* and *sector* mappings. Only the first three are suitable for data management in message passing systems.

Set-associative mapping represents a compromise between the simplicity of the direct mapping and the performance of the fully-associative mapping [6,7]. A simple mapping technique comparable with direct mapping is used to determine the set in which a data item may reside. The set must then be searched to test if the data item is present. Here, the set-associative organization attempts to provide the performance of full-associativity with the simplicity of a direct mapped cache, and has become the most common placement policy for memory management systems.

A replacement policy is necessary to determine which cache positions will be overwritten when the cache becomes full. In cache management systems, *Least Recently Used* (LRU) has been the most popular of the replacement policies. When a data item is referenced, it is marked as being the most recently used, and all the others are modified accordingly. Then when a write occurs the least recently used entry is selected for overwriting.

3.4. Clusteral Models

Initial task allocation is implicit with the volume partitioning approach, that is, the tasks to be done at a processing element are determined by the portion of the volume data that was assigned initially. As shown in Figure 2 (b), the partial results of these tasks will need to be combined in order to produce the desired pixel values for rendering the image. Here we discuss two clusteral models, hierarchical and distributed, which may be used to facilitate this combining process.

The need to combine the partial results may be solved by sending them all to the system controller where they can be combined prior to rendering. For a large number of processing elements, such an approach can cause a serious bottleneck at the system controller and lead to a significant degradation of overall system performance. A more efficient solution is to distribute at least some of the combining computations to the processing elements and have the system controller perform only a limited number of these operations.

A straightforward approach to distributed combining would be to divide the image plane into a number of conceptual regions. Specific processing elements can now be assigned the job of combining all the partial results for one such region. While such an approach will avoid the bottleneck at the system controller, this static allocation takes no account as to how the tasks may be distributed within the system. The tasks which constitute a particular ray are determined by the current viewpoint. In volume visualization, this viewpoint is constantly moving. Therefore, the partial results may need to be sent from the processing elements where they were calculated, to the physically remote processing element which is doing the combining.

The clusteral model exploits the coherence within task allocation to reduce "long distance" communication. A number of processing elements within the configuration are *conceptually* grouped together to form a cluster, as shown for the 16-processing element torus in Figure 3. All the partial results from the processing elements of a cluster are combined within this cluster.

A *hierarchical* cluster model allocates one processing element of the cluster to perform all the combinations within that cluster. Combining the partial results at this “central” combining processing element reduces communication within the cluster to a near-neighbor pattern. The additional computational effort required by this processing element to combine the partial results means that some of its own tasks may have to migrate to other processing elements in order to maintain an optimum load balance within the system. The implementation ensures that, where possible, a task will migrate to processing elements within the same cluster. This means that the movement of the data associated with the task only needs to be fetched from a neighboring processing element. In the event that load balancing requires that tasks should migrate outside a cluster then the system will allocate these tasks to idle processing elements as “close” as possible to where it originated. The partial results from these migrated tasks are returned to the appropriate cluster for combining.

A *distributed* clusteral approach attempts to reduce the need for load balancing by having all processing elements within each cluster perform an equal portion of the combining operations. Although this may reduce the need to migrate tasks and their associated data, such a distributed model does have the disadvantage that the partial results are no longer passed to a neighbor, the central processing element, but may now be required to be passed further. In configurations where there is no alternative route between processing elements within a cluster, sending the partial results to the appropriate processing element will place an additional communication burden on the central processing element.

3.5. Configuration

The performance of a distributed memory multiprocessor depends in large part on the efficiency of the message transfer system that provides the interface between the co-operating processing elements. To achieve the most efficient performance, the configuration chosen should be well suited to the communication patterns inherent in the parallel implementation [2]. The communication patterns implicit in the clusteral models suggest either a tree or a torus would be the most suitable configuration for our parallel implementation of volume visualization.

A tree of degree *d* and height *h* consists of a single processing element at the top level, the *root processing element*, connected to d other processing elements, each of which is a ”root” processing element of a subtree of degree *d* and height *(h-1)*. The processing elements at the lowest level of the tree, the *leaf processing elements*, are only connected to their “parent” tree. Any leaf processing element wishing to communicate with another leaf processing element must thus do so via branch processing elements further “up” the tree.
The hierarchical structure of a tree configuration makes it well suited to the clusteral approach. Each branch processing element combines the partial results from its immediate “children” and passes this result upwards. The system controller is situated at the root of the tree and performs any final combining of results before passing the newly computed pixel values to be rendered.

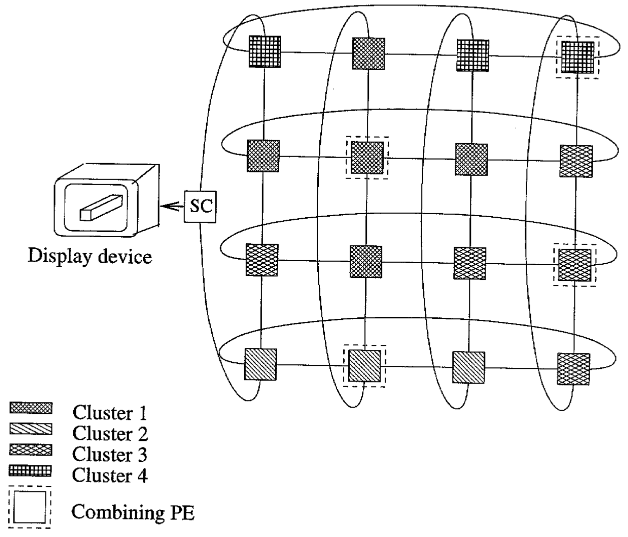

Figure 3: Clusters within the 16-processing element torus

A torus configuration consists of rings of rings of processing elements. To minimize the diameter of the torus it is preferable that the number of processing elements within the horizontal rings is approximately the same as the number of processing elements in the vertical rings. Figure 3 shows how the clusters may be defined within a 16-processing element torus. The system controller, labelled SC in the figure, once more performs any final combination of partial results prior to rendering.

		Processors				
		8	16	32	64	128
Diameter	Torus	3	4	6	7	12
	Ternary Tree	4	5	6	8	10
Average Distance	Torus	1.50	2.00	3.00	4.00	5.64
	Ternary Tree	1.97	2.91	3.93	5.01	6.25

Table 1: Comparison of configurations

Although tree configurations are conceptually better suited to the clusteral model, tori have lower diameters and average interprocessor distances, as shown in Table 1. The torus configurations are thus more appropriate for complex communication patterns, such as those required for the parallel implementation of volume visualization [2].

4. RESULTS

To show the performance improvements that the clusteral approach can provide by reducing communication overheads, two volume data sets: a volume frame of the Mandelbrot set in the quaternion, shown in Figure 8; and, a medical MRI scan of a human head, Figure 9, were visualized. The results have been obtained on a Meiko system of sixty four T800 transputers arranged in both tree and torus configurations with a volume data size of *128 X 128 X 128* voxels for rendering and of *256 X 256 X 256* voxels for visualization. For visualization the volume data was rotated about an arbitrary axis and a set of frames were visualized.

The advantages of the clusteral approach can be seen in Figure 4. This graph compares the speed-up obtained using a random volume partitioning strategy with the hierarchical clusteral model on tree configurations. The inherent bottlenecks for global communication in the tree configurations have an increasing effect on the system performance as the number of processing elements is increased. Nevertheless, the benefits of using hierarchical clusters can clearly be seen.

Figure 5 shows how the choice of configuration influences overall system performance. The graph shows speed-up obtained using hierarchical clustering on both tree and torus configurations. For lower numbers of processing elements the choice of configuration has little effect. However, for the larger systems, the lower diameters and average interprocessor distances of the torus configurations, and their lack of bottlenecks, provides an increasing improvement in system performance.

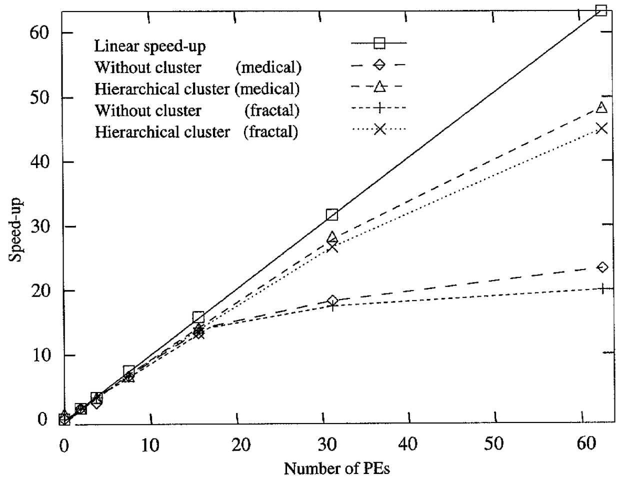

Figure 4: Tree configurations with and without hierarchical clustering

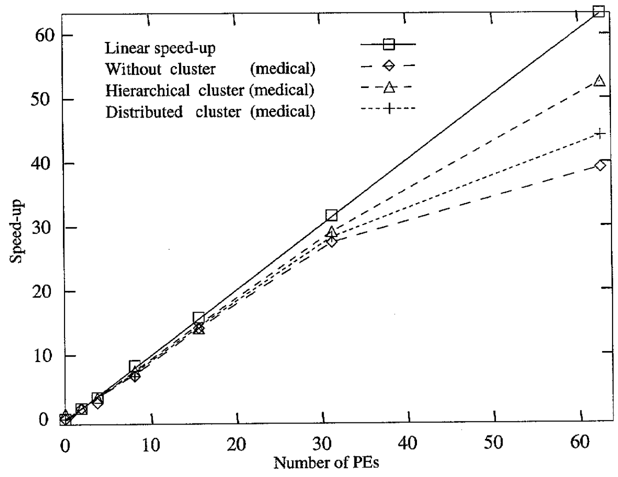

Figure 6: A comparison of the clusteral approaches

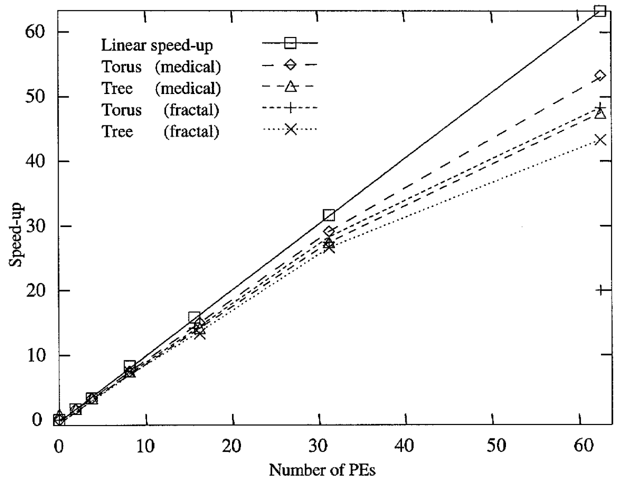

Figure 5: Hierarchical clustering on tree and torus configurations

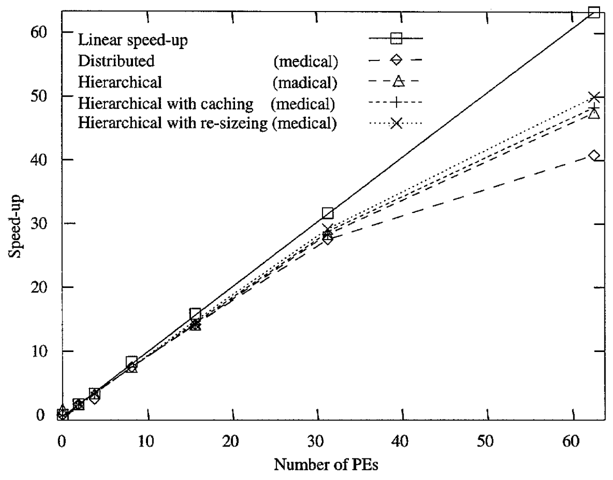

Figure 7: A comparison of the clusteral models with and without dynamic cluster re-sizing and caching

Figure 6 shows the speed-up obtained on the torus configurations for the distributed and hierarchical clusteral approaches, compared with no clustering. For the sake of clarity, only the results for the medical volume data are shown. As can be seen in the graph, for large configurations, the increasing communication overheads have an increasing effect on overall system performance. The hierarchical clustering implementation for rendering on the 64 processing element system produced a speed-up of 53.5, better than the distributed approach which had a speed-up of 44.1, and a significant improvement over the speed-up of 39.4 obtained for the implementation without the clusteral model.

Finally, Figure 7 shows the speed-up on a torus for distributed clustering, hierarchical clustering, hierarchical clustering with dynamic cluster re-sizing and hierarchical clustering with caching. Hierarchical clustering with cluster re-sizing offers the best performance for parallel volume visualization. Therefore average speed-up for volume visualization are given to show that the speed-up of parallel systems is improved by using cluster re-sizing. For a few frames, system performance may not be good as expected but for a large number of image frames performance will be better as can be seen from the figure.

On the other hand, a caching technique for clusteral models may be needed for further performance increase but this performance increase is limited for small rotation angles. For large rotations, cache coherence may be loosed so that caching may not be useful for large rotations. However, most of the volume visualization process can be taken as a sequence of small rotations. The hierarchical clustering implementation with cluster re-sizing for visualization on the 64 processing element system produced a speed-up of 50.1, better than the clustering with caching which had 48.6, hierarchical clustering which had speed-up 47.8, and a significant improvement over the speed-up of 41.2 obtained for the implementation with distributed clusteral model.

5. CONCLUSION

The results presented in this paper show that an efficient parallel implementation of volume visualization is possible using the volume partitioning method of allocating tasks. Addition of a clustering scheme to this approach reduces the communication overheads by enabling the combination of partial results to occur on processing elements which are “physically close” to those processing elements which performed the corresponding tasks. Although limited, there is still a need to communicate data amongst processing elements to facilitate load balancing. The results confirm that, for large multiprocessor systems, torus configurations are better suited than tree configurations for such a communication need.

Despite the good performance that has already been achieved (a speed-up of 53.5 for volume rendering and 50.1 for volume visualization on 64 processing elements), improvements will still need to be made if the visualization of volumes on our system is to be made *interactive*. Future work will examine complexity reduction schemes which will render an approximation of the volume data between successive view points. Once the desired new view point has been reached, progressive refinement techniques will be used to obtain desired image quality from these approximations. Recent developments in hardware and software also make it possible to implement volume visualization techniques on heterogeneous distributed parallel systems by using Java technology. For an interactive visualization, these techniques will be implemented on a heterogeneous system by using Java technology in a future work.

6. REFERENCES

[1] Chalmers A. G., Evaluation of interconnection networks, The 2 Austrian-Hungarian Workshop on Transputer Applications, Budapest, 1994.

[2] Chalmers A. G. and Tidmus J. P., Practical Parallel Processing: An introduction to problem Solving in parallel, International Thomson Publishing, London, 1996.

[3] Goel V. and Mukherjee A., An optimal parallel algorithm for volume ray casting, Visual Computer, 12(26):26-39, 1996.

[4] http://www.k isit.or.jp/ppram, 1997.

[5] Kaufman A., Höhne K., and Schröder P., Research issues in volume visualization, IEEE Computer Graphics and Applications, pages 63-67, 1994.

[6] Köse C. and Chalmers A., Dynamic data management for parallel volume visualization, UK Parallel 1996, Springer Verlag, 1996.

[7] Köse C. and Chalmers A. G., Memory management strategies for parallel volume rendering, In B. O'Neil, editor, 19.th World Occam and Transputer User Group meeting, Nottingham, 1996.

[8] Köse C. and Chalmers A. G., Profiling for efficient parallel volume visualization, Parallel Computing special edition on Parallel Graphics & Visualization., 23:943-952,1997.

[9] Levoy M. S., Volume rendering: display of surfaces from volume data, Computer Graphics and Applications, 8(3), 1988.

[10] Levoy M. S., Efficient ray tracing of volume data, ACM Transactions of Graphics, 9(3),1990.

[11] Ma K. L. and et al., Parallel volume rendering using binary-swap compositing, IEEE Computer Graphics and Applications, page 67-73, 1994.

[12] Mackerras P. and Corrie B., Exploiting data

coherence to improve parallel volume rendering, IEEE Parallel & Distributed Technology, pages 8-16, 1994.

[13] Neumann U., Volume reconstruction and parallel rendering algorithms: A comparative study, Ph.D. thesis, The University of North Carolina at Chapel Hill, Department of Computer Science.1993.

[14] Neumann U., Communication cost for parallel volume rendering algorithms, IEEE Computer Graphics and Applications, pages 49-58, 1994.

[15] Reinhard Erik, Jansen F. W. and Chalmers Alan.G., Overview of parallel photo-realistic graphics, Eurographics Star-State of art Reports 1998.

[16] Walker C., Hardware for transputer without transputers, 1996.

[17] Westover L. A., Splatting: A parallel, feed-forward volume rendering algorithm, Technical report, The University of North Carolina at Chapel Hill, Department of Computer Science, 1991.

[18] Yagel R. and Machiraju R., Data-parallel volume-rendering algorithms, Visual Computer, 11(6):319-338, 1995.

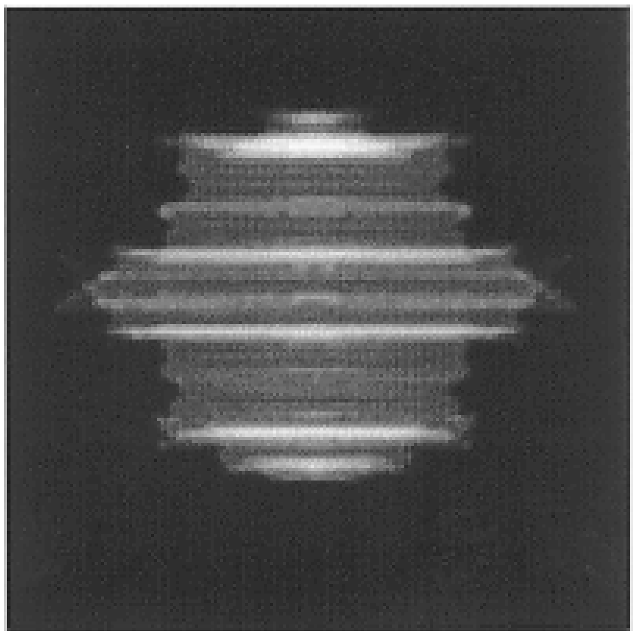

Figure 8: Fractal image in the quaternion

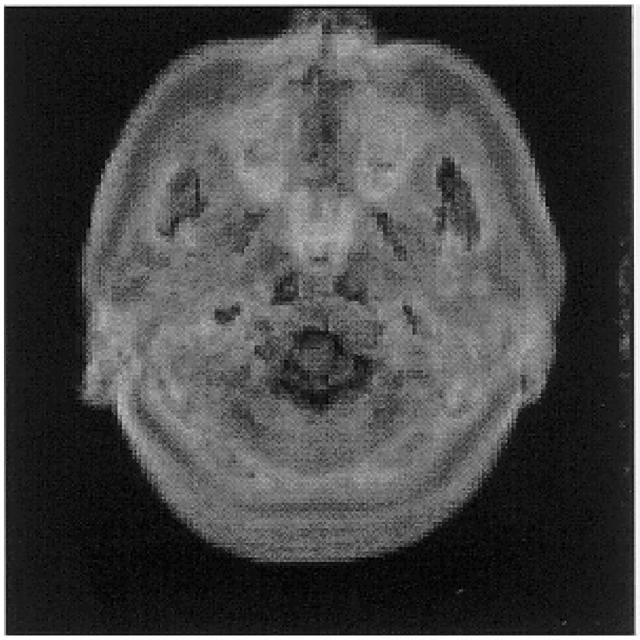

Figure 9: Medical scan of a slice of a human head