VoxelFlow: A Parallel Volume Rendering Method for Scientific Visualization

Asish Law, Roni Yagel, and D.N. Jayasimha

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio

{law, yagel, jayasim}@cis.ohio-state.edu

Abstract

Volume Visualization techniques are often applied to sampled data obtained from MRI and CT-scans as well as to simulated data computed by computational fluid dynamics (CFD) methods. In such applications, a sequence of images is animated by incrementally rotating the volumetric object about its center, so that the scientist can view the structure from all angles. In this paper, we propose an efficient method, called VoxelFlow, for generating rendered images of volume data which can be used in animating rotations of objects. The algorithm uses a look-ahead scheme of voxel dataflow and exploits the communication features of modern scalable multicomputers to achieve near linear speedup. Simulated results are shown for implementations on mesh architectures like the Intel Paragon and torus architectures like the Cray T3D. Computational and communicational load balancing issues among processors are also addressed.

1. Introduction

Medical data obtained from MRI and CT-scanners act as good sources for volume visualization applications. This technique is also applied to studies in computational fluid dynamics (CFD) where powerful computers simulate natural phenomena in 3D. Such visualization applications can greatly benefit from real time rendering of the scene. However, the size of the data in this situation may sometimes be too colossal to fit in a single computer memory. Due to this reason, it often becomes infeasible to apply direct volume visualization techniques for generating numerous images for animation on a uniprocessor machine.

Several methods exist for the rendering of volumes. One class of these methods converts the voxel representation into surface primitives [7]. However, methods of this class suffer from various disadvantages that mainly arise from the ambiguity of determining the exact position of the surface. As an alternative, **direct methods** have been developed to render volumes. These methods can be classified as *object order* or *image order*. Object order methods require the enumeration of all voxels of a volume

Submitted to: Int. Conf. Computer Applications in Eng. and Medicine

and the determination of the affected pixels on a screen [14]. Image order techniques, on the other hand, determine all the voxels of a volume which affect a given pixel on the screen [6]. Hybrid methods exist in which the volume is traversed in both object and image order [2]. Westover [14] employs the synonymous terms *feed-forward* and *backward-feed* for object and image order methods, respectively. We shall also use these terms to classify volume rendering methods. The algorithm we describe here does not assume any specific algorithm and can accommodate any one of the existing methods as the algorithm running in each processor.

Some implementations of direct volume rendering have recently been reported for a variety of parallel architectures. A few of them exist on experimental parallel processors, such as the Princeton Engine [12] or the DASH parallel system [10]. Other implementations have been conducted on commercially available parallel machines such as the AMT DAP [1], Connection Machine CM-2 ([11], [12]), MasPar MP-1 ([15], [17]), Intel iPSC-2 [9], nCube [4], Cray Y-MP [13], IBM PVS [8], Silicon Graphics multiprocessor workstation [5] and network of Suns [14].

The viewing algorithms adopted in the afore-mentioned implementations are many and varied. Several are based on backward-fed methods. In [9] and [10] ray-casting is used, while in [11] our template-based ray-casting [16] was implemented. Others, including [1], [11], and [17] adopted a feed-forward method, wherein the multiple shear decomposition was implemented. The feed-forward method ([4], [8]) on the other hand, included splatting as the reconstruction method of choice.

Efficient implementations of volume rendering methods on parallel machines are dependent on the class of the architecture, namely SIMD (Single-Instruction-Multiple-Data) or MIMD (Multiple-Instruction-Multiple-Data), the architecture of each processor of the machine ensemble (pipeline or vector architectures), memory organization (completely shared, hierarchical, or only local) and the interconnection network (mesh, hypercube, omega, or a bus). Also essential are the supported parallel programming paradigms and tools.

One measure of efficient implementation is the equity of computational load among all processors of the ensemble, or the *load balance* achieved. Another metric is the amount of synchronization (on pure shared memory machines) or communication (on ensembles with local memories) that arises. An even better metric is the ratio of the total communication to computation. Small ratios are desirable, since communication costs which are high on parallel processors should be kept small relative to the amount of computation. One way to achieve a small value of this ratio is to use large or coarse grains. A *grain* is the amount of work assigned to each processor. For machines with local memory, large grain sizes are a necessity for efficient implementation.

In this paper, we describe a parallel volume rendering method for generating images by incrementally rotating the object for animation. Such animations have wide range of applications in many disciplines of science such as biomedicine and engineering. In the following section, we describe the method and its implementation details, including memory requirements and load balancing issues. The importance of data partitioning, mapping, scheduling and assignment to reduce communication and synchronization overhead are taken into consideration during the design process. Simulated results for the implementations on the Cray T3D are shown in Section 3. In Section 4 two applications of our method, in medicine and CFD, are described. We conclude by discussing, in Section 5, the advantages, drawbacks, and possible improvements to our method.

2. The VoxelFlow Method

The parallel volume visualization technique described here for animating rotations of objects, exploits temporal coherence between scenes when there is little change in the screen position. To reduce the complexity in the description of the algorithm, we illustrate the method for the case of a 2D grid projected on a 1D "screen" as shown in Figure 1.

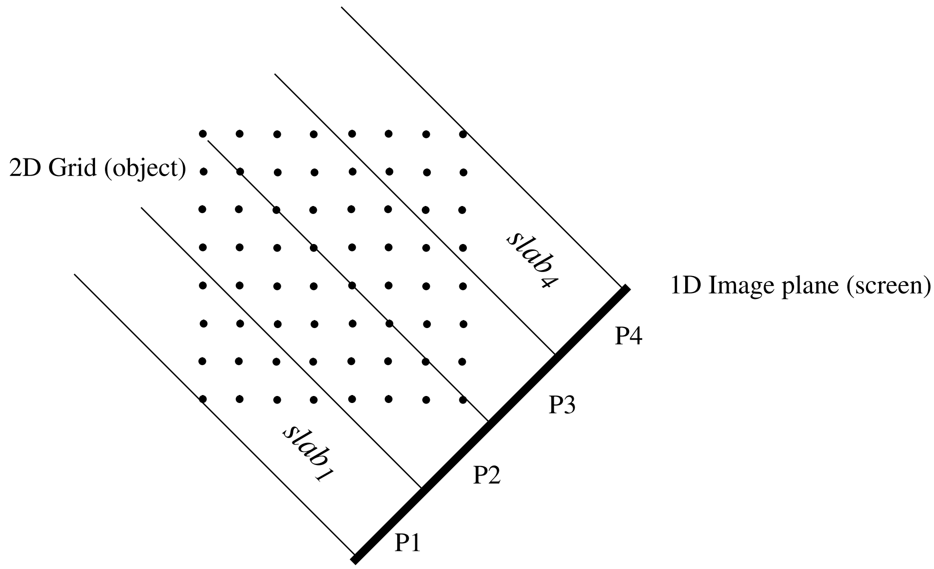

FIGURE 1. A 2D object being projected on a 1D image plane. The image plane is divided into 4 disjoint parts and assigned to the 4 processors (P1, P2, P3, P4). The object data is distributed in such a way that the i^{th} slab of data, which is locally available at each processor, is sufficient to generate the final image for that processor without communication.

Figure 1 shows a 1D screen which has been equally divided into four contiguous disjoint segments. Each of these image space segments is assigned to a different processor (P1, P2, P3, and P4). The object space is then partitioned in a way such that each processor will have in its local memory all the voxels required to generate the final image of the segment of the screen it has been assigned to. In Figure 1, we see that if the four areas of the object are allotted to the four processors as shown, then each processor will contain the voxels it needs for parallel ray casting of its portion of the screen. The portion of the object contained within two parallel lines - signifying the object partition boundaries of two adjacent processors - is referred to as a **slab** of the object (grid). The thickness of the slab is given by the width of the screen divided by the number of processors to ensure image space load balancing among processors.

The image and the object space partitioning as described above achieve a situation where no communication of data or combination of images is necessary to generate the final image. This condition holds as long as the screen remains static at the current position. Now we extend this method to generate a sequence of images by rotating the screen in small incremental angles about the center of the object. The method described below exploits temporal coherence in small rotations and minimizes communication by requesting only those voxels from other processors which are not currently available in the local memory. As seen in Figure 2(a), when only a small change in the view angle is desired, most of the data is already available in the processor (black area). Some small amount of information is needed from other processor (grey area) and some information can be discarded (or passed to other processors that might need it). This flow of voxel data between adjacent processor gives our method its name: *VoxelFlow*. A similar method to what we have described so far as been developed in parallel by [2]. As the rotation angle increases the size of the black area decreases, that is, most of the data have to be brought in (communicated) from adjacent processors. If the angle of rotation is very large some of the information might be needed which currently resides more than one processor away.

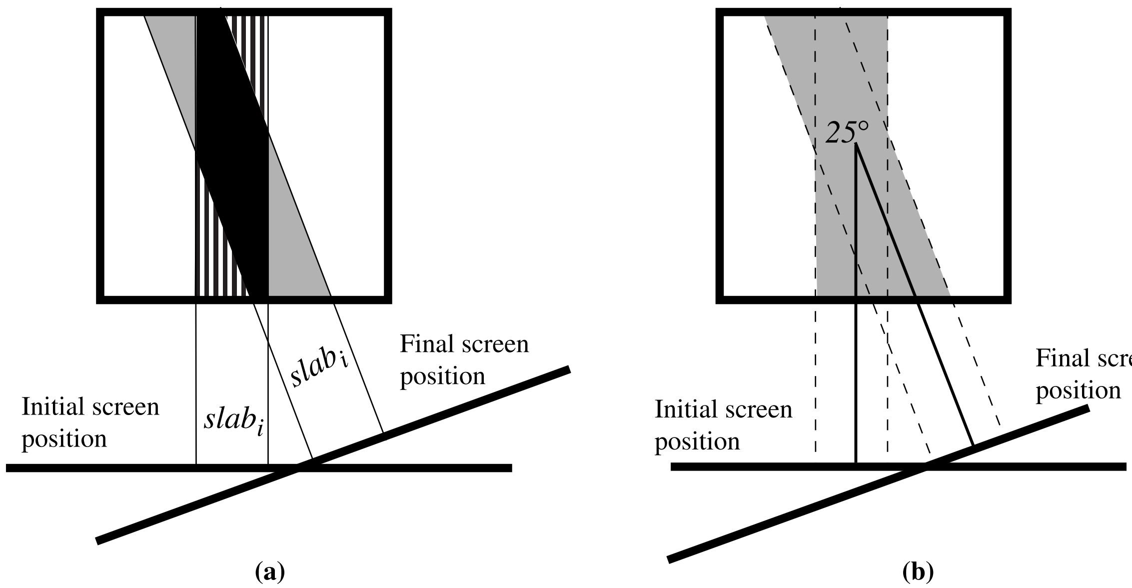

FIGURE 2. (a)When the viewpoint moves from the initial screen position to the final position the grey area contains those data elements that need to be brought in form adjacent processors, the line textured area can be discarded, and the black area consists of the data that need not be moved. (b) If the processor contains all the information in the shaded area, than all viewpoints in the 25° range can be accommodated without any communication.

Depending on the availability of local memory, we can assign and store additional data in each processor in a way such that the processors now will hold voxels needed for generating images for a range of rotational angles. Thus, instead of assigning a slab of the object to a processor, we assign a cross-section of the object which looks like a skewed X, as shown in Figure 2(b). It can be seen that for generating a sequences of images in small rotational increments between the initial and the final screen positions, the only part of the object needed by this processor is shown by the shaded area (which looks like a skewed X). Needless to say, a larger range of angles would demand a larger memory to store a wider X-like cross-section of the object.

After the appropriate voxels are assigned to each processor in a manner described above, they can generate images at each of the angles within the range required for animation of the object. The range of rotations, where all the images can be generated with the available data, is referred to as a **rotation phase**, or a **rendering phase**, or simply a **phase.** In Figure 2(b) a phase of 25° is supported. As all the required voxels are available at each processor for generating images within this range, no communication of voxels is required within a phase. Only after the screen reaches its final position in the range, redistribution of data will be needed and it is only at this stage that communication of voxels will take place. The communication is kept to a minimum by requesting and transferring only those voxels which are not locally available but will be needed in the next rotation phase. Depending on the direction of rotation of the screen, some voxels which are currently present in the local memory of a processor can be discarded at the end of a phase to make space for the voxels which are acquired from the other processors. Before these voxels can be discarded, any pending requests to this processor should be considered. This is because some of the voxels which are currently held by a processor may be the only copy in the system and may be requested by some other processor for the next phase of rotation.

A few important issues arise in the VoxelFlow method of data redistribution. Firstly, the X-like data distribution causes overlap of data among adjacent processors, leading to a more than N^3 total space requirement, for a volume of size $N \times N \times N$. On the other hand, within a rotation phase, linear speedup is attained, which suggests a space-time trade-off. Even if we have the additional memory needed, an intelligent method of data storage in memory must be devised. The voxels received at the end of a rotation phase should be placed in memory along with the voxels which were already present locally. The memory update scheme should ensure that voxel retrieval during the rendering phase does not become very complicated and time consuming.

The communication of minimal number of voxels at the end of a rotation phase also pays a toll in the exact determination of the needed voxels for the next phase. Along with the computation of the voxels, communication overhead will also be incurred in the process. Most of the multicomputers marketed today provide a communication processor at each node, which allows for parallel asynchronous communication with computation. A few popular multicomputers which provide this facility are the Intel Paragon, the Cray T3D, Meiko CS, and the Stanford Dash. Such communication processors provide a valuable resource for data acquisition in our method. Depending on the direction of rotation of the screen, we resort to a look-ahead data acquisition scheme. The direction of rotation provides us with a clue as to what additional data a processor must acquire to set up the next rotation phase. To expedite the process, we predetermine the required voxels using this information and initiate the communication process using the communication processors, so that at the end of the current rotation phase, the data for the next phase has already arrived at the processors's communication port buffer. This sort of look-ahead data acquisition eliminates communication time altogether (wire-hiding). Only some **postprocessing** remains to be done for appropriately updating the memory with the acquired data. Determination of the voxels needed by the adjacent processors and memory update make up the majority of the postprocessing stage.

Another important issue which should be discussed is the computational mapping of the VoxelFlow algorithm to a given topology so that the dilation, congestion and expansion of the embedding is optimal. The block partitioning of the image and the object space fits best to a linear array of processors. In the linear array, adjacent processors are assigned adjacent screen segments and overlapping X-like object segments. With this arrangement, the maximum screen rotation in a phase is limited to such an angle where all the additional data needed during the next rotation phase is available in the adjacent processors only. This facilitates the post processing and communication steps, by eliminating the need to send request messages to all other processors. The data acquisition information about adjacent processors can be stored in every processor so that at the end of a rotation phase, a processor will automatically send the voxels needed by the adjacent processors for the next rotation phase. In section 2.1, a block-cyclic image partitioning scheme is suggested for attaining load balance among processors. In this scheme, a ring of processors fits best for our application. The embedding of our algorithm is important as, in either case, it is known that a linear array or a ring can be optimally embedded into most of the popular topologies like the mesh or the hypercube. Also, using a ring topology with only adjacent processor communication eliminates channel contention in congestion-1 embedding.

Figure 3 gives the algorithm for generating images for incremental rotations between 0° and 360°. The rotational increments are done in steps of *rot_step* degrees. In Section 2.2 we formulate a relation between the number of processors and the maximum rotation angle such that the voxels required in the next rotation phase can be obtained from the immediately adjacent processors only. Section 2.1 suggests a block-cyclic partitioning of the image and the object space. As the results indicate, considerable load balance in communication and computation is achieved with this sort of partitioning.

Calculate thickness of slab from screen width and number of available processors

Assign appropriate screen coordinates to each processor

Distribute appropriate X-like object segments to each processor

for r = 0° to 360° step *rot_step* **do**

    for each processor P **do**

        generate portion of the image assigned to P

    if r = maximum rotation in a phase **then**

        acquire required voxels from adjacent processors

        update local memory

FIGURE 3. Algorithm for generating images for rotational animation of volume data.

2.1 Load Balancing with Block-Cyclic Data Distribution

Communicational and computational results from the first row in Figure 4 and first four rows of Table 1 show that the load is not balanced among the four processors with block distribution of data (i.e., with cyclicity 1). Three ways of load balancing were considered.

1. Assuming that the objects are uniformly distributed within the volume, the object could be distributed to the processors in a way such that equal parts of the *volume* are assigned to the processors rather than equal sections of the screen.
2. When a processor completes generation of the part of the image it is assigned to, it borrows some work from its neighbors, so that some of the neighbor's work load is reduced and at the same time the borrowing processor also remains busy.
3. If the object distribution is not uniform, a fine grain block-cyclic distribution scheme can be used, so that all the processors will process almost equal number of voxels. That is, the volume is divided to many more parts than the number of processors. Than, these subspaces are distributed to processor in a random or cyclic fashion.

The first method turned out to be a rather complicated programming task with no formidable gain. With this sort of data distribution, the load could still be unbalanced with equal probability as in the case of the block distribution. The second method of load balancing works efficiently when distributed shared memory and efficient caching systems are used. The third method of block-cyclic distribution of data provided a simple but efficient solution to load balancing. Results (Figure 4 and Table 1) show that with increasing cyclicity, almost optimal load balance is achieved both in communication and computation. Strictly speaking, the computation time at each processor will depend on the scene, as the actual number of voxels which are occupied and processed may vary across processors and greatly influence their rendering time. The percentage of occupied voxels in a scene is referred to as the **coverage** of the scene. In the results shown here, we assume complete coverage, implying that all the voxels which make up the scene must be processed for generating the final image. Even with partial and non uniform coverage, the probability of balancing the load at each processor increases with increasing cyclicity.

TABLE 1. The average number of voxels transmitted in each phase and the average number of voxels processed per 1^o of rotation.

Processor #	Cyclicity	Max rotation angle in a phase	Average # of voxels transferred to the left processor	Average # of voxels transferred to the right processor	Average # of voxels processed per rotation step
1	1	30.0	0	587	2077
2			665	1205	6116
3			1198	671	6116
4			583	0	2077
1	2	15.0	676	681	3509
2			689	760	4685
3			757	690	4685
4			678	678	3509
1	3	10.0	755	713	4090
2			711	683	4105
3			686	737	4097
4			737	759	4097
1	4	7.5	780	744	3894
2			742	738	3632
3			738	769	3802
4			770	781	3970
1	5	6.0	757	756	4038
2			756	758	4017
3			757	760	4129
4			758	759	4210

However, there is a trade-off between the level of cyclicity and the total processing time. With increasing cyclicity for a fixed number of processors, the width of the slabs diminishes (since we want to restrict communication only to be between adjacent processors). As shown in the next section, the maximum angle to which the rotations can occur in a phase also becomes smaller. This leads to more frequent, computationally expensive postprocessing, giving rise to larger overhead in the algorithm.

Table 1 shows the average number of voxels transferred to the left and the right processors in a ring at the end of each phase of rotation. The average number of voxels processed per degree of rotation are also shown. It can be seen that cyclicity 3 is sufficient to achieve almost optimal load balancing both in computation and in communication.

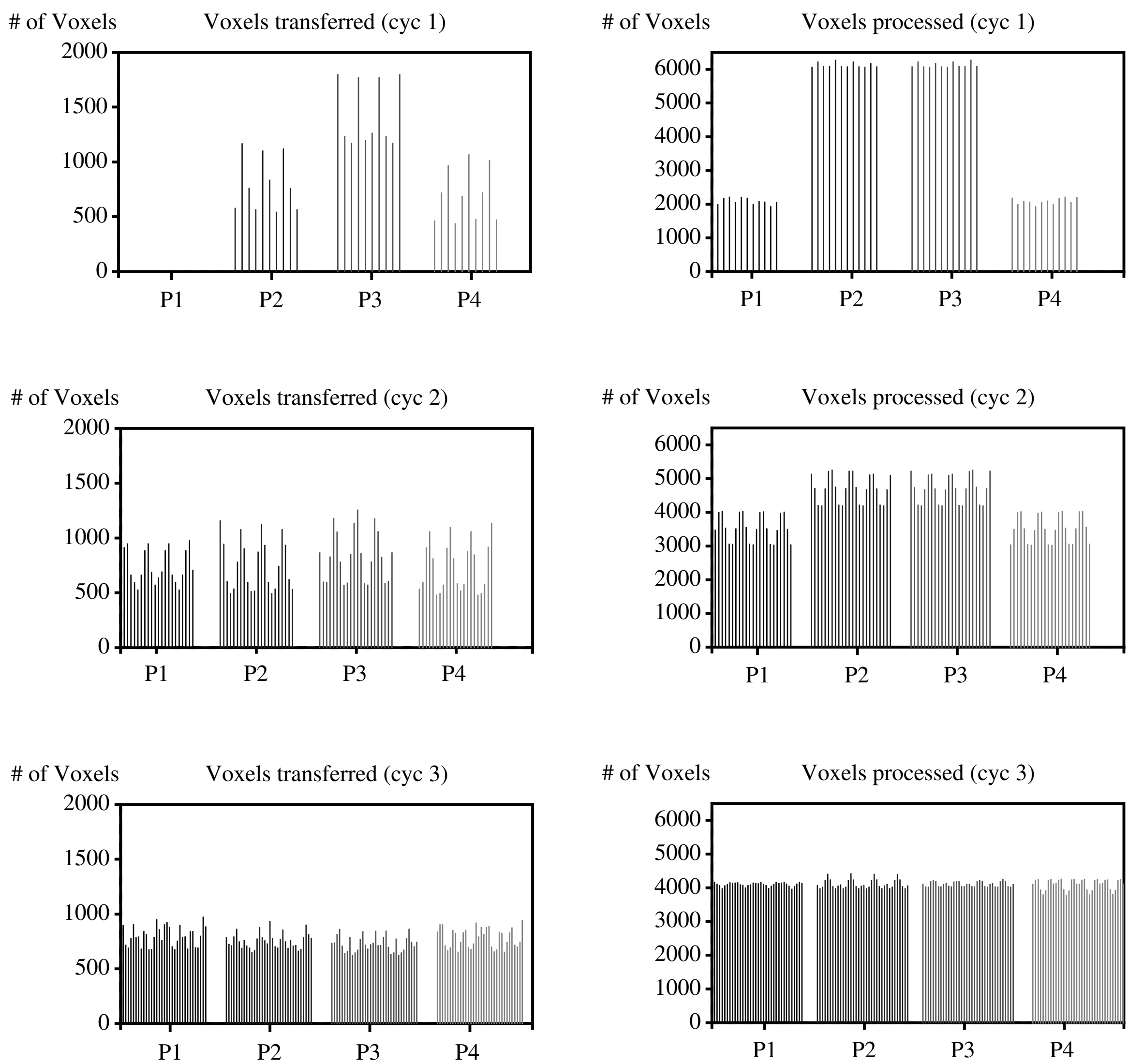

FIGURE 4. The number of voxels transferred (left column) to the neighboring processor, and processed (right column), by each of the 4 processors at the end of each rotation phase. The graphs show communication patterns for data distributions from cyclicities 1 to 3. For average numbers refer to Table 1. The increase in the number of vertical lines in the graphs with increase in number of cycles is because of the fact that with increasing number of cycles, the maximum rotation angle decreases, leading to an increase in the number of phases.

2.2 A Formulation of the Maximum Rotation Angle

The maximum rotation angle α specifies the maximum angle such that the data acquisition needed for the next phase is limited to the adjacent processors only, and no data from non-adjacent processors is needed. The following formulation shows that the maximum angle is independent of the volume resolution when the number of processors and the cyclicity remain the same.

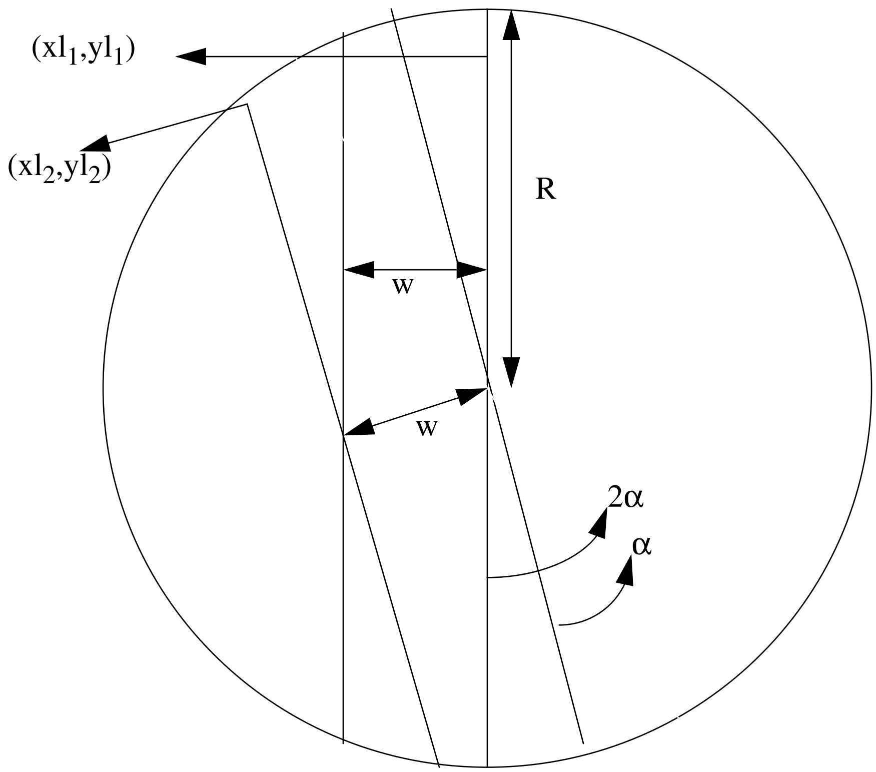

FIGURE 5. The circle used for finding the maximum rotation angle α. 2R gives the diagonal of the object and w is the width of a slab. The lines indicate the initial and final position of the boundaries of the image space partition of two adjacent processors. The dotted lines shows the final position of one processor at the end of the next rotation phase.

In order to determine the maximum rotation angle, it is noted that this angle is limited by the position of the slab, where the slab length becomes the maximum. This happens when the screen is subtended at 45° to the X axis. Thus, in the formulation, we need to be concerned only with this position. A circle, with the diagonal of the object as the diameter, is circumscribed around the object (see Figure 5). As the object lies totally within this circle, the maximum rotation angle for the actual object will be at least as large as the maximum rotation angle found here. Also, the maximum slab length occurs for the slab whose edge passes through the center of the circle, i.e., the diameter of the circle. Let,

w = slab width = resolution/ (# processors * cyclicity)
α = maximum rotation angle
R = resolution / 2

To simplify calculations, the center of the circle (also the center of the grid) is assumed to be at the origin (0,0). This does not affect our formulation as our interest is just in determining the rotational angle, which is independent of location.

Figure 5 shows the circle used for finding the maximum rotation angle. From the figure, we see that the point (xl_1,yl_1), where the diameter intersects the circle at an angle 2α is given by

$$(xl_1, yl_1) = (-R\sin(2\alpha), -R\cos(2\alpha))$$

and the point (xl_2,yl_2) where the adjacent slab edge intersects the circle at an angle α is given by

$$(xl_2, yl_2) = \left(-w\cos\alpha - (\sqrt{R^2 - w^2})\sin\alpha, yl_2\right)$$

Taking into consideration the x-coordinate only, data required for the next rotation phase can be obtained only from the adjacent processors if $xl_2 < xl_1$. If this condition fails then data needed in this phase will have to be fetched from non-adjacent processors also.

Replacing the values of xl_1 and xl_2 we get:

$$-R\sin(2\alpha) > -w\cos\alpha - (\sqrt{R^2 - w^2})\sin\alpha$$

$$2R\sin\alpha\cos\alpha < w\cos\alpha + (\sqrt{R^2 - w^2})\sin\alpha$$

The value of α can be easily obtained from the above equation. The results show that the value of α does not depend on the size of the object if the number of processors and the level of cyclicity remain the same, i.e., if the width of the slab (w) remains the same. For an object of size 128×128 the results are shown below. The number of processors is kept constant at 4, while the cyclicity is varied. The effective number of processors is equivalent to the actual number of processors times the number of cycles with block distribution (cyclicity 1). From Figure 6, we see that the maximum rotation angle decreases asymptotically with increasing cyclicity, when the number of processors is kept constant.

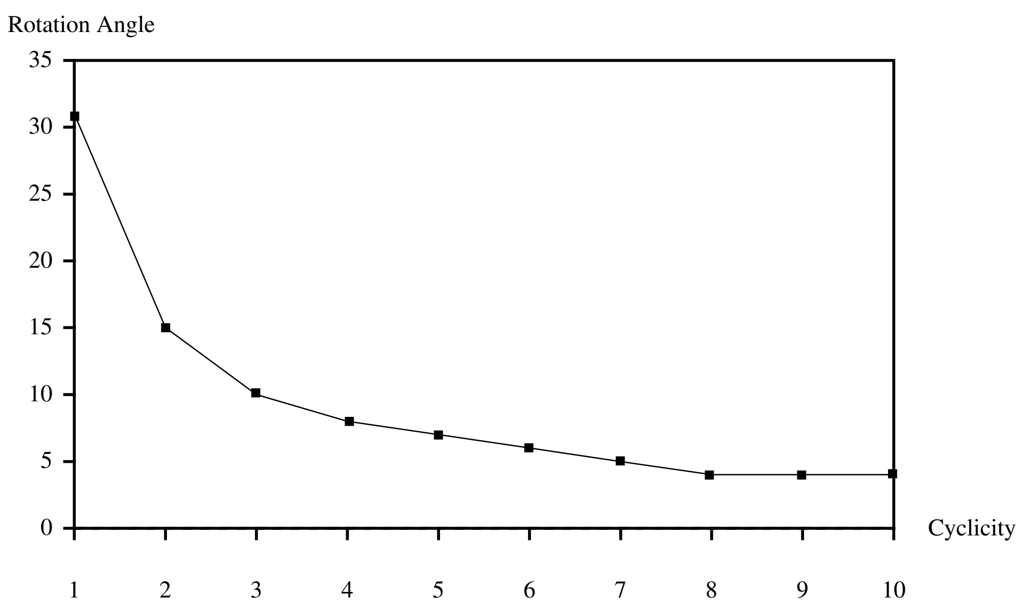

FIGURE 6. The dependance of maximum rotational angle in a phase, so that the communication for required voxels for the next phase can be acquired from adjacent processors only. This restriction will eliminate any potential message congestions in the network.

3. Simulator Setup and Results

3.1 Simulator Setup

The method discussed in the previous section has been implemented on a CSIM based simulator developed by one of the authors of this paper. The setup can simulate mesh, hypercube and torus topologies, with multiple virtual channels using wormhole routing. The timings used to simulate our method are given below, to conform with the timings for the Cray T3D. Some of the features, like the virtual channels, were not necessary at this stage of the project. The embedding of the algorithm into a ring network along with the restriction of only nearest-neighbor communication, circumvents the problems of channel or node congestion and thus the use of virtual channels. A future extension of the method to incorporate non-nearest-neighbor (multiple-hop) message transmission will require the use of virtual channels to efficiently route the messages from source to destination.

1. Clock cycle time (based on T3D) = 6.67 nsec. (150 MHz)
2. Communication time (cycles) per byte (flit) using wormhole routing (t_c) = 1 cycle.
3. Message start-up time = 225 * t_c = 225 * 6.67 nsec. = 1.5 microsecond.

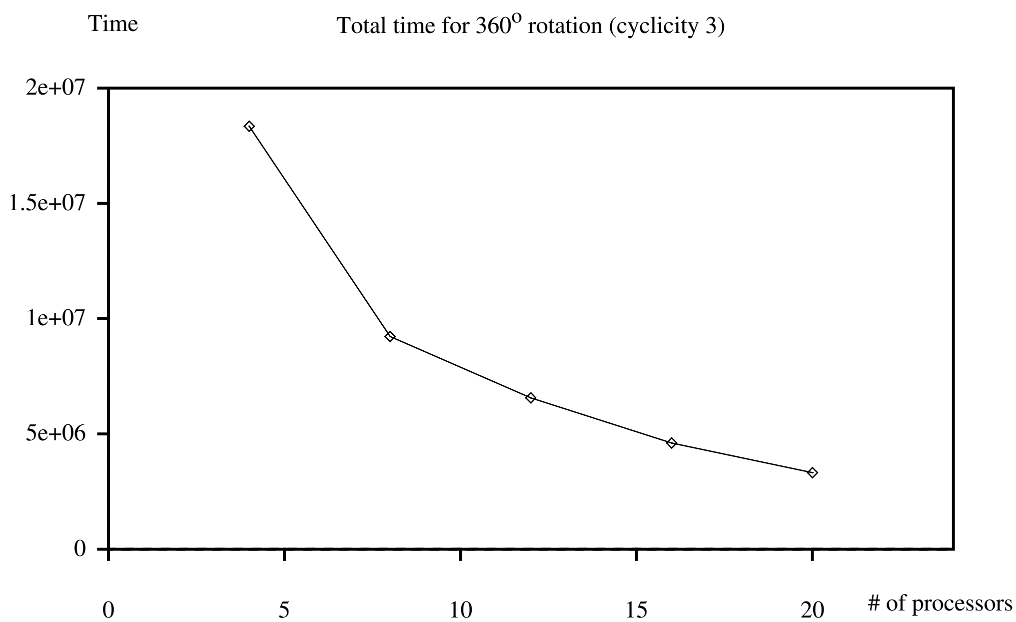

FIGURE 7. The time taken for generating images for 360° of rotation with rotation steps of 1° for different number of processors.

3.2 Results

Figure 7 shows the total time taken for generating 360 images with different number of processors. Due to limitations of the simulator, timing data was collected for a maximum of 20 processors, but the trend of the curve shows an asymptotic decrease in time with increase in the number of processors. The total time for processing each voxel is taken to be the sum of the transformation time, the shading time, the splatting time and the composition time (0.132 + 1.178 + 1.748 + 0.543 = 3.601 microseconds). These figures were obtained from [8].

Figure 8 shows the timing diagram for generating a sequence of 360 images using four different rotational increment methods. The images are generated with rotational increments of 1° and the maximum rotational angle in a phase being 30°. The bottommost curve shows the ideal timing, i.e., with speedup 1. This is possible when the complete data set is duplicated at each processor. It should be mentioned that the time required for initial data distribution and the time required for displaying the images are ignored here. Thus, in the ideal case, no data communication is needed. On the other extreme, if every voxel has to be fetched individually before processing, the fetching time, which includes start-up costs and latency costs, will be incurred for each voxel processed. This situation is shown by the topmost curve in Figure 8. The middle two plots show the timings for two slightly different incremental schemes. In the example shown here, the first scheme fetches the data for each rotation step [2] and the other fetches the data once every 30°. In the former scheme, three sources of overhead contribute to each rotation step: determination of the required data for the next step, fetching this data from the adjacent processor, and updating the local memory with the acquired data. Some of the latency overhead can be saved by pre-fetching the data, thus overlapping communication with computation. In contrast, the second method of fetching data once every 30° incurs these costs only once every 30°. This overhead is shown as bumps in the lower-middle timing diagram in Figure 8. As each processor is self-sufficient during a phase, the slope of this curve exactly follows the ideal curve within each 30° phase.

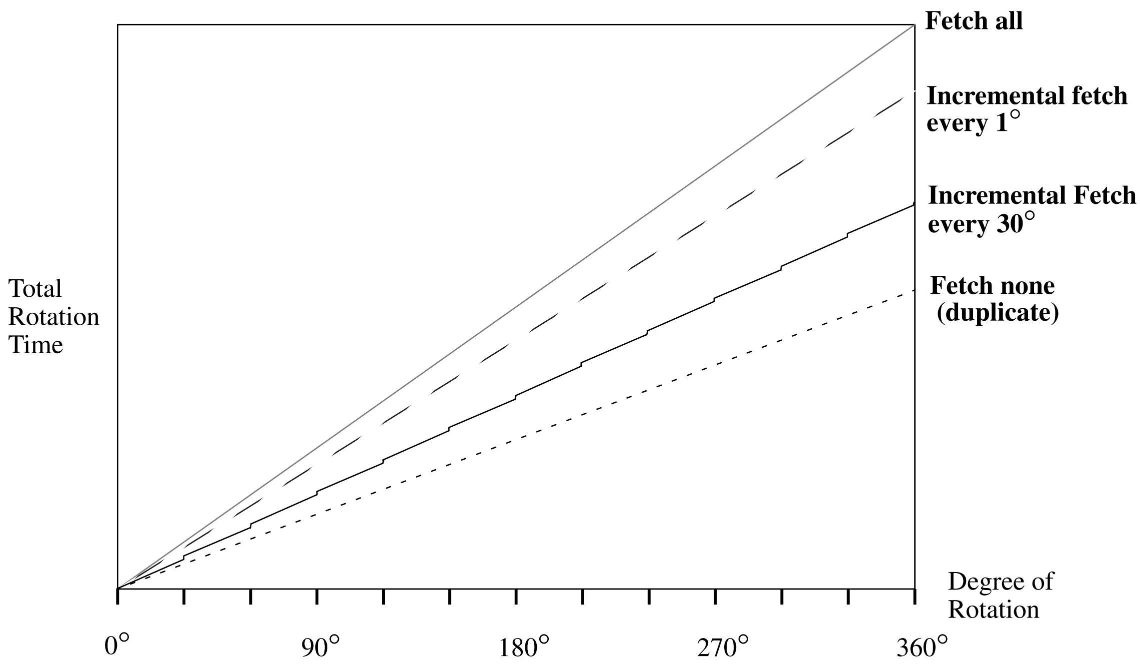

FIGURE 8. Accumulative timing diagrams for the generation of 360 images by four different implementation methods of incremental rotation scheme. Top: Every voxel fetched. Bottom: The whole database duplicated at each processor. Upper-middle: Data incrementally fetched once for every rotation step or 1°. Lower-middle: Data incrementally fetched once for every 30° of rotation.

At first glance, it would seem that our scheme of fetching a Large amount of Data only a Small number of Times (LDST) will require more time to update the local memory than fetching a Small amount of Data a Large number of Times (SDLT). A closer inspection shows that the total memory update time remains almost the same for a complete 360° rotation of the screen, as in either scheme, the total amount of data transfer and memory updates will be the same for each processor.

The major gain lies in the fact that our method (LDST) incurs only (360/30 - 1) or 11 start-up costs, 11 data request calculations, and 11 large memory updates, as opposed to (360/1 - 1) or 359 start-up costs, 359 data request calculations, and 359 small memory updates in the SDLT scheme. It may be appropriate to mention at this point that the efficacy of our system increases as the incremental rotation step size decreases. The slope of the upper and the upper-middle curves in Figure 8 will increase while the lower-middle curve will follow the ideal curve, independent of the rotation step size. This widens the gap between the LDST and the SDLT curves. For example, with rotation step sizes of half degree, the SDLT method will require (360/0.5 - 1) or 719 postprocessing steps, while the number of postprocessing will remain the same (11) for the LDST scheme. On the other hand, the reverse is also true. The slope of the upper-middle curve (SDLT) decreases with increase in step size, thus bringing it closer to the lower-middle curve (LDST). In the limit, when the step size becomes 30° in this example, the two curves will coincide and the SDLT method performs better for rotation step sizes greater than 30°.

4. Applications

Computational Fluid Dynamics (CFD) is an active research area. CFD research involves large 3D volumes of data. The data from CFD simulations often contains many data values per 3D location (e.g., velocity, pressure, density, energy, temperature). Recently, scientific visualization systems have been developed to aid CFD researchers in the interpretation of these computational data sets. Techniques used in these systems include isosurface rendering, stream-lines, contour plots, and 3D volume visualization.

Current visualization systems for CFD simulations fail to accurately display the entirely of the 3D data from the CFD simulations. CFD simulations model 3D flow phenomena. Surface based ordering techniques, stream-lines, and contour plots can only capture a small fraction of data in a 3D simulation, resulting in important information being hidden or not easily discernible in the resulting images. Volume visualization is an important tool for discerning 3D data, especially 3D scalar data. In [3] Ebert et al., have described volume rendering methods for CFD visualization.

Figure 9 shows a typical volume rendering of the flowfield static pressure generated using the computed Navier-Stokes flowfield data between two adjacent compressor rotor blades and is rendered by a ray-caster which employs a changing opacity transfer function to emphasize gradient. Without any shading of the scene, it becomes difficult to comprehend the 3D nature of their behavior at a single instant of time. The extension of the 3D rendering procedure to include the temporal variation will provide even greater detail which will be of value in understanding these features. The incremental rotation scheme developed here can effectively provide the motion parallax to perceive the depth of the flow field in an effective manner.

Volume visualization techniques are also frequently applied to data acquired from MRI and CT-scanners. Visualization of such data provide the physician with a computer generated image of the region of interest. Figure 9 shows one such image generated from data collected from a head MRI. Techniques have been developed to easily manipulate the object in such a way that the viewer can disclose the interior of the object. By specifying appropriate transparencies and/or specific cut-planes, such manipula-

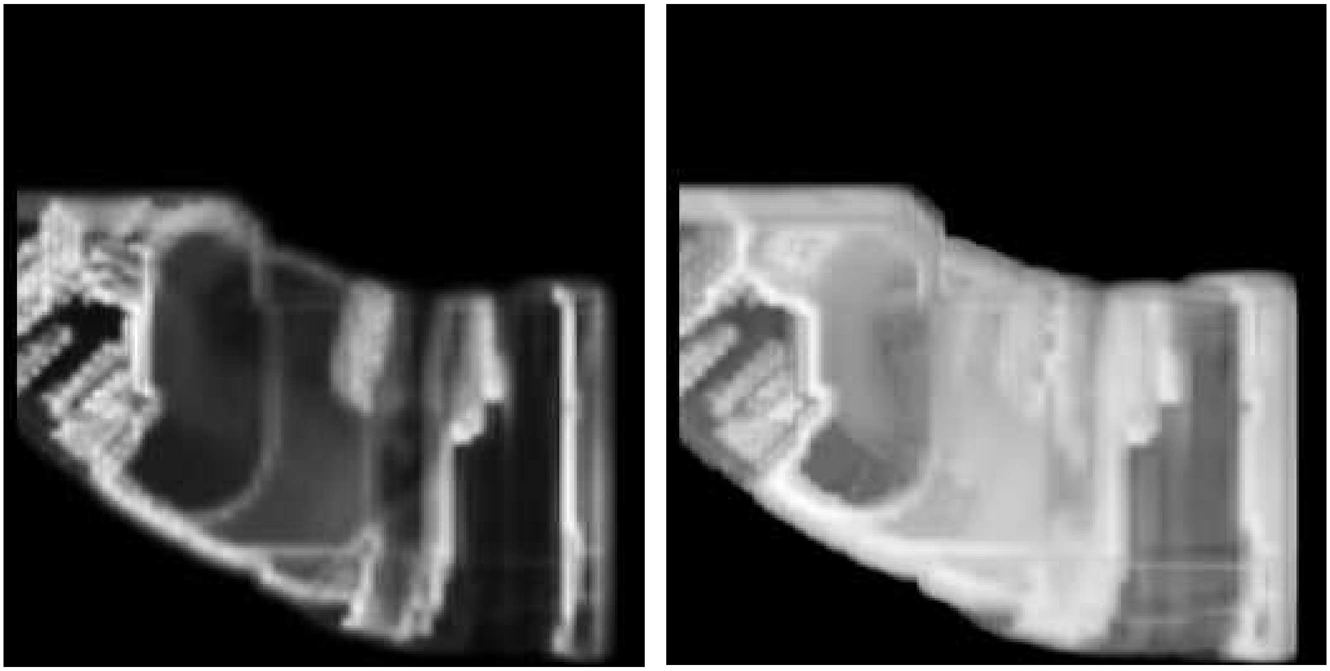

FIGURE 9. CFD: Changing opacity transfer function to emphasize gradient. The figure reveals the quantity of information that can be lost in traditional visualization systems.

FIGURE 10. Medical: Visualizing the interior of the volume obtained from head MRI. A cut-plane is used to disclose the interior of the head.

tions give the physician a powerful tool to view the volume in a useful way. In Figure 9, the volume has been cut, using a cut plane, at a certain point to visualize the interior of the brain. Although the static image is effective by itself, the physician often likes to view the rest of the brain by incrementally moving the screen (or viewing angle) around the head. Such applications falls in direct purview of the incremental rotation scheme developed here. Considering the enormity of these volume data sets along with the time taken for generating each image, the incremental rotation scheme developed on parallel computers provide considerable speedup to achieve almost real time performance for the animator.

5. Discussion and Future Work

In this paper, we have described a hybrid model for animating rotations of objects on a parallel computer. On one extreme, when the data set is small enough to be duplicated in the local memory of each processor, no data communication is needed during the rendering phase. On the other extreme, the total memory of the whole computer is just large enough to fit the database. In this case, only a predetermined non-overlapping set of data can be placed in the local memory of each processor. This is referred to as the **resident set**. While generating an image, the voxels at another processor's memory is fetched on demand. These voxels can be overwritten or lost when the local memory becomes full and additional voxels are fetched. The method described in this paper works best with local memory sizes between these two extremes, which allows for data overlap to a certain degree.

The parallel implementation of the incremental rotation scheme raises some important communicational overhead and load balancing issues. As discussed in Section 2.1, the block cyclic method of data distribution achieves considerable load-balancing both in communication and in computation. To alleviate some of the overhead, most current multicomputers provide special purpose hardware including the DMA channels to overlap communication and computation. Using asynchronous sends and receives, it is possible to partially or totally eliminate the communication time. The requests are scheduled in such a way that the message routers utilize the communication channels while the CPUs remain busy generating the images. Using the DMA channels, the local memory is updated with the incoming message content without the intervention of the CPU. This increases the utilization of all the processors and also the communication channels.

Designing the algorithm to work most efficiently on a ring architecture has some inherent advantages. A ring of processors can be easily embedded in most of the more popular topologies with dilation-1, expansion-1 and congestion-2. This makes it simple to implement on almost any type of architecture. For example, on a mesh topology like the Intel Paragon, a ring can be embedded by mapping the array onto an Hamiltonian Cycle in the mesh. For a binary tree, a pre-order traversal of the tree nodes will result in a ring. Gray codes can be used to embed a ring in a hypercube topology.

Apart from the issues discussed above for designing the algorithm, the best advantage of the system lies in the simplicity by which almost linear speedup is achieved. As discussed in the previous section, the efficacy of the method deteriorates with increase in rotation step size. In general, during an animation process, the viewing angle does not change abruptly and big jumps in screen orientations are avoided, thus making this scheme extremely useful in any general purpose animation.

Among the other advantages of the algorithm lies the fact that perspective ray casting method can be designed as a straightforward extension of the parallel viewing algorithm developed here. Also, no combining of images is required. The data is distributed in a such a manner that each processor has the voxels required to generate the final image of the part of the screen assigned to it. In such a case, there are no issues of image overlap.

In spite of the advantages mentioned above, the method is not free from drawbacks. While resorting to a block cyclic distribution of data, there is a high likelihood of computational and communicational load balancing in the animating process. At the same time, the maximum rotational angle reduces with increasing cyclicity or with increasing number of processors. This requires a more frequent postprocessing. The resulting performance tends towards the SDLT scheme of incremental rotation, which requires data fetching in each rotation step. Also, the method is not particularly suited for rotations with large rotational steps. In this case, enormous amount of data will be transferred between processors. In fact, for sufficiently large rotational steps, almost all the data required for ray casting will have to be fetched from neighboring processors.

The determination of the voxels for the next phase, the mapping of the acquired data in memory for fast retrieval, and catering to the requests of the neighboring processors have made both the algorithm and the data storage scheme in memory quite complicated, especially when extended to 3D. The complication of the method can be partly justified considering the parallelism and efficiency gained with the method. Better memory mapping or sophisticated data structures should be devised, both to decrease the complication and the overhead incurred in the parallel volume rendering system.

Finally, it is assumed that the object rotation or the screen rotation takes place about the center of the volume. This may not be a major drawback, for this sort of data visualization is predominant in most applications, especially in the arena of medical data visualization. For other applications, the method has to be extended to include rotations which are not about the center of the object.

6. Conclusion

In this paper, we have presented a parallel volume visualization technique for animating rotations of objects and described two of its applications. The method takes advantage of temporal coherence to perform a look-ahead voxel acquisition from other processors. The design of the scheme is done in a way such that only near-neighbor communication is needed thus alleviating network congestion. Careful design of the algorithm and its embedding in popular network topologies has helped considerably reduce the overhead due to parallelism. The incremental rotation scheme described here finds good application in the fields of medicine (to view data from MRI or CT-scans) and CFD (computational fluid dynamics), which incorporate voluminous data impossible to fit in a general purpose uniprocessor memory. The simplicity of the algorithm to achieve near linear speedup is its main advantage. It is our intent to device another similar scheme, which will further reduce the overhead inherent in the postprocessing step of the current method, leading to better processor/channel utilization and speedup.

Acknowledgments

We would like to thank the Ohio Supercomputer Center for allowing the use of the CRAY T3D. We thank Jeff May, Kant Patel, and Roshan Rao for the use of the multiprocessor simulator they developed. This project was supported by the National Science Foundation under grant CCR-9211288.

7. References

1. Cameron, G.G, and Underill, P.E.,“Rendering Volumetric Medical Image Data on a SIMD Architecture Computer”, SPIE, Visualization in Biomedical Computing 1992, Vol. 1808, pp. 137- 142.
2. Cohen D. and S. Fleishman, “An Incremental Rotation Algorithm for Parallel Volume Rendering”, unpublished report, Ben Gurion University.
3. Drebin, R. A, Carpenter, L., Hanrahan, P.,“Volume Rendering”, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 65-74.
4. Ebert D. S., R. Yagel, J. Scott, and Y. Kurzion, “Volume Rendering Methods for Computational Fluid Dynamics Visualization”, Proceedings of Visualization'94, Washington, DC, October 1994, pp. 232-239.
5. Elvins, T. T.,“Volume Rendering on a Distributed Memory Parallel Computer”, Proceedings of Visualization’92, Boston MA, October 1992, pp. 93-98.
6. Fruhauf, M.,“Volume Rendering on a Multiprocessor Architecture with Shared Memory: A Concurrent Volume Rendering Algorithm”, Proceedings of Second Eurographics Workshop on Visualization in Scientific Computing, 1992.
7. Levoy, M., “Display of Surfaces from Volume Data”, IEEE Computer Graphics and Applications, Vol. 8, No. 5, May 1988, pp. 29-37.
8. Lorenson, W. E., Cline, H. E., “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics, Vol. 21, No. 4, July 1987, pp. 163 - 169.
9. Machiraju R. and R. Yagel, “Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors”, SUPERCOMPUTING'93, Portland, Oregon, November 1993, pp. 699-708.
10. Montani, C., Perego, R., Scopigno R., “Parallel Volume Visualization on a Hypercube Architecture”, Proceedings of 1992 Workshop on Volume Visualization, October 1992, Boston, MA, pp. 9-15.
11. Nieh, J., Levoy, M.,“Volume Rendering on Scalable Shared-Memory MIMD Architecture”, Proceedings of 1992 Workshop on Volume Visualization, Boston, MA, pp. 17-24.
12. Schroder, P., Salem, J. B., “Fast Rotation of Volume Data on Data Parallel Architecture”, Proceedings of Visualization’91, San Diego, CA, pp. 50-57.
13. Schroder, P., Stoll, G.,“Data Parallel Volume Rendering as Line Drawing”, Proceedings of 1992 Workshop on Volume Visualization, October 1992, Boston, MA, pp. 25-32.
14. Stredney, D., Yagel, R., May, S. F., Torello, M., “Supercomputer Assisted Brain Visualization with an Extended Ray Tracer”,1992 Workshop on Volume Visualization, Boston, MA, pp. 33-38.
15. Westover, L., “Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm”, Ph.D Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, July 1991.
16. Wittenbrink, C., Somani, A., “2D and 3D Optimal Parallel Image Warping Techniques”, private communication, 1992.
17. Yagel, R., Kaufman, A., “Efficient Methods for Volume Rendering”, Computers and Graphics, Proceedings of Eurographics’92, September 1992, Cambridge, England, pp. 153-157.
18. Veznia, G., Fletcher, P. A., Robertson, P. K., “Volume Rendering on the MasPar MP-1”, Proceedings of 1992 Workshop on Volume Visualization, October 1992, Boston, MA, pp. 3-8.