PARAVOL: Parallel Volume Rendering for Virtual Medicine

Roni Yagel, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio; *Don Stredney*, The Ohio Supercomputer Center, Columbus, Ohio; *Gregory Wiet*, Department of Otolaryngology, The Ohio State University Hospitals, Columbus, Ohio; and *Asish Law*, Department of Computer and Information Science, The Ohio State University, Columbus, Ohio

ABSTRACT: *PARAVOL is a system under construction that combines software and hardware renderers. The system consists of the* Fuzzy-Set Renderer (FSR), *which runs on a Silicon Graphics Reality Engine, and the* Active Ray-Tracer (ART), *which runs on the Cray T3D. FSR utilizes hardware-based texture mapping to deliver real-time, medium quality, volume rendered images. It is geared mainly to provide a navigation aid and a tool for preliminary data exploration. Interaction is multisensory, combining various input devices and future haptic feedback. Rendering parameters are passed from the FSR front end to the ART renderer. It is based on a ray-stacking mechanism that supports latency hiding by postponing computation on inactive rays. It optimizes memory usage and utilizes a cache-only-memory organization to achieve high quality rendering while demonstrating linear speedup.*

1 Introduction

Volume-based systems are becoming attractive as processing power and memory capacity of today's systems approach a critical mass. From virtual surgery planning to concurrent design, volume-based design has the potential to greatly enhance the capabilities of surgeons and engineers. However, processing speed and data size requirements call for the use of either a hardware-based solution or a multiprocessing approach. In this paper, we report on our efforts in both of these directions and describe a system that will integrate them.

Speed and data size are two of the most important reasons that parallel computers have become inevitable for real-time rendering and animation of volumetric models. During an animation process, several images are generated by repeatedly rendering the scene. In each frame, either some objects in the scene change their positions or the viewer changes his position or viewing direction. Such animations have prevalent applications in the field of realistic rendering, science, and medicine. For example, medical data obtained from MRI (Magnetic Resonance Imaging) are good sources for volume visualization [12]. Because of the enormity of the data and, consequently, the time needed to generate each image, employing a uniprocessor for the task of rendering numerous animation frames often becomes infeasible.

1.1 3D Rendering

Our goal is to render a 3D model, residing in *object-space*, on a 2D *image-space*, the screen. We distinguish between two types of models – surface and volume. Surface models represent 3D scenes with surface patches such as polygons (Figure 1). A *volume* is a regular 3D grid of *voxels*. A voxel is the 3D equivalent of the 2D pixel. Figure 2 shows a volume made up of 8×6×5 voxels, each of size 1×1×1. Each voxel is characterized by its position in the 3D grid, and may have a color and opacity associated with it. Some sources of such voxel-based volumes are MRI, confocal microscopy, 3D simulated data, and synthetic 3D models.

There are two main approaches to volume rendering: ray casting and splatting. In ray casting, rays are cast into the object space through the screen pixels (see Figure 2) [17]. For each ray, the volume is sampled at regular intervals along the ray. The values of the samples (color and opacity) are composited from front to back until either the composited opacity becomes unity or the ray exits the object space. The composited color of the ray is the final color of the screen pixel. Our ART algorithm for the Cray T3D is a ray caster.

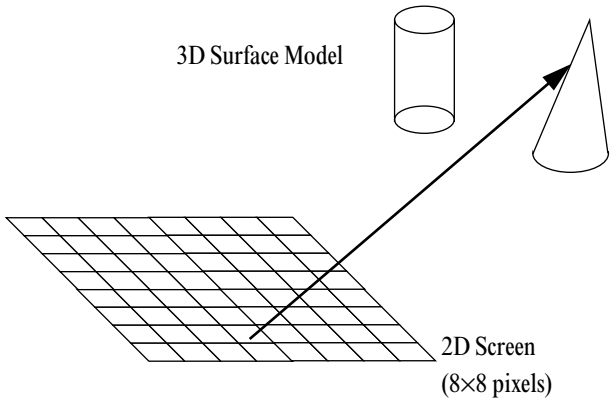

Figure 1: A 3D surface model projected onto an 8×8 screen. Only one ray is shown in the figure.

The second approach to volume rendering is splatting [23], by which the rendering algorithm scans the volume from back to front; it starts at the voxel farthest from the eye (the black voxel in Figure 2) and scans the whole volume, ending at the voxel closest to the eye (the striped voxel in Figure 2). Each voxel is drawn to the screen as a small cloud, simulating the distribution (splatting) of the voxel's intensity onto a neighborhood of pixels. Our FSR uses texture-mapping hardware to implement a variation of the splatting algorithm.

1.2 Parallel 3D Rendering

Various approaches exist to parallelize the process of 3D rendering. At one extreme, the size of the entire model is small enough to be replicated at each node. The screen is divided into a number of segments, the number being equal to the number of processors. Each processor is assigned the responsibility to generate the final image for one of these segments. In this method, no data communication is needed to generate the final image, because all the objects that may be needed to generate the respective image segments are available at the local memory of each processor. Here we have ignored the communication required to display the final image. This approach is referred to as *parallel rendering with no dataflow*.

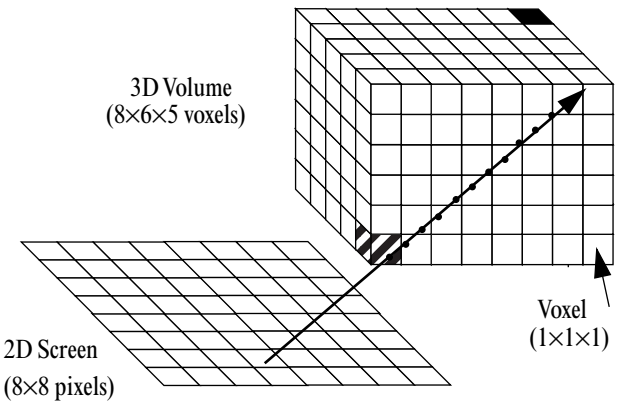

Figure 2: A 3D surface volume model consisting of 8×6×5 voxels. The model is projected onto an 8×8 screen. The dark circles along the ray show the sampling points of the volume along the ray.

At the other extreme, the total memory in the entire parallel system may be just enough to hold a single copy of the 3D model, leading to no data replication. Two different approaches are generally taken in such cases: the *ray dataflow* approach, or the *object dataflow* approach. In the former, the model is broken up into several partitions, and each of the partitions is statically assigned to a processor. During rendering, several rays are initiated in parallel by some processors. As the rays traverse through different volume partitions, they are accordingly passed to the respective processors. The receiving processors, in turn, continue tracing the ray for the part of the volume available locally and then either pass the ray to some other processor to continue tracing the ray or return the final value of the ray to the processor that initiated this ray. Proper partitioning of the volume and proper assignment of these partitions to the processors are essential for exploiting spatial and temporal coherency. Some examples of such an approach can be found in [1][3][6].

In the object dataflow approach (for example [9] [13] [14] [16] [20]), the 3D scene is partitioned similar to the partitioning in the ray dataflow approach. In addition, the screen is also partitioned and assigned to the processors. During the rendering stage, the processors cast rays for the screen segments assigned to them. For each ray, a processor fetches from other processors the data that are not available locally but are required to complete tracing the ray. For the first few rays, almost all the data must be fetched and cached locally, but subsequent rays take advantage of spatial coherence, so that most of the data that were fetched for the previous rays are also needed (and are available) for the current ray. With a local cache of considerable size, temporal coherence (between frames in an animation sequence) can similarly be exploited, so that almost all the data that are required for one frame are available in the local cache to generate the next frame.

For the object dataflow approach, several factors affect the performance of the system. First, the allocation scheme of the scene (data) partitions plays an important role: if processors are assigned partitions in a static manner, then all the demand-driven fetches will be placed in the local cache. These fetched data are likely to be removed, depending on some cache replacement policy. The amount of coherency that can be exploited depends mainly on the size of the cache. A small cache may not be able to take advantage of even ray-to-ray coherence, while a large cache may exploit even frame-to-frame coherence. The size of the cache line determines how much data is transferred for each request. As each processor potentially tries to send some data to every other processor, the cache line and the initial data allocation influence the amount of congestion that can arise in the network. Therefore, the algorithm should be designed to utilize the cache as efficiently as possible.

The underlying architecture also has a significant effect on the performance of object dataflow parallel rendering methods. If the architecture uses a shared (or distributed shared) memory, then the complete volume can be considered as shared data among all the participating processors, and non-local data are fetched using fast, hardware-based reads. This mechanism requires the existence of directories to track down the owning processor of the data. Hardware-based centralized and distributed schemes are two popular ways to implement the directories. On the other hand, no notion of shared data exists if the underlying architectural model supports only a distributed memory organization. The required data must be fetched using explicit user-level message passing protocols in software, and the user has to define all the synchronization mechanisms within the rendering program.

For all these approaches, several issues must be considered while designing a parallel system. Some of the more important issues are load balancing, network congestion, parallelization overheads, coherency exploitation, algorithm embedding, and suitability to general-purpose parallel computers.

The design of PARAVOL includes an interactive front-end that allow users to interact with the volume model in real time and a back-end parallel rendering that produces higher quality images in interactive rates (but not real time). In the next section we describe the real-time, front-end FSR algorithm. In Section 3 we describe the robust back-end ART algorithm. We describe, in Section 4, one of the applications we are exploring on the system – endoscopic sinus surgery. We conclude, in Section 5, with a short summary of our work.

2 The Fuzzy Set Renderer (FSR)

We have developed a visual interface using splats [23] that can produce frame rates up to 20 Hz using an SGI ONYX/RE. This same system produces graphics at a rate of 12Hz using an SGI CRIMSON/RE.

The *Volume Splatter* relies on the notion of *fuzzy voxel set*, which consists of a subset of the volume's voxels. For each voxel in the original volume we evaluate a transfer function $F:(\nabla, \rho) \rightarrow t$, where ∇ and ρ are the gradient and the density of the given voxel and, *t* is the inclusion criterion. We include a voxel in the *fuzzy set* if it has a large enough *t* value (above some user-defined threshold). Note that if we pick *F* to be a projection onto the second coordinate, ρ, we do merely thresholding on the density. We name each voxel passing the *F* threshold a "*splat*". The idea of fuzzy voxel set is similar to *semi-boundaries* and *shells* used by [22]. However, unlike previous methods, which choose the voxels for the set by segmentation methods, our approach chooses the voxels to be included by their contributions to the final image. This process which effectively rejects all the voxels that contribute little or nothing to the final image, greatly reduces the burden placed on the rendering pipeline.

The resulting subset of voxels, the fuzzy set, is ordered in the same fashion as the original volume: we hold slices of splats, where each slice contains rows of splats. The only difference is that the number of elements (splats) in each row may not be equal. Each row of splats is a sparse vector of original voxels; thus, for each splat in a row, we maintain its position in 3D space. In addition, we maintain the normal at each splat, which we calculate based on the information in all its twenty-six adjacent voxels.

The volume splatting algorithm takes as input a fuzzy set. The algorithm traverses the fuzzy set in a back-to-front order. For each member of the set it renders a rectangle facing the viewer, textured with a splat texture. The splat texture contains an image of a fuzzy circle, with opaque center and transparent circumference. Various functions can be used to govern the decay of opacity in this circle of influence, and we use a Gaussian function [23]. We also implemented a faster version of the rendering algorithm in which, instead of rectangles, we render enlarged points on the screen. These points have constant opacity and, therefore, generate images with some visible artifacts; however, because points are very simple graphic primitives, this method supports higher rendering speeds.

We control the material properties of the splats; however, for reasons of speed, we vary only opacity and diffuse reflection of the material for each splat. We define multiple light sources (infinite and local) and use the GL light routines to shade the splats. Transforming the rectangles, scan-converting the textured rectangles, and compositing colors and opacities are performed by the SGI graphic hardware.

2.1 Performance

Our renderer is composed of two processes, one that performs some preprocessing and one that renders. The general design of the renderer borrows some of the ideas in Iris Performer [19]. However, while Performer operates in three phases – traversal, culling, and rendering we have only two – culling and rendering. Our culling phase is a little more complex than Performer's. Most of the polygons in a standard Performer scene are independent of the viewer's eye point. Only in special cases (e.g., billboards) does Performer rotate polygons to make them face the viewer. In our case, all the polygons (splats) must face the viewer. In orthographic viewing we create all the viewer-facing polygons by translating a single polygon around the volume. Therefore, during the culling phase, we do not treat a fuzzy set as a stream of independent splats but as a geometric structure with some regularities.

For culling, we hold our splats as sparse vectors aligned with the original volume major axis. We cull splats by culling every such sparse vector. This operation reduces the complexity of the problem from n^3 to $n^2 \log n$. The *log(n)* term comes from a binary search for the sparse vector location where clipping occurs. The output of the culling process is a display list containing all the information for the most rapid rendering of an image. Our render task takes this display list and renders it.

We run our splat renderer on a multiprocessor Onyx with Reality Engine graphic hardware and a single Raster Manager (RM) board. Extracting a fuzzy set out of a 128^3 volume takes

≈60 seconds. Depending on the choice of splat threshold, we can control the resulting number of splats in the fuzzy set. For a fuzzy set with 50,000 splats lighted by four light sources at infinity, we get render rates of about twenty frames per second for point splats and about seven frames per second for textured rectangular splats. Although initialization must be repeated whenever the user changes the transfer function or loads another dataset, for many visualization operations involved in data exploration FSR provides very attractive rendering speed.

2.2 Discussion

Several drawbacks exist. First, the renderer relies on preprocessing to reduce the number of fuzzy voxels. If all the object is rather transparent, namely, most voxels contribute to the final image, the performance of this algorithm will greatly degrade. Moreover, the speed we achieved is based on simplifications in the illumination procedure. The transfer function is precomputed and cannot be changed in real time. Some advanced rendering features such as reflections are not possible. Various operations on light sources, colors, and material attributes are limited.

Also, we developed a volume deformation algorithm [15] that will allow us to simulate, for example, surgery-specific operators. This algorithm is based on deforming the sight ray used to render the volume. FSR, which is based on splatting (rather than ray casting) cannot incorporate these techniques.

We, therefore, need to implement a software-based algorithm that will place no limiting conditions on the rendering procedure. The only way to achieve some reasonable throughput from such an algorithm is through a parallel implementation. In the next section we describe our active ray tracing (ART) algorithm and its Cray T3D implementation.

3 The Active Ray Tracer (ART)

We adopted an object dataflow approach to parallel rendering on a distributed memory machine. The screen is divided into several regions, which are assigned to the processors in a cyclic manner. The object space is partitioned into equal-sized cells containing the objects in the 3D scene. For example, when we render a 128^3 volume, we may divide it into (4096) cells, each of size 8×8×8× voxels. Each processor maintains the local status of all cells in the 3D space. If a cell is available locally, its status is *valid*; otherwise it is in an *invalid* state. A processor can use only those cells whose status is marked *valid*.

Each processor keeps track of a random, disjoint subset of the cells in a data structure called the *directory*. The processor records in the directory the list of all processors holding a copy of the cells. The randomization process alleviates hot-spotting at specific nodes. The processor holding the directory is referred to as the *home node* for these cells. Each cell has a home in exactly one processor. Whenever a cell is unavailable locally, a request is first passed to the home node of that cell. The home node searches its directory to determine the node closest to the requesting node that has a copy of the requested cell. It then instructs that processor to send a copy of the cell to the requesting processor.

During the rendering stage, a processor is responsible for generating the image of the assigned screen regions only. All the rays to be traced are queued up in a *ray stack*. The processor starts tracing the rays at the top of the stack. Each ray is advanced through the 3D space as long as all the cells along its way are available locally. If the progress of a ray is interrupted because of unavailability of a cell, the ray becomes *inactive* and is therefore put back in the stack. At the same time, a request is sent to the home node of that missing cell. The processor then searches in the stack for an active ray, that is, a ray that was waiting for some data that has already arrived. The algorithm repeatedly scans the ray stack until an active ray is found (in which case it is traced) or until the stack becomes empty (in which case the algorithm moves to generate the next frame). In the case of recursive ray tracing, a special illumination model has to be formulated to support this model of stopping the processing of one ray while computing another [24].

Now we describe how the cell requests are handled in this distributed directory parallel renderer. The information corresponding to each cell in the directory is a list of all processors that have a copy of the cell. No state information is needed, because the cells are always read-only. This assumes that the animation takes place by altering the viewing parameters only (position and direction of the screen) – the objects do not move in the scene. Such animations are prevalent in computer graphics and scientific visualization. The handling of the more general case, in which objects are allowed to move, is also possible and is currently under investigation.

Five kinds of messages are used to maintain the directories. The first three correspond to a cell request, while the last two are used for cell invalidations.

1. **RQST**: This message is used for requesting a cell not available locally. The requesting processor passes the request for the cell to the home node. The home node, upon receiving this request, traverses its directory looking for the processor closest to the requesting processor that holds a copy of the requested cell.
2. **FRWD**: The home node forwards the request to the closest processor, instructing it to pass the requested cell to the requesting processor. Upon receiving a FRWD message, a processor sends the appropriate cell (a DATA message) to the requesting processor.
3. **DATA**: This message type indicates that a requested cell has arrived. The processor receives the data and updates its memory accordingly.

These three steps are shown as a graph in Figure 3a. This procedure of acquiring a cell from another node is termed a *3-hop request system*, because three hops are required to finally receive a requested cell. When the home node is itself the closest to the requesting processor, it becomes a 2-hop request system. All the steps are performed asynchronously. This is important

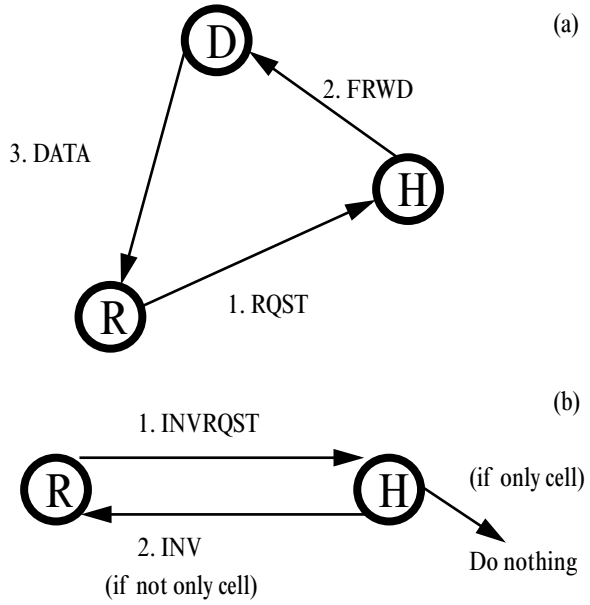

Figure 3: (a) A 3-hop system for requesting data (cells) from other processors. (b) A 2-hop invalidation process for discarding a cell from a processor's memory. R=requesting processor, H=home node, and D=closest processor containing the

because the processors do not wait to receive particular messages. Moreover, the RQST and FRWD messages are typically very small and do not affect network data traffic.

The above method of acquiring non-local data differs from previous object dataflow algorithms. In earlier algorithms, the home node always contains the requested data, so no FRWD message is needed. This adversely affects the system's performance in two ways. First, the home node always expends some memory for statically storing some data not needed by it. Second, sending of the (possibly large) requested data uses larger network bandwidth by traversing more channels in the network. With data migration, a processor stores only those data that are required for the generation of the parts of the image assigned to the processor, thus utilizing the local memory efficiently. By forwarding the cell request to the closest processor, it utilizes the network channels in an efficient manner also. A short message (FRWD) travels across the network to the closest processor, while a large message (DATA) travels a short distance to the requesting processor. In the results section we will see that the time required to determine the closest node and the time spent for the extra hop (FRWD message) are negligible.

We observe that in the case in which one is careful to assign adjacent screen segments to adjacent processors, it is very likely (and for ray casting it is always so) that the requested cells will be found at, and fetched from, adjacent processors. This unique attribute of our approach allows for the efficient exploitation of spatial and temporal coherency.

In addition to the 3-hop mechanism for data fetching, we need a 2-hop procedure (Figure 3b) for invalidating data that are no longer required locally. A processor that wants to discard a cell needs to request permission to do so from the cell's home node since this data might consist of the only copy of the cell in the entire system.

4. **INVRQST**: Whenever a processor's local memory exceeds a pre-specified limit, an LRU scheme is used to purge some locally available cells. Before removing a cell, the processor asks the home node's permission to do so by sending an INVRQST message. When a home node receives the invalidation request, it checks whether this is the last copy of the cell in the system. If other copies exist, it sends an invalidation message (INV) to the processor to purge the cell from its memory. Otherwise, it does nothing.
5. **INV**: Upon receiving this message, a processor is sure that it can purge the cell from its memory, and does so.

The different types of messages described above are handled similarly to polling-based Active Messages [5]. While the processor proceeds from one ray to the next in the stack, it polls the buffers to check for any pending messages, using non-blocking probes. The messages, if any, are received, and appropriate action is taken immediately. Home nodes forward the requests for cells to the appropriate nodes. A forwarded message is handled by immediately sending the requested cell to the requesting processor, and invalidation messages are handled accordingly. All such pending messages are dealt with before proceeding to the next ray.

This interleaving process has a two-fold effect on the performance of the algorithm. First, the communication latency is hidden by the computation process (rendering), except start-up costs. Second, it avoids deadlock due to filling up of buffers. By constantly polling and emptying the buffers, a processor avoids the indefinite filling up of its buffers, and by sending only a small number of messages after tracing each ray, a processor avoids hot-spot congestion at other nodes.

3.1 Results

The object migration parallel rendering method has been implemented on a 32 node Cray T3D at the Ohio Supercomputer Center. The T3D is a scalable massively parallel (MPP) system. Each PE is a DEC chip 21064, having 64 MBytes of memory per node. The PEs are connected by a very fast, bidirectional 3D torus interconnection network that has four virtual lanes per node in each direction. The system can be used either in message-passing mode or in shared-memory mode; we chose the first paradigm to incorporate our algorithm. Each processor runs at a peak clock speed of 150 MHz (clock cycle time = 6.67 nsec.). The Cray T3D supports deterministic wormhole routing, and communication takes place at 150MB/sec/link/direction, the message start-up time being 1.5 microseconds.

To analyze the attributes of our implementation we tested various aspects of it by varying some of the parameters that control its performance. We have tested the impact on perfor-

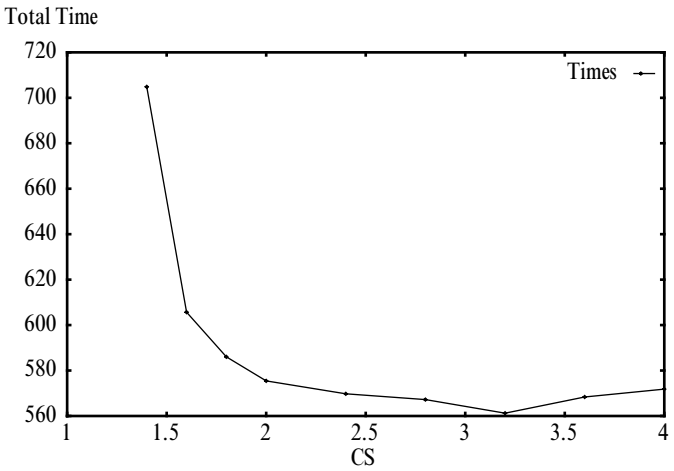

Figure 4: Rendering time as a function of cache size. The total amout of cache in all eight processors is equal to CS*volume size (CS = cache size).

mance of cache size, cell size, number of processors, and volume resolution. We measured rendering times and number of cells fetched in various scenarios.

When cache size is decreased, we expect more thrashing to happen because some cells that are removed from a small cache might be needed again while rendering the same frame. In Figure 4 we observe that when the total amount of cache in all eight processors approaches 2.5 times the volume size, processing time stabilizes at around 560 seconds. In this test we rendered 60 frames of a 128^3 volume, using cell size of 8^3. A volume of this size occupies 2MB of memory. In the case of 8 processors, we will not need more than 640K cache at each node to perform optimally.

There are two main reasons for the time penalty suffered when cache size decreases. The first is thrashing which can be measured by looking at the number of cell fetches. The second is the overhead of our algorithm that needs to scan the ray stack many more times.

Table 1. Rendering times, as a function of the number of processors, for generating 30 frames.

Processors	128^3	256^3
1	2153	-
2	1062	-
4	557	3915
8	281	1968
16	142	996

Table 1 shows the total times (in seconds) taken to generate 30 frames of 128^3 and 256^3 volumes. Screen sizes (that is, number of rays traced) are 256^2 and 512^2, respectively). Times for a small number of processors are not available in the case of 256^3 because of lack of memory. The graph in Figure 5 demonstrate the almost linear speedup achieved with our algorithm. We attribute this speedup to two main reasons: the first is our ability to conceal communication overhead by effective latency hiding based on ray stacking; the second is that other overheads such as send and receive start-ups and directory maintenance are negligeable and amount to approximately 0.1% of the total time. The maintenance of the LRU replacement policy is somewhat costly (about 2.5%) and tuning the system to the optimal cell and cache sizes can yield significant benefits. The efficiency for the 256^3 volume is measured relative to the timing for 4 processors.

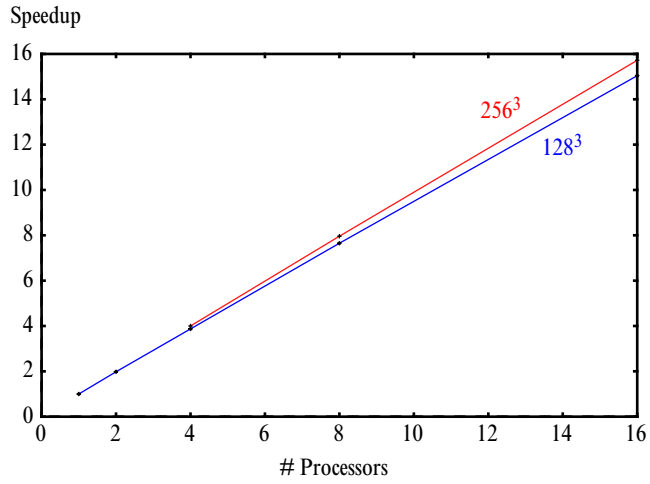

Figure 5: Speedup results for volume rendering of 128^3 and 256^3 volumes on 1 to 16 processors.

4 Application: Endoscopic Sinus Surgery

Endoscopic Sinus Surgery (ESS) is currently the procedure of choice for the treatment of medically resistant recurrent acute and chronic sinusitis. Its rationale is based on the simple premise of improving the natural drainage of the sinuses into the nasal airway.

The paranasal sinuses are composed of a series of labyrinthine passageways and spaces located in the lower forehead, between the eyes, at the center of the cranial base, and behind the cheeks. Several factors complicate surgery in these areas. First, the sinuses are surrounded by vital structures such as the eye socket and its contents, the internal carotid artery, which supplies blood to the brain, and the brain itself. Such structures lie within millimeters of the sinus boundaries. Second, to avoid external incisions, access to the sinuses is through the nostril, which precludes direct visualization and manipulation of the sinus structures by the naked eye. The technique consists of visualizing landmark structures within the nasal cavity and sinuses, excising and “biting” out diseased tissue, probing and suctioning under direct visualization through the endoscope or via a video monitor with attached endoscopic camera, as shown in Figure 6. The nostril becomes a fulcrum of rotation because the scope and other instruments must pass through this narrow area to gain access to the deeper sinus structures. Recognition of key structures by both visual and haptic queues is paramount to a safe, minimally invasive, and adequate technique.

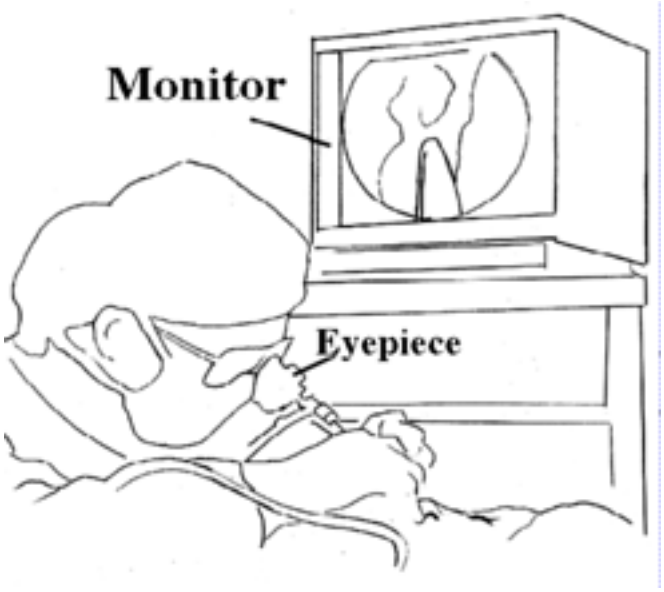

Figure 6: An illustration showing a surgeon looking through the eyepiece while the video image is displayed on the monitor in the back.

Today the technique is taught by a stepwise approach. The learning process is labor intensive and can be frustrating because models and cadavers can show only limited variability and interaction. There is no way to actually practice the techniques learned without using either cadaver material or live subjects. A realistic simulator of the paranasal sinuses will allow for a quicker understanding of the three-dimensional anatomy and allow a safe and realistic environment for the novice surgeon to learn the techniques. A true simulator must have realistic haptic as well as visual display. Many of the surgical maneuvers require subtle haptic as well as visual cues. Inside the sinuses, many of the tissues look similar, and it is their "feel" that allows the expert surgeon to distinguish between such structures as another air-filled passage versus the thin bone that covers the brain and eye. One can be removed aggressively, while the other's disruption may lead to a disastrous result for the patient. Haptic display is essential if the simulation is to be more than just a "fly through". Finally, to convey a sense of reality the system must deliver real-time photo-realistic rendering of the 3D environment.

Our system (see Figure 7) consists of a mock patient head housing electromechanical mechanisms that provide the force reflection to the user. Reconstructed anatomy derived from magnetic resonance and computed tomography data is registered with the model of the face. The 3D volumetric model is rendered to the screen using the FSR algorithm described in Section 2. A surface model of the surgical instruments is integrated with the internal view of the reconstructed anatomy. The digital referents of the instruments are graphically updated through information provided by the sensors inside the model head. One of the instruments is the endoscope, whose location provides the viewpoint and viewing vector for the user and the generation of the scene. Viewing and shading parameters can be sent to the ART renderer, (described in Section 3) for high-quality rendering.

We have integrated a five degrees-of-freedom (5DOF) probe, the Microscribe^{TM} by Immersion Corporation of San Jose. This probe presents an intuitive interface for the users, allowing them to work in a natural way within the patient model. The user views the internal anatomy, which has been reconstructed in the computer, by direct observation of the computer monitor. This method is similar to viewing the surgical field via a monitor. Eventually, we will incorporate the video directly to the scope as this technique is employed by experts.

Figure 7: The Microscribe^{TM} held by a surgeon inserted into the nasal cavity of a mock patient head. The monitor displays a view of the virtual patient in real time.

The efforts of FSR and ART are not yet fully integrated under the PARAVOL system, and some components such as volume morphing [15], high quality illumination, and visibility preprocessing [25] are being developed. We are still working to complete each renderer and complete the implementation of the umbrella PARAVOL system.

5 Conclusion

In this paper, we have described PARAVOL, a system under construction that combines hardware-based interactive rendering with parallelized software-based high quality rendering. The first method, called FSR, exploits today's texture mapping hardware such as the Reality Engine. FSR also relies on the extraction of some subset of (fuzzy) voxels. The parallel algorithm, termed ART, runs on the Cray T3D machine. It uses a distributed directory organization for tracking the cells required by a processor for generating images during an animation sequence. Unlike existing methods, it exploits both spatial and temporal coherence in an animation sequence to reduce communication between nodes. The main advantages of the method are efficient local memory utilization, effective latency hiding, reduction in network congestion, and static load balancing. These are the factors that contribute to the linear

speedup of our algorithm. We advocate in this paper the construction of a system that combines the advantages of hardware-based rendering algorithm as a front-end to parallel software-based renderer. We believe that only such a solution will be able to satisfy the ever growing demands of virtual medicine.

6 Acknowledgments

This work was partially supported by National Science Foundation Grant CCR-9211288, Department of Defense USAMRDC 94228001, and by the Advanced Research Projects Agency Contract DABT63-C-0056.

7 References

1. D. Badouel, K. Bouatouch, T. Priol. “Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data”. *SIGGRAPH ‘90 Parallel Algorithms and Architecture for 3D Image Generation Course Notes*. pp. 185-198.
2. H. Burkhardt III, S. Frank, B. Knobe, J. Rothnie. “Overview of the KSR1 Computer System”. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.
3. J.G. Cleary, B. Wyvill, G.M. Birtwistle, R. Vatti. “Multiprocessor Ray Tracing”. Research Report 83/128/17, University of Calgary, October 1983.
4. B. Corrie, P. Mackerras. “Parallel Volume Rendering and Data Coherence”. *Proceedings of Parallel Rendering Symposium*, October 1993, pp. 23-26.
5. T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser. “Active Messages: a Mechanism for Integrated Communication and Computation”. Proceedings of *the 19th International Symposium on Computer Architecture*. Gold Coast, Australia, May 1992, pp. 256-266.
6. M. Dippe, J. Swensen. “An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis”. *Computer Graphics* 18(3), 1984. pp. 149-158.
7. Foley J.D., A. van Dam, S.K. Feiner, J.F. Hughes. *Computer Graphics, Principles and Practice*, second edition, Addision Wesley, 1992.
8. N. Green, M. Kass, G. Miller. “Hierarchical Z-Buffer Visibility”, *Proceedings of SIGGRAPH’93*, pp. 231-238.
9. S.A. Green, D.J.Paddon. “Exploiting Coherence for Multiprocessor Ray Tracing”. *IEEE Computer Graphics & Applications*, November 1989, pp. 12-26.
10. E. Hagersten, S. Haridi, D.H.D. Warren. “The Cache-Coherence Protocol of the Data Diffusion Machine”. Michel Dubois and Shreekant Thakkar, editors, *Cache and Interconnect Architectures in Multiprocessors*. Kluwer Academic Publishers, 1990.
11. T. Joe, J.L. Hennessy. “Evaluating the Memory Overhead Required for COMA Architectures”. *IEEE Computer*, September 1994, pp. 82-93.
12. A. Kaufman. *Volume Visualization*, IEEE CS Press, 1991.
13. H. Kobayashi, H. Kubota, S. Horiguchi, T. Nakamura. “Effective Parallel Processing for Synthesizing Continuous Images”. *New Advances in Computer Graphic*s. Proceedings of CGI ‘89, pp. 343-352.
14. H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, Y. Shegei. “Load Balancing Strategies for a Parallel Ray-Tracing System Based on Constant Subdivision”. *The Visual Computer*. 1988. pp. 197-209.
15. Y. Kurzion R. Yagel. “Volume deformation using Ray Deflectors”. *The 6th Eurographics Workshop on Rendering*. Dublin, June 1995, pp. 21-32.
16. A. Law, R. Yagel. “CellFlow: A Parallel Rendering Scheme for Distributed Memory Architectures”. *International Symposium on Parallel and Distributed Processing Techniques and Applications.* Athens, Georgia, November 3-4, 1995.
17. M. Levoy. “Display of Surfaces from Volume Data”, IEEE Computer Graphics and Applications, Vol. 8, No. 5, May 1988, pp. 29-37.
18. W. Messerklinger. *Endoscopy of the Nose*. Urban and Schwartzenburg, Inc. Baltimore, Maryland, 1978.
19. J. Rohlf J. Helman. “IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics”, *Proceedings of SIGGRAPH ‘94*. Orlando, Florida, July 1994, pp. 381-395.
20. J. Nieh, M. Levoy. “Volume Rendering on Scalable Shared-Memory MIMD Architecture”. *Proceedings of 1992 Workshop on Volume Visualization.* Boston, MA, pp. 17-24.
21. P. Stenstrom, T. Joe, A. Gupta. “Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures”. *Proceedings of the 19th Annual International Symposium on Computer Architecture*. pp. 80-91. May 1992.
22. J.K. Udupa D. Odhner. “Interactive Surgical Planning: High-Speed Object Rendition and Manipulation Without Specialized Hardware”. *Proceedings of the First Conference on Visualization in Biomedical Computing*. pp. 330-335, May 1990.
23. L. Westover. “Footprint Evaluation for Volume Rendering”. *Proceedings of SIGGRAPH ‘90, Computer Graphics*, 24(4):367-376, August 1990.
24. R. Yagel and J. Meeker. “Priority Driven Ray Tracing”. Technical Report OSU-CISRC-8/95-TR35, Department of Computer and Information Science, The Ohio State University, August 1995.
25. R. Yagel and W. Ray, “Visibility Computation for Efficient Walkthrough Complex Environments”, accepted to *PRESENCE*, April 1994.
26. K. Ying, P. Schmalbrock, B.D. Clymer, “Echo-Time Reduction for Submillimeter Resolution Imaging with a Phase Encode Time Reduced Acquisition Method”. *Magn. Reson. Med.*, 33:82-87, 1995.