[KB88] Arie Kaufman and Reuven Bakalash. Memory and processor architecture for 3D voxel-based imagery. *IEEE Computer Graphics and Applications*, 8(11):10–23, November 1988.

[KBCY90] Arie Kaufman, Reuven Bakalash, Daniel Cohen, and Roni Yagel. A survey of architectures for volume rendering. *IEEE Engineering in Medicine and Biology Magazine*, 9(4):18–23, 1990.

[Lev89a] Marc Levoy. Design for a real-time high-quality volume rendering workstation. In *Chapel Hill Workshop on Volume Visualization Conference Proceedings*, pages 85–92. Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[Lev89b] Marc Levoy. *Display of Surfaces From Volume Data*. PhD thesis, The University of North Carolina at Chapel Hill, 1989.

[Mea85] Dr. Donald J. Meagher. Applying solids processing methods to medical planning. In *Proceedings of NCGA*, pages 101–109. National Computer Graphics Association, April 1985.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *Proceedings of the San Diego Workshop on Volume Visualization*, 1990.

[OUT85] Toshiaki Ohashi, Tetsuya Uchiki, and Mario Tokoro. A three-dimensional shaded display method for voxel-based representations. In *Proceedings of EUROGRAPHICS 1985*, pages 221–232. National Computer Graphics Association, September 1985.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. In *Computer Graphics*, pages 253–259. ACM Siggraph '84 Conference Proceedings, 1984.

[Sab88] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. In *Computer Graphics*, pages 51–58. ACM Siggraph '88 Conference Proceedings, 1988.

[TG88] Filippo Tampieri and Donald Greenberg. Experimental distributed processing system for global illumination algorithms. Technical report, Cornell University, 1988.

[UK88] Craig Upson and Michael Keeler. Vbuffer: Visible volume rendering. In *Computer Graphics*, pages 59–64. ACM Siggraph '88 Conference Proceedings, 1988.

[Wes89] Lee Westover. Interactive volume rendering. In *Conference Proceedings of the Chapel Hill Workshop on Volume Visualization*, 1989.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. In *Computer Graphics*. ACM Siggraph '90 Conference Proceedings, 1990.

[WG91] Jane Wilhelms and Allen Van Gelder. A coherent projection approach for direct volume rendering. In *Computer Graphics*. ACM Siggraph '91 Conference Proceedings, 1991.

[Whi91] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett, October 1991.

[ZT89] O. C. Zienkiewicz and R. L. Taylor. *The Finite Element Method*. McGraw-Hill Book Company, 1989.

References

[BBN88] BBN Advanced Computers, Inc. *Programming in C with the Uniform System*, revision 1.0 edition, October 1988.

[BBN89] BBN Advanced Computers, Inc. *Inside the TC2000 Computer*, preliminary edition, August 14 1989.

[BL90] Andrew Burke and Wm Leler. Parallelism and graphics: an introduction and annotated bibliography. In *SIGGRAPH Course Notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[BP90] Didier Badouel and Thierry Priol. An efficient parallel ray tracing scheme for highly parallel architectures. In *Proceedings of the Fifth Eurographics Workshop on Graphics Hardware*, September 1990.

[Cha90] Judith Ann Challinger. Object-oriented rendering of volumetric and geometric primitives. Master's thesis, University of California, Santa Cruz, 1990.

[Cro90] Franklin C. Crow. Parallel computing for graphics. Technical report, Xerox Palo Alto Research Center, 1990.

[CWBV83] John G. Cleary, Brian Wyvill, Graham M. Birtwistle, and Reddy Vatti. Multiprocessor ray tracing. Technical Report 83/128/17, University of Calgary, 1983.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *Computer Graphics*, pages 65–74. ACM Siggraph '88 Conference Proceedings, 1988.

[DS84] Mark Dippé and John Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. In *Computer Graphics*, pages 149–158. ACM Siggraph '84 Conference Proceedings, 1984.

[EKM+91] John Ellis, Gershon Kedem, Richard Marisa, Jai Menon, and Herb Voelcker. Breaking barriers in solid modeling. *Mechanical Engineering*, 113(2):28–34, Febuary 1991.

[Fle88] C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics*. Springer-Verlag, 1988.

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics*, pages 79–88. ACM Siggraph '89 Conference Proceedings, 1989.

[Fuc77] Henry Fuchs. Distributing a visible surface algorithm over multiple processors. In *Proceedings of ACM*, pages 449–451, October 1977.

[GRB+85] Samuel M. Goldwasser, R. Anthony Reynolds, Ted Bapty, David Baraff, John summers, David A. Talton, and Ed Walsh. Physician's workstation with real-time performance. *IEEE Computer Graphics and Applications*, 5(12):44–56, December 1985.

[Jac88] D. Jackél. Reconstructing solids from tomographic scans - the PARCUM II system. In *Advances in Computer Graphics Hardware II*, pages 101–109. Springer International, 1988.

structures such as the transfer functions or viewing parameters, followed by rerendering. In that case, some of the data structures and code will be in the local instruction or data caches, giving improved performance.

The parallel raycasting method intuitively seems to be more easily extended to render other types of volumetric datasets. Those datasets in which neighboring cells are not implicit (such as finite element meshes) will require an explicit visibility graph as opposed to the implicit one used here for a regular rectilinear dataset. In particular, pointers to adjacent cells will be needed in addition to the counts. For curvilinear computational meshes, ordering constraints may need to be imposed between two cells that are not even neighbors since the mesh can be curved in $\Re^3$. These complications will result in even larger memory requirements for the storage of the visibility graph which may be prohibitive for large volumetric datasets.

6.4 Future Work

This research is being continued along several directions:

- Development of schemes for local memory utilization will reduce the overhead incurred in remote references.
- The projection approach is an important method for volume rendering. It would be interesting and worthwhile to design a more efficient task generator or decomposition for this algorithm.
- In interactive use, two of the most likely updates will be to the transfer functions and to the viewing parameters. It may be possible to achieve very fast image update rates for changing transfer functions by obtaining, ordering, and caching all the samples for each pixel and requiring only compositing operations for each change to the transfer function. Changing the view would require the volumetric dataset to be resampled. Since the transfer function and viewing parameters are currently propagated to each processor's local memory, disseminating updates to these data structures will be an issue. It may be that they will need to be in shared memory which will increase contention. The tradeoffs will need to be explored.
- Design of parallel algorithms which can utilize coherency in an efficient manner. This will be particularly important for efficient rendering of embedded geometric primitives.
- Extensions to the algorithms to handle curvilinear and finite element datasets.

7 Acknowledgements

Special thanks to Nelson Max for the use of his code implementing the projection algorithm [MHC90] and for discussions and ideas on the techniques for parallelization of the projection method. Allen Van Gelder, Kim Taylor, Jane Wilhelms, and Scott Whitman contributed insightful commentary and encouragement. Thanks also to the staff of the Massively Parallel Computing Initiative at Lawrence Livermore National Laboratories for use of the machine and for their timely response to questions and problems.

its last task, until all processors have completed their last tasks. The load imbalance is seen to affect the speedup of the scan-line-per-task approach for large n (see fig. 6.3).

- **Task time**: for the scan-line-per-task approach, this is the sum of the time each processor spends rendering a scan line. For the pixel-per-task approach, this is the sum of the time each processor spends rendering pixels. For the projection method, this is the sum of the time each processor spends projecting cells to the image, excluding the explicit synchronization required for the use of the ready list and visibility graph.
- **Get cell synchronization**: for the projection method, this is the sum of the time each processor spends getting a cell to render off the ready list.
- **Update graph synchronization**: for the projection method, this is the sum of the time each processor spends updating the visibility graph after rendering a cell, including possibly moving additional cells to the ready list.
- **Task overhead**: for all methods, the sum of time for each processor in which that processor was not in one of the above states. This is primarily the time required for task generation, but may be affected by other unknown operating system inefficiencies. In particular, it can be seen in figure 6.4 that the GenOnA() task generator is extremely inefficient for large n.

6.3 Conclusions

The parallel algorithm for the projection method given in this paper does not scale well with the number of processors. The synchronization requirements for the proper ordering of cell rendering generate a significant amount of overhead as n increases. The main inefficiencies for the scan-line-per-task approach for the raycasting method are load imbalance for large n, and the penalty for using globally shared memory. The number of tasks, and therefore task length, has a significant effect on scalability as can be seen by the inefficiencies in task generation for raycasting with a pixel per task.

The results of this research indicate that the raycasting method has some advantages for parallelization on this type of architecture. In particular, rays (pixels) can be processed completely independently with no ordering constraints. However, it is crucial that rays be grouped together to form a single task as needed for efficient processor utilization.

The projection method, on the other hand, does have ordering constraints which complicate task generation. The techniques used in this project for the parallelization of the projection approach require large amounts of memory in order to store the visibility graph, the use of locks around critical sections of code, and atomic operations. There are other ways the parallelization of the projection method could be approached. For instance, several cells at a time could be rendered per task. The determination of cell groupings to form tasks and the ordering of these tasks is an open problem. In addition, these algorithms may be difficult to generalize to more complex meshes.

Raycasting using a task consisting of rendering an entire scan line shows promise for achieving interactive rendering speeds on large datasets. For the volumetric dataset that was benchmarked, the serial rendering time of a little over 9 minutes was reduced to 12 seconds using 100 processors. A modified algorithm which makes better use of local memory may result in code that will perform at interactive rates. In addition, the benchmarked times measured were for the first execution which includes the propagation of data structures and code to each processor. Interactive use will typically involve a loop of small changes to data

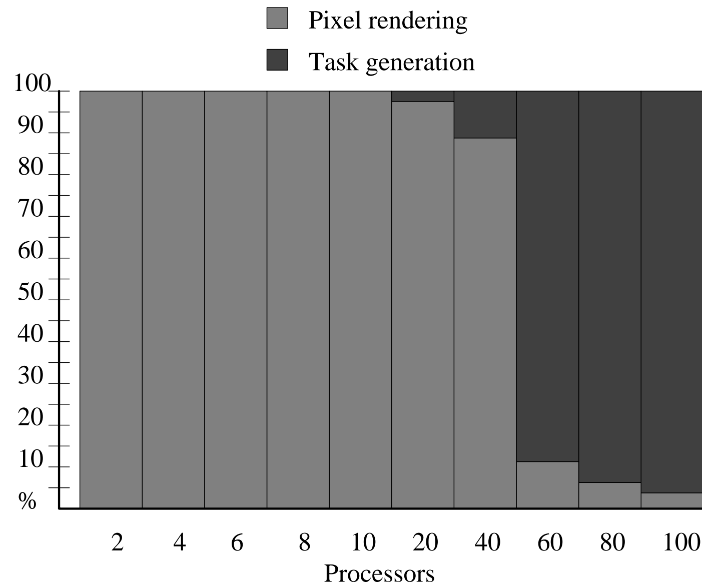

Figure 6.4: Execution profile - raycasting with one pixel per task.

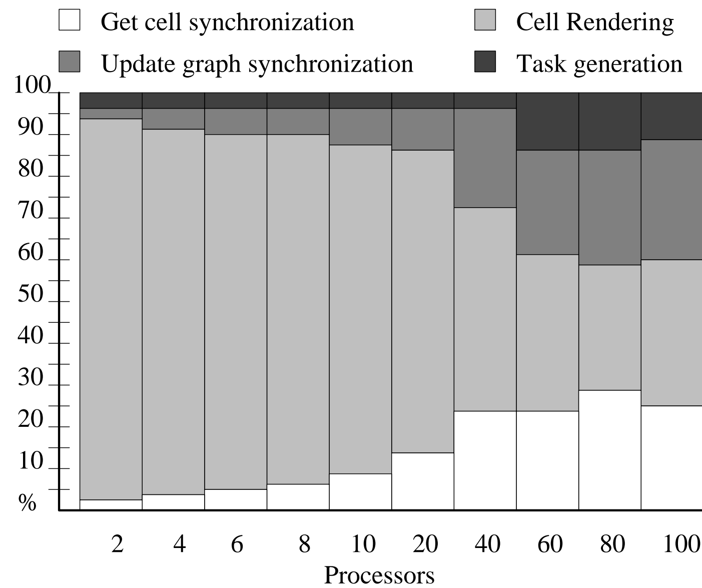

Figure 6.5: Execution profile - projection with one cell per task.

for the scan-line-per-task approach. The measurements collected include:

- **Startup time**: the sum of the time each processor spends getting started, that is from the time the code goes parallel to the first execution of a task. It is during this time that data structures which are to be propagated to each processors local memory are moved. The startup time was found to be negligible for all three approaches, but may become an important factor as the rendering approaches interactive speeds.
- **Load imbalance**: the sum of the time each processor spends waiting after finishing

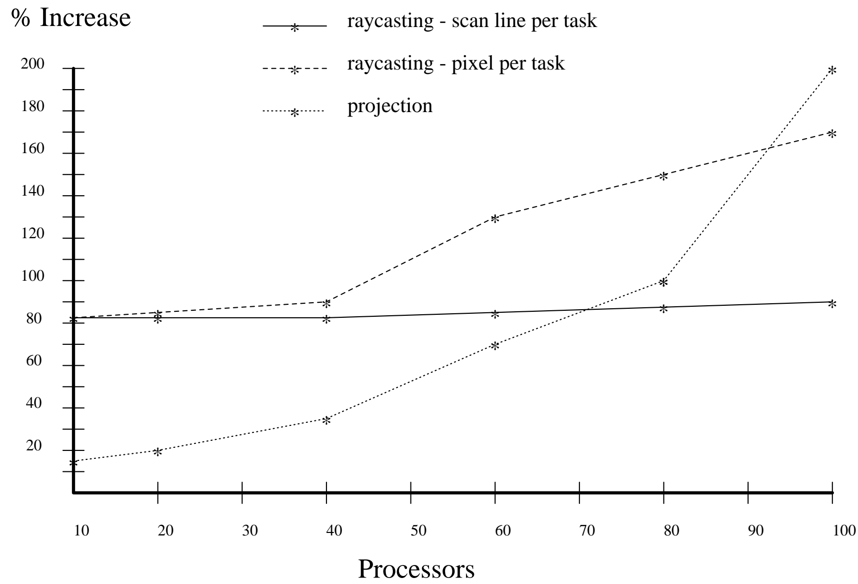

Figure 6.2: Percent increase in average task size.

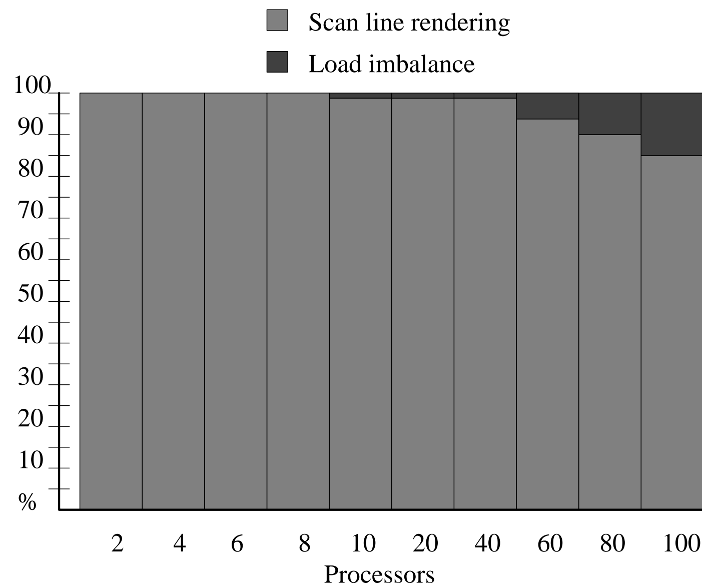

Figure 6.3: Execution profile - raycasting with one scan line per task.

6.2 Execution Profiles

Figures 6.3, 6.4, and 6.5, illustrate the execution profiles of the three approaches for various numbers of processors. All measurements are given as a percentage of the total processor time which is defined as the number of processors multiplied by the time spent between going parallel and all processors completing their tasks. Only the most significant measurements are shown. States which account for under 2% of the total processing time are not shown. Thus, task generation does not appear in figure 6.3 because it is negligible

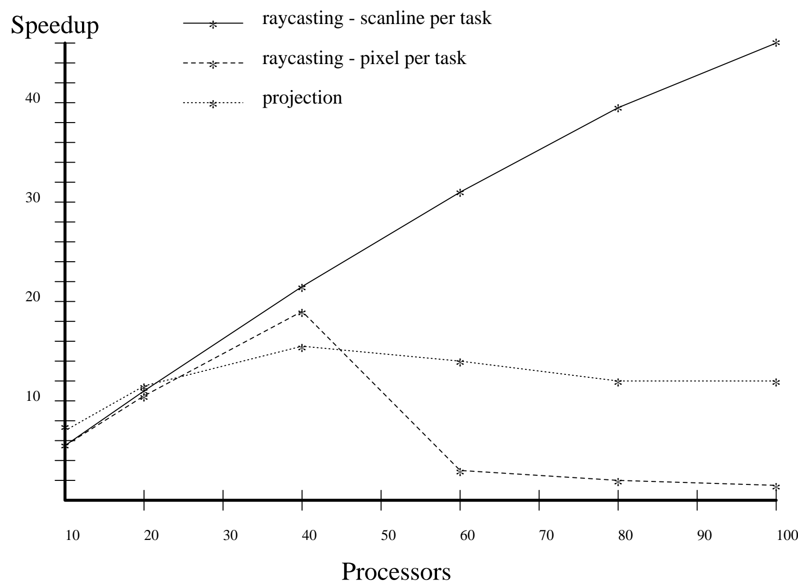

Figure 6.1: Speedup graphs.

Only the portion of the algorithm which renders the image has been benchmarked. The time required to initialize the Uniform System, read a volumetric object in and initialize it, and output the resulting image to a file or display device, is not included in these speedup measurements. All measurements have been averaged over three runs. Not much variation was seen between runs.

Although the results and analysis presented here are for one volume, similar behavior has been seen with other datasets. For the projection method, all three viewing cases were benchmarked with similar results. No significant performance differences were seen between the three different viewing cases.

The volumetric dataset used to produce these benchmarks was a 100x120x16 electron density map for Staphylococcus Aureus Ribonuclease contributed by Dr. Chris Hill of the University of York. The dataset was scaled by 5 to produce a 512x512 image. This viewing configuration generated 512 tasks for the scan-line-per-task approach; 262,144 tasks for the pixel-per-task approach; and 176,715 tasks for the projection approach. Figure 6.1 shows the speedup graphs for all three approaches. T_1 is about 550 seconds for the raycasting algorithms and 313 seconds for the projection approach. Figure 6.2 gives the percentage increase in the average length of the tasks. For the scan-line-per-task approach, a task is rendering one scan line. For the pixel-per-task approach, a task is rendering one pixel. For the projection approach, a task is rendering one cell. It can be seen that at 10 processors, there is an increase in task length of 80% for the raycasting approaches and 6% for the projection approach. It is not completely clear what factors influence this, but the amount and locality of global memory accesses influencing efficient use of the processor caching mechanism may play a large part. For all methods, the average task length increases with the number of processors, reflecting the increasing memory and switch contention. This contention is exacerbated by a small task length (many tasks), probably due to numerous accesses for task generation.

The viewing case is easily determined by the number of back-facing faces. One back-facing face corresponds to viewing case one in figure 5.1, two back-facing faces to viewing case two, and three back-facing faces to viewing case three. During an initialization phase each cell is initialized with a count of the number of cells it directly obscures. One or more cells which are farthest from the viewpoint will initially have count=0. After the counts are initialized, the ready list is initialized to include those cells with count=0.

The ready list and visibility graph are updated using atomic operations by the parallel tasks as each cell is rendered. After a cell is rendered the counts of the neighbors which directly obscure it are decremented. When a count becomes 0, the cell is added to the ready list. Depending on the view, from one to three counts will be decremented and checked for zero. Decrementing a count is an atomic operation, as is the addition of cells to the ready list. The ordering on the cells that is enforced by this process will ensure that the compositing operations on each pixel in the image will be appropriately ordered.

Summary of Parallel Projection Algorithm

The task generator GenOnI() is used to generate one task for each cell in the volumetric dataset. Each task executes the following operations:

- Lock the ready list and attempt to remove a cell for processing. If one is not available, release the ready list, wait for a short period of time, and retry.
- Render the cell that was obtained from the ready list [MHC90].
- Update the visibility graph using an atomic decrement. If this updating process identifies cells that are ready to be added to the ready list, then that list will need to be locked since additions to the list must also be atomic. Once updating is completed the processor is free to acquire a new cell for rendering.

Memory Management for Projection

Memory for the projection approach is managed in the same way as for the raycasting approach, with the addition of the two extra data structures that are required for task ordering:

- The visibility graph is scattered in shared memory in the same way as the scalar volume. The array of pointers to the scattered visibility graph is propagated to each processor's local memory.
- The ready list is located in shared memory.

6 Results

6.1 Measured Speedup

The measured speedup reported in this section has been calculated as T_1/T_n where T_1 is the time for the algorithm to execute on a single processor using all local memory, and T_n is the time for the algorithm to execute on n processors.

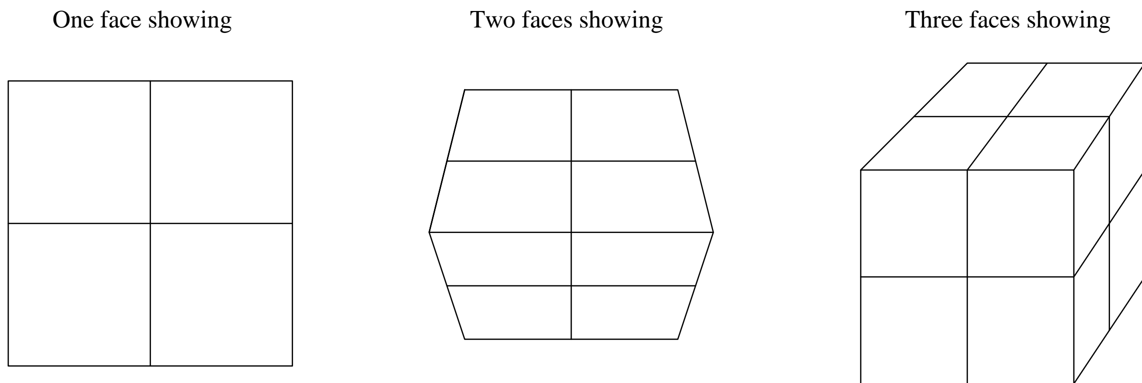

Figure 5.1: Three viewing cases affecting parallelism in the projection method.

5.2 Parallelization of the Projection Method

Processor Management for Projection

Task management for the projection algorithm is complicated by the fact that the compositing operations for each pixel must be ordered. There are three viewing cases which will affect the number of cells that are available to be rendered in parallel (fig. 5.1):

- One face visible: Initially an entire plane of cells will be available for rendering in parallel. Each cell rendered will affect the visibility of one other cell.
- Two faces visible: One row of cells will initially be available for rendering in parallel. Each cell that is rendered will affect the visibility of two other cells.
- Three faces visible: Initially there will be only one cell available for rendering. Each cell rendered will affect the visibility of three other cells.

For the projection method two additional data structures are kept in order to drive the parallelism:

- The *ready list* is a list of cells which are currently available for rendering. Each task removes a single cell from this list for rendering.
- The *visibility graph* indicates when a given cell can be transferred to the ready list. Cells are rendered from back to front. A cell can be rendered when all cells which it obscures have been rendered. For each cell in the volume, a count is kept of the number of cells which are directly obscured by the given cell (i.e. are adjacent and behind the cell). When this count reaches zero, the cell may be transferred to the ready list.

For the special case of a rectilinear mesh, all that is required in the visibility graph are the counts of obscured cells. The identity of the obscured cells is inherent in the structure of the mesh. In a more general mesh, more information will need to be kept in the visibility graph. In particular, pointers to obscured cells will be required, and the initialization of the visibility graph will be much more complex. It is important to note that the visibility graph depends on the viewpoint and thus must be modified to reflect changes in the viewing specifications.

5 Parallel Volume Rendering

In order to experiment with various parallel volume rendering algorithms, an object-oriented volume renderer [Cha90] has been ported to the BBN TC2000. This volume renderer initially used the raycasting algorithm as the method for rendering an image, so the code has been extended to provide the projection approach as another rendering method [MHC90]. The serial version of this volume renderer provides a platform upon which different parallel algorithms can be experimented with. Three primary issues involved in the parallelization of the rendering algorithms are discussed here. They include task generation for the raycasting and projection methods, and memory management. After discussing these issues, results and conclusions will be presented. All of the approaches presented here utilize a parallel projection for viewing, although the methods are extensible to perspective projections.

5.1 Parallelization of the Raycasting Method

Processor Management for Raycasting

Two approaches have been implemented for parallel rendering using the raycasting method:

- Using GenOnI() a task is generated for each scan line of the image.
- Using GenOnA() a task is generated for each pixel of the image.

Memory Management for Raycasting

Data structures have been allocated in such a way as to minimize the memory and switch contention. Scattering data across memories has been done whenever possible for large data structures, and as much information as possible is kept in local memory. Data structures in shared memory include:

- The scalar data volume is scattered across the globally shared memory of the processors. The data structure is scattered by Z planes with each processor storing one or more planes of constant Z.
- The image memory is scattered across the globally shared memory. Each processor stores one or more scan lines of the image.

Data structures in process private memory:

- The description of the volume to be rendered, including such things as its dimensions, the array of pointers to the Z planes of the shared scalar volume, transformation matrix, color and opacity lookup tables, etc.
- The description of the image to be created including its dimensions and the array of pointers to the locations of the scan lines in globally shared memory.
- Description of world characteristics such as defined light sources, viewing specifications, etc.

4 The BBN TC2000

The machine used in this project is a BBN TC2000 located at the Massively Parallel Computing Initiative at Lawrence Livermore National Laboratories. This particular machine is configured with 128 processors and 2GB of main memory. In this section an overview is given of the important hardware and software features of this machine.

The BBN TC2000 is a multiprocessor architecture with a distributed shared memory [BBN89]. The TC2000 processors access the shared memory through an interconnection network called the Butterfly switch. The architecture is modular and scalable and can be configured to contain between 1 and 512 function boards.

A software library called the *Uniform System* is provided by BBN for controlling the parallel execution and memory use of an application. This library supplies functions for memory and processor management that are callable from C, Fortran, or C++. The goal and design philosophy of the Uniform System is to provide functionality to an application program in such a way that the full bandwidth, both memory and processor, of the machine is utilized. For a full description of the Uniform System the reader is referred to the TC2000 documentation [BBN88].

The Uniform System implements a large virtual address space that is shared by all processors. This approach allows the programmer to treat all processors as identical workers. Each processor has two kinds of memory at its disposal. Process private or local memory is used for storage of all global or static variables, the heap, and the stack. Globally shared memory is made available and managed through the use of Uniform System functions. Functions are provided to allocate shared memory, to scatter large data structures across several memories of the machine, and to propagate or copy process private data between local memories of different processors.

Processors are treated as a group of identical workers. Applications are structured into two parts: functions which may perform the various application tasks in parallel, and one or more task generation functions which specify the next task for execution. Several basic task generators are supplied by the Uniform System, or the application may provide a specialized one. Claimed benefits of this approach include:

- The generator mechanism is very efficient. It is implemented in one process per processor with each processor executing a tight loop of generate task - execute task with no context switches.
- A program written this way is insensitive to the number of processors.
- The load is balanced dynamically.

Many task generators, both synchronous and asynchronous, are provided. The two used in this project are:

- GenOnI(worker, range)

  Generates tasks of the form

  worker(0, i)

  for $0 \leq i < range$.

- GenOnA(worker, range1, range2)

  Generates tasks of the form

  worker(0, i1, i2)

  for $0 \leq i1 < range1$ and $0 \leq i2 < range2$.

The most useful visualization tools allow a researcher to interactively explore a dataset. One intent of this research is to determine whether it is possible to reach interactive rendering speeds for volume rendering through the use of parallel processing. Both the raycasting and projection approaches to volume rendering seem to be amenable to parallelization. In particular, this research addresses the following questions:

- How to parallelize the two volume rendering approaches?
- What kind of speedup is attained? What are the inefficiencies?

Two goals which will guide the choice of algorithms include:

- Algorithms should be scalable with respect to number of processors, volume size, and image size.
- Algorithms should be extensible to more complex meshes.

3 Related Work

Previous efforts to achieve interactive rates for rendering of voxel-based objects through the use of parallel architectures come primarily from the medical imaging and solid modeling communities. Specialized architectures have been developed to speed rendering in those applications. A survey of some of these systems has been done by Kaufman [KBCY90]. Many of these systems were initially designed for rendering solid models and were later applied to medical imaging. In most cases, the voxels that are visible are also opaque. These architectures do not handle the semi-transparent volumetric models and rendering techniques described in the previous section. The specialized architectures that have been proposed or developed for solid modeling applications include: the **3DP^4** [OUT85], **Insight** [Mea85], the **PARCUM II** [Jac88], and the **RayCasting Engine** [EKM$^+$91]. Architectures designed for medical imaging applications include: the **Voxel Processor** [GRB$^+$85], and the **Cube** [KB88]. Levoy has proposed a parallelization of the raycasting approach on the **Pixel-Planes 5**, a parallel architecture developed for computer graphics [FPE$^+$89, Lev89a].

Related efforts in the parallelization of computer graphics algorithms are surveyed by Burke and Leler [BL90] and Crow [Cro90]. Fuchs proposed a technique for distributing the z-buffer hidden surface removal algorithm over a distributed-memory MIMD architecture [Fuc77]. Cleary, et al. give an algorithm for ray tracing which utilizes a world-space decomposition on a distributed-memory MIMD system [CWBV83]. The rays are represented as messages passed between the processors. Dippé and Swensen extend this approach to achieve better load balancing by using an adaptive world-space decomposition [DS84]. Parallelization of global illumination algorithms in a distributed workstation environment has been addressed by Tampieri and Greenberg [TG88]. Parallelization of the ray tracing algorithm on a distributed-memory MIMD architecture using an image-space decomposition is presented by Badouel [BP90]. This algorithm depends on an implementation of shared virtual memory with local caching. Whitman explores several image-space decompositions and scheduling strategies on a shared-memory MIMD machine [Whi91]. A detailed analysis of the overhead incurred in the parallelization is presented.

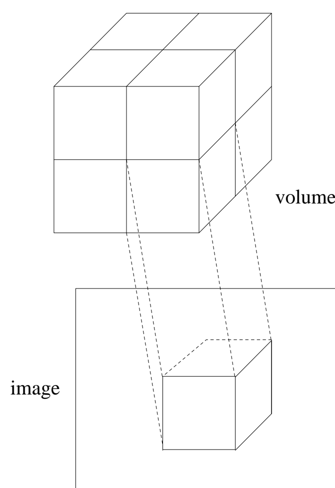

Figure 2.3: Volume rendering using the *projection* algorithm.

any order.

2.2 Rendering by Projection

A *cell* consists of eight neighboring scalar values in the volumetric dataset which form the corners of a hexahedron. *Projection* is an object-space algorithm in which each volume cell is sampled to give its contribution to every pixel in the image onto which the cell projects [UK88,Wes89,Wes90,MHC90,WG91]. For a given cell we are essentially integrating across the depth of the cell to give a contribution at each pixel the cell affects. An illustration of this process is given in figure 2.3.

In this algorithm, the primary loop is through the cells of the volumetric dataset. Each cell may be projected to the image independently, however compositing operations from cells which project to the same pixel must be ordered. The compositing algorithm used may specify either front-to-back or back-to-front ordering.

2.3 Why Parallel Volume Rendering?

Volume rendering is a computationally intensive process. A rectilinear volumetric dataset may consist of 256^3 samples, or 16MB of data if the scalar data values are bytes. The time required to render an image from such a volume will typically be something like several minutes to an hour, depending on which of the existing algorithms are used and the desired image resolution and quality.

Volumetric datasets which are arranged on a regular rectilinear lattice are of the simplest form. More general lattices are commonly used by researchers to generate volumetric datasets. The lattice spacing may be irregular, as in some computational fluid dynamics applications where portions of the space need to be sampled/computed at a higher resolution than others. Or the mesh may be curved to match the simulation geometry. Computational meshes in $\Re^3$ made up of tetrahedral or hexahedral elements that have been shaped are also common in computational fluid dynamics and finite element analysis applications [Fle88, ZT89].

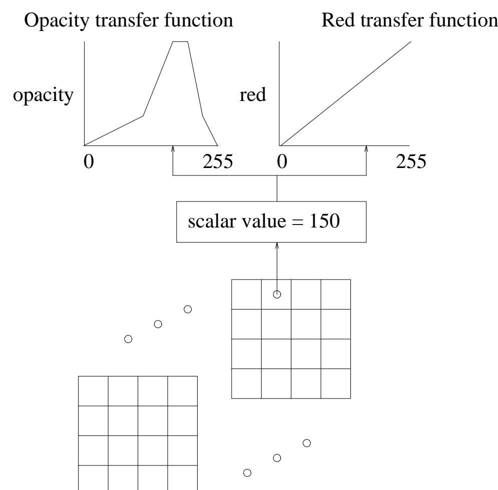

Figure 2.1: Volume rendering using transfer functions.

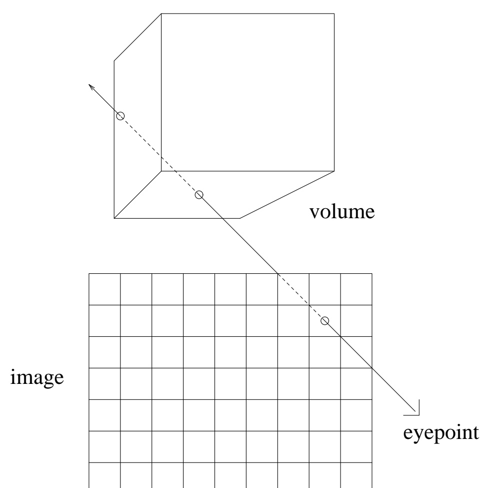

Figure 2.2: Volume rendering using the *raycasting* algorithm.

must then be composited or accumulated into the image [PD84]. There are currently two algorithms which are widely used for sampling the entire volume, raycasting and projection.

2.1 Rendering by Raycasting

Raycasting is an image-space algorithm which involves casting a ray from the viewpoint through each image pixel and testing for intersection with the volume [UK88,Lev89b,Cha90]. If a ray intersects the volume, the contents of the volume along the ray are sampled and composited and the resulting value is taken as the pixel contents. In essence, we are integrating along the length of the ray that is passing through the volume of data. This process is illustrated in figure 2.2.

In this algorithm, the primary loop is through the image pixels. Each ray may be processed completely independently from any other ray, and the rays may be processed in

1 Introduction

A *volumetric dataset* is a collection of scalar data in which each datum has an associated (usually implicit) location in three-dimensional space ($\Re^3$). *Volume rendering* is a powerful, but computationally intensive, computer graphics technique for visualizing volumetric datasets. This paper presents results of investigations into approaches for volume rendering using parallel processing on a shared-memory MIMD architecture.

Volume rendering has been shown to be a useful technique for visualizing the results of scientific computations. One motivation for this research is to provide a powerful interactive tool for the visual analysis of the results of simulations. In this context, it is desirable for the analysis tool (volume renderer) to be made available on the same machine as the simulation is running on. This will facilitate closer coupling of scientific simulations and visualization tools in the future, ultimately leading to the ability to interactively steer scientific computations based on visual feedback. As the trend towards putting large scientific simulation applications on massively parallel machines continues, it is important that similar research is conducted on the parallelization of visualization techniques.

Volume rendering is introduced in section 2, with specific attention to the raycasting and projection approaches currently in use. The intent of this research is to compare the requirements, approaches, and speedup attained by the parallelization of these two algorithms. A brief survey of related work is given in section 3, followed by an overview of the hardware and software features of the BBN TC2000, a shared-memory MIMD machine used for this project (section 4). Specific approaches for the parallelization of the two volume rendering algorithms are described in section 5. Results in the form of speedup measurements and execution profiles are presented in section 6, followed by a discussion of ongoing research.

2 Volume Rendering

Volume rendering refers to the process of generating an image from a volumetric dataset. The dataset may have been sampled or computed over one of many distributions in $\Re^3$. In its simplest form, the dataset is arranged in regular intervals on a rectilinear lattice or mesh. An example of this might be many magnetic resonance images stacked together in memory to create a three-dimensional array of data. Each image sample has an associated implicit location in $\Re^3$. The research presented in this paper is based on volumetric datasets which are arranged on a regular rectilinear lattice.

How can such an entity be visualized so as to allow internal structure to be seen while preventing loss of information? This question has led researchers to a technique for rendering volumetric datasets called direct volume rendering [DCH88,UK88,Sab88,Lev89b]. Using this technique, the entire volume of data can be rendered using various levels of transparency related to the scalar data values in order to see the interior structure. There are many algorithms for accomplishing this; figure 2.1 shows how lookup tables (commonly called transfer functions) can be used to obtain color and opacity values for a specific scalar data value in the array [UK88]. In general, the sample points desired within the data array will not lie directly on those sample points that are given. An interpolation method will be employed to generate the scalar data value at the desired sample point.

Rendering the volumetric dataset involves sampling the entire dataset in some fashion and mapping each sampled value to a color and opacity. The contribution from each sample

Parallel Volume Rendering on a Shared-Memory Multiprocessor

Judy Challinger

UCSC-CRL-91-23
Revised
March 1992

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Volume rendering is a powerful, but computationally intensive, computer graphics technique for visualizing volumetric datasets. This paper presents results of investigations into techniques for volume rendering using parallel processing on a shared-memory, multiple-instruction, multiple-data (MIMD) architecture. In particular, two widely used algorithms for volume rendering, *raycasting* and *projection*, have been parallelized on a BBN TC2000, and their performance has been measured and analyzed. Preliminary results indicate that the raycasting approach to volume rendering has some advantages for parallelization on this type of architecture.

Keywords: volume rendering, parallel, shared memory, multiprocessor, computer graphics, scientific visualization