

## INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

# UMI

A Bell & Howell Information Company  
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA  
313/761-4700 800/521-0600



EXPLOITING COHERENCY IN PARALLEL ALGORITHMS FOR  
VOLUME RENDERING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for  
the Degree Doctor of Philosophy in the Graduate  
School of the Ohio State University

By

Asish Law, B.Tech., M.S.

\* \* \* \* \*

The Ohio State University

1996

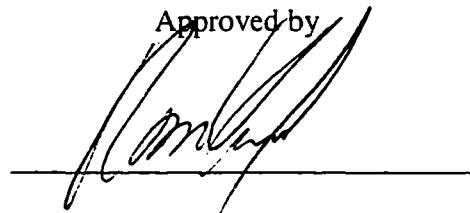
Dissertation Committee:

Roni Yagel, Adviser

Richard Parent

Dhabaleswar Panda

Approved by

A handwritten signature in black ink, appearing to read 'Roni Yagel', is written over a solid horizontal line.

Adviser

Dept of Computer and Information Science

**UMI Number: 9710599**

---

**UMI Microform 9710599  
Copyright 1997, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized  
copying under Title 17, United States Code.**

---

**UMI**  
300 North Zeeb Road  
Ann Arbor, MI 48103

*Copyright by*  
*Asish Law*  
*1996*

## ABSTRACT

Volume visualization has emerged as a prominent off-shoot of graphics for viewing and manipulating scientific datasets, such as those obtained from MRI, CT-scans, and CFD, or volumes which are generated by voxelizing geometric models. Direct volume rendering has the mechanisms for allowing simple manipulation techniques and easy viewing of the inside of objects. However, the size of these volumes tends to be so large that even the most powerful uniprocessor machines are unable to provide the much desired interactivity in such environments.

In this dissertation, we have designed and implemented three scalable parallel volume rendering algorithms on the Cray T3D. Our methods suggest new paradigms and alternatives to traditional ways of parallel rendering in general, and parallel volume rendering in particular. They are designed in such a way that the complete local memory is used only for that data which are required by a processor. Most earlier algorithms made use of the hardware cache only for exploiting spatial coherency; the performance of these algorithms are prone to degrade with increasing dataset sizes or decreasing cache sizes. Utilizing the local memory as another level of cache, *i.e.* software cache, was not explored for parallel rendering. In addition, ways to hide latency and minimize network congestion in each of these algorithms are novel approaches in themselves. In addition, previous algorithms are not applicable to render colossal datasets (possibly in compressed form), as they are prone to thrashing. We propose a new algorithm that combines the advantages of both the object-order algorithms and image-order algorithms to solve the problem of thrashing and provide the most coherent screen traversal scheme.

In summary, this research has primarily focussed on some of the as yet unexplored problems in parallel volume rendering, e.g., latency hiding, optimal local memory utilization, optimal screen traversal, reducing network congestion, and portability, and has suggested exclusive ways to eliminate some or all of these. Our coherent algorithms demonstrate scalability to a very high degree, with potential to improve even further. With the advent of high-resolution scanners, the dataset sizes approach the limits of disk storage. The coherent algorithms developed in this dissertation will provide state-of-the-art methods for visualizing such large datasets in the future.

**To My Family**



## ACKNOWLEDGMENTS

I would like to express sincere appreciation to Dr. Roni Yagel for his guidance and insight throughout the research. Dr. D.N. Jayasimha's critical comments during the initial stages gave a solid building block to this research. Thanks to the other members of my advisory committee, Drs. R. Parent and D. Panda, for their valuable suggestions and comments. I am very grateful to the staff at the Ohio Supercomputer Center, especially Tim Rojmajzl and Al Stutz, without whose help this work would not have been possible. The support and recommendations provided by all the members of our Volume Visualization group have helped me improve the quality of my research and dissertation. And finally, to my wife, Anandi, I offer sincere thanks for your unshakable faith in me and your willingness to endure with me the vicissitudes of my endeavors.

## VITA

March 25, 1965	Born - Calcutta, India
1988	B.Tech. (Hons.), Indian Institute of Technology, Kharagpur, India
1991	M.S., Biomedical Engineering, The Ohio State University
1994	M.S., Computer and Information Science, The Ohio State University

## PUBLICATIONS

1. R. Yagel, D. Reed, A. Law, P.W. Shih, N. Shareef, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing", *Proceedings IEEE Symposium on Volume Visualization 96*, San Francisco, October 1996, pp. 55-62.
2. A. Law, R. Yagel, "The Active-Ray Approach to Rendering on Distributed Memory Multiprocessors", *Proceedings IEEE Eighth Symposium of Parallel and Distributed Processing, SPDP 96*, New Orleans, October 1996, pp. 414-421.
3. A. Law, R. Yagel, D.N. Jayasimha, "Parallel Volume Rendering for Scientific Visualization", *ISCA Journal of Computers and Their Applications*, Vol. 3, No. 3, December 1996.
4. A. Law, R. Yagel, "An Optimal Ray Traversal Scheme for Visualizing Colossal Medical Volumes", *Proceedings of Visualization in Biomedical Computing VBC 96*, Hamburg, Germany, September 1996, pp. 33-42, Karl H. Hoehne, Ron Kikinis (ed.), Springer, 1996. Lecture notes in computer Science Vol. 1131.
5. A. Law, R. Yagel, "Exploiting Spatial, Ray, and Frame Coherency for Efficient Parallel Volume Rendering", *Proceedings of GraphiCon 96, The 6th International Conference on Computer Graphics and Visualization in Russia*, St. Petersburg, Russia, July 1996, pp. 93-101.
6. A. Law, R. Yagel, "Multi-Frame Thrashless Ray Casting with Advancing Ray-Front", *Proceedings of Graphics Interface, GI 96*, Toronto, Canada, May 1996, pp. 70-77.

7. R. Yagel, D. Stredney, G. Wiet, A. Law, "PARAVOL: Parallel Volume Rendering for Virtual Medicine". The Cray User Group Meeting, Fairbanks, Alaska, Sept 1995. pp. 131-138.
8. A. Law, R. Yagel, "CellFlow: A Parallel Rendering Scheme for Distributed Memory Architectures", Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 95, Athens, Georgia, November 1995. pp. 3-12.
9. A. Law, R. Yagel, D.N. Jayasimha, "VoxelFlow: A Parallel Volume Rendering Method for Scientific Visualization", Proceedings of the ISCA Conference of Computer Application in Engineering and Medicine, Indianapolis, Indiana, March 1995, pp. 260-264.
10. A. Law, R. Yagel. "Voxel-Based Morphing". Tech. Report # OSU-CISRC-4/93-TR15.

## FIELDS OF STUDY

Major Field: Computer and Information Science

Field of Specialization: Computer Graphics

Minor Field: Parallel Computing

Minor Field: Algorithms

## TABLE OF CONTENTS

	<u>Page</u>
Abstract .....	.ii
Dedication .....	.iv
Acknowledgments .....	.v
Vita.....	.vi
List of Tables .....	.xi
List of Figures .....	.xii
Chapters:	
1. INTRODUCTION .....	1
1.1. Volume Rendering .....	3
1.2. Acceleration Techniques for Volume Rendering .....	6
1.2.1. Software Based Acceleration .....	7
1.2.2. Hardware Based Acceleration .....	9
1.2.3. Acceleration Via Indirect Volume Rendering .....	10
1.2.4. Acceleration With Low-Quality Images .....	11
1.2.5. Acceleration With Parallel Processing .....	11
1.3. Problem Statement .....	12
1.3.1. Motivation .....	12
1.3.2. Issues in Parallel Rendering .....	13
1.4. Overview of the Dissertation .....	14
1.4.1. Exploiting Temporal Coherency .....	14
1.4.2. Exploiting Spatial Coherency .....	14
1.4.3. Exploiting Volume Coherency .....	15
1.5. Summary of Contributions and Significance .....	15

<b>2. PARALLEL VOLUME RENDERING ALGORITHMS</b> .....	18
2.1. Types of Parallelism .....	19
2.1.1. Image Partition .....	19
2.1.2. Object Partition .....	20
2.1.3. Hybrid .....	22
2.1.4. Shear-Scale-Warp .....	22
2.2. Issues in Parallel Volume Rendering .....	23
2.2.1. Architecture .....	23
2.2.2. Embedding Topology .....	25
2.2.3. Scalability .....	28
2.2.4. Data Partitioning .....	30
2.2.5. Partition Distribution .....	31
2.2.6. Load Balancing .....	33
2.2.7. Coherency .....	34
2.2.8. Latency Hiding .....	36
2.2.9. Network Congestion .....	37
2.2.10. Portability .....	38
2.3. Summary of the Issues in PVR Algorithms .....	38
2.3.1. Summary of the Features of PVR Algorithms .....	42
2.3.2. Summary of Performance of PVR Algorithms .....	45
2.4. Open Issues .....	48
<b>3. EXPLOITING TEMPORAL COHERENCY: THE CellFlow ALGORITHM</b> .....	50
3.1. CellFlow: The General Method .....	51
3.2. Design Issues .....	53
3.2.1. Screen and Scene Subdivision .....	53
3.2.2. Load Balancing .....	54
3.2.3. Algorithm Embedding .....	54
3.2.4. Network Congestion .....	54
3.2.5. Memory Mapping .....	54
3.2.6. Latency Hiding .....	55
3.3. Volume Rendering Incremental Rotation .....	55
3.4. Implementation Details .....	58
3.5. Results .....	58
3.5.1. Scene and Screen Description .....	59
3.5.2. Load Balancing .....	61
3.5.3. Overheads .....	61
3.5.4. Scalability .....	63
3.5.5. Effect of PRA .....	65
3.6. Discussion and Future Work .....	65
3.7. Conclusion .....	67
<b>4. EXPLOITING SPATIAL COHERENCY: THE ActiveRay ALGORITHM</b> .....	68
4.1. Distributed Memory Implementation of Parallel Rendering .....	68
4.2. Illumination Model for Active Ray Tracing .....	73
4.3. Design Issues .....	76
4.3.1. Load Balancing .....	76
4.3.2. Network Congestion .....	76
4.3.3. Latency Hiding .....	76

4.3.4. Memory Overheads .....	77
4.3.5. Deadlock Avoidance .....	77
4.4. Results .....	78
4.4.1. Load Balancing .....	78
4.4.2. Effect of Irregular Animation .....	78
4.4.3. Effect of Software-Cache Size .....	80
4.4.4. Effect of Cell Size .....	80
4.4.5. Scalability .....	82
4.5. Discussion and Future Work .....	84
4.6. Conclusion .....	85
5. EXPLOITING VOLUME COHERENCY: THE RayFront ALGORITHM .....	86
5.1. Exploiting Coherency for Efficient Rendering .....	87
5.2. Method .....	88
5.2.1. Screen and Scene Subdivision .....	89
5.2.2. Preprocessing .....	89
5.2.3. Ray Casting .....	89
5.3. Results .....	95
5.3.1. Number of Frames per Phase .....	95
5.3.2. Comparison .....	96
5.3.3. Load Balance .....	98
5.3.4. Scalability .....	99
5.4. Discussion and Future Work .....	99
5.5. Conclusion .....	100
6. THE UNIPROCESSOR RayFront ALGORITHM FOR VISUALIZING COLOSSAL MEDICAL VOL- UMES .....	102
6.1. Introduction .....	102
6.2. Method .....	104
6.2.1. Multi-Frame Thrashless Volume Rendering Revisited .....	105
6.2.2. Enhancements .....	106
6.2.3. Extension to Arbitrary Frame Animation .....	106
6.3. Results .....	108
6.3.1. Timings for Non-Compressed Volumes .....	108
6.3.2. Effect of Cell Size .....	109
6.3.3. Cost of Overheads .....	109
6.3.4. Simultaneous Multi-Frame Rendering of Compressed Volumes .....	112
6.4. Discussion .....	113
7. CONCLUSIONS .....	114
7.1. CellFlow vs ActiveRay vs RayFront .....	117
7.2. Cray T3D vs Convex SPP .....	120
7.3. Cluster of Workstations (COWs) .....	123
7.4. Extensions and Future Research .....	124
7.5. Conclusion .....	130
List of References .....	131

## LIST OF TABLES

Table	Page
2.1.	Summary of the features of the PVR algorithms (continued on next page).....43
2.2.	Summary of performance of the PVR algorithms (continued on next page).....46
3.1.	Description of various volumes and respective screen sizes .....59
3.2.	Frame rendering times for various volumes as a function of number of processors. All times are in seconds. ....64
4.1.	Rendering times as a function of cell size. ....80
4.2.	Frame rendering times for various volumes as a function of number of processors. All times are in seconds. ....83
5.1.	The first 5 passes of the ray-casting algorithm with advancing ray-front for the example shown in Figure 5.5.....94
5.2.	Frame rendering times for various volumes as a function of number of processors. All times are in seconds. ....101
6.1.	The volumes, along with reading, rendering, and total times (in seconds). ....109
6.2.	Total times (reading + rendering) in seconds, as a function of Cell Size. The minimum times are shown in bold. .... 111
6.3.	Degradation factor of our algorithm as compared to a normal ray-caster..... 111
6.4.	Average rendering times (decompressing + reading + rendering) in seconds, as a function of number of frames in a phase. Total number of frames = 20 ..... 111
7.1.	Summary of the features and performance of the CellFlow, ActiveRay, and RayFront algorithms described in Chapter 3, Chapter 4, and Chapter 5, respectively. (continued on next page)..... 115
7.2.	Comparison of frame times (in seconds) for the ActiveRay algorithm on Cray T3D and Convex SPP..... 121
7.3.	Comparison of frame times (in seconds) for the RayFront algorithm on Cray T3D and Convex SPP..... 121
7.4.	Frame times (in seconds) for the ActiveRay algorithm on a cluster of DEC-alpha workstations, with Ethernet and FDDI connections. .... 123
7.5.	Frame times (in seconds) for the RayFront algorithm on a cluster of DEC-alpha workstations, with Ethernet and FDDI connections. .... 123

## LIST OF FIGURES

Figure	Page
1.1.	A 3D volume consisting of 8×6×5 voxels. Each voxel is a single unit in each dimension. The volume is rendered onto an 8×8 screen. A ray is shot from each screen pixel. Only one ray is shown in the figure. The circles along the ray shows the sampling points of the volume along the ray. ....5
1.2.	An example of forward projection rendering. Each voxel is projected onto the screen using a transformation matrix (T), and its contribution is combined with all the pixels it affects. Only one voxel projection is shown in the figure, and the screen area it effects is shown by the checkered pattern. ....5
2.1.	Bus-based configuration of the network, connecting the processors and the shared memory. Each node is provided with a processor and a hardware cache. ....26
2.2.	Different topologies: (a) 4×4 2D mesh, (b) 4×4×3 3D mesh, (c) 15-node full binary tree, (d) 16-node (24) hypercube, (e)8-node ring. ....27
2.3.	A 4-processor multistage interconnection network (MIN) as used in Omega.....29
2.4.	Distributed memory architecture for PVR algorithms.....29
2.5.	Volume partitions: (a) slices, (b) slabs, (c) shafts, (d) cells. ....32
2.6.	Summary of the methods and issues arising in the design of PVR algorithms. (a) methods used for PVR, (b) volume partition schemes, (c) image or screen partition schemes, (d) partition distribution methods, (e) load balancing schemes, (f) types of coherency exploitation, (g) ways to hide latency, (h) network topologies, Summary of the methods and issues arising in the design of PVR algorithms. (i) communication patterns, (j) architectures, (k) programming models. (Continued on next two pages) .....39
3.1	Object space is divided into cells. If all dark cells are locally available, the screen region R can be rendered from viewpoint A without any communication, but not from point B. If padding is also locally available (light grey cells) rendering from A, B and many other points in between does not require any communication. Some top white cells may be needed when the viewer moves to point C. ....52
3.2	A set of 2D objects being projected on a 1D image plane, which is divided into 4 disjoint parts and assigned to 4 processors (P1, P2, P3, P4). The object data is distributed in such a way that the ith slab of data is sufficient to generate the final image for that processor without communication. ....52
3.3	(a) When the viewpoint moves from the initial screen position to the final position the grey area contains the data elements that need to be brought in from other processors, the striped area can be discarded, and the black area the data that need not be moved. (b) If the processor contains all the information in the shaded area then all viewpoints in the 25° range can be accommodated without any communication.....57



3.4	An object of size $16 \times 16$ is divided into cells of size $4 \times 4$ each. Each cell thus contains 16 voxels. The top-left cell is extruded into a 3D cell, which contains $4 \times 16 \times 4$ voxels. ....	57
3.5	(a) SOD128, (b) Simple128, (c) Head256, (d) Capsid256 and Capsid512.....	60
3.6	Frame times for (a) for Simple128 and (b) for Capsid256 volumes on 8 processors, and with cyclicity varying from 1 to 4.....	62
3.7	(a) Average Sending, Receiving, and Postprocessing overheads for each frame with a fixed PRA of 10 ( $P=8, C=4$ ). (b) Average Sending, Receiving, and Postprocessing overheads with varying PRA ( $P=8, C=4$ ).....	62
3.8	The speedups of the parallel incremental rotation algorithm for different number of processors.....	64
3.9	(a) Average number of cells sent, received, and retained per phase of rotation ( $P=8, C=4$ ). The topmost curve is the total number of cells in memory at a time, which is the sum of the number of cells retained and the number of cells received. The bottom curve shows both the number of sends and receives. (b) Fraction of cells received at the end of each phase relative to the number of cells retained. ....	66
4.1.	(a) A volume made up of $32 \times 24 \times 15$ voxels is divided into $8 \times 6 \times 5$ cells each of size $4 \times 4 \times 3$ voxels. Each processor is home to 60 random cells in a 4-processor system. (b) A screen divided into 64 tiles of equal size and distributed cyclically to 4 processors, P1, P2, P3, and P4. For example, the black cells and the dark tiles are assigned to P1.....	70
4.2.	(a) A 3-hop system for requesting data (cells) from other processors. R is the requesting processor, H the home node, and D is the closest processor containing the requested cell. (b) A 2-hop invalidation process for discarding a cell from a processor's memory.....	70
4.3.	Average times spent by each processor for rendering each frame as a function of tile size. (a) for simple128, and (b) for capsid256 scenes.....	79
4.4.	Communication overhead when the eye 'jumps' after every ten frames, compared to a smooth animation.....	79
4.5.	Rendering time as a function of software-cache size on 16 processors. The total amount of cache in all sixteen processors is equal to $CS \times \text{volume size}$ ( $CS = \text{cache size}$ ). ....	81
4.6.	(a) Number of iterations through the ray list as a function of software-cache size while rendering twenty frames. (b) Average number of cells received by each processor as a function of software-cache size. ....	81
4.7.	Speedup results for rendering several volumes on 1 to 128 processors. ....	83
5.1.	(a) A volume made up of $32 \times 24 \times 15$ voxels is divided into $8 \times 6 \times 5$ cells each of size $4 \times 4 \times 3$ voxels. Each processor is home to 60 random cells in a 4-processor system. (b) A screen divided into 64 tiles of equal size and distributed cyclically to 4 processors, P1, P2, P3, and P4. For example, the black cells and the dark tiles are assigned to P1.....	89
5.2.	Ray-casting algorithm with advancing ray-front. ....	91
5.3.	Illustration of the linked list data structure used for efficiently advancing only certain rays through a cell. The example shows that there are 3 rays entering cell [10,12,4], they are 42, 27, and 10.....	93
5.4.	The modified ray-casting algorithm with advancing ray-front. ....	93
5.5.	An example of advancing ray-front with 11 rays. The figure shows the advancement of the ray-front for the first 5 passes only. The 2D object space is divided into cells, and the numbers in each cell indicate its position in the FTBL. ....	94
5.6.	For all viewing positions in region I, and when viewed towards the center of the volume, the FTB order of the cells are as shown. $\times$ denotes the center of the volume. ....	95
5.7.	(a) Times and (b) Number of cells received with number of frames generated in each phase of the algorithm for generating 30 frames. ....	97

5.8.	Comparison of four different screen traversal schemes - scan-line, spiral, hilbert, and rayfront. The graph shows the times taken for generating 30 frames in the animation.....	97
5.9.	Time spent by each processor for rendering 30 frames as a function of tile size, (a) for simple128, and (b) for capsid256 volumes. ....	98
5.10.	Speedups exhibited of the ray-front algorithm for different volumes. ....	101
6.1.	For all viewing positions in region I, and when viewed towards the center of the volume, the FTB order of the cells are as shown. × denotes the center of the volume. A ray is also shown which traverses cells 13, 19, 20, 26, 27, 33, 34, in order. ....	105
6.2.	(a) An FTB order for eye in the left bottom side (red arrow) that requires the maintenance of arbitrary number of ray segments for some other eye positions (black arrow). (b) An FTB order that requires at most two ray segments for any ray orientation. ....	107
6.3.	(a) Effect of cell size for different datasets. (b) A closer look at the lower left part of the graph. ....	110
6.4.	Speedups achieved for different volumes as FPP is varied between 1 and 20. Total number of frames remains the same at 20. ....	112
7.1.	Graphs comparing the speedups of the CellFlow, ActiveRay, and RayFront algorithms on four datasets. (a) simple128, (b) sod128, (c) head256, and (d) capsid256. ....	118
7.2.	Graphs comparing the speedups of four datasets on Cray T3D and Convex SPP, (a) simple128, (b) sod128, (c) head256, and (d) capsid256. ....	122

## CHAPTER 1

### INTRODUCTION

The field of 3D graphics has become very prevalent in the past few years for comprehension and manipulation of computer simulated 3D scenes. The realistic images generated in graphics come at an enormous computational expense; an anti-aliased image with reflections, refractions, shadows, and texture mapping of a moderately complex scene taking minutes or even hours to generate. Furthermore, the size and complexity of such 3D scenes has grown more rapidly than the computing power of uniprocessor machines.

One important source of large 3D scenes is the volumetric datasets. Volume visualization has emerged as a prominent off-shoot of graphics for viewing and manipulating scientific datasets, such as those obtained from Magnetic Resonance Imaging (MRI), Computed Tomography (CT-scans), and Computational Fluid Dynamics (CFD), or volumes which are generated by voxelizing geometric models. Unlike surface-based graphics, direct volume rendering has the mechanisms for allowing simple manipulation techniques (like cutting, sculpting, etc.) and easy viewing of the inside of objects (for transparent objects). However, the size of these volumes tends to be several magnitudes larger than that used in surface graphics. Even the most powerful uniprocessor machines are unable to provide the much desired interactivity in such environments.

In this dissertation, we have designed and implemented three coherent algorithms on the Cray T3D, which have proven to be extremely scalable. These algorithms are primarily based on efficient utilization of local memory and attempt to hide the latency of locally unavailable objects. They suggest new paradigms and alternatives to traditional ways of parallel rendering. Optimization of local memory usage has largely been neglected in the past, as previous algorithms have concentrated on small to medium sized datasets. Our algorithms are designed in such a way that the complete local memory is used only for that data which are required by a processor. These methods present the most efficient memory usage paradigms. Most earlier algorithms made use of the hardware cache only for exploiting coherency; the performance of these algorithms are prone to degrade with increasing dataset sizes or decreasing cache sizes. Utilizing the local mem-

ory as another level of cache, i.e. the software cache, was not explored for parallel rendering. In addition, ways to hide latency and minimize network congestion in each of these algorithms are novel approaches in themselves.

Designing a parallel algorithm to render colossal datasets (e.g., the Visible Human) needs a complete paradigm shift. Previous algorithms are no longer applicable to render such large datasets (possibly in compressed form), as they are prone to thrashing, a phenomenon which should be avoided at all costs. A new algorithm is sought that will provide the required efficiency and remove the problem of thrashing for rendering colossal datasets. The RayFront algorithm combines the advantages of both the object-order algorithms (e.g., no thrashing, regularity of access, object space coherency) and image-order algorithms (e.g., opacity clipping, better image quality, simplicity, and usage of other acceleration techniques) to solve the problem of thrashing and provide the most coherent screen traversal scheme.

In summary, this research has primarily focussed on some of the as yet unexplored problems in parallel volume rendering, e.g., latency hiding, optimal local memory utilization, optimal screen traversal, reducing network congestion, and portability, and has suggested exclusive ways to eliminate some or all of these. Our coherent algorithms have demonstrated scalability to very high degrees, with potential to improve even further. With the advent of high-resolution scanners, the dataset sizes approach the limits of disk storage. The coherent algorithms developed in this dissertation will provide state-of-the-art methods for visualizing such large datasets in the future.

In this chapter, we start with a brief introduction to volume rendering, along with the various schemes used to accelerate this time consuming process. Multiprocessing machines provide a viable platform for efficiently gaining speedup to render large volumes. The crux of this dissertation deals with the design of efficient parallel algorithms for volume rendering. Later in this chapter, we provide the issues involved in designing parallel rendering algorithms, and the motivation behind this research. It should be stated that although we have implemented our algorithms for volumes, they are equally applicable to polygonal based models as well. In the following discussion, objects may thus refer to voxels (in the case of volumetric models), or to polygonal objects (in case of polygonal models).

In the following chapter (Chapter 2), we have done a comprehensive survey of existing parallel volume rendering methods. The survey reveals some of the as yet unexplored issues, and we have attempted to provide efficient solutions to overcome these drawbacks. Chapter 3, Chapter 4, and Chapter 5 discuss these methods

in sufficient detail, while Chapter 6 describes a classical uniprocessor application of one of the methods (the RayFront in Chapter 5).

## 1.1 Volume Rendering

In the past few years, volume visualization has emerged as a powerful technique for the representation, manipulation, and rendering of volume data. Unlike traditional graphics techniques, which represent 3D objects as geometric surfaces and edges commonly approximated by polygons and lines, volume data are 3D entities that may have information inside them. On one hand, these volumetric entities might not consist of surfaces and edges, and on the other, they may be too voluminous to be represented geometrically. Volume visualization techniques provide the mechanisms that make it possible to reveal and explore the inner or unseen structures of volumetric data and allow visual insight into transparent and complex datasets.

A *Volume* is a regular 3D grid of *voxels*. A voxel is the 3D equivalent of the 2D pixel. Figure 1.1 shows a volume made up of  $8 \times 6 \times 5$  voxels, each of size  $1 \times 1 \times 1$ . Each voxel is characterized by its position in the 3D grid, and may have associated with it a color and opacity. Medical data obtained from MRI (magnetic resonance imaging) and CT-scanners (computed-tomography) act as good sources for volume visualization applications. This technique may also be applied to studies in CFD (computational fluid dynamics) where powerful computers simulate natural phenomena in 3D. Volume representation is very effective for other applications in which the acquired data are in an inherently volumetric form. These applications include biology (e.g., confocal microscopy), geoscience (e.g., seismic measurements), industry (e.g., inspection), meteorology, molecular systems (e.g., electron density maps), and 3D image processing (e.g., time varying 2D images). A comprehensive treatment of the field of volume visualization can be found in [53].

In one approach to volume rendering (Figure 1.1) [75][110], rays are cast into the volume through the screen pixels. 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 (FTB) until either the composited opacity becomes larger than some pre-specified threshold value, or until the ray exits the volume. The composited color of the ray is the final color of the screen pixel. This approach to volume rendering is referred to as *backward projection*, *image order*, or *ray casting*. Ray casting was first adopted by Tuy and Tuy for rendering binary volumes [110], and then extended to multi-valued ray-casting by Levoy [75][76], Upson and Keeler [112], and Sabella [97].

The other popular approach to volume rendering is the *forward projection* suggested by Frieder, et al. [33], and later improved to splatting by Westover [116][117]. In this approach (Figure 1.2), each voxel is projected on the screen, and the final color of a pixel is the accumulated effect of all the voxels which project on this pixel. This approach is also known as *object order*. Examples of forward projection volume rendering of binary volumes can be found in [33] and [40].

A third approach to volume rendering adopts the advantages of both object-order and image-order algorithms, and is known as *hybrid projection*. Herman and Liu [47] used this approach for binary volumes, whereas Drebin, et al. [28], Upson and Keeler [112], and Wilhelms and van Gelder [122] used it for rendering multi-valued volumes. Wittenbrink and Somani [126] used their permutation warp to implement a hybrid method on SIMD parallel machines.

Generating a reasonable quality image of a volume is extremely computation and memory intensive. For each pixel on the screen, a ray is cast into the volume and samples are taken at regular intervals along the ray. In order to generate an image of reasonable quality, approximately  $O(N^3)$  samples have to be generated and composited, where  $N^3$  is the total number of voxels in the volume. This problem is even more exacerbated when higher quality images are desired, as in anti-aliasing. Anti-aliasing can be done by any of the following methods:

1. supersampling the pixels in the image lattice, by shooting multiple rays from each pixel.
2. supersampling the volume, by taking closer samples along the ray.
3. applying higher-order filters, e.g., tri-linear or cubic interpolation filters, during volume sampling.
4. applying filters to the pixels in the image lattice.

Applying any of the above anti-aliasing methods involves a several-fold increase in computation time. In general, the sampling rate along the ray is kept at the smallest distance required to obey Shannon's sampling theorem. The theorem's requirement that the sampling rate be above the Nyquist rate implies that  $t \leq 0.5 \times \text{voxel\_size}$ , where  $t$  is the distance between adjacent ray origins in screen space and distance between samples along a ray. Incorporating illumination effects, shadows, and texture mapping on the fly demands even more computation power. All these factors contribute to increase the image generation time of reasonable sized volumes ( $256^3$  or  $512^3$  volumes) to minutes or even hours.

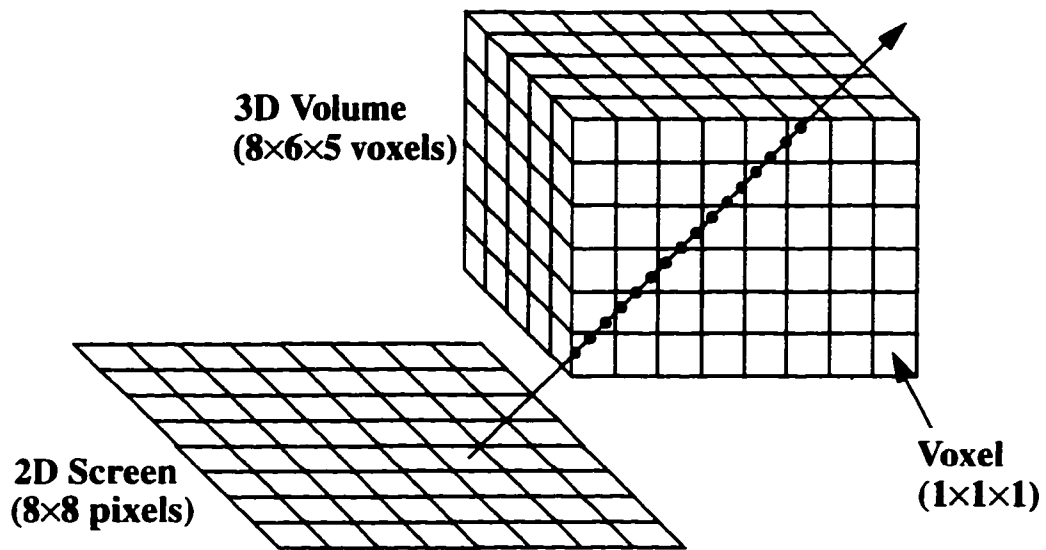


Figure 1.1. A 3D volume consisting of  $8 \times 6 \times 5$  voxels. Each voxel is a single unit in each dimension. The volume is rendered onto an  $8 \times 8$  screen. A ray is shot from each screen pixel. Only one ray is shown in the figure. The circles along the ray shows the sampling points of the volume along the ray.

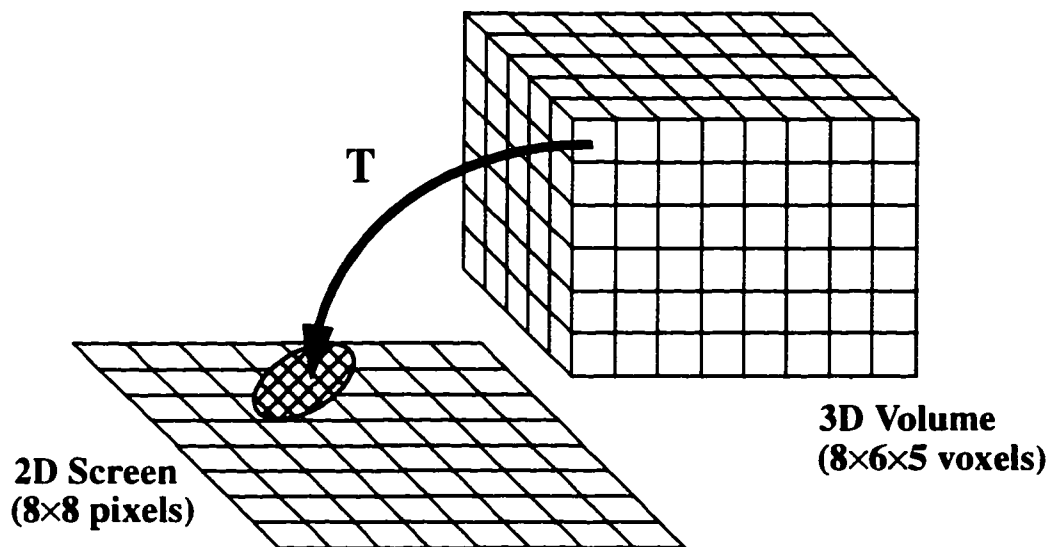


Figure 1.2. An example of forward projection rendering. Each voxel is projected onto the screen using a transformation matrix ( $T$ ), and its contribution is combined with all the pixels it affects. Only one voxel projection is shown in the figure, and the screen area it effects is shown by the checkered pattern.

## 1.2 Acceleration Techniques for Volume Rendering

Brute force volume rendering, as illustrated above, involves enormous computational effort for quality rendering of volumes with medium to high resolutions. For example, consider a volume of resolution  $N \times N \times N$  used for generating an image of size  $M \times M$ . For forward projection algorithms, four steps are involved in generating the image: traversal, transformation, rasterization, and display. Let us assume that on an average, a voxel projects on  $k$  screen pixels. Thus the brute-force volume rendering process will need  $f_t \times N^3$  operations for transforming all the voxels in the volume, where  $f_t$  is the number of operations needed for transforming each voxel. In addition,  $f_c \times k \times N^3$  operations are needed to composite the effects of all the voxels, where  $f_c$  is the number of operations needed for compositing with each pixel. This gives the total rendering time for object-order methods as  $(f_t + k \times f_c) \times N^3$ . In case of volume rendering using ray-casting, let us assume that on an average there are  $n$  samples along each ray, where  $n$  will be dependent on the volume resolution,  $N$ . Let  $f_s$  be the number of operations spent on acquiring and compositing each sample. This gives the total rendering time for ray-casting algorithms as  $n \times f_s \times M^2$ .

The number of operations,  $k$ ,  $f_c$ , and  $f_s$ , used above depends on the quality of rendering desired, and also on the features included, like illumination. To determine the enormous computational power needed by the rendering process, let us consider generating a  $512^2$  image of a  $256^3$  volume using ray-casting with zero-order interpolation, and with 256 samples along each ray. Each sample requires at least  $3 \times 3 = 9$  floating point calculations (for all the three color channels, rgb). Generating images at real-time rates require the frame generation time to be less than 0.03 secs, implying  $256 \times 9 \times 512^2$  FP operations in 0.03 secs. This poses a demand for more than  $20 \times 10^9$  FLOPS! This figure will surely escalate by several orders of magnitude, when additional features like illumination, shadowing, or texture mapping, and higher order interpolation filters, like trilinear or cubic, are introduced for generating images of better quality. Object-order methods for volume rendering also demand similar machine power. Clearly, even the fastest uniprocessor machines currently available are unable to provide such high computational power.

As a result, much research has been devoted to reducing the time required for generating images of volumes. Software and hardware solutions that expedite the volume rendering process are collectively referred to as *acceleration techniques*. Some of these techniques have been borrowed from those used for traditional ray-tracing of polygonal objects, e.g., using bounding boxes. Others are applicable to volume rendering only. Apart from software and hardware based acceleration methods, indirect volume rendering (e.g., isosurf-



ing) and generating approximate images may also be regarded as ways to speed up the rendering process. Last, but not the least, parallel computers provide an efficient environment to obtain raw computation power for attaining real-time rendering of large volume datasets. Although the focus of this work is acceleration by parallelism, we briefly survey other approaches also.

### 1.2.1 Software Based Acceleration

Software based acceleration techniques are those that either resort to efficient ways of calculations (e.g., by avoiding floating-point calculations), or exploit some form of coherency (e.g., inter-ray coherency). In the case of forward projection, each voxel has to be first transformed to image space and then projected onto the screen. Its effect on each pixel is then composited to the already available value for that pixel. The inherent ordering of the voxels within the volume precludes the necessity to sort them in front-to-back fashion.

A popular method for accelerating the process of ray casting involves the use of *octrees*. Octrees are hierarchical spatial decomposition graphs, and can be efficiently organized to skip empty sub-volumes since they will not contribute to the final image. Levoy [76] used octrees for efficient ray-casting of volumes. Octrees were also used by Laur and Hanrahan [64], where each node in the octree has an error approximation associated with it. A node's children are splat only if the representation error of their parent is more than a threshold. Goldwasser [39] adopts a similar *divide-and-conquer* approach to avoid transforming each voxel in the volume. The volume is organized in an octree structure, with similar voxels grouped into an octree leaf node. The aggregate of voxels within a node are then treated as a single entity and transformation and compositing calculations are applied to them as a whole.

Some acceleration techniques involve savings during transformation of voxels by avoiding repetitive floating point operations. The *table driven transformation* method adopted by Frieder, et al. [33] stores all the multiplication calculations required for the transformation matrix multiplication into a look-up table (LUT). The transformation of a voxel can then be accomplished simply by accessing the LUT, the entry accessed in the table depending on the xyz coordinates of the voxel. Machiraju and Yagel [83] exploit coherency within the volume to implement a novel *incremental transformation* scheme. A seed voxel is first transformed using the normal matrix-vector multiplication. All other voxels are then transformed in an incremental manner with just three extra additions per coordinate.

Forward projection volume rendering can be accelerated by first transforming the volume from voxel space to pixel space by employing a decomposition of the 3D affine transformation into five 1D *shearing transformations* [44]. Each of these 1D transformations require only one floating point addition. The transformed voxel is projected onto the screen in a FTB order. Lacroute and Levoy [62] use a similar approach. Their *shear warp* method involves a warping step where the slices of the volume are sheared and then warped so that all the rays are then parallel to one of the major axes. The transformed voxels then become aligned with one of the major axes, alleviating the task of resampling. Vezina, et al. [113] has also adopted the shear/scale operations for implementation of volume rendering on SIMD MPPs like MasPar MP-1, and Lacroute [63] has implemented his shear-warp algorithm on the SGI Power Challenge.

Backward projection volume rendering methods can be also be accelerated by either exploiting some form of coherency, or by resorting to efficient ways of ray calculations. Yagel [129] has proposed several efficient methods for accelerated ray-casting of volumes. For example, Yagel, et al. [130] use discrete versions of rays for ray-casting of volumes. A 26-connected ray is used for traversing most of the empty spaces. As the ray comes close to occupied voxels, it is changed to a 6-connected ray. Yagel and Kaufman's [131] *template based* approach also achieves good speedup for parallel projection ray casting. In this approach, a ray-template is formed, which is used to generate an image for all the pixels on a base plane. This intermediate image is then projected to form the final image.

The most popular acceleration methods rely on spatial coherency of volumes to avoid sampling of empty spaces. Bounding boxes have been used in the past for expediting the process of ray tracing. This method can also be used in case of volume rendering. The object is surrounded by a tightly fit box. Rays are first intersected with the bounding box and start their actual volume traversal from this intersection point rather than the volume boundary. Rays that do not hit any bounding box are simply assigned the background color. Avila, et al. [8] strives to have a better fit by allowing a convex polyhedral envelope to be constructed around the object.

Yagel and Shi [132] introduced a method called *space leaping* for skipping over empty spaces in the volume. The method is not useful for rendering a single frame, but temporal coherency between slowly changing frames is exploited to determine the empty spaces that can be skipped. During rendering of a frame, the closest voxel encountered for each ray is recorded. These voxel locations are then transformed according to the position of the next frame. When the next frame is being generated, the position of the transformed voxels

are used to skip empty spaces. Information about empty spaces may also be recorded during a preprocessing step. For example, Cohen and Shefer [20] used *proximity clouds* to skip over empty spaces in the volume. With each voxel in the volume is associated the closest distance to an occupied voxel (in any direction). While sampling a voxel, the coded distance can be safely skipped along the ray before another occupied voxel can be encountered. In a contemporary work, Sramek [104] proposed a similar idea as proximity clouds, but called his metric as *chessboard distance*. Zuiderveld, et al. [137] present an efficient technique, using *distance transforms*, for computing a volume that contains the radial distance from each point in the data volume to the closest “interesting” point.

From results reported in published articles, software based acceleration techniques have gone a long way in reducing the rendering times for manipulation and interpretation of volumes. Achieving real-time rendering rates, though, is still a far-fetched goal. As processor speed doubles every year, so does the size of the volumes to be rendered. Hardware solutions become imperative under such situations. Another cost effective way for achieving real-time rates would be to combine the software-based acceleration techniques with general purpose parallel computers, which offer teraflops of computation power. In the following section, we see some ways by which available polygon rendering hardware can be utilized for volume rendering, and in the following chapter, we will focus on several ways to take advantage of massively parallel processing machines for achieving high frame rates.

### **1.2.2 Hardware Based Acceleration**

The hardware based polygon renderers, like [2][84][34] are often utilized to render volumes as well. For example, if sufficient texture memory is available, the volume is considered as solid texture and stored in the texture memory. A number of planes, each parallel to the image plane, are used to slice the volume at regular intervals. The texture-mapped planes (polygons) are then composited to form the final image. Cullip and Neumann [25], and Cabral, et al. [15] use the texture memory available on high-performance graphics workstations to render volumes. In another approach, the effect of each voxel can be considered to take the form of a polygon (a splat [117]). Each splat is transformed (in hardware), and the individual splats are then composited (in hardware) in a front-to-back or back-to-front manner. Yagel, et al. [134], Laur and Hanrahan [64], and Wilhelms and van Gelder [123] use this approach to utilize the graphics hardware.

Although these hardware based methods are extremely fast, a compromise has to be made with the quality of the image. Hardware based methods are also limited by the flexibility of the rendering models, like using different illumination models, incorporating realism like refractions, reflections, and shadows.

Although there are only a handful of commercially available hardware volume renderers (e.g., Pixar/Vicom Image Computer and Pixar/Vicom II [107]), several research prototypes are under investigation. Kaufman, et al. [54] and Stytz, et al. [107] have done a survey of various hardware implementations of volume renderers. In the next chapter, we will survey some specific hardware volume rendering implementations that demonstrate a close resemblance with popular parallel rendering algorithms.

### 1.2.3 Acceleration Via Indirect Volume Rendering

As an alternative to the direct volume rendering techniques (e.g., ray-casting and splatting), the volume data can be first converted into geometric primitives in a process called *surface-tracking*. In this process, the user specifies a seed voxel. Surface tracking begins with at the seed location, and traces the surface of the object in either depth-first or breadth-first manner. Only one connected surface can be tracked with this method. Several boundary detection methods proposed by Liu [77], Artzy [6], and Udapa [107][111] used surface tracking using a user-specified seed.

In a similar approach, called *isosurfacing*, all the voxels in the volume are visited without any user intervention. A test is conducted at each voxel to determine whether the specified surface passes through it: the voxel is approximated by a polygon in case the surface does pass through it. The geometric primitives (e.g., polygon mesh, contours), resulting from either surface tracking or isosurfacing are then rendered to the screen by conventional geometric rendering. The advantage of such an approach is that once the polygon mesh for the isosurface is generated (as a preprocessing step), available graphics hardware may be utilized. Isosurfacing also involves reduction in data size. On the other hand, surface extraction is associated with loss of inside information of the volume. Whenever a different isosurface is required, the polygon mesh generation algorithm has to be invoked. Lorensen and Cline [79], Livnat, et al. [78], Wilhelms and van Gelder [123], Shen and Johnson [101], and Itoh and Koyamada [50] are only a few authors who have proposed novel isosurfacing techniques for regular and irregular volume grids.

### 1.2.4 Acceleration With Low-Quality Images

Frequently, the quality of images is compromised for gaining rendering speed. *Progressive refinement* is the technique of producing “rough” images at high frame rates when a user is modifying view parameters and progressively better images when user input ceases [13]. With a ray casting renderer, a rough image is computed quickly by *undersampling* the image lattice. Undersampling can be done either by regular but sparse sampling in all the three dimensions, or by raising the threshold on the variance allowed in adaptive sampling, or by lowering the alpha cutoff threshold of accumulated opacity. Laur and Hanrahan’s hierarchical splatting method [64] can also be accelerated by raising the error approximation threshold for generating lower quality images.

Most shear-warp and shear-scale algorithms, like 3-pass affine transforms, also involve fast but slightly inaccurate image generation techniques. Only Wittenbrink [126] claims to have a more accurate image generation scheme than other object order shearing algorithms. Wu and Brady [128] offers an approximate computation method for generating volume rendered images used for animation. Their method takes advantage of ray-to-ray coherency to store partial results from earlier frames to generate an image for the current frame.

### 1.2.5 Acceleration With Parallel Processing

For generating high-quality images, not compromising flexibility at the same time, software renderers are still the order of the day. However, the size of the data may sometimes be too colossal to fit in a single computer memory and the time required for rendering can be unacceptable. Some software rendering methods have been able to achieve interactive frame rates only for low resolution volumes [62] but, in general, they compromise with flexibility. For example, they assume that illumination calculations are already applied to the volume during a preprocessing step, and several copies of the volume exist simultaneously in memory. Due to these reasons, it often becomes infeasible to apply direct volume visualization techniques for generating numerous images for animation on a uniprocessor machine. Parallel computers become indispensable in such situations. The various approaches taken to parallelize the volume rendering process and the issues lying therein are surveyed in the next chapter.

## 1.3 Problem Statement

### 1.3.1 Motivation

The field of 3D graphics has become very prevalent in the past few years for comprehension and manipulation of computer simulated 3D scenes. Various illumination models have been proposed to provide lighting effects and realism to the generated images. These images may also be enhanced using anti-aliasing techniques, and complex surface details can be handled using various texturing schemes. The inclusion of each of these features in the rendering algorithms comes at an enormous computational expense; a realistic image with reflections, refractions, and shadows of a moderately complex scene taking minutes or even hours to generate. Furthermore, the size and complexity of such 3D scenes has grown more rapidly than the computing power of uniprocessor machines.

One important source of large 3D scenes is the volumetric datasets. Volume visualization has emerged as a prominent off-shoot of graphics for viewing and manipulating scientific datasets, such as those obtained from Magnetic Resonance Imaging (MRI), Computed Tomography (CT-scans), and Computational Fluid Dynamics (CFD), or volumes which are generated by voxelizing geometric models. Volumes can be seen as a set of sampled points in a 3D spatial grid, each sample indicating scalar, vector, or tensor elements. For example, the value may stand for temperature or stress for datasets obtained from CFD or FEM (Finite Element Method) sources, or it may indicate density values when obtained from medical sources. Unlike surface-based graphics, direct volume rendering has the mechanisms for allowing simple manipulation techniques (like cutting, sculpting, etc.) and easy viewing of the inside of objects (for transparent objects). However, the size of these volumes tends to be several magnitudes larger than that used in surface graphics. e.g., the Visible Human dataset from the National Library of Medicine [1] is 40 GB in size. Even the most powerful uniprocessor machines are unable to provide the much desired interactivity in these environments.

General purpose parallel computers and dedicated parallel graphics accelerators have become indispensable in such situations. Several parallel rendering algorithms have been proposed for different architectures, each with its mechanisms to remove the overheads associated with the parallelization process. Although interactive display (5-10 frames/sec.) of moderate sized volumes are now becoming feasible, real time animation (30 frames/sec.) of high resolution volumes still remains a far-fetched goal.

In this dissertation, we have taken a leap towards achieving the goal of real-time animation of large volumes. We take a closer look at some of the important factors that have a detrimental effect on a parallel renderer's performance. Our survey of existing PVR algorithms surfaces new bottlenecks and we have proposed alternate parallel rendering methods that make efficient use of the local memory, incorporate effective latency hiding, and minimize network congestion. Results show that our system has the potential to perform better than other existing parallel rendering systems for a variety of scientific datasets. Our implementations also demonstrate scalability superior to existing parallel rendering algorithms.

### 1.3.2 Issues in Parallel Rendering

Designing parallel rendering algorithms poses some challenging problems. First, we note that this class of problems falls under the purview of unstructured or irregular parallel algorithms, where the pattern of data communication between processors is unpredictable. The most challenging of all the problems is to design an algorithm that minimizes the overheads associated with the parallelization process, e.g., the communication overhead. *Data partitioning*, *load balancing*, *coherency exploitation*, *latency hiding*, and *portability* are the most important issues that a parallel rendering algorithm designer faces for efficient utilization of the resources. All these factors combine to affect the *scalability* of the algorithm, which provides an idea about the amount of speedup gained by using a larger number of processors. Ideally one would expect to speed up an algorithm by a factor that equals the number of processors (linear speedup). Because of the associated overheads, parallel rendering algorithms almost never achieve this ideal situation, although there is a continuing effort to bring the scalability as close as possible to this goal.

Most previously proposed algorithms have focused on data distribution, load balancing, and coherency exploitation, but have overlooked the effect of communication latency, which proves to be a bottleneck in improving scalability. Proposals for efficient local memory utilization is absent from literature, and the effect of screen traversal has largely been neglected. Designing algorithms to run most efficiently on a particular machine has also made them less portable to machines with different or heterogeneous topologies. Our methods take all these factors into account. It provides algorithms with an optimal screen traversal scheme, ample latency hiding, efficient utilization of local memory, and good static load balancing. In addition, our algorithms are independent of the network topology, and they demonstrate high scalability with respect to data and machine sizes.

## 1.4 Overview of the Dissertation

We have designed and implemented three coherent algorithms on the Cray T3D [Chapter 3, Chapter 4, and Chapter 5], which have proven to be extremely scalable, with scope for further improvement. These algorithms are primarily based on efficient utilization of local memory and attempt to hide the latency of locally unavailable objects. The irregularity and unpredictability of the problem at hand makes the incorporation of these features all the more difficult. The issues underlying the design of parallel rendering algorithms are discussed in detail in the next chapter.

### 1.4.1 Exploiting Temporal Coherency

In the first phase of our research [Chapter 3], we looked at data distribution, memory allocation, and load balancing issues. We determined that instead of statically allocating parts of the volume to each processor, the local memory utilization could be optimized if *only* the required data were made locally available, that is, data which will be needed by a processor to generate its assigned portion of the screen. Depending on the local memory availability, this data allocation scheme can be extended to provide what is termed as *frame-to-frame coherence*, where data are allocated in such a way that a processor is self-sufficient to generate a number of slowly changing frames [65][66][67] with no need for additional communication. If the animator happens to have the knowledge of subsequent screen positions, then the data needed for the next set of frames can similarly be fetched. This allows for effective overlapping of computation and communication (latency hiding) as well, as data required for the next set of frames may be fetched and be made available in local memory while the current set of frames are being generated.

### 1.4.2 Exploiting Spatial Coherency

The above method finds good application when subsequent screen positions are known *a priori* and works well for incremental rotations of the screen. The second phase of the research [Chapter 4] was concerned with exploiting frame-to-frame coherence in a more general way, i.e., for real time animation where the screen positions are not restricted to rotations only. Furthermore, this method is more amenable to be extended to ray tracing as well. Several memory optimizations were made and a *directory-based protocol* was designed to further optimize memory utilization [69]. Local memory was used to store only that data which was needed by the processor. It was seen that with sufficient local memory size to hold all the objects for a single frame, the data requirement for the next (close) frame was just 5% of the volume. No memory



was assigned for static data; objects *migrated* within the system; and a *directory* was used to trace an object's location. Latency hiding was achieved by queueing all the rays for which data needed to be fetched from other nodes (*inactive rays*), while rays for which all the required data were locally available (*active rays*) were traced to completion. Furthermore, our screen assignment scheme ensured that the non-available data at a node were always available from one of its neighboring nodes. We also devised a *forwarding* scheme which minimized network congestion, thus taking advantage of *network coherency* as well. The overall system provided enough generalization and independence from the underlying network. Considerable load balancing, sufficient latency hiding, network congestion control, and utilization of frame-to-frame coherency all combine to make this algorithm scalable.

### **1.4.3 Exploiting Volume Coherency**

Having optimized the memory utilization and scalability of our algorithms, we turned our attention to rendering colossal volumes [68]; datasets too large to fit in the total memory of the ensemble of processors. Sometimes such volumes are even stored on remote disks in a compressed form [71]. *Thrashing* is a phenomenon which is often encountered in such situations, where objects are fetched multiple times (either from remote processors or from disk) during the generation of a single frame. In the third phase of our research [Chapter 5 and Chapter 6], we devised a method which is not only thrashless for a single frame generation, but retains its thrashless property across a number of slowly changing frame positions. In short, the sequence in which data are fetched and the way in which rays are traversed on the screen (*ray coherency*) ensures that no data are fetched more than once for a number of frames. Our scheme eliminates one of the burning problems encountered in parallel rendering of huge databases, viz., thrashing.

## **1.5 Summary of Contributions and Significance**

Parallel rendering is by no means a mature field. Efficient parallel rendering is an elusive goal, with more questions raised than answered. As parallel computers become more powerful and more affordable, integrating graphics and parallel applications will be a central issue for the graphics and visualization community. Efficient parallel rendering algorithms are a prerequisite to this process.

Previous algorithms for parallel volume rendering have mainly been divided into two categories, object-order and image-order. The scene is partitioned and statically assigned to processors. Data that are required

but not available locally are fetched and cached (in hardware) only on demand. These are ultimately removed based on some replacement policy.

The algorithms proposed in this dissertation suggest new paradigms and alternatives to traditional ways of parallel rendering. Optimization of local memory usage has largely been neglected in the past, as previous algorithms have concentrated on small to medium sized datasets. Our CellFlow algorithm [Chapter 3] and the all-cache approach [Chapter 4] optimizes the local memory usage such that it is used only for that data which are required by the local processor. These methods present the most efficient memory usage paradigms. Most earlier algorithms made use of the hardware cache only for exploiting coherency, the performance of these algorithms are prone to degrade with increasing dataset sizes or decreasing cache sizes. Utilizing the local memory as another level of cache, i.e. the *software cache* [Chapter 4], was not explored for parallel rendering. In addition, ways to hide latency and minimize network congestion in each of these algorithms are novel approaches in themselves.

Designing a parallel algorithm to render colossal datasets, like the Visible Human, needs a complete paradigm shift. Previous algorithms are no longer applicable to such large datasets (possibly in compressed form), as they are prone to thrashing, a phenomenon which should be avoided at all costs. A new algorithm is sought that will provide the required efficiency and remove the problem of thrashing for rendering colossal datasets. The algorithm described in Chapter 5 combines the advantages of both the object-order algorithms (e.g., no thrashing, regularity of access, object space coherency) and image-order algorithms (e.g., opacity clipping, better image quality, simplicity, and usage of other acceleration techniques) to solve the problem of thrashing and provide the most coherent screen traversal scheme.

The versatility of the parallel thrashless rendering system opened new avenues for applications in the area of rendering colossal medical volumes on uniprocessor machines as well. The basic algorithm remains unchanged, except that the unavailable cells are fetched from disk (possibly in compressed form), rather than from other processors. The enormously high cost involved in constantly thrashing to disk, and probably decompressing the same cell multiple times adds undesirable overheads to the rendering system. The uniprocessor RayFront algorithm described in Chapter 6 provides an effective solution to this problem. As in the multiprocessor method, it minimizes the number of times data are read from disk, and consecutively, the number of times they are decompressed for rendering a number of frames in an animation sequence.

The accomplishments and contributions to the field of parallel volume rendering is summarized below.

- An incremental rotation scheme [Chapter 3] that exploits temporal coherency to minimize communication, accomplishes effective latency hiding, and utilizes local memory optimally.
- A method with the most efficient local memory utilization [Chapter 4], among the ones proposed in literature. This “cache-only” scheme minimizes the communication overheads, and achieves effective latency hiding, providing a general purpose animation algorithm at the same time.
- A thrashless method for visualizing colossal datasets [Chapter 5], which also maintains its thrashless property across a number of similar frames. This method has shown considerable promise for visualizing compressed volumes as well [Chapter 6], i.e., volumes so big that they have to be stored on disk in a compressed form.
- The thrashless property of the above method also inherently provides an optimal screen traversal method, and has outclassed the most efficient screen traversal scheme proposed so far, i.e., the Hilbert transform [136].
- The scalability demonstrated by the above algorithms are far superior to those reported in the literature.
- The algorithms are transparent to the underlying network topology and thus are portable to parallel machines with any topology.
- Unlike previous methods, our algorithms are primarily designed to perform efficiently on DM architectures, so that the widely available and unused computing resources, like the NOWs and COWs, may be effectively utilized for such enduring tasks like volume rendering.

In summary, this research has primarily focussed on some of the as yet unexplored problems in parallel volume rendering, e.g., latency hiding, optimizing local memory utilization, optimal screen traversal, reducing network congestion, and portability, and has suggested exclusive ways to eliminate some or all of these. Our coherent algorithms have demonstrated scalability to very high degrees, with potential to improve even further.

## CHAPTER 2

### PARALLEL VOLUME RENDERING ALGORITHMS

The numerous acceleration techniques presented in Chapter 1 undoubtedly highlight the need to make volume visualization more effective. Interactive or real time rendering algorithms are a precursor to this goal. Rapid visualization techniques will provide the scientist with interactive exploration of the data sets. Altering viewing parameters, changing opacity and other transfer functions, are some of the features that can be quickly modified to make crucial decisions. General purpose massively parallel processors (MPPs) provide a highly viable platform for such an application. Parallel rendering has the potential for high-performance in effectively producing real-time visual output. Realizing this potential requires careful analysis of the algorithms in light of a system's architectural parameters. As we will see later, significant obstacles remain, principally in the areas of scalability, load balancing, and image composition.

In this chapter we focus on the parallel volume rendering (PVR) algorithms proposed in the past. We will first discuss the various approaches taken to parallelize volume rendering, highlight some of the important issues underlying the design of these approaches, and finally give an insight into some of the open issues tackled in this dissertation. This survey mainly focuses on the issues underlying the design of PVR algorithms for regular (rectilinear) volumes only. For sake of completeness, we have also included a few specialized hardware implementations of PVR which bears a close resemblance with general purpose multicomputer algorithms. In particular, four such architectures are considered: Fuch's Pixel-Planes 5, Kaufman's Cube, Goldwasser's Voxel Processor, and Ohashi's 3DP. A survey of parallel rendering in general can be found in [23], parallel polygon rendering on shared memory architectures in [118][120], specialized architectures for volume rendering in [54] and [107], parallel ray-tracing in [72], and a partial list of PVR algorithms in [124].

## 2.1 Types of Parallelism

Parallelism exists both in image-order and object-order volume rendering methods, and it has been successfully exploited by researchers to propose a variety of algorithms. For example, the independence of rays in ray casting makes it a good candidate for parallel implementation. Similarly, the independence of transforming voxels one at a time makes object-order methods amenable to parallelization.

In general, two types of parallel machines have become popular - shared memory (SM) and distributed memory (DM). In the former, data may be physically distributed among the nodes, but all the processors view it as a single coherent data space. The data coherency management is transparent to the users. In contrast, no notion of shared data exists in DM machines; information exchange between processors is carried out by explicit message passing between them. These machines will be discussed in more detail in Section 2.2.1.

PVR algorithms can be grouped into two main categories: *image partition* (IP) and *object partition* (OP). Two other categories are now becoming popular: *hybrid* and *shear-scale-warp* (SS). Below we discuss these approaches to parallel rendering, their advantages and disadvantages, and their possible modifications. The issues affecting the performance of these approaches will be dealt with in the next section (Section 2.2).

### 2.1.1 Image Partition

Image partition (IP) algorithms partition the screen into a number of regions (typically the number of regions equals the number of processors), which are then assigned to processors. Although image partitioning is equally applicable to object-order or image-order PVR methods, they are generally associated with the latter. Processors concurrently and independently process rays in the image region(s) assigned to them. If the complete volume is assumed to be replicated in the local memory of each processor, no interaction between processors is necessary during the rendering stage. Communication takes place only during image gathering, when each processor sends its image segment to the display processor. On the other hand, when the memory is too limited to allow complete replication, communication during rendering becomes inevitable, the granularity of which is implementation dependant.

IP algorithms are well suited for shared memory (SM) or distributed shared memory (DSM) architectures, where the complete volume is kept in shared memory. Examples of such implementations include those by Challenger [17], Nieh and Levoy [90], Palmer, et al. [93], and Goel and Mukherjee [37]. Uniprocessor vol-

ume rendering programs can be ported to these machines with minimal changes in the code. In these implementations, required portions of the volume are fetched automatically (using hardware) by simply reading the relevant data. Hierarchical hardware caches provide means for exploiting spatial coherency that helps to reduce the latencies of remote fetches. As we will see later, dynamic load balancing schemes can also be easily applied to IP techniques. Several acceleration techniques (see Chapter 1), like early ray termination and space leaping, can be applied to IP algorithms, and several types of coherency (spatial and frame) can be exploited to reduce communication overheads.

The disadvantage of this class of algorithms is the computationally expensive resampling that is required for generating high quality images. A solution to this has been proposed and has given rise to the *shear-scale-warp* algorithms (see Section 2.1.4). Also, these methods are prone to excessive amounts of data communication between nodes, especially when screen positions change abruptly; the amount of data communicated being highly view dependent.

Not many IP methods have been reported on message passing (MP) or distributed memory (DM) machines, the primary reasons being the enormous cost associated with volume redistribution, and usage of complicated data fetching schemes. Corrie and Mackerras [22], Westermann [115], and algorithms described in Chapter 3, Chapter 4, and Chapter 5 implement IP algorithms on DM machines. In all these cases, the local memory is treated as another level of cache (the software cache) to exploit spatial coherency and reduce communication. Cohen and Fleishman [19] exploit temporal coherency for incremental rotations on DM machines, while Shroder and Stoll's parallel IP algorithm [100] is designed for DM SIMD architectures. Fuch's Pixel Planes 5 architecture [34] also can be used for volume rendering using the IP approach. Specialized processors, called Renderers, are responsible for the initial classification and shading of the volume. During rendering, rays are cast into the volume by another set of processors, called the Graphics Processors, and the required voxels are provided by the Renderers.

### **2.1.2 Object Partition**

The second class of PVR algorithms are the object partition (OP) techniques, where the volume is divided into several sub-volumes, and each sub-volume is assigned to a processor. Each processor generates an independent image (either by ray-casting, or by splatting or voxel projection) of the sub-volume present locally. The individual sub-images from all the processors are combined into a final image in a separate compositing

step. Although ray-casting also involves some sort of object partitioning, OP parallel rendering algorithms have been popularly associated to the method described above.

Goldwasser was one of the first authors to suggest a hardware implementation of OP PVR on their Voxel Processor machine [38]. A set of processors (called Processing Elements) generate partial images of the sub-volume assigned to them. Another set of processors (called Intermediate Processors) then composite the partial images from a cluster of PEs. An Output Processor finally generates the complete image from the sub-images received from the IPs. A similar hardware implementation was also adopted by Ohashi, et al. in their 3DP architecture [92], but uses only one hierarchical level for merging all the sub-images in a FTB order, no IPs are involved in their case.

The ray-casting approach to generating partial images from the local sub-volumes has also been referred to as *segmented ray casting*, and was adopted by Hsu [49], Ma, et al. [82], Camahort and Chakravarty [16], and Neumann [89]. The *voxel projection* (VP) approach, on the other hand, were used by Challinger [17], Stredney, et al. [106], Machiraju and Yagel [83], and OP algorithms proposed by Wittenbrink [124]. The partial images generated by either of these approaches are then composited into a final image. Compositing can either be done in a *centralized* manner [30], in a *hierarchical tree-like* manner as in Machiraju and Yagel [83], or in a *binary-swap* manner as in Ma, et al. [81], in Wittenbrink and Harrington [125], in Hansen, et al. [45], and in Rowlan, et al. [96]. The amount of communication required in these methods is, in general, much smaller than in case of IP algorithms, it is view independent, and more importantly, it is deterministic. OP algorithms may also be implemented on DM machines with sufficient ease.

OP algorithms suffer from several drawbacks. First, the image compositing stage is generally not scalable. The partial images have to be composited in a certain order. This requirement imposes some serialization during the compositing stage. Moreover, compositing processors (especially in case of centralized compositing) are targets of severe bottlenecks. As such, load balancing also becomes a concern during this stage. Second, although load balancing is automatically achieved during the rendering stage (as far as the number of voxels processed is concerned), several volume regions may have only a few occupied voxels, making actual load balancing a more difficult job. Equalizing load becomes even more complicated if we realize that reducing the sub-volume sizes gives rise to many more partial images which have to be subsequently composited. Dynamic load balancing with these methods opens more problem avenues, such as for correct sort-

ing of the partial images. Finally, acceleration techniques like early ray termination cannot be properly applied, leading to a lot of wasteful computations.

### 2.1.3 Hybrid

A third group of PVR algorithms try to utilize the advantages of both the IP and the OP approaches, and thus is referred to as the *hybrid* approach. In Montani, et al.'s [85] hybrid method, the complete volume is replicated in clusters of processors. Each cluster is also assigned with an image partition. The volume within a cluster is partitioned among the processors that form the cluster. A cluster is responsible for generating the sub-image of the screen segment assigned to it. Each cluster uses the OP approach to generate its image segment(s). The partial sub-images generated by the processors within a cluster are composited within the cluster to form the final image of the screen segment(s). The final sub-image(s) from the clusters are then sent directly to the display.

Hybrid algorithms seem to be more promising than OP mechanisms, as dynamic schemes for load balancing is a possibility. Not many researchers have explored hybrid methods for PVR, but preliminary results reported by Montani, et al. offer lucrative grounds for more active research in this area. On the other hand, the requirement of volume replication in each cluster may prove to be a bottleneck when higher resolution volumes are considered for rendering.

### 2.1.4 Shear-Scale-Warp

The shear-scale-warp (SSW) PVR algorithms were primarily designed to assuage the resampling expenses in IP methods. This method is also primarily IP, but is preceded by a shear-scale step. The volume slices are first sheared (and scaled, in case of perspective), after which the rays can be traced as if they are parallel to one of the major axes. This considerably simplifies the resampling process during rendering. Regular communication of volume data takes place between processors after the shearing, such that processors have the voxels required for resampling. Following the shear-scale-warp step, the beam of voxels perpendicular to the image plane are simply composited to obtain the final color of a pixel; no resampling is needed. Shroder and Salem [99] and Vezina, et al. [113] apply the shear/scale method for implementing rotations on SIMD machines, while Amin, et al. [3] and Lacroute [63] extend the latter's shear-warp algorithm on DM and SM (or DSM) MIMD machines respectively.



SSW algorithms are considered as one of the fastest ways of rendering volume datasets. Their implementation on parallel machines have accordingly attracted attention. The gain in speed comes at the cost of slightly poorer quality images, and the possible requirement of maintaining multiple copies of the volume simultaneously in memory [3].

## 2.2 Issues in Parallel Volume Rendering

In this section, we provide a comprehensive survey of the issues that affect the design, implementation, and performance of a PVR system. Highlighting the features and performance of each of the PVR algorithms in literature not only gives one a quick overview of the popular approaches, but it also allows us to understand the common trends followed by the researchers. It opens avenues to new and unresolved issues that need to be tackled. Alternative schemes are thought of, and efficiency becomes the primary driving force for taking these alternative routes. Of course, a thread joins all the issues together to form one complete efficient system, and efficiency has to be compromised in one aspect of the design to balance it with another. It may be impossible to design a parallel algorithm with the most efficient individual features, but this survey reveals some of the design choices made by researchers to achieve their goal in efficient parallel volume rendering. Issues such as type of projection, illumination, shading, interpolation, etc. that effect the rendering speed on uniprocessor machines, are not considered in this survey.

### 2.2.1 Architecture

Two types of parallel programming paradigms are often used for multiprocessor volume rendering: *Single-Instruction-Multiple-Data* (SIMD) and *Multiple-Instruction-Multiple-Data* (MIMD). All the processors in an SIMD machine perform the same instruction in lock-step fashion. SIMD implementations of PVR algorithms generally try to regularize communication so that it is restricted among neighboring nodes only. This is because regular communication happens over fast networks (e.g., XNet on MasPar MP-1), as opposed to point-to-point communication over slower global network. Designing algorithms to optimize these communication patterns poses a difficult problem. Coherency that exist in volumes is often advantageously exploited to make data transfer regular. Shear-scale methods for volume rendering has shown considerable promise for implementation on SIMD machines, as can be found in several implementations like those described in [99][113][126].

The restriction of regularizing communication makes algorithms designed for SIMD machines not too scalable. Moreover, SIMD machines are, in general, suitable for OP PVR methods. As seen earlier, OP methods suffer from numerous drawbacks. Due to these limitations of SIMD machines and the difficulty that arise to design efficient algorithms on these machines, MIMD machines have now become popular. As the name suggests, no synchronization is required among the processors for running multiple instructions in parallel. A special class of MIMD programming paradigm is the *Single-Program-Multiple-Data* (SPMD) model. The same program is spawned to all the processors; the processor in turn executes only a certain portion of the code, depending on its responsibilities. The responsibilities are assigned based on the pids (processor ids) of the processors. Most current PVR algorithms use this programming model.

MIMD multi-computers are available in various forms, most of which can be grouped into the following three categories: *shared memory* (SM), *distributed shared memory* (DSM), and *distributed memory* (DM) (see Figure 2.4). DM machines are sometimes referred to as *message passing* (MP) machines.

Most shared memory MPPs are available with a globally shared bus, (Figure 2.1) which offers low scalability. It is important to realize which algorithms assume a global shared address space. As seen in the previous section, these algorithms are less amenable to portability and scalability than those designed for DM architectures.

DSM and DM machines may share the same topology and memory organization, as shown in Figure 2.4. Nodes may be connected in any of the scalable topologies described in the previous section. In the former, the physically distributed memory among the processors are accessed using a single address space, each processor maintaining a portion of the total shared memory. Examples of a DSM machine is the Stanford DASH or the Convex SPP multicomputer. Message passing between nodes is kept hidden from the user. The scalability of the underlying network makes such architectures much more scalable than those which are bus-based. For DM machines, the address space is local to each processor, and a processor cannot access any processors' memory except its own. Communication between nodes is carried out by explicit user-specified message passing only, which, in general, is more expensive than the low level communication on DSM machines. Explicit message passing requirements also puts a burden on the user to provide synchronizations within their PVR programs.

With the advent of fast interconnection networks like FDDI and ATMs, distributed environments will become the choice of parallel computing. Distributed volume rendering is now becoming popular on net-

work of workstations (NOWs) [117], cluster of workstations (COWs) and heterogenous environments [81][135]. These environments generally do not assume shared memory, and thus DM PVR algorithms are portable to distributed and heterogenous environments.

Object order volume rendering methods are quite amenable for implementation on *vector* or *pipelined* architectures as well. Machiraju and Yagel [83] formulated a method that exploits voxel-to-voxel coherency to reduce transformation calculations. Their method is extremely amenable to vectorization. Stredney, et al. [106] have reported such an implementation on the vector machine, the Cray Y-MP.

As seen from the above discussion, the efficiency and portability of PVR algorithms have a strong correlation to the kind of MIMD multi-computer it is originally designed for. Design efforts continue as different combinations of topologies, architectures, and methods are tried. From the past efforts, it is evident that OP methods can be implemented on DM parallel computing environment, while IP methods hold more ground in SM (DSM) multiprocessors, primarily due to the simplicity and efficiency in portability and design. Among the authors who have defied these general rules are the IP implementations by Yoo, et al. [135], Corrie and Mackerras [22], Westermann [115], Cohen and Fleishman [19], Amin, et al. [3], and Law and Yagel [67][68][69] on DM architectures, and the OP implementation by Challinger [17] on DSM architecture.

## 2.2.2 Embedding Topology

The underlying topology of the interconnection network also has substantial impact on the performance of the parallel renderer. For example, parallel algorithms designed for hypercubes will not perform optimally when implemented on meshes, and algorithms optimized for meshes will perform sub-optimally when ported to tree architectures. Some algorithms, for example, those designed for rings, may be portable, as rings may be efficiently embedded into other scalable topologies with considerable ease.

Different network topologies evolved primarily to provide network scalability for designing MPPs with thousands of processors. On one extreme, bus-based processor configurations (Figure 2.1) offer limited architecture scalability. The processors are connected to each other and to the memory via a globally shared bus. The memory is organized in an hierarchical manner. Each processor maintains a local hardware cache, which is much faster than the globally shared memory. The primary reason for the unscalability of such parallel processing machines is that the shared bus quickly becomes a source of contention. This is exemplified by implementations on SGI Power Challenge by Palmer, et al. [93] and then by Lacroute [63]. Their algo-

rithms show efficiencies of 75% (on 32 nodes) and 74% (on 16 processors) respectively. Machiraju and Yagel [83] also used the shared bus architecture of IBM PVS to achieve low efficiency of 60% on only 10 processors.

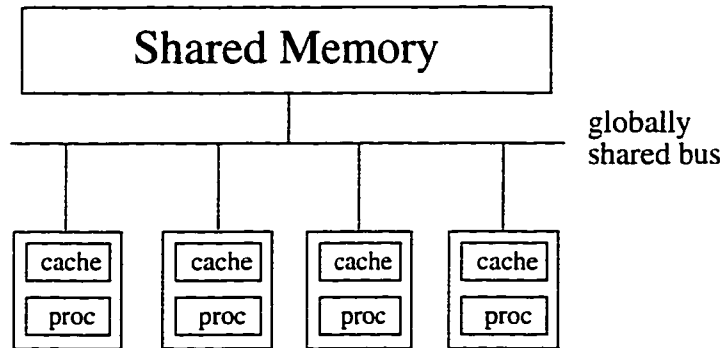
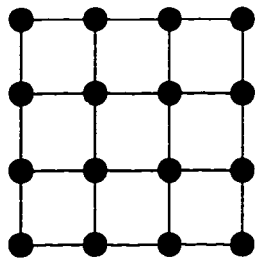


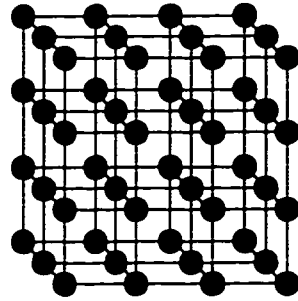
Figure 2.1. Bus-based configuration of the network, connecting the processors and the shared memory. Each node is provided with a processor and a hardware cache.

On the other end of the spectrum lies the scalable networks, like meshes and hypercubes. These network topologies have shown considerable promise for scalable performance. Literature reveals that the most popular embedding topologies for PVR algorithms have been the *2D and 3D meshes* (Figure 2.2a and Figure 2.2b, respectively). Several algorithms have been proposed on these machines, some examples of SIMD implementations being Vezina [113] and Wittenbrink [126] on MasPar MP-1, and Hsu [49] on DECmpp 12000. Instances of MIMD implementations on meshes include Nieh and Levoy [90] and Lacroute [63] on DASH, Corrie and Mackerras [22] on Fujitsu AP 1000, Wu and Brady [128] on WaveTracer Zephyr, Neumann [89] on Touchstone Delta, Goel and Mukherjee [37] on MasPar MP 1200, and Law and Yagel [70] on Cray T3D.

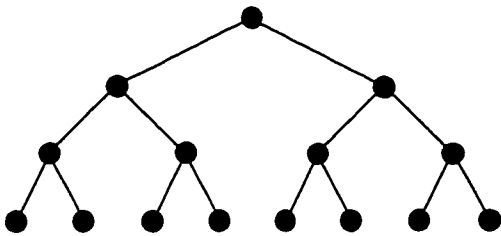
*Hypercubes* and *trees* (Figure 2.2d and Figure 2.2c respectively) are other popular topologies. Hypercube implementations include Schroder and Salem [99] on CM-200, Montani, et al. [85], Wu and Brady [128], and Elvins [30] on nCube-2, and Schroder and Stoll [100] on the Princeton Engine. Several instances of fat-tree implementations can be found on CM-5, some of them being Ma, et al. [81], Camahort and Chakravarty [16], Westermann [115], and Amin, et al. [3]. The configuration of a fat-tree is the same as that of a tree (Fig-



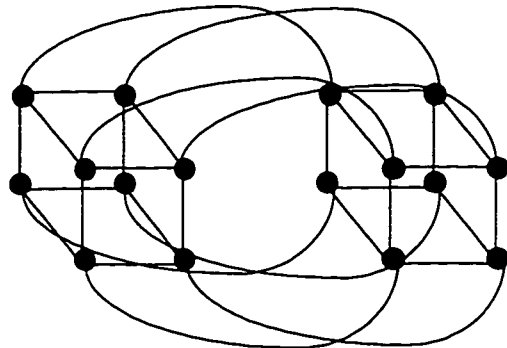
(a)



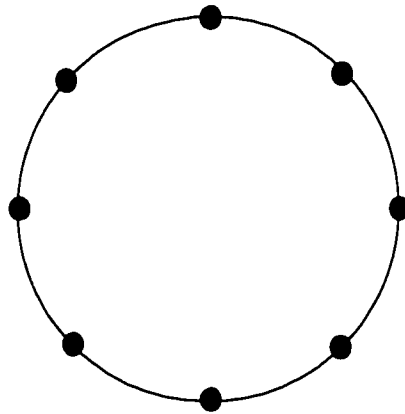
(b)



(c)



(d)



(e)

Figure 2.2. Different topologies: (a) 4x4 2D mesh, (b) 4x4x3 3D mesh, (c) 15-node full binary tree, (d) 16-node ( $2^4$ ) hypercube, (e) 8-node ring.

ure 2.2c), except that the former has more links between nodes located higher up in the tree. Multiple links help to avoid congestion that may arise due to more messages traversing through the upper nodes of the tree.

Interconnections demonstrating intermediate level scalability include the *multi-stage interconnection networks* (MIN) (Figure 2.3) and *rings* (Figure 2.2e). As such, these architectures have not gained much importance in the parallel rendering community. Only Schroder and Stoll [100] have reported a 1D ring implementation of their PVR algorithm on CM-2. MINs provide an interim solution to the shared bus contention problem, where processors are connected to memory modules by a partially scalable interconnection network, e.g., the Omega network (Figure 2.3). In order to provide accessibility from each processor to every memory module, multiple stages of interconnection is required. For example, the number of stages required for the Omega network using  $2 \times 2$  switches is  $\log_2 P$ ,  $P$  being the number of processors. There are  $P/2$  switches at each stage. The partially scalable interconnection network of MINs provide partial scalability to PVR algorithms implemented on them, as is shown by Challenger on Butterfly BBN TC 2000 [17], by Wittenbrink and Harrington on Proteus [125], and by Rowlan, et al. on IBM SP-1 [96]. The efficiencies reported by them are 46% (on 100 processors), 69% (on 32 processors), and 22% (on 64 processors), respectively.

### 2.2.3 Scalability

One of the most important issues in designing any parallel algorithm is its *scalability*. Although all parallel algorithms follow the rule of diminishing returns, scalable algorithms provide increasingly superior throughput for large number of processors. An associated term, *speedup*, is also often used for this measure. Speedup and scalability are given as follows:

$$Speedup = \frac{T_1}{T_p}$$

$$Scalability = \frac{Speedup}{P}$$

where  $T_1$  and  $T_p$  are the rendering times respectively on one and  $P$  processors.

While the raw frame generation time indicates the speed of the renderer, scalability of a parallel algorithm provides a rough estimate of the amount of overheads influencing the parallel design. All the above issues,

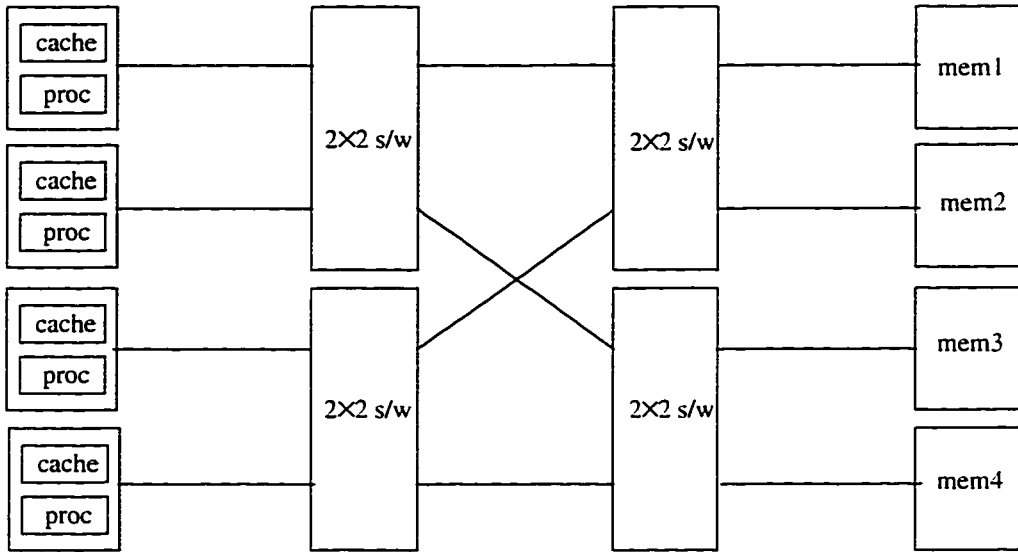


Figure 2.3. A 4-processor multistage interconnection network (MIN) as used in Omega.

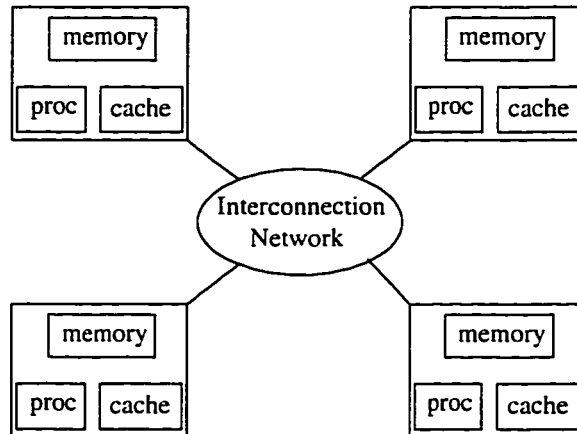


Figure 2.4. Distributed memory architecture for PVR algorithms.

like load imbalance, latency, network congestion, etc. combine to bring down the scalability of a parallel algorithm, and so measures have to be taken to minimize these parallelizing overheads.

Among the notable ones who demonstrate high scalability are Nieh and Levoy [90] with greater than 80% scalability on 48 DASH processors, Corrie and Mackerras [22] with 83% utilization on 128 Fujitsu AP-1000 processors, and Montani, et al. [85] with 74% efficiency on 128 processors of nCube 2. Our algorithms presented in the next three chapters also offer high scalability of about 75% on 128 processors of Cray T3D. On the other hand, the parallel algorithm suggested by Elvins [30] is not scalable for more than 8-16 processors on moderately sized volumes.

## 2.2.4 Data Partitioning

One of the prime issues in the design of a parallel renderer is the way to partition the data. Neumann [89] has done a theoretical analysis (verified by experiments) on the relationship between data partitioning and communication costs on mesh connected multicomputers. The four most popular ways of partitioning the volume data are: *slices*, *slabs*, *shafts*, and *cells* (see Figure 2.5). For OP methods, Neumann showed that cells exhibit the maximum independence of performance to changes in view direction. Performance of the OP algorithms are view dependent for all other partitioning techniques. This is due to the fact that while slices and slabs are partitioned only in one dimension and shafts are partitioned by planes in two dimensions, cells are partitioned in all three dimensions.

That cells offer the best performance among all OP alternatives is confirmed by the fact that it had been the popular choice of most OP methods reported in literature. Only Stredney, et al. [106] and Machiraju and Yagel [83] used slabs for their object-order projection algorithms to take maximum advantage of voxel-to-voxel coherency and vector processing, and Elvins [30] used slice based partitioning for dynamic load balancing purposes.

Most IP methods have also used cells as the volume partitioning scheme. Shafts were used only by Vezina, et al. [113] for their shear/scale SIMD implementation, while slices were used by Challinger [17] and Amin, et al. [3] for dynamic load balancing purposes. Slabs found use in the hybrid MIMD algorithm of Montani, et al. [85], and the incremental rotation IP MIMD algorithms of Cohen and Fleishman [19], and Shroder and Stoll [100]. Furthermore, in case of SM (or DSM) MIMD algorithms, the volume is automatically divided



into *pages*, as in Nieh and Levoy [90], and in Lacroute [63], which is the granularity at which communication takes place between processors.

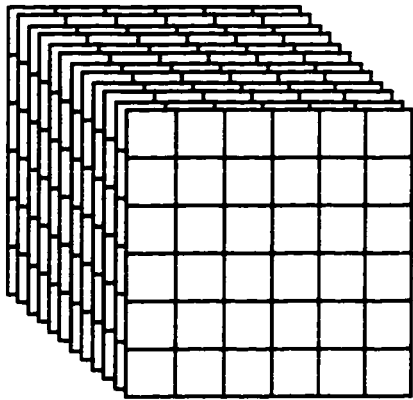
Similar to the volume partitioning, the decision on image partitioning is also dictated by the desire to maximize the amount of coherency that can be exploited within a partition. *Scan-lines*, *stripes* (both vertical and horizontal), or *tiles* emerged as the most popular choices. To maintain spatial coherency in IP methods, IPs should be made as large as possible, which is achieved at the cost of poor load balance (see Section 2.2.6). Thus a compromise is reached by losing coherency in order to equalize load more effectively. Tiles have been shown to maintain more coherency than scan-lines, because of their compact space-filling attributes. Tiles have been used by Nieh and Levoy [90] and also by Law and Yagel [68][69] in the ActiveRay and RayFront algorithms described in Chapter 4 and Chapter 5 respectively, while stripes were the choice of Challenger [17], Yoo, et al. [135], and the CellFlow algorithm proposed by Law and Yagel [67] (described in Chapter 3).

In several OP methods, image is not partitioned. Individual processors generate the complete image from the sub-volume present locally. These partial images are then composited in the final compositing step. In case the screen is partitioned, processors send the respective portions of the images to appropriate processors, where the individual sub-images are first sorted and then composited to form the final image. Ma, et al. [81] propose a recursive image partitioning technique that achieves good load balancing during the compositing stage also.

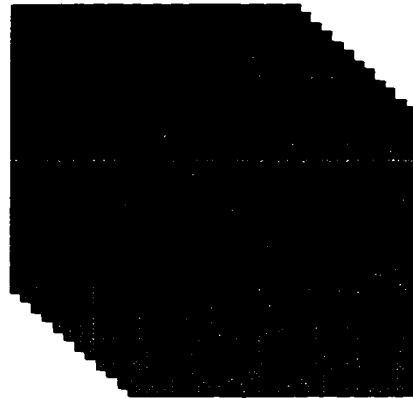
### 2.2.5 Partition Distribution

Once the volume and the image are partitioned, they are distributed to the processors. Partition distribution is generally done in such a way so as to exploit coherency and minimize communication and other parallelization overheads. It is to be noted that volume and image partition distributions do not function independently of each other. For example, for IP methods, the image is partitioned and distributed to processors, while the required volume segments are fetched from other nodes on demand. Once the final image is complete for the image partition, the volume present locally will conform with the image partition.

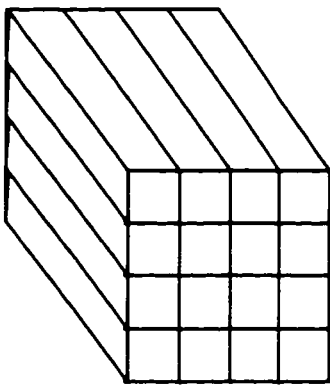
Although partition distribution and load balancing are separate issues, there is a close resemblance between the two. For both IP and OP methods, there are two ways of distributing partitions: *static* and *dynamic*. For static distribution in IP methods, several IPs are generated (typically, # of IPs =  $k \times$  # of processors, where  $k$



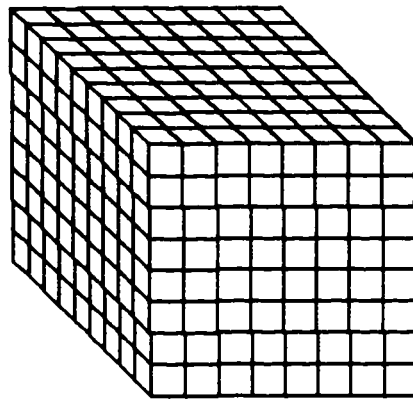
(a)



(b)



(c)



(d)

Figure 2.5. Volume partitions: (a) slices, (b) slabs, (c) shafts, (d) cells.

is an integer), these IPs are then distributed to the processors in a *block* manner (in case  $k=1$ ), or in an *interleaved* or *random* manner (in case  $k > 1$ ). This approach is taken in methods proposed by Hsu [49], Cohen and Fleishman [19], and in several methods proposed by Law and Yagel [70]. Interleaved or random static IP distributions are more apt for static load balancing among processors than block distribution.

Even interleaved or random IP distribution is often not sufficient to balance the load among processors. Dynamic schemes for IP assignment are then adopted. For example, Corrie and Mackerras [22] and Westermann [115] use the worker-arm paradigm to assign IPs on-demand to the working processors. In Nieh and Levoy's [90] proposal, IPs in the form of tiles are initially distributed to the processors in a static manner. During rendering, tiles belonging to one processor may be "stolen" by other idle processors depending on the work load at the processors.

For OP methods, dynamic OP distribution is sometimes difficult to achieve, as the compositing stage demands the images to be sorted in a certain order, although Challinger [17] does resort to this technique by distributing blocks of volume (cells) to processors on-demand, and Elvins [30] distributes slices of the volume on-demand. Even static interleaved OP distributions are not particularly helpful, as the larger number of partitions result in a large number of partial images, proportionately slowing down the compositing stages.

## 2.2.6 Load Balancing

One of the most crucial factors affecting the efficiency of a parallel algorithm is the load balance among the participating processors. The unpredictability of the scene and ray paths in graphics makes the issue of data and computation distribution a challenging task. Any trivial distribution scheme is prone to load imbalance among the computing nodes. Some processors will be unfairly assigned more work than the other. This, in turn, leads to unacceptably poor speedup (and thus low scalability). Two approaches have been taken for attaining load balance: *static* and *dynamic*. Static load balancing schemes determine the computation distribution a priori, which remains unaltered during run-time. In contrast, dynamic load balancing schemes assign computations to the processors as and when required.

The partition distribution schemes discussed above has a one-to-one relation to the way load balancing is achieved. Static [106][69] and dynamic [22][90][115] load balancing schemes for IP methods are as described above. Similarly, [17] and [30] provide examples of dynamic load balancing for OP methods.

In addition to the *worker-farm* and *task stealing* paradigms used for load balancing, coherency is often exploited to predict the load on each processor. For example, Palmer, et al. [93] use their *first moment of work-load distribution* to determine the optimal IP that will balance load. Similarly, Montani, et al.'s hybrid scheme [85] involves dynamic image partitioning based on the work-load estimated from earlier frames.

During the rendering stage, OP methods are automatically load balanced if all the voxels are processed, irrespective of whether they are occupied or empty. In cases where the empty voxels are skipped over, load imbalance arises with OPs of equal size. Unequal sized sub-volumes, determined in a preprocessing step, partially alleviates the load imbalance problem in these algorithms. As an example, Rowlan, et al. [96] adjust the location of the OP cutting planes to improve performance.

A majority of object decomposition methods neglect the load balancing issue during the image compositing stage. Compositing of the sub-images is generally done in a centralized or a tree-based manner. These approaches are prone to load imbalance particularly when the image size is large. For example, in the centralized scheme [30], all the processors except the compositing processor are idle during the compositing stage. Similarly, during hierarchical compositing [83], recursively half the number of processors at each level of the hierarchy becomes idle. Ma, et al. [82] propose a novel solution to load balancing during the compositing stage using their binary-swap algorithm. In this method, the complete image at each processor is recursively halved and sent to its partner processor. This continues in several stages ( $\log_2 P$ ), ultimately each processor is left with the final image of just one small IP. All the processors participate in the compositing procedure, and remain busy until the final image is generated.

### 2.2.7 Coherency

In graphics, coherency has been defined as the uniformity that exists in time and space [108]. Different types of coherency pertaining to parallel rendering are *spatial coherency*, *temporal coherency*, *volume coherency*, *volume-space coherency*, and *network coherency*. While spatial and temporal coherency pertain to IP algorithms, volume or voxel-to-voxel coherency are more applicable to OP approaches.

A fair amount of coherency exists between slowly changing frames in an animation sequence. Such temporal coherency has been exploited by Cohen and Fleishman [19] and by Law and Yagel [67] (described in Chapter 3) to minimize the data transferred between adjacent processors in case of incremental screen rotations around the volume. Both these algorithms determine the small amount of additional data which is

needed by each processor when the screen rotates around the volume by a small amount. Amin, et al. [3], determine the incremental data transfer required for rotations using the Lacroute-Levoy shear-warp algorithm [62].

SM (or DSM) multiprocessors provide a convenient platform for IP approaches, as uniprocessor volume rendering codes are easily parallelizable on these machines. Such SM machines use their hierarchical *hardware cache* structure to exploit spatial coherency and reduce latency overheads. Challinger [17], Nieh and Levoy [90], and Lacroute's [63] SM (or DSM) implementation of IP algorithms exploit spatial coherency only to the level of the (small) hardware cache. On SM machines, Palmer, et al. [93], exploit temporal coherency as well.

Due to the high cost of such hardware caches, they are generally small in size, and as such are prone to *thrashing* with increasing volume sizes. To circumvent this undesirable situation, some authors like Corrie and Mackerras [22], Law and Yagel [69] (Chapter 4), and Westermann [115] have extended the exploitation level of spatial coherency by taking advantage of the local memory as well. They use part of the local memory as a *software cache*, where data fetched from remote nodes may be stored. This provides a much larger local working cache area for exploiting spatial coherency.

In the *object migration* approach proposed by Law and Yagel [69], and described in Chapter 4, spatial coherency has been raised to yet another level, as no local memory is reserved for static data allocation. Volume partitions migrate and replicate at nodes, and the complete local memory is available to be used as a software cache. They also propose a directory-based scheme for determining the location of volume segments in the ensemble of processors.

The amount of spatial coherency that can be exploited is a function of volume size. With colossal volumes, thrashing begins to deteriorate the performance, as the same portions of the volume has to be fetched multiple times across the slow network. To avoid such performance-limiting situations, several screen traversal schemes, like *scan-line*, *spiral*, *Hilbert curve*, *Peano curve*, etc., have been proposed [7][136] to exploit *volume coherency*. These methods try to traverse the pixels on the screen in such a manner that objects once fetched may be used over and over again before they are discarded using some replacement strategy. As volume size increases compared to cache size, any of these screen traversal algorithms are prone to thrashing. Law and Yagel [68] has proposed a method (Chapter 5) that exploit both volume , spatial, and temporal coherency simultaneously to eliminate thrashing in parallel volume rendering of colossal volumes.

Coherency also exists in volumes that can be utilized to reduce voxel processing times. This type of coherency is not necessarily applicable to parallel implementation only, but has been adopted in several PVR systems. For example, Machiraju and Yagel [83], and Stredney, et al. [106] use the *voxel-to-voxel coherency* existing between adjacent voxels, beams, and slices to resort to incremental calculations for transforming object-space voxels to screen-space. Similarly, Lacroute [63] and Amin, et al. [3], exploit *volume coherency* to skip over empty portions, and exploit *volume coherency* to skip over pixels which have already accumulated enough opacity. Such coherency helps speed the rendering of sparse volumes by a factor of more than two.

### 2.2.8 Latency Hiding

The term latency hiding refers to the ability of the algorithm to overlap communication with useful computation. Communication is the main source of overhead in any parallel algorithm design, and in most cases, it is inevitable. Performance of the parallel algorithm can be sufficiently improved by overlapping this communication with computation. Latency hiding can be achieved to some extent by either *prefetching data* or by *postponing processing*.

Surprisingly enough, latency hiding has not gained much attention among the parallel graphics community. This may be due to the fact that in these applications, latency hiding is difficult to achieve. As the scene and the ray paths are unpredictable, it is difficult to either predict or prefetch objects. Coherency can sometimes be exploited to predict which objects may be needed in the near future. For example, Law and Yagel [67][69] have developed two schemes for hiding latency (Chapter 3 and Chapter 4). In [67], a *look-ahead scheme* is used to fetch objects by predicting the motion of the screen, and in [69], computations of rays waiting for a certain object are postponed while the requested object is being fetched. Overlapping is achieved by immediately processing only those rays for which objects are locally available, while a request is sent for objects that are missing. Montani, et al. [85] also hides latency by postponing ray processing, while Westermann [115] exploits coherency to pre-fetch required data and hide latency.

Some authors, like Lacroute [63], have opted to consider *hardware caching* as yet another way to hide latency. In his SM implementation, volume segments are fetched and stored locally in extremely fast hardware caches. These data, in turn, can be used during the processing of other rays, or during the generation of other frames, depending on the size of the cache. Fetching data locally from the cache, instead of from remote memory, drops the latency of these fetches substantially.

## 2.2.9 Network Congestion

Only a few authors have taken the detrimental effect of network congestion into account for designing their parallel rendering systems. If possible, communication should be restricted to neighboring processors only. Global communication patterns, all-to-all broadcasts, and non-neighbor communication are some incidents that give rise to severe network congestion. Such communication patterns are sometimes regularized so that the network bandwidth can be utilized to its maximum extent.

Most SIMD implementations [100][113] resort to regular communication between nodes. In these architectures, regular communication takes place over efficient channels (e.g., XNet on MasPar-1), as opposed to point-to-point communication over the slower global network. Goel and Mukherjee propose an optimal algorithm [37] on MasPar-1200 that does the compositing of the segmented images in  $\log_2 p^3$  steps on a hypercube architecture,  $p^3$  being the total number of processors.

In case of MIMD machines, avoiding network congestion is a more difficult task. Some algorithms designed for these machines exploit temporal coherency to accomplish nearest neighbor communication. The incremental rotation algorithms proposed by Law and Yagel [67] (in Chapter 3), Cohen and Fieshman [19], and Amin, et al. [3] try to restrict the communication only between adjacent processors. They resort to regular communication only by shifting incremental amounts of data between adjacent processors. Moreover, if the algorithms are carefully designed such that adjacent IPs are assigned to adjacent processors, the required volume data is most likely to be available at close-neighboring processors only, as evidenced by Law and Yagel in [69], and by Corrie and Mackerras in [22]. Wittenbrink [126] also achieved non-conflicting communication patterns with his parallel permutation-warp approach. In this method, all the source and the target (transformed) processor pairs are connected by non-conflicting paths between them, thus avoiding network congestion during the transformation stage.

OP methods generally do not resort to communication during the rendering stage. During the compositing stage also, hierarchical image compositing methods can avoid congestion by mapping a tree in the underlying network. Binary-swap compositing involves some network congestion as the communication of the image segments is generally global. But, at the same time, the size of the image segments decreases as the distance of communication increases, thus keeping the congestion under control. Centralized compositing schemes perform the worst, as the channels near the compositing processor are targets of severe congestion.

### **2.2.10 Portability**

Last but not the least, *portability* plays an important role in the parallelization process. With a sea of parallel commercial machines, a parallel algorithm designer has to keep in mind the ease with which an algorithm designed for a particular parallel machine can be ported to other parallel machines. For example, most SIMD parallel volume rendering algorithms are portable to other SIMD machines with similar topology, and uniprocessor algorithms can be easily ported to SM (or DSM) machines without much code modification. Among the specific algorithms, Ma, et al.'s [81] binary swap algorithm is well suited for hypercube networks, while methods using hierarchical compositing are suitable for tree architectures. In general, DM implementations can be easily ported to DSM environments, but not vice versa.

## **2.3 Summary of the Issues in PVR Algorithms**

Figure 2.6 summarizes the various issues discussed in the last section.



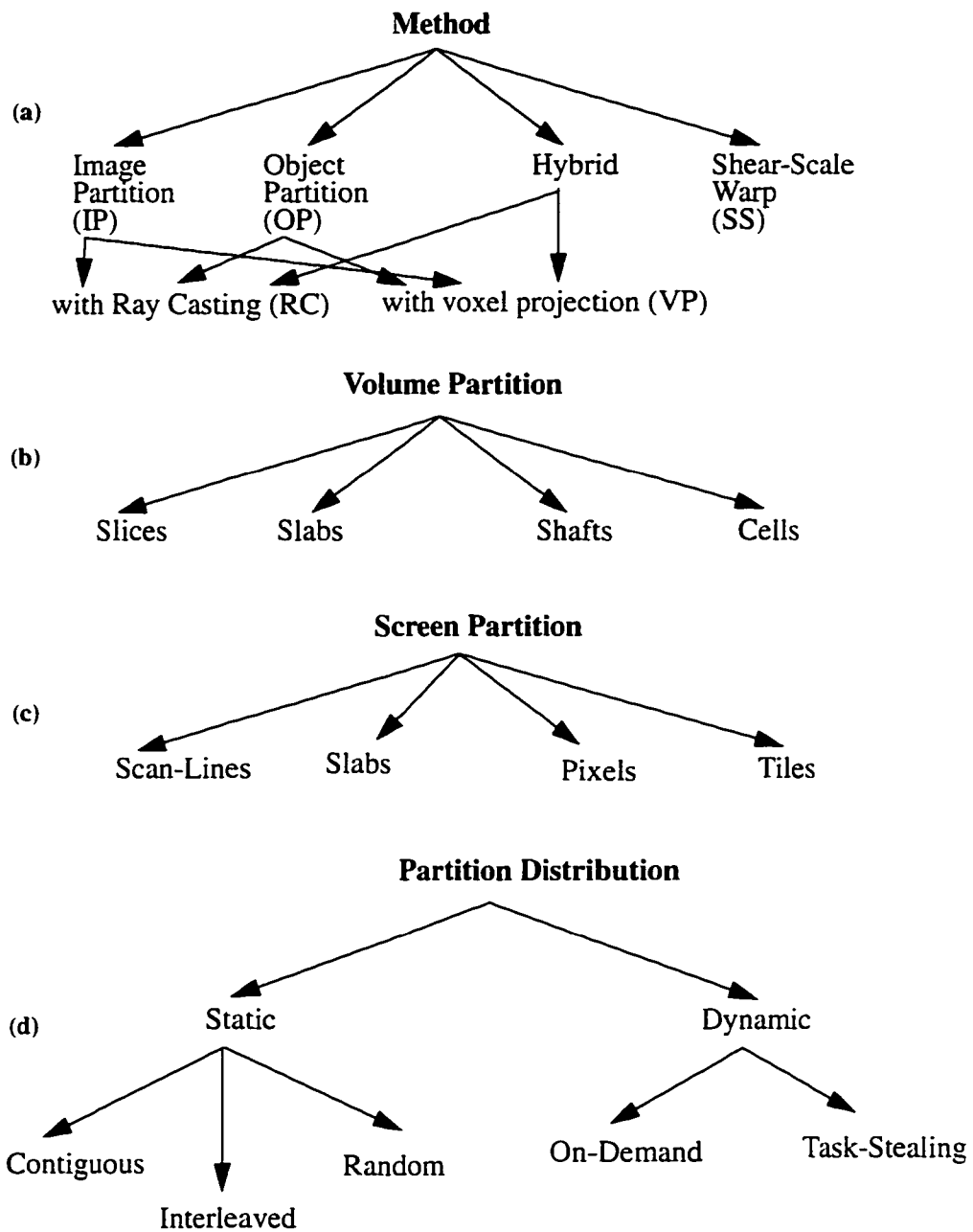


Figure 2.6. Summary of the methods and issues arising in the design of PVR algorithms. (a) methods used for PVR, (b) volume partition schemes, (c) image or screen partition schemes, (d) partition distribution methods, (e) load balancing schemes, (f) types of coherency exploitation, (g) ways to hide latency, (h) network topologies, Summary of the methods and issues arising in the design of PVR algorithms. (i) communication patterns. (j) architectures. (k) programming models. (Continued on next two pages)

Figure 2.6 (contd.)

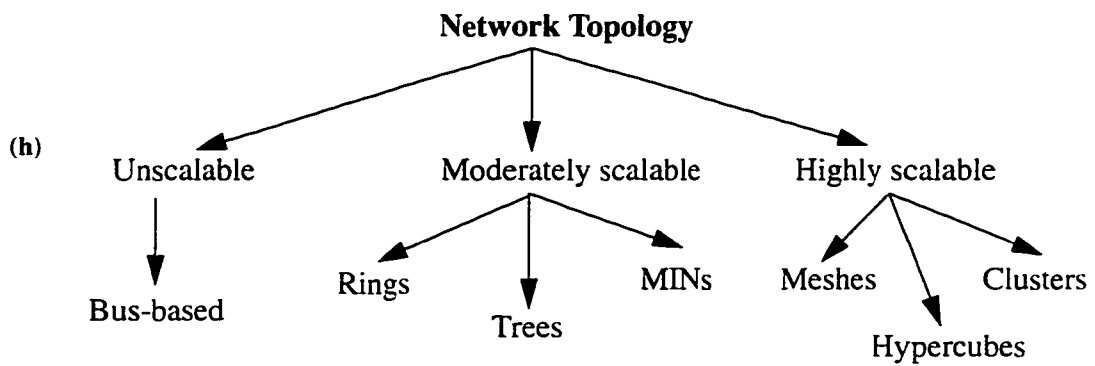
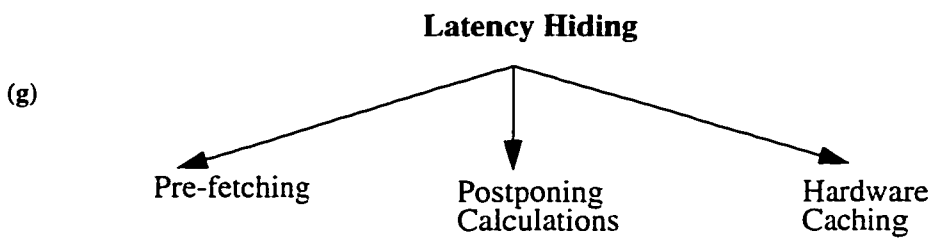
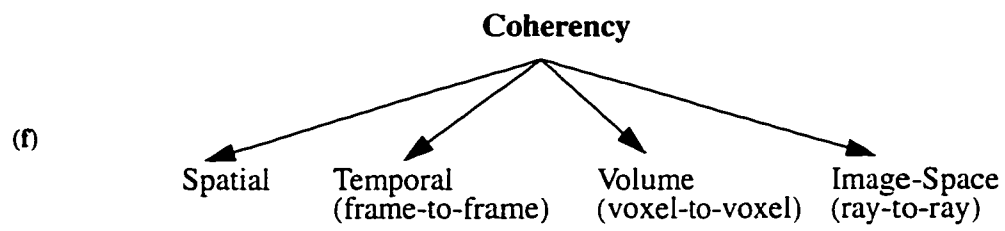
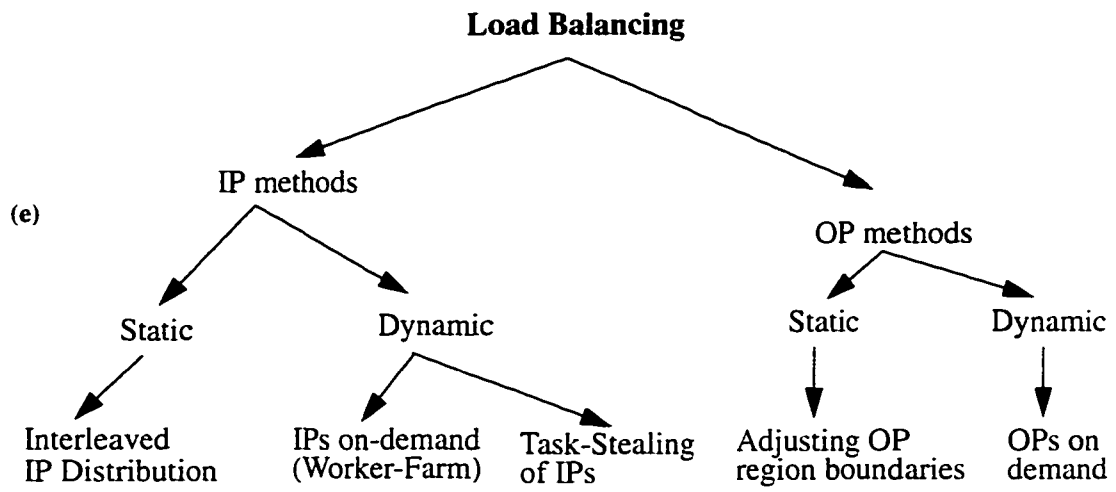
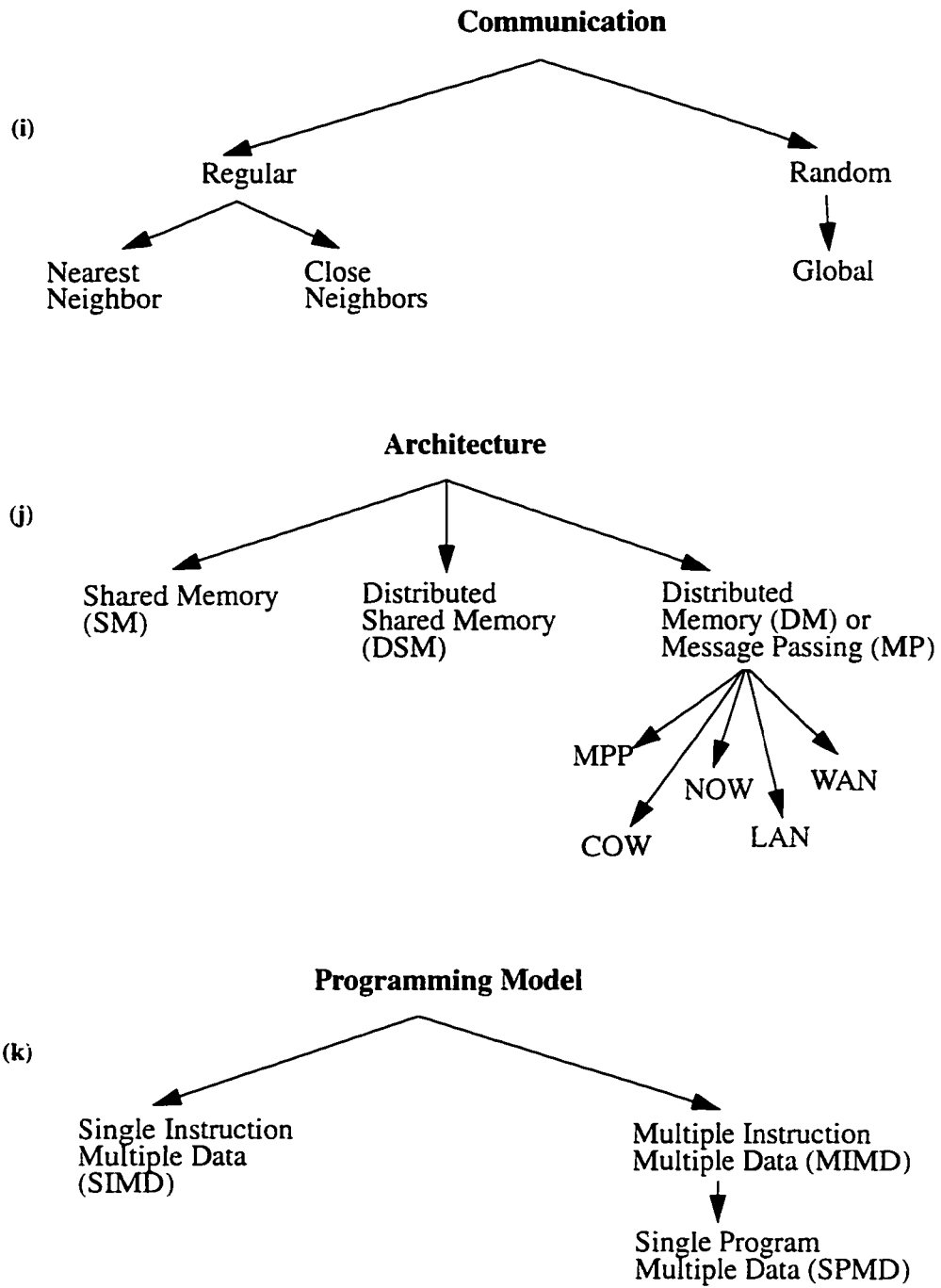


Figure 2.6 (contd.)



### 2.3.1 Summary of the Features of PVR Algorithms

In this section, we summarize the features of a number of PVR algorithms proposed in literature. The tabular representation of the summary reveals some areas where sufficient attention has not been diverted in the past. Table 2.1 gives a summary of the features of the PVR algorithms. The following notations are used in the table: IP (Image Partitioning), RC (Ray Casting), VP (Voxel Projection), Sp (Splatting), PW (Permutation Warping), SS (Shear-Scale), SSW (Shear-Scale-Warp). For example, OPwVP means it is an object partition method where the partial images are generated by voxel projection of the sub-volume available locally. “-” in the table indicates that either the information is not available in the paper, or the feature was not considered in the reported implementation.

First Author [ref]	Method	Data Part	Part Distrib	Load Balancing	Coherency	Latency Hiding	Comm	Portability
Goldwasser [38]	OPwVP	cells	static	-	-	-	-	-
Fuchs [34]	IP	-	-	-	-	-	-	-
Ohashi [92]	OPwVP	cells	static	-	-	-	-	-
Kaufman [52]	VP	beams	inter-leaved	-	-	-	-	-
Schroder [99]	SS	voxels	to virtual proc	-	-	-	global	-
Yoo [135]	IP	cells	dynamic	-	spatial	-	-	most parallel architectures
Challinger [17]	IP	slices	inter-leaved	scan-lines on demand	spatial	-	global	SM
	OPwVP	cells		blocks on demand				
Vezina [113]	SS	shafts	-	automatic during proj	spatial	-	nearest neighbor	-
Montani [85]	hybrid	slabs	-	based on estimated work-load	-	postponing ray processing	between adjacent nodes	-
Nieh [90]	IP	pages	inter-leaved	dynamic using tiles queue	spatial	only during adaptive image sampling	global	SM
Schroder [100]	IP	slabs	-	by pipelining	-	-	nearest neighbor by circular shifts	SIMD
Stredney [106]	OPwVP	slabs	-	automatic	voxel-to-voxel	-	-	-
Elvins [30]	OPwSp	slices	on demand	slices on demand	image	-	global	-
Hsu [49]	OPwRC	cells	cyclic	by cyclic distribution	-	-	global	-
Ma [81]	OPwRC	cells	-	binary swapping during IC	-	-	-	-
Corrie [22]	IP	cells	-	worker farm	spatial	-	neighbor	DM
Wittenbrink [126]	OPwPW	slabs or cells	-	static	-	-	non conflicting	-
Wu [128]	summed proj	-	-	-	temporal	-	-	binary tree

Table 2.1. Summary of the features of the PVR algorithms (continued on next page)

Table 2.1 (contd.)

First Author [ref]	Method	Data Part	Part Distrib	Load Balancing	Coherency	Latency Hiding	Comm	Portability
Camahort [16]	OPwRC	cells	static	-	-	-	nearest neighbor	-
Neumann [89]	OPwRC	cells	static	-	-	-	global	-
Machiraju [83]	OPwVP	slabs	static	automatic	voxel-to-voxel	-	non conflicting	binary tree
Rowlan [96]	OPwRC	blocks	static	adjusting cutting planes	-	-	global	DM machines
Wester-mann [115]	IP	cells	-	dynamic: tiles on demand	spatial	pre-fetching	global	MIMD
Goel [37]	IP	cells	static	-	-	-	regular	hypercube
Palmer [93]	IP	-	replicated in clusters	dynamic work-load distribution	temporal	-	only to display images	-
Cohen [19]	IP	slabs	dynamic	static	temporal	-	adjacent procs only	MIMD
Amin [3]	SSW	slices	dynamic	dynamic with scan lines	temporal and volume	-	local neighborhood	MP machines
Lacroute [63]	SSW	pages	inter-leaved	interleaved and task stealing	spatial and volume	-	global	SM and DSM machines
Law [67]	IP	cells	dynamic	static with cyclic tile distribution	temporal	pre-fetching	adjacent procs only	MIMD
Law [68]	IP		random		volume	pre-fetching	random	
Law [69]	IP		dynamic		spatial and temporal	by post-poning ray processing	nearest neighbor	
Witten-brink [125]	OPwPW	cells	static	-	-	-	global	MIMD machines

### 2.3.2 Summary of Performance of PVR Algorithms

Table 2.2 summarizes the performance of all the PVR algorithms mentioned in the last section. We have tried to be as comprehensive as we could. For example, because of space constraints, not all results presented in the paper are reported here. Interested readers are referred to the corresponding papers for detailed information, like the lighting model (if used), filter size used (for anti-aliasing), and number of iso-surfaces rendered. Also, memory compromises made are not reported here. For example, several authors like Yoo, et al. [135] and Lacroute [63] compromise memory for speed. They store various features like normals, or even pre-shaded volumes to gain speed during parallel rendering. Attempt has also been made to be as consistent as possible across papers, so as to provide a fair comparison between different methods. For example, similar dataset sizes are reported wherever possible.

In Table 2.2, the following notations are used: In the Arch column, SM (Shared Memory), DM (Distributed Memory or Message Passing), DSM (Distributed Shared Memory), SIMD (Single Instruction Multiple Data), MIMD (Multiple Instruction Multiple Data), MIN (Multi-stage Inter-connection Network). DM is synonymous with machines which use explicit message-passing (MP) to communicate between them.

First Author [ref]	Machine	Topology	Arch	# of Nodes	Volume Size	Image Size	Time (secs/frame)	Efficiency
Goldwasser [38]	Voxel Processor	hierarchical	-	64	$256^3$	$512^2$	0.04	-
Fuchs [34]	Pixel-Planes 5	ring	SIMD/MIMD	32	$256^3$	$1280 \times 1024$	0.10	-
Ohashi [92]	3DP	hierarchical	-					-
Kaufman [52]	Cube	-	SIMD	512	$512^3$	$512^2$	0.062	-
Schroder [99]	CM-200	toroidal hypercube	DM/SIMD	64k	$128^3$	$128^2$	0.324	0.96
Yoo [135]	Pixel-Planes 5	ring	DM/SIMD/MIMD	7 rend 16 GP	$128^3$	$640 \times 512$	0.70	-
Challinger [17]	BBN TC2000	MIN	DSM/MIMD	100	$500 \times 600 \times 80$	$512^2$	12	0.46
Vezina [113]	MasPar MP-1	2D mesh	DSM/SIMD	16k	$256^3$	$256^2$	2.52	-
Montani [85]	nCube-2	hypercube (cluster of rings)	DM/MIMD	128	$97^2 \times 116$	$350 \times 250$	5.14	0.74
Nieh [90]	DASH	cluster/mesh	DSM/MIMD	48	$256^3$	$416^2$	0.70	0.83
Schroder [100]	PE	hypercube	DM/SIMD	32k	$128^3$	-	0.25	0.584
	CM-2	1D ring		1024		$128^2$	0.028	-
Stredney [106]	Cray Y-MP	-	Vector	-	$256^3$	-	0.71	-
Elvins [30]	nCube 2	hypercube	DM/MIMD	32	$256^2 \times 90$	$200^2$	2.14	-
Hsu [49]	DECmpp 12000	2D mesh	DSM/SIMD	16k	$128^2 \times 112$	$128^2$	0.24	0.36
Ma [81]	CM-5	fat tree	DM/MIMD	512	$256^3$	$256^2$	0.90	0.875
	Hetero	-		32			9.00	-
Corrie [22]	Fujitsu AP1000	2D torus mesh	DM/MIMD	128	$256^2 \times 109$	$512^2$	45.00	0.83
Wittenbrink [126]	MasPar MP-1	2D mesh	DSM/SIMD	16k	$128^3$	$128^2$	0.50	-
Wu [128]	WaveTracer Zephyr	3D mesh	DSM	8k	$128^3$	-	0.19	-
	nCube	hypercube	SPMD	16			0.10	-
Camahort [16]	CM-5	fat tree	DM/SPMD	64	$256^3$	-	23.72	0.935
Neumann [89]	Touchstone Delta	2D mesh	DM/MIMD	216	$192^3$	$256^2$	0.20	0.27

Table 2.2. Summary of performance of the PVR algorithms (continued on next page)



Table 2.2 (contd.)

First Author [ref]	Machine	Topology	Arch	# of Nodes	Volume Size	Image Size	Time (secs/frame)	Efficiency
Machiraju [83]	IBM PVS	global bus	SM/MIMD/Vector	10	$100^3$	$173^2$	1.672	0.60
Rowlan [96]	IBM SP-1	Omega switch (MIN)	DM/MIMD	64	$128^3$	$512^2$	3.58	0.22
Wester-mann [115]	CM-5	fat tree	DM/MIMD	64	$256^3$	$256^2$	6.60	1.05
Goel [37]	MasPar MP1200	2D mesh (hypercube)	DSM/SIMD	4096	$256^3$	-	11.11	-
Palmer [93]	Power Challenge	global bus	SM/MIMD	32	375 MB of VH	$640 \times 486$	0.40	0.75
Cohen [19]	sim	1D ring	DM/MIMD	-	$500^3$	-	-	-
Amin [3]	CM-5	fat tree	DM	128	$256^2 \times 167$	$256^2$	0.085	0.29
Lacroute [63]	SGI Power Challenge	global bus	SM/MIMD	16	$256^2 \times 225$	$256^2$	0.077	0.74
	DASH	clusters/2D mesh	DSM/MIMD	32			0.17	0.61
Law [67]	Cray T3D	3D torus	DM/SPMD	128	$256^3$	$256^2$	0.38	0.62
Law [68]							0.20	0.58
Law [69]							0.16	0.78
Wittenbrink [125]	Proteus (Intel i860)	clusters/cross-bar	DM/MIMD	32	$256^3$	$256^2$	4.32	0.69

## 2.4 Open Issues

Table 2.1 and Table 2.2 reveal some of the unexplored issues in parallel rendering. For example, the empty entries in the “Latency Hiding” column of Table 2.1 indicates that this important issue has largely been neglected in earlier systems. In this dissertation, we propose two ways to achieve considerable latency hiding of locally unavailable objects - first, by predicting and prefetching data [Chapter 3], and second, by postponing computations [Chapter 4].

The “Comm” column also shows that most researchers did not consider the communication factor in their design. Global communication of data is prevalent, which leads to higher latencies and higher network congestion. Near neighbor communication tends to bring down these overheads. Our communication schemes exploit spatial and network coherency such that global communication of large data is avoided and data movement is restricted to near neighbor only.

Portability is yet another factor which has not gained satisfactory attention. Algorithms that are particularly designed for shared memory machines cannot be easily ported to distributed memory architectures. Several systems have been optimized to work better on certain topological networks, that do not perform well when ported to other architectures. Our algorithms do not assume any underlying topology, and are well suited for distributed memory implementations, even on such communication limited environments like NOWs and COWs.

A closer look at the performance table (Table 2.2) also leads to several unexplored avenues in parallel rendering. First, most of the researchers have restricted themselves to rendering low to medium resolution volumes ( $128^3$  to  $256^3$ ). The primary reason may be the inefficiency caused by constant thrashing in relatively small memory systems. Large resolution volumes ( $>512^3$ ), like the Visible Human dataset (32 GBytes), require new methods that avoid the thrashing problem. In Chapter 5 and Chapter 6, we propose a versatile scheme that has provided optimal performance in such cases on both uniprocessor and multiprocessor machines.

The most vital issue is the efficiency of a parallel algorithm. Algorithms that do not take sufficient measures to remove the parallelization overheads are prone to lower efficiencies on larger machines (with more processors). On the other hand, a careful scrutiny for removing these overheads results in higher scalability, as evidenced by our algorithms.

As Neumann mentions in [89], parallel volume rendering with object migration has not been explored before. Nieh and Levoy [90] point out that an implementation of PVR on COMA machines may be useful. We show such an implementation on DM machines (Chapter 4). With NOWs becoming more popular, our implementations are more amenable to be ported to such multicomputer and heterogenous environments. By exploiting various types of coherency, like temporal, spatial, and network, the proposed algorithms try to improve the computation to communication ratio, attempt to hide the latency of locally unavailable objects, reduce network congestion, optimize the use of local memory, and provides independency to the underlying topology. Our solutions to these issues have brought the performance of shared memory and distributed memory implementations of rendering algorithms closer: we report almost real-time speeds of 20 frames/sec. and interactive speeds of 5 frames/sec. for  $128^3$  and  $256^3$  volumes respectively. In addition, eliminating these overheads have resulted in the scalability of our algorithms far superior to those reported in literature.

Certainly, several other areas like dynamic load balancing and optimal algorithm embedding need to be addressed, but in this dissertation, we have tried to eliminate some of the more daring overheads infesting a parallel rendering system. Alternative schemes are proposed and an attempt has been made to bring the scalability as close to linear as possible. Future research must be dedicated to alleviating these overheads, so as to improve the efficiency and scalability of the parallel rendering system.

## CHAPTER 3

### EXPLOITING TEMPORAL COHERENCY: THE *CellFlow* ALGORITHM

In the first phase of our research, we focussed our attention on data distribution, local memory utilization, and load balancing issues. Previous IP algorithms distributed data statically to processors, which where then fetched and cached locally only on demand. We determined that instead of statically allocating parts of the volume to each processor, the local memory utilization could be optimized if *only* the required data were made locally available; that is, data which will be needed by a processor to generate its assigned portion of the screen. Depending on the local memory availability, this data allocation scheme can be extended to provide what is termed as *frame-to-frame* or *temporal coherence*, where data are allocated in such a way that a processor is self-sufficient to generate a number of slowly changing frames with no need for additional communication. In this chapter, we describe an IP PVR method, called *CellFlow*. In addition to exploiting temporal coherence, it achieves effective latency hiding by pre-fetching non-local objects, and attains significant load balance using an interleaved IP distribution mechanism.

*CellFlow* is an animation system that exploits frame coherence to implement a look-ahead scheme of object dataflow. The implementation of this scheme uses the communication features of modern scalable multi-computers to achieve good speedup by means of latency hiding. We demonstrate the performance of our approach in the field of volume rendering by implementing incremental rotation of the volumetric object about its center. The main advantages of the algorithm are: its simplicity, its optimal embedding in popular network topologies, and minimal, congestion-free communication among processors. Results are shown for implementation on the Cray T3D, a distributed memory 3D torus architecture. Computation and communication load balancing issues among processors are also addressed.

### 3.1 CellFlow: The General Method

CellFlow is a distributed memory parallel animation system which is used to generate a sequence of images by slightly changing the view position in each step. We have implemented one specific aspect of CellFlow - the incremental rotation of voxel-based objects, where the frames are rendered using parallel projection ray casting (Section 3.3), and shown how various types of coherency can be exploited to attain good speedup. We assume that during an animation process, the screen position changes in a smooth or incremental manner. The type of objects one needs to render (e.g. surfaces, volumes) and the rendering algorithm (e.g. z-buffer, ray casting) used to render the volume are not restricted by the CellFlow scheme.

In the CellFlow approach, each processor is responsible for producing the final image of exclusive screen partitions, called *regions* of the 2D screen. The 3D model (e.g. polymesh, volume) is divided into *cells* by embedding the model in a 3D regular grid (similar to [17][22]). These cells are the basic unit of communication, much like the notion of pages in memory systems. Whenever a processor determines that the extent of a cell lies within the space rendered to its screen region, this cell is transferred as a single unit to the processor, hence the name *CellFlow*. The initial data distribution is done in such a way that each processor has all the data required to generate the image for its region of the screen. In Figure 3.1, the dark grey squares represent those cells available locally at the processor that are assigned to render the region R from the viewpoint A. Thus, for screen position A no communication is needed between processors to generate the final image in region R.

In order to avoid the need for communication in the case of slight changes in rendering parameters, *padding* is provided around the initial data distribution (light grey squares in Figure 3.1), so that even if the viewer changes its position slightly, each processor can still generate the complete image of the region assigned to it with no need for data communication. The amount of padding which can be provided depends on the additional memory available at each node, and determines the range of viewing positions and orientations that can be rendered without the need of additional cells. This range of views is referred to as a *phase*. As the viewer position changes in small increments, it approaches the boundary of the phase. When this boundary is crossed, not all the data needed is available locally and communication between processors becomes inevitable.

Depending on the direction of movement of the viewer, we propose a *look-ahead* data acquisition scheme for latency hiding. As soon as the amount of effective padding falls below a certain limit, additional cells are

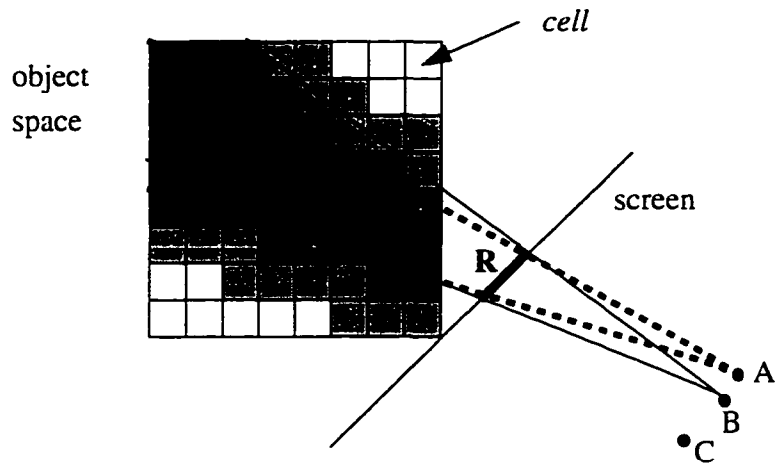


Figure 3.1 Object space is divided into cells. If all dark cells are locally available, the screen region R can be rendered from viewpoint A without any communication, but not from point B. If padding is also locally available (light grey cells) rendering from A, B and many other points in between does not require any communication. Some top white cells may be needed when the viewer moves to point C.

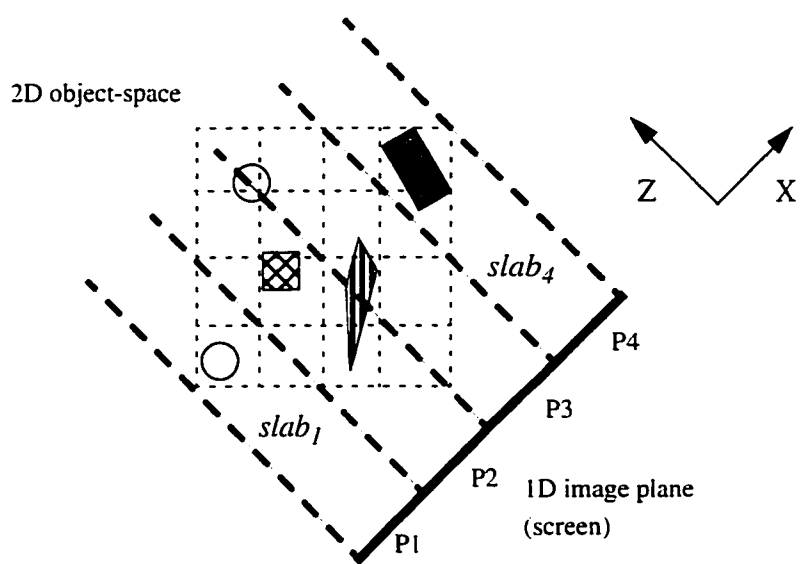


Figure 3.2 A set of 2D objects being projected on a 1D image plane, which is divided into 4 disjoint parts and assigned to 4 processors (P1, P2, P3, P4). The object data is distributed in such a way that the *i*th slab of data is sufficient to generate the final image for that processor without communication.

acquired from other processors. A minimum padding is thus dynamically maintained, and most of the communication latency is hidden by the rendering process. The processors continuously send requests to other processors for data required to maintain their padding. While these requests are carried out in the background by a dedicated communication processor (router), the processor is free to render the data that it currently has. If we assume gradual changes in viewing parameters, the padding will always guarantee that a processor never has to wait for a missing cell. In Figure 3.1, when the user moves from A to B, the system will request some of the top white cells in anticipation of continued motion towards point C.

Some replacement policy is used for discarding cells that are not needed. Depending on the direction of movement of the viewer, data which are outside the padding region and farthest away from the direction of viewer movement are discarded first. In Figure 3.1, when the user moves from A to B, the bottom part of the pad (light-grey area) will be discarded first.

In the next section, we discuss some design issues such as screen subdivision, optimal selection of the padding, load-balancing among processors, elimination of network congestion, and efficient algorithm embedding. The main motive for considering these issues is to minimize the overheads inherent in the parallelization process, thus attaining better scalability.

## **3.2 Design Issues**

### **3.2.1 Screen and Scene Subdivision**

Figure 3.2 shows a 1D screen that has been equally divided into four contiguous disjoint segments. Adjacent image space segments are assigned to adjacent processors (P1, P2, P3, and P4). The object space is partitioned into cells which are then distributed to processors in a way such that each processor will have in its local memory all the cells required to generate the final image of its screen segment (for parallel projection ray-casting). It is assumed that the 1D screen is oriented along the x-axis, with the z-axis perpendicular to the screen as shown in Figure 3.2. The portion of object-space 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. In addition, some padding is also provided to each processor on either side of its slab (not shown in Figure 3.2). The amount of padding depends on the amount of local memory available.

### **3.2.2 Load Balancing**

It is imperative that the rendering and communication load among the processors will be unbalanced with block distribution of the screen (as described above). Instead, a fine grain block-cyclic distribution scheme is used here. The screen is divided into many more parts than the number of processors. Then, these screen subspaces are distributed to the processors in a cyclic fashion. This static block-cyclic distribution of data provides a simple but efficient solution to load balancing. With increasing cyclicity, good load balancing can be achieved both in communication and computation.

### **3.2.3 Algorithm Embedding**

The computational mapping of the CellFlow algorithm to an underlying topology is done in such a way 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. A ring of processors fits best for the block-cyclic partitioning described above (for load balancing). The embedding of our algorithm is critical as a linear array or a ring can be optimally embedded in most popular topologies like the 2D and 3D meshes or the hypercube.

### **3.2.4 Network Congestion**

Network congestion is avoided by observing that all the cells needed by a processor to set up a subsequent phase are either already available at its nearest neighbors, or they will be available at the nearest neighbors before the end of the current phase. This implies that before the culmination of the current phase, every processor's needs can be fulfilled by its nearest neighbors only, irrespective of the size of the phase. This information is also useful for avoiding deadlock due to filling up of buffers.

### **3.2.5 Memory Mapping**

An efficient method of data storage in memory must be devised. The cells received at the end of a phase should be placed in memory along with the cells which are already present locally. The memory update scheme should ensure that cell retrieval during the rendering phase does not become very complicated and time consuming. We use a scheme similar to a complete directory for keeping track of all the available cells in memory, and a LIFO stack to keep track of free cell space in memory. Each processor maintains the local



availability information about all the cells in object-space and the location of all the cells which are present locally.

### 3.2.6 Latency Hiding

The algorithm utilizes the communication routers available with most modern multi-computers to hide the latency. Such routers provide a valuable resource for data acquisition in our method. Depending on the direction of movement of the screen, we resort to a look-ahead data acquisition scheme (also called a pre-fetching scheme). A processor predetermines the required cells (i.e., cells not available to maintain a minimum padding) using this information and initiates the communication process between the processors. The processors use their routers to send the requested data, so that at the end of the current phase, the requested data for the next phase has already arrived at the receive buffers of the requesting processors. This sort of look-ahead data acquisition eliminates communication time altogether (except message start-up times), effectively achieving wire-hiding or latency-hiding.

### 3.3 Volume Rendering Incremental Rotation

The parallel volume visualization technique described here for animating rotations of objects, exploits coherence between scenes (temporal coherence) when there is little change in the viewing parameters. 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 3.3. The 3D extension of the algorithm is described at the end of this section.

The image and the object space partitioning described above achieve a situation where neither communication of data nor combining of images is necessary to generate a single 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. It minimizes communication by requesting from other processors only those cells that are not currently available in the local memory. As seen in Figure 3.3a, when only a small change in the view angle is desired, most of the data are already available in the processor (black area). Some small amount of information is needed from other processors (grey area) and some information can be discarded after passing them to other processors that might need it (striped area). The two wedge shaped grey areas in Figure 3.3 correspond to the padding in the Cell-Flow scheme. This flow of voxel data between adjacent processors gave our earlier method its name: *Voxel-*

*Flow* [65]. As the rotation angle increases, the size of the black area decreases, that is, most of the data has to be brought in (communicated) from other processors.

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 hold cells needed for generating images for a range of rotational angles. Instead of assigning a slab of object-space to a processor, we assign a cross-section of the object-space which looks like a skewed X, as shown in Figure 3.3b. For generating a sequence 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. A larger range of angles would demand a larger memory to store a wider X-like cross-section of the object, and will inflict larger amounts of data replication.

The range of rotations where all the images can be generated with the available data is referred to as a *rotation phase*, which is a special case of the *general phase* we defined previously. The maximum angle by which the screen can be rotated within a rotation phase is referred to as the *phase rotation angle (PRA)*. In Figure 3.3b a phase with  $PRA=25^\circ$  is supported. As all the required cells are available at each processor for generating images within this range, no communication of cells is required within a phase. Communication takes place only after the screen reaches its final position in the PRA, when redistribution of data is needed. The communication is kept to a minimum by requesting and transferring only those cells which are not locally available but will be needed in the next rotation phase.

The 2D incremental rotation scheme described so far is extended to 3D by extruding the 2D object (see Figure 3.4). There is no subdivision of the volume in the extruded dimension (Y-dimension in this case). Each cell now contains a beam of voxels. The screen is divided in the form of stripes and distributed to the processors in the same manner as the segments in the 1D screen case. The advantage of this type of object and screen subdivision is that the storage and computational complexity of the postprocessing is independent of the Y dimension. A processor has to calculate the cell requirement for one slice of the volume only, and this remains the same for all the other slices. Rotations about the three axes can also be considered independently. For example, rotations purely about the Y-axis are handled exactly as in the 1D screen case. Absolutely no extra communication is required for screen rotating purely about the X-axis. Rotations about the screen's Z-axis can be attained simply by rotating the image itself.

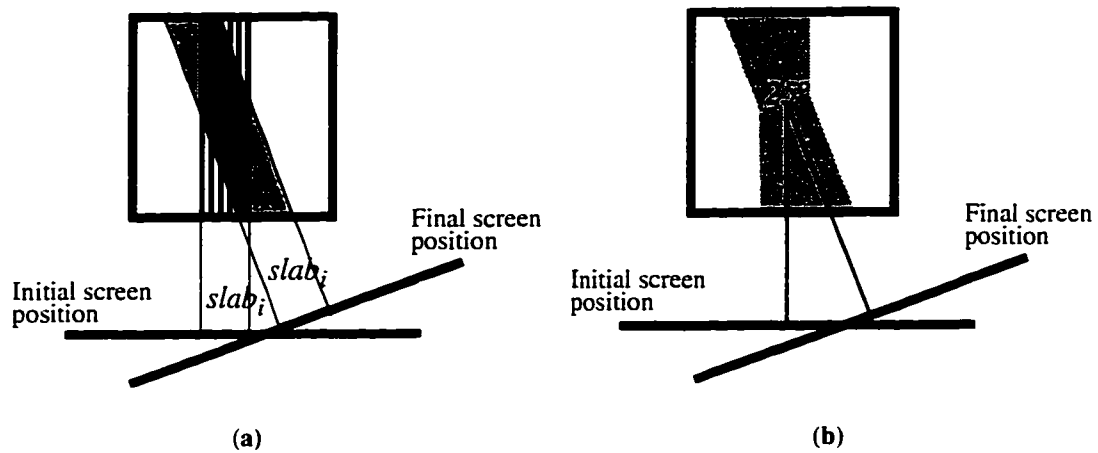


Figure 3.3 (a) When the viewpoint moves from the initial screen position to the final position the grey area contains the data elements that need to be brought in from other processors, the striped area can be discarded, and the black area the data that need not be moved. (b) If the processor contains all the information in the shaded area then all viewpoints in the  $25^\circ$  range can be accommodated without any communication.

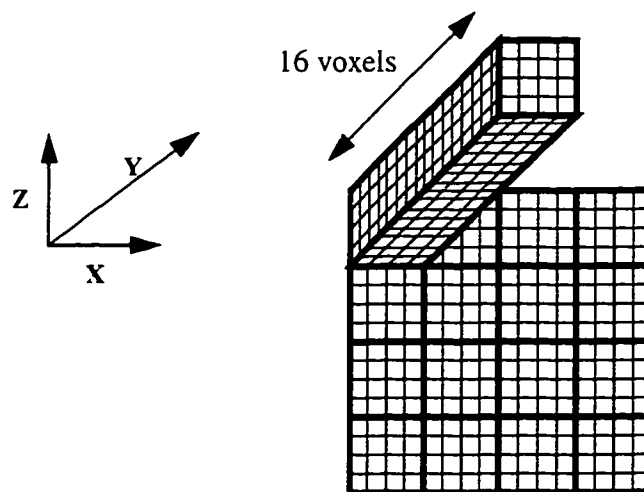


Figure 3.4 An object of size  $16 \times 16$  is divided into cells of size  $4 \times 4$  each. Each cell thus contains 16 voxels. The top-left cell is extruded into a 3D cell, which contains  $4 \times 16 \times 4$  voxels.

### 3.4 Implementation Details

The parallel incremental rotation system is implemented on the 128-processor Cray T3D available at the Ohio Supercomputer Center. It is a scalable massively parallel system, using PVM as the message-passing library. The animation process is distinctly divided into three stages. Each processor goes through a *preprocessing* stage, where the initial data distribution and screen assignments take place. Each subsequent phase is divided into two stages – the *rendering* stage and the *postprocessing* stage.

In the preprocessing stage, each processor assigns itself some contiguous regions of the screen. The number of regions equals the cyclicity of data distribution, and the position of the regions depends on the processor's id. A host processor reads the data from disk and distributes the cells to the appropriate processors. Each processor then requests additional cells to get ready for the first phase. During the rendering stage, the processors generate images for all the frames in the phase. The postprocessing stage is involved with the determination and acquisition of all the cells which will be needed for the next phase. To hide latency, the request for the cells needed for the next phase is sent before a processor enters the current rendering stage. Each processor sends some of the requested cells to the appropriate processors at the end of each frame generation. The processors also poll and empty their receive buffers using non-blocking receives at this time. By the end of the current rendering stage, all the cells needed to set up the next phase are available at the requesting processor, and they are now ready to enter the next phase without waiting for any more messages. In this way, the communication latency is totally hidden by the computation process (rendering stage), except the start-up costs.

The other effect of the interleaving process is to avoid deadlock due to filling up of buffers. By constantly polling and emptying the buffers, a processor avoids the indefinite filling up of its buffers. By sending only a small number of cells at the end of each rendering stage, a processor also avoids hot-spot congestion at other processors.

### 3.5 Results

The object migration parallel rendering method discussed in the previous sections has been implemented on Cray T3D, a scalable massively parallel system (MPP). The system currently supports 128 processing elements at The Ohio Supercomputer Center. 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 vir-

tual 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.

### 3.5.1 Scene and Screen Description

Table 3.1 summarizes the volume datasets used in the remainder of this dissertation. It shows the volume and screen sizes, the opacity used for each occupied voxel, and indicates whether the occupied voxels were illuminated. High opacity values indicate that the ray traversal will be halted as soon as the first occupied voxel is encountered, while low opacity values ensures that the ray will traverse its entire path inside the volume boundaries. In case of illuminated scenes, voxels were illuminated on the fly using three light sources. Image produced by each of the three algorithms described in this chapter and in Chapter 4 and Chapter 5 are exactly of the same quality and are shown in Figure 3.5. Only one ray was processed per pixel; volume sampling was done at unit intervals along the ray; and sample values were determined with zeroeth order interpolation.

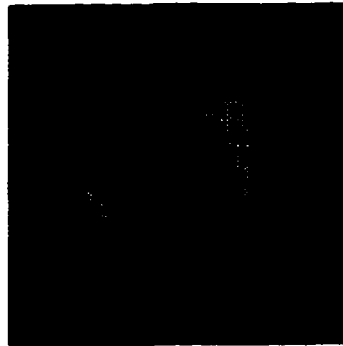
Volume	Volume Size	Cell Size	Screen Size	Opacity	Illumination	Image
SOD128	$128^3$	$8^3$	$128^2$	1.00	Y	Figure 3.5a
Simple128	$128^3$	$8^3$	$128^2$	0.01	N	Figure 3.5b
Head256	$256^3$	$16^3$	$256^2$	1.00	Y	Figure 3.5c
Capsid256	$256^3$	$16^3$	$256^2$	0.01	N	Figure 3.5d
Capsid512	$512^3$	$16^3$	$512^2$	0.01	N	Figure 3.5d

Table 3.1. Description of various volumes and respective screen sizes

Simple128 is a  $128^3$  volume consisting of two spheres and a cube. SOD128 is another  $128^3$  volume consisting of electron densities, and was obtained from UNC Chapel Hill volume repository. The volumes Capsid256 and Capsid512 each consist of 252 spheres. The spheres are organized in the shape of the molecular structure of a capsid, taking the shape of a dodecahedron. Head256 is a  $256^3$  volume obtained from CT-scan also taken from the UNC Chapel Hill volume repository.



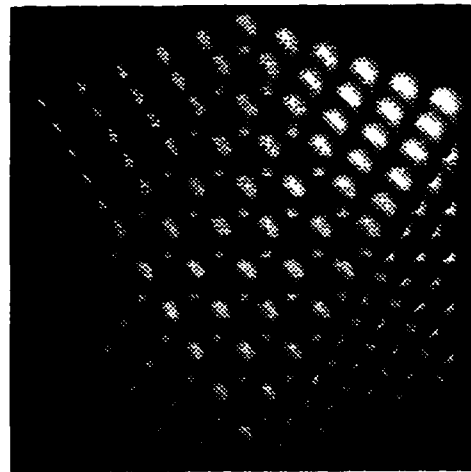
(a)



(b)



(c)



(d)

Figure 3.5 (a) SOD128, (b) Simple128, (c) Head256, (d) Capsid256 and Capsid512

The screen is rotated  $120^\circ$  about the y-axis in steps of  $1^\circ$ , the phase size being  $10^\circ$ , resulting in 12 phases. The timings for a complete rotation of  $360^\circ$  will be proportionally increased. The results shown below do not include the preprocessing time or the time taken to display the final images.

### 3.5.2 Load Balancing

Figure 3.6 shows the rendering times taken by each processor for various cyclicities to render the simple128 and capsid256 scenes described above. These results are shown for the animation process on 8 processors. It is evident that the rendering times reduce with increasing cyclicity.

For both the volumes, almost a 100% improvement can be seen in rendering times when cyclicity increases from one to four. In short, the static block-cyclic distribution scheme adopted in our method attains significant load balancing, and helps to reduce the total rendering times. From the graphs in Figure 3.6, it can also be seen that cyclicity 4 is sufficient for considerable load-balancing among processors for these objects.

### 3.5.3 Overheads

There are three kinds of overheads introduced by our parallelization process: a) postprocessing overheads, b) overheads due to sending the requested cells to the requesting processors, and c) overheads due to receiving and updating the local memory with cells obtained from other processors. These overheads are shown in Figure 3.7a. Compared with the rendering times (Figure 3.6), the timings shown in this figure are not very significant. This implies that except the start-up (receiving and sending), the wire latency of the cells transferred between processors are totally hidden by the rendering process. The sending times shown in Figure 3.7a are only the start-up times taken by each processor to send cells to other processors. The receiving times include both the receiving start-up times and the time taken to place the received cell at an appropriate place in memory. Because of this, the receiving times are slightly higher than the sending times, even though the number of cells sent and the number of cells received at the end of each phase are almost the same. The postprocessing time consists of the time to determine all the needed cells for the next phase and the start-up times to send these request messages to other processors.

Figure 3.7b shows the overheads incurred with a change in PRA (phase rotation angle). As expected, the total time taken for sending or receiving is independent of PRA. This is because for a total rotation of  $120^\circ$ , the total number of cells transferred between processors remains the same, irrespective of the size of a

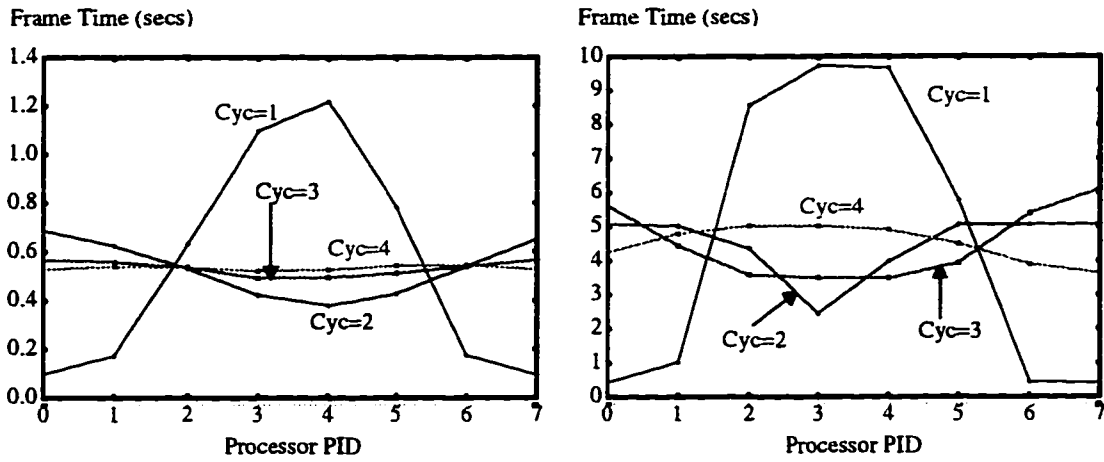


Figure 3.6 Frame times for (a) for Simple128 and (b) for Capsid256 volumes on 8 processors, and with cyclicity varying from 1 to 4.

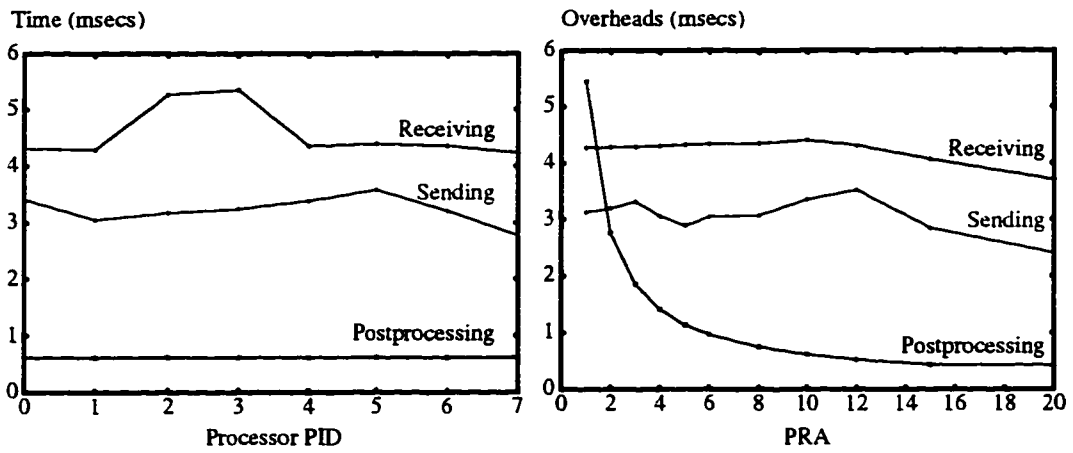


Figure 3.7 (a) Average Sending, Receiving, and Postprocessing overheads for each frame with a fixed PRA of  $10^\circ$  ( $P=8$ ,  $C=4$ ). (b) Average Sending, Receiving, and Postprocessing overheads with varying PRA ( $P=8$ ,  $C=4$ )



phase. If the phase size is large, then a large number of cells will be transferred a small number of times, and vice versa. The figure also shows that as the PRA increases (and therefore the number of phases decreases), the postprocessing time decreases proportionally. At the end of each phase, the postprocessing stage determines all the cells to set up the next phase. The time taken in this stage is independent of the size of the phase and approximately remains constant. Therefore, the total postprocessing time is directly proportional to the number of phases (or inversely proportional to the PRA). This suggests that the PRA should be maintained as large as possible depending on the memory availability at each node.

### 3.5.4 Scalability

Table 3.2 gives the frame times for several datasets on different number of processors, and Figure 3.8 deals with the scalability of our algorithm. An efficiency of more than 60% is seen for up to 128 processors. Almost real-time speeds of 20 frames/sec. for  $128^3$  volume and interactive speeds of 2.5 frames/sec. for  $256^3$  volumes have been achieved. The efficiency can be further improved by reducing the overheads. One way to reduce the sending and receiving times would be to combine a number of cells before sending. For example, if the send/receive buffers allow space for 4 cells to be packed into a single message, then 3 send/receive start-ups can be saved, leading to a reduction of sending cost by 3/4ths of the current value. As mentioned in Section 3.5.3, the total postprocessing time by each processor is directly proportional to the number of phases. Thus, although the rendering time is inversely proportional to the number of processors, the total postprocessing time by each processor remains independent of the number of processors. This situation tends to make the algorithm unscalable, as for a very high number of processors, the postprocessing time begins to dominate over the rendering time.

In more practical situations, the width of the slabs required by each processor decreases with increasing number of processors. Thus, for the same size of the object, the memory requirement of each processor reduces as the number of processors increases, allowing for a higher PRA. This reduces the total postprocessing time. In our experiments, we have just studied the effect of changing the number of processors with all other parameters remaining unaltered. The PRA dependency on memory availability will require a more detailed study of larger objects

Number of Processors	Simple128	SOD128	Head256	Capsid256
1	3.98	7.55	30.20	29.89
2	2.00	3.90	15.87	16.09
4	1.04	1.97	8.30	7.85
8	0.53	1.01	4.27	4.05
16	0.27	0.52	2.20	2.08
32	0.14	0.27	1.17	1.10
64	0.08	0.15	0.72	0.63
128	0.05	0.08	0.45	0.38

Table 3.2. Frame rendering times for various volumes as a function of number of processors. All times are in seconds.

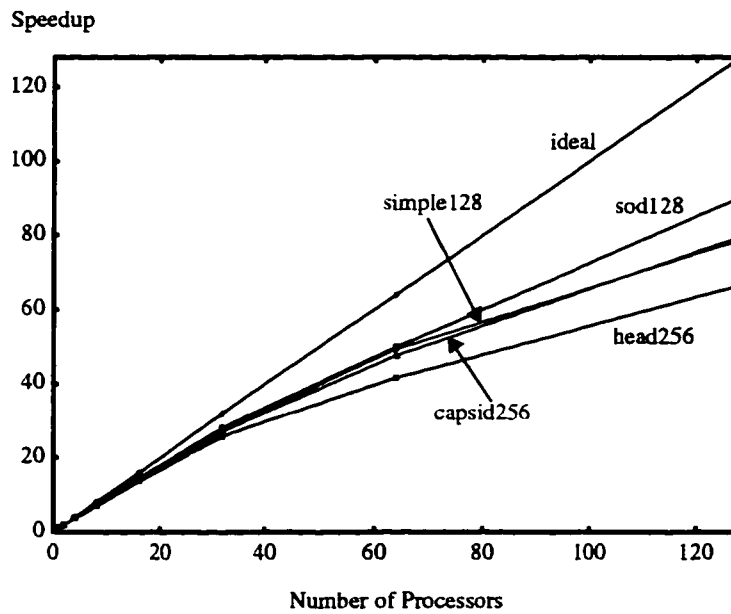


Figure 3.8 The speedups of the parallel incremental rotation algorithm for different number of processors.

### 3.5.5 Effect of PRA

Figure 3.9 shows the effect of PRA on the number of cells sent, received, and retained in each phase. From Figure 3.9a it can be seen that only a few cells are transferred at the end of each phase, compared to the number of cells which are already present (retained) at a node. The average fraction of the number of cells received at the end of a phase compared to the number of cells retained is shown in Figure 3.9b. As expected, the fraction increases with PRA, and for higher PRAs the curve tends to flatten, implying that for higher PRAs, the fraction of cells required does not change much. The reason for this is the fact that as the PRA is increased, an overlapping occurs between the various X-like slabs with a processor. With this overlap, some more cells can be found locally. This is also the reason why the “retained” curve in Figure 3.9a also begins to increase slightly for PRAs above 12°.

The *replication factor* is given by the ratio of the total number of cells in the entire system to the total number of cells in the entire volume. We have observed that the replication factor increases linearly, but not as fast as the PRA.

## 3.6 Discussion and Future Work

The main drawback of the system is that it is not particularly suited for rotations with large rotational steps. However, in most general purpose animation systems, abrupt changes in screen positions are rather rare. The primary advantage of the system lies in the simplicity by which good speedup is achieved. Another advantage of the method is its design for optimal embedding in most popular topologies. The screen and scene subdivision is done in such a way that there are no issues of image overlap, eliminating the need for image combining. The interleaving process between rendering and sending/receiving cells offers effective latency hiding and avoids deadlock due to filling up of buffers.

Since we have implemented the general CellFlow approach for one case of volume rendering, obvious future extensions include the exploration of arbitrary volume rotation, perspective volume rendering, and the application of the CellFlow approach to polygon rendering. Extensions to the basic CellFlow method include providing mechanisms to deal with non-incremental abrupt change in viewing parameters, and dealing with dynamic scenes. The idea of “rotational padding”, as used in this chapter, can also be extended to provide “general padding” to each processor, so that they maintain additional data for generating images for some frames which are close enough from the current frame.

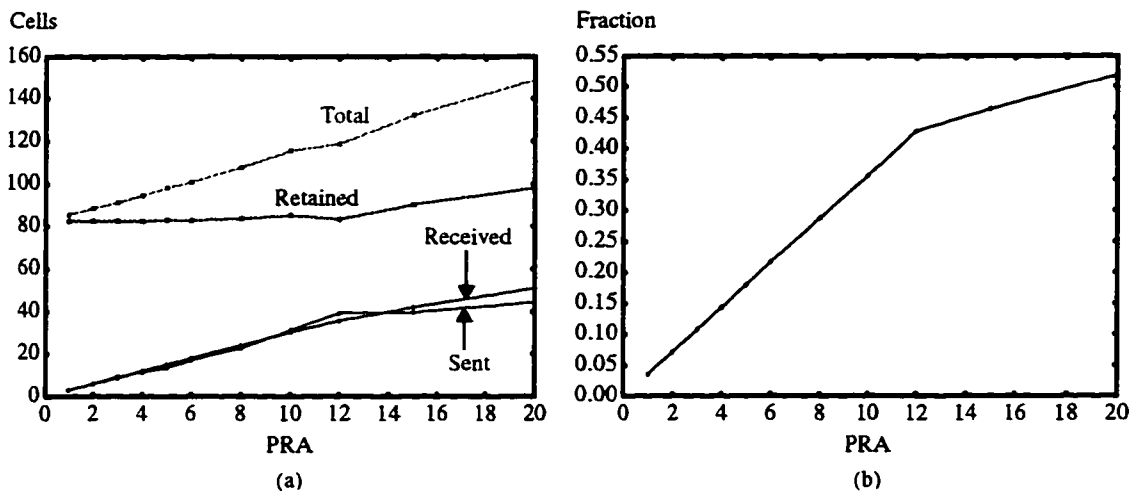


Figure 3.9 (a) Average number of cells sent, received, and retained per phase of rotation ( $P=8$ ,  $C=4$ ). The topmost curve is the total number of cells in memory at a time, which is the sum of the number of cells retained and the number of cells received. The bottom curve shows both the number of sends and receives. (b) Fraction of cells received at the end of each phase relative to the number of cells retained.

### **3.7 Conclusion**

In this chapter, we have proposed a distributed memory parallel animation scheme and implemented one of its specific applications – the incremental rotation of voxel-based objects with parallel projection ray casting. This rendering scheme finds good application in the fields of medicine (to view data from MRI or CT-scans) and CFD, which incorporate voluminous data impossible to fit in a general purpose uniprocessor memory. This scheme should be extended to perspective and polygonal models, in a way so as to reduce the overheads inherent in the postprocessing step of the current method, leading to better processor/channel utilization and speedup.

## CHAPTER 4

### EXPLOITING SPATIAL COHERENCY: THE *ActiveRay* ALGORITHM

The CellFlow method, presented in the previous chapter, finds good application when subsequent screen positions are known *a priori* and works well for incremental rotations of the screen. The second phase of the research aimed at removing some of the deficiencies of the CellFlow algorithm and was concerned with exploiting frame-to-frame coherence in a more general way, i.e., for real time animation where the screen positions are unpredictable.

In earlier object dataflow parallel rendering systems, the data representing the 3D scene is statically distributed among processors and objects are fetched and cached only on demand. Most previous object dataflow methods were implemented on shared memory architectures and exploited spatial coherence only to reduce hardware cache misses. In this chapter, we propose an efficient model for object dataflow parallel volume rendering on message passing machines. The 'Active Ray Tracing' algorithm is introduced and its ray storage mechanism is used to support latency hiding by postponing computation on inactive rays. Memory usage is optimized by letting objects migrate and replicate at different processors rather than the common static assignments. Our cache-only-memory approach uses a distributed-directory scheme to trace the location of objects at other nodes. A mechanism to minimize network congestion was implemented which optimizes channel utilization. Unlike previous methods, our approach can benefit from temporal coherence and effectively minimizes communication costs in successive frames. Finally, we propose an extension of the Active Ray approach to recursive ray tracing also.

#### 4.1 Distributed Memory Implementation of Parallel Rendering

In this work, we have adopted an object dataflow approach to parallel rendering on a distributed memory parallel machine. The screen is divided into several regions (in the form of tiles) which are assigned to the processors in a cyclic manner (Figure 4.1b). 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  $8 \times 8 \times 8$  voxels (Figure 4.1a). 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*. As a side benefit of cell decomposition, all empty cells in the volume may be detected and marked. At rendering time, the rays may skip the empty cells along their paths.

Each processor keeps track of a random, disjoint, subset of the cells, in a data structure called the *directory*. It 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 for a cell is referred to as the *home node* for that cell. 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 list*. Each ray in the list 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 due to unavailability of a (non-empty) cell, the ray becomes *inactive* and is therefore put back in the ray-list. At the same time, a request is sent to the home node of the missing cell. The processor then proceeds to the next active ray in the list, that is, a ray that was waiting for some cell and has now arrived. The algorithm repeatedly scans the ray-list until an active ray is found (in which case it is traced), or until the list becomes empty (in which case the algorithm moves to generate the next frame). In the case of recursive ray tracing a special illumination model, described in Section 4.2, has to be formulated to support this model of stopping the processing of one ray while computing another.

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 has a copy of the cell. No state information is needed, as 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 when objects are allowed to move in the scene, is suggested in the Section 4.5.

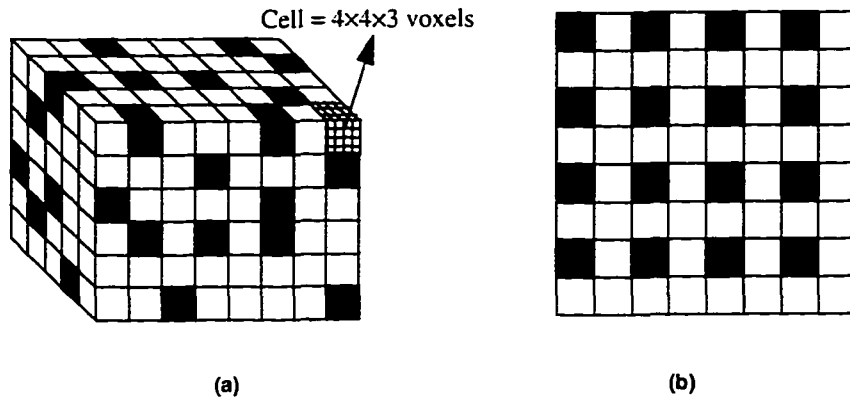


Figure 4.1. (a) A volume made up of  $32 \times 24 \times 15$  voxels is divided into  $8 \times 6 \times 5$  cells each of size  $4 \times 4 \times 3$  voxels. Each processor is home to 60 random cells in a 4-processor system. (b) A screen divided into 64 tiles of equal size and distributed cyclically to 4 processors, P1, P2, P3, and P4. For example, the black cells and the dark tiles are assigned to P1.

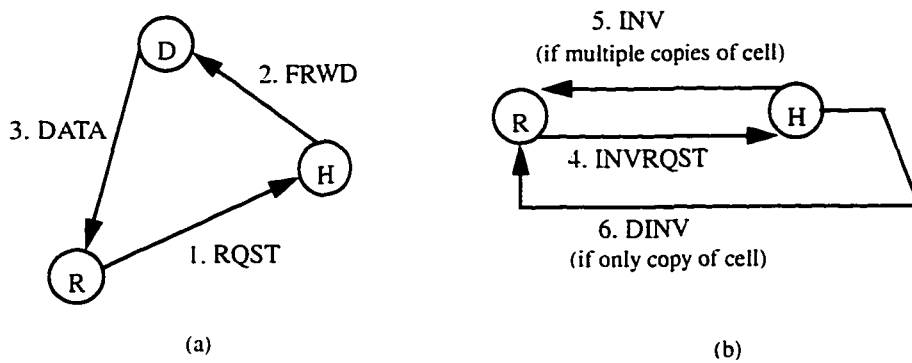


Figure 4.2. (a) A 3-hop system for requesting data (cells) from other processors. R is the requesting processor, H the home node, and D is the closest processor containing the requested cell. (b) A 2-hop invalidation process for discarding a cell from a processor's memory.



Six kinds of messages are used for maintaining the directories in the animation process. The first three correspond to a cell request, while the last three 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 4.2a. This procedure of acquiring a cell from another node is termed a *3-hop request system*, as 3 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 as 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 local memory stores only that data which are required during its own animation process, 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 in our application. As far as network congestion is concerned, forwarding of a cell to the closest processor may not exhibit significant advantages when implemented on MPPs with fast and tightly coupled communication networks (like the Cray T3D). At the same time, such considerations may prove beneficial when wide area distributed parallel renderers are designed with high latency interconnections.

In various instances, animations take place by making minor alterations to the viewing parameters. In such animation sequences, if one is careful to assign adjacent screen segments to adjacent processors, it is very likely that the requested cells will be found at and fetched from adjacent processors. This is the unique attribute of our approach that allows for the efficient exploitation of spatial, temporal, and network coherency.

In addition to the 3-hop mechanism for fetching cells, a 2-hop procedure is needed (Figure 4.2b) for invalidating cells that are no longer required locally, and to make space for other cells in turn. A processor that would like to discard a cell requires permission to do so from the home-node, since it might have the only copy of this cell in the 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, it 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 sends a message (DINV) to the processor, not to purge the cell. Deadlocks which may arise in these situations are averted as described in Section 4.3.5.
5. **INV**: Upon receiving this message, a processor is sure that it can purge the cell from its memory, and does so.
6. **DINV**: Upon receiving this message, the processor puts back the cell at the end of the LRU list.

The different types of messages described above are handled similarly to polling-based Active Messages [29]. While proceeding from one ray to the next in the list, buffers are polled to check for any pending messages using non-blocking probes. The messages, if any, are received and appropriate action 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 with 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. This is described further in Section 4.3.3. 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 receive buffers, and by sending only a small number of messages after tracing each ray, a processor avoids hot-spot congestion at other nodes.

## 4.2 Illumination Model for Active Ray Tracing

In the basis of the 'active ray tracing' approach lies our ability to postpone the computation of an inactive ray. In the case of ray casting this poses no major problem. However, in the case of recursive ray tracing, where the value of the primary ray (shot from the eye) depends on the values computed by the secondary rays, we must be able to render these rays independently and then be able to update the pixel value when their partial results become available.

The intensity at a screen pixel  $(P_x, P_y)$  is denoted by  $I_\lambda(P_x, P_y)$ , where  $\lambda$  indicates wavelength (e.g., RGB). If the ray that passes through  $(P_x, P_y)$  does not intersect any object,  $I_\lambda(P_x, P_y)$  is set to some background intensity. If that ray intersects some object  $O$ , then  $I_\lambda(P_x, P_y)$  is assigned the intensity of  $O$ 's surface at the intersection point. This is typically computed by some rendition of the Global Illumination Equation:

$$I_\lambda(O_x, O_y, O_z) = I_{a\lambda}k_aO_{d\lambda} + \sum_{i=1}^m S_iA_iI_{\lambda i} [k_dO_{d\lambda}(N \cdot L_i) + k_s(N \cdot H_i)^n] + k_sI_{r\lambda} + k_tI_{t\lambda} \quad (1)$$

where:

$O_x, O_y, O_z$  coordinates of the 3D point that we illuminate.

$I_{a\lambda}$  ambient light intensity.

$O_{d\lambda}$  diffuse color of object.

$m$  number of light sources.

$k_a, k_d, k_s, k_t$  ambient, diffuse, specular, and transmission coefficients.

$S_i$  light-source  $i$  occlusion factor.

$A_i$  light-source  $i$  attenuation factor.

$I_{\lambda i}$  light-source  $i$  intensity.

$N$  surface normal at  $O_x, O_y, O_z$ .

$L_i$  direction to light-source  $i$  from  $O_x, O_y, O_z$ .

$H_i$  half-way vector between  $N$  and the viewer.

$n$  specular reflection exponent.

$I_{r\lambda}$  intensity of the reflected ray.

$I_{t\lambda}$  intensity of the transmitted ray.

The first two terms in Equation 1 compute what is commonly called *local illumination*. The last two expressions compute the *global illumination*; they model the influence of other objects in the environment. The terms  $S_i$  are computed by following shadow feelers to the light sources. The values of  $I_{r\lambda}$  and  $I_{t\lambda}$  are computed by recursively tracing secondary rays (such as reflection and transmission rays, respectively). As long as this recursive computation of the global illumination is not complete, no value can be assigned to the pixel at  $(P_x, P_y)$ . The illumination equation used by our active ray tracer is basically a reordering of the summation in Equation 1 which allows for non-recursive handling of secondary rays. Likewise, the processing of the shadow-feelers differs from the way traditional ray tracers handle these rays since the active ray tracer might defer processing shadow feelers until a later time.

In the active ray tracing approach, we give each ray a unique identifier number  $\psi$ . Given a ray  $\psi$ , its parent is denoted by  $P(\psi)$ . For primary rays  $P(\psi)$  is undefined.

We associate, with each ray, a weight  $\omega(\psi)$  and defined:

$$\omega(\psi) = \begin{cases} 1 & \text{primary ray} \\ k_s \omega_{P(\psi)} & \text{reflection ray} \\ k_t \omega_{P(\psi)} & \text{transmission ray} \end{cases} \quad (2)$$

We denote by  $C(\psi)$  the coordinate of the pixel the ray  $\psi$  belongs to. That is, a primary ray traced through the pixel at  $(P_x, P_y)$  and all its descendants will have the same value of  $C(\psi) = (P_x, P_y)$ . We denote by  $R_\lambda^\psi$  the intensity contributed by the local illumination components in Equation 1 at the point where a ray  $\psi$  first intersects an object. We also assume that  $S_i = 1$ ; that is, for the time being we do not deal with shadowing effects. We can now extract from Equation 1 the following expression for  $R_\lambda^\psi$ :

$$R_\lambda^\psi = I_{a\lambda} k_a O_{d\lambda} + \sum_{i=1}^m A_i I_{\lambda i} [k_d O_{d\lambda} (N \cdot L_i) + k_s (N \cdot H_i)^n] \quad (3)$$

If we expand the recursive terms in Equation 1,  $I_{r\lambda}$  and  $I_{t\lambda}$ , we will eventually have a large sum solely consisting of terms similar to the ones in Equation 3, namely, local illumination computations at the intersection points of all the rays in the ray tree. Therefore, the intensity of the pixel  $I_\lambda(P_x, P_y)$  can be formulated as the sum of all the local illuminations, as defined in Equation 3, computed by all the rays of the ray tree rooted at  $(P_x, P_y)$ . That is

$$I_{\lambda}(P_x, P_y) = \sum_{\forall \psi: C(\psi) = (P_x, P_y)} \omega(\psi) R_{\lambda}^{\psi} \quad (4)$$

At run time, when a ray  $\psi$  is popped from the queue and traced, the resulting value is added to the pixel value by:

$$I_{\lambda}(C(\psi)) = I_{\lambda}(C(\psi)) + \omega(\psi) R_{\lambda}^{\psi} \quad (5)$$

In Equation 3 we assume that no occlusion exists between the point of ray-object intersection and any of the light sources  $i$ , hence the unity assumption for  $S_i$ . At a later stage, shadow-feelers are processed. At that time, if a light source  $i$  is (partially) occluded, some light intensity should be subtracted from the pixel.

Each shadow ray, when spawned by ray  $\psi$  and sent to light source  $i$ , is associated with light intensity denoted by  $I_{\lambda i}^{\psi}$ , which we define to be

$$I_{\lambda i}^{\psi} = A_i I_{\lambda i} [k_d O_{d\lambda} (N \cdot L_i) + k_s (N \cdot H_i)^n] \quad (6)$$

that is,  $I_{\lambda i}^{\psi}$  is the intensity contributed by the light source  $i$  to ray  $\psi$  (following the assumption  $S_i = 1$ ). Now Equation 3 becomes:

$$R_{\lambda}^{\psi} = I_{a\lambda} k_a O_{d\lambda} + \sum_{i=1}^m S_i I_{\lambda i}^{\psi} \quad (7)$$

which can be re-written as:

$$R_{\lambda}^{\psi} = I_{a\lambda} k_a O_{d\lambda} + \sum_{i=1}^m I_{\lambda i}^{\psi} - \sum_{i=1}^m (1-S_i) I_{\lambda i}^{\psi} \quad (8)$$

Therefore, at run time, when a shadow-feeler, spawned by ray  $\psi$  towards light source  $i$ , is traced, it returns the value denoted by  $S_i$ . Then, the value of the pixel at  $C(\psi)$  is adjusted by:

$$I_{\lambda}(C(\psi)) = I_{\lambda}(C(\psi)) - \omega(\psi) (1-S_i) I_{\lambda i}^{\psi} \quad (9)$$

## **4.3 Design Issues**

### **4.3.1 Load Balancing**

Rendering and communication times for different processors indicated that the load among the processors is unbalanced with block distribution of the screen (see Section 4.4.1). We resort to a static block-cyclic screen distribution to attain considerable load-balancing among processors. Results show that the amount of computational and communicational load balancing increases with increasing cyclicity for various scenes.

### **4.3.2 Network Congestion**

Network congestion is reduced by forwarding the cell request to the processor closest to the requesting processor containing a copy of the requested cell. The request (RQST) and forward (FRWD) messages just contain information about the requested cell and the requesting processor. These small messages do not consume much network bandwidth. Moreover, by forwarding the requested message to the closest processor, a small number of links are used to deliver the cell (possibly a large message) to the requested processor. Since communication overhead can be extremely low on some architectures, it could be the case that such optimization is not really necessary.

### **4.3.3 Latency Hiding**

The latency of the unavailable cells is hidden by not allowing a processor to explicitly wait for a requested cell. A list of unfinished rays is used for this purpose. At the start of a frame generation, a local ray-list contains all the incomplete rays to be traced by the processor. Rays which traverse to completion using the locally available cells are removed from the list. If, at any time, a ray is blocked due to the unavailability of a cell, the processor sends a request for the required cell, and proceeds with the next unfinished ray in the list. It iterates through the list until all the rays finish tracing. While the processor traverses the list of unfinished rays, the requested cell travels to the node. Results for ray-casting show that after the initial frame, 2-3 iterations of the list are sufficient to advance all the rays to completion.

Earlier distributed memory object dataflow algorithms did not hide the latency of these requests. Some implementations (shared memory in particular) may use pre-fetch mechanisms to hide latency, but for rendering algorithms, it often becomes difficult to determine which objects need to be fetched, and so the processor must explicitly wait for the requested cell.

#### **4.3.4 Memory Overheads**

The maintenance of the directory, the hiding of the latency, and the memory management of cells pay a toll as memory overheads in the algorithm. These overheads can be put into two categories: overheads for maintaining the objects, and overheads for maintaining the rays. We discuss each of these below.

Local memory available for data is segmented into contiguous regions, and each region is used to store the contents of a cell. The object overheads consist of maintaining the directory for the home cells, for keeping track of free regions in data memory, and for storing all the information of each cell in the volume. A list is also used to maintain the LRU (Least Recently Used) for cell replacement.

The size of a local directory is the number of cells a node is home to, which equals the total number of cells in the volume divided by the number of processors. Each cell can potentially be shared by all the processors. The width of the directory thus equals the number of processors. The total memory overhead for maintaining the directory in the entire system therefore becomes proportional to the number of cells in the volume.

To keep track of free regions available to store newly fetched cells, a list of free pointers is maintained. The size of the list equals the total number of cells that can fit in the local data memory. For each cell in the volume, a processor stores its home node, its local status (valid, invalid, or requested), a pointer to the location where it is stored in local memory, and a pointer to its location in the directory (if it is itself the home node).

The memory overheads due to the rays consist of maintaining the ray list, and some additional information stored for each ray. The ray list is an array of pointers indicating all the unfinished rays. As the rays are not traced to completion, information like their starting point, direction, and accumulated opacity need to be retained in addition to their accumulated colors. The length of the ray-list and the ray information list equals the total number of rays to be traced by a local processor. The number of rays traced by each processor is given by the screen size divided by the number of processors.

#### **4.3.5 Deadlock Avoidance**

As mentioned earlier, deadlock due to filling up of communication buffers is avoided by frequently polling and emptying the buffers. Deadlock can also happen due to filling up of the software cache. The potential for such a situation is detected when a certain percentage of local memory is filled by the cells. In this case, instead of sending an invalidation request to the home node, the cell itself is sent back to the home node. In

the worst case, all the cells will be sent back to their respective home nodes. We keep two levels for cell replacement in local memory, first (about 80%) to invoke the LRU replacement of cells, and the other (about 95%) to avoid deadlock. This means that when the local memory is 80% full, cells are removed using the INVRQST-INV protocol, and when the local memory is 95% full, then the cells are simply sent to their home nodes. For ray casting, it is seen that the above figures are sufficient to avoid this type of deadlock.

## **4.4 Results**

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

### **4.4.1 Load Balancing**

Figure 4.3 shows the times spent by each processor for rendering the simple128 and capsid256 volumes on 8 processors. Severe load imbalance exists for larger tile sizes, while it gets equally distributed as tile size becomes smaller. Load balance also results in decreasing the average time required to generate each frame. From these figures we see that tile size of 8×8 is sufficient to attain ample load balancing.

### **4.4.2 Effect of Irregular Animation**

It is obvious that our method benefits from temporal coherency. That is, when the change in viewing parameters is incremental, only small number of cells will be fetched and this communication cost can be hidden. We tested the behavior of our system in a case where the animation is not smooth; after every ten frames the eye is made to 'jump' to an arbitrary new position in space. The number of cells fetched is shown in Figure 4.4. This test was conducted on sixteen processors rendering the capsid256 volume. Despite the extended communication, most of this overhead can be effectively hidden by the ray storage mechanism. Rendering times were 1.99 seconds per frame for the smooth animation (3° per frame), compared to 2.33 seconds per frame for the 'jumpy' one. We also observed that in either case the number of ray-list traversals never exceed four.



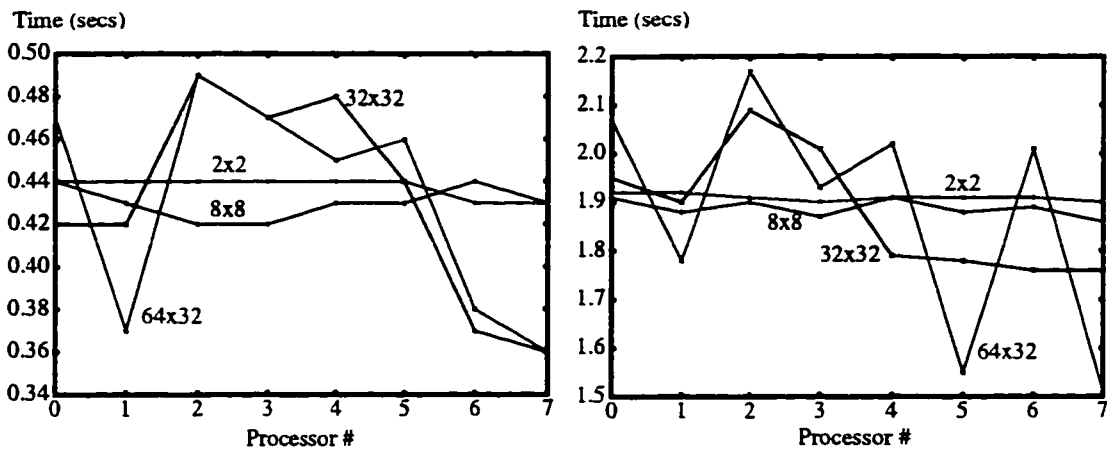


Figure 4.3. Average times spent by each processor for rendering each frame as a function of tile size, (a) for simple128, and (b) for capsid256 scenes.

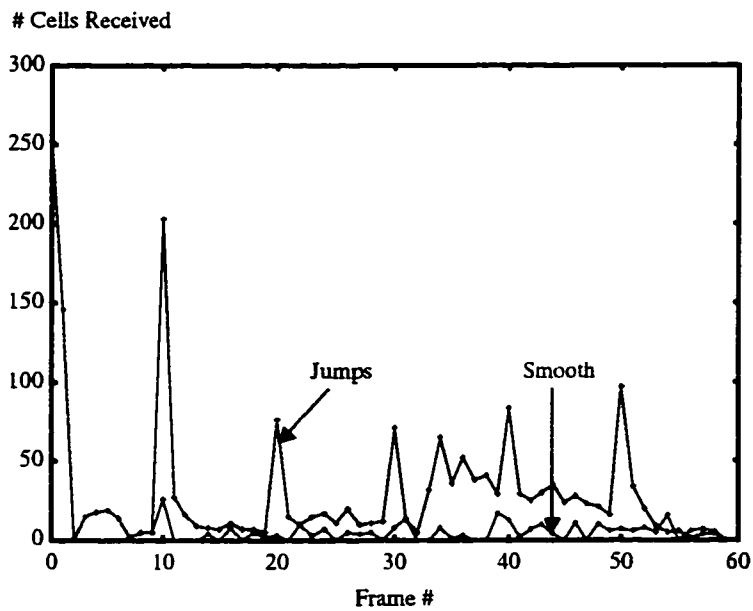


Figure 4.4. Communication overhead when the eye 'jumps' after every ten frames, compared to a smooth animation.

### 4.4.3 Effect of Software-Cache Size

When cache size (CS) is decreased we expect more thrashing to happen since some cells that are removed from a small cache might be needed again while rendering the same frame. Figure 4.5 shows the normalized rendering times for generating images for the simple128 and capsid256 volumes on 16 processors. Times are normalized with respect to the time taken with complete replication, i.e., with CS = 16. We observe that when the total amount of cache in all sixteen processors approach 4–6 times the volume size, processing time stabilizes. This indicates that in the case of rendering a  $256^3$  volume (16MB) on 16 processors, we will not need more than 4MB software-cache at each node to perform optimally. As a comparison to a parallel ray caster with static memory allocation, frame times were 1.64 seconds at cache size 1.2 and 0.71 seconds at cache size 3.0 while rendering the simple128 volume on 8 processors. The corresponding times for the ActiveRay algorithm were 1.05 seconds and 0.632 seconds, respectively.

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 cells fetched. The second is the overhead of our algorithm for scanning the ray-list many more times. The cost involved in these two sources of slowdown are demonstrated in Figure 4.6.

### 4.4.4 Effect of Cell Size

We also tested the algorithm's performance as a function of the cell size. Cell sizes, as used here, are analogous to the size of cache lines in SM or DSM machines. In Table 4.1 times are shown for cell size of  $8^3$  and up to  $32^3$  voxels (because of communication buffer size limitations, larger cell sizes are extremely inefficient). Times decrease and then increase as cell size increases. Average frame times (in seconds) are shown for rendering the capsid256 volume on 16 processors using software-cache size (CS) of 3.5 times the volume size and for rendering the capsid512 volume on 32 processors. The reasons for the time decrease are because

Cell Size	Capsid256 on 16 nodes	Capsid512 on 32 nodes
$8^3$	2.11	-
$16^3$	1.33	4.04
$32^3$	2.97	2.89

Table 4.1. Rendering times as a function of cell size.

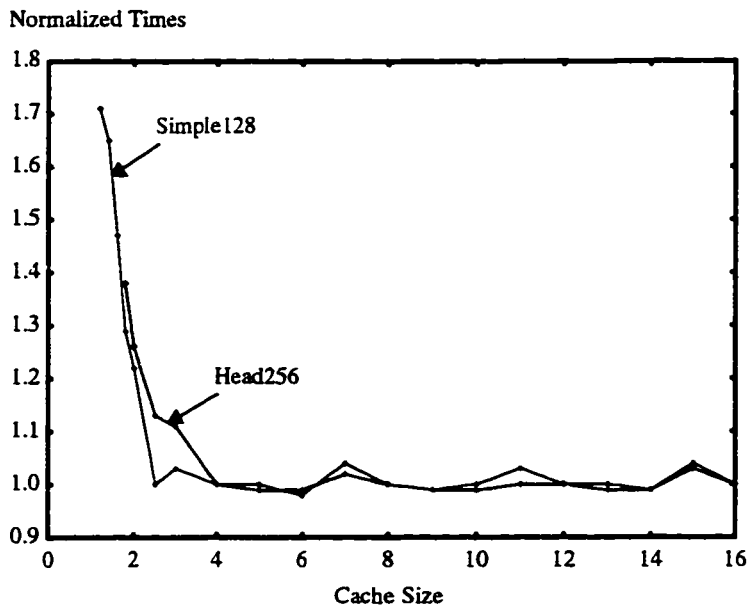


Figure 4.5. Rendering time as a function of software-cache size on 16 processors. The total amount of cache in all sixteen processors is equal to  $CS \cdot \text{volume size}$  ( $CS = \text{cache size}$ ).

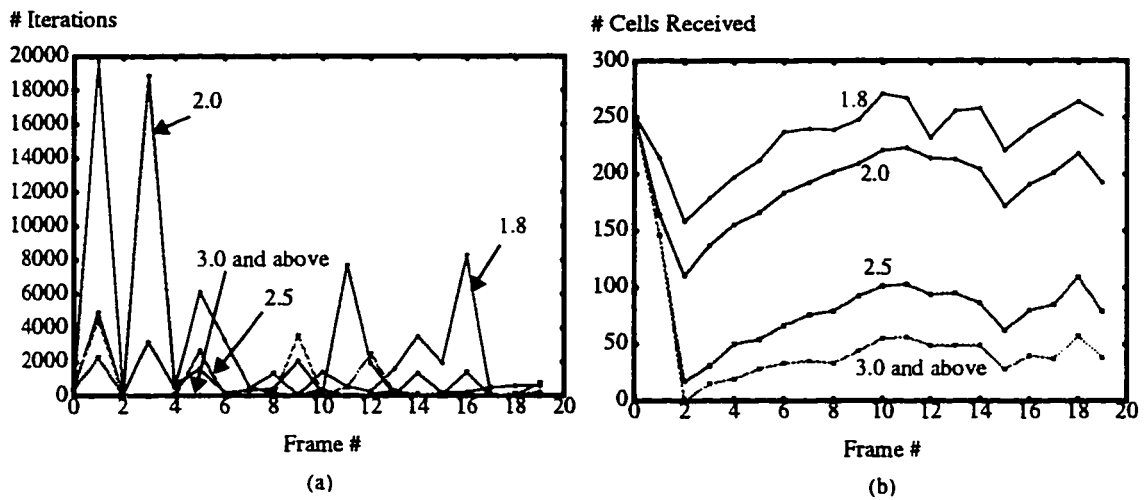


Figure 4.6. (a) Number of iterations through the ray list as a function of software-cache size while rendering twenty frames. (b) Average number of cells received by each processor as a function of software-cache size.

rays have to intersect less cells, ray-list management is reduced, less communication start-ups are needed, and less LRU list maintenance is needed. However, when cell size increases even further, less cells are labeled as empty cells and therefore less cells are skipped. Also, communication of larger cells may lead to increase in network congestion. Thus, a compromise must be reached which gives optimal performance.

#### **4.4.5 Scalability**

Table 4.2 shows the average frame times (in seconds) taken to render the simple128, sod128, head256, capsid256, and capsid512 volumes, respectively. Times for small number of processors are not available in the case of capsid512 volume due to lack of memory.

For volumes employing early ray termination (e.g., head256), the algorithm shows 15%-20% improvement in timing, compared to the times reported in Table 4.1, attributed mainly to early ray termination and less cell communication. The graph in Figure 4.7 demonstrate the excellent speedup achieved with our algorithm. The fastest known algorithms achieve 75% speedup on 16 processors [63], or 50% and 30% on 64 processors and 128 processors [3], respectively. Our implementation surpasses these as well as all other recently reported algorithms. We attribute this speedup to two main reasons: the first is our ability to conceal communication overhead by effective latency hiding based on ray storage; the second is the fact that other overheads, such as, send and receive start-ups, and directory maintenance are negligible and amount to approximately 4% of the total time. The maintenance of the LRU replacement policy is somewhat costly (about 5%) and tuning the system to the optimal cell and software-cache sizes can yield significant benefits as shown in Section 4.4.3. The efficiencies for the capsid512 volume are measured relative to the timing for 8 processors.

Number of Processors	Simple128	SOD128	Head256	Capsid256	Capsid512
1	3.47	5.86	16.25	15.62	–
2	1.76	2.92	7.98	7.66	–
4	0.89	1.51	4.07	3.89	–
8	0.45	0.80	2.18	1.94	13.97
16	0.23	0.42	1.14	1.01	7.19
32	0.12	0.23	0.59	0.53	3.69
64	0.07	0.13	0.33	0.29	1.98
128	0.04	0.07	0.17	0.16	1.09

Table 4.2. Frame rendering times for various volumes as a function of number of processors. All times are in seconds.

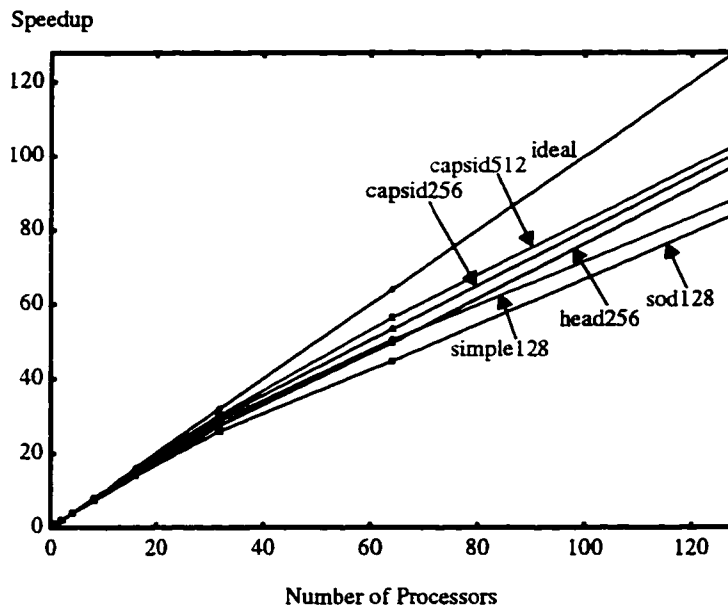


Figure 4.7. Speedup results for rendering several volumes on 1 to 128 processors.

## 4.5 Discussion and Future Work

Object-dataflow approaches are known to take advantage of spatial coherency to reduce the number of misses [22][42]. For sufficient local memory sizes, this helps to reduce the amount of communication between processors compared with ray-dataflow approaches. The hit ratio can further be increased if frame-to-frame coherence is also exploited in an animation sequence. The hit ratio directly effects the overheads incurred by the system. Each object fetched from non-local memory incurs a start-up cost, and if it is not a pre-fetch data acquisition scheme, it also incurs a latency cost. These costs are very low if shared memory machines having the ability to pre-fetch non-local data are used - and as such, these overheads are small compared to the rendering cost. On distributed memory machines, the start-up cost is significant and rapidly becomes a performance degradation factor if coherency is not exploited. A distributed memory algorithm should thus be designed in a manner so as to reduce start-up costs (or the number of times non-local objects are fetched). Combining this with the data acquisition scheme described here brings the performance of the shared memory and the distributed memory systems closer.

In the algorithm presented in this chapter no memory is used for static data allocation as used in earlier object dataflow schemes. This implies that our algorithm can exploit spatial and temporal coherency to a much higher degree than the earlier algorithms. With increasing scene sizes, the earlier algorithms will have decreasing software-cache sizes. Our algorithm, on the other hand, utilizes the complete local memory more efficiently by storing only that data required for the generation of the current frame.

The primary advantage of the distributed-directory algorithm lies in its ability to attain comparable speedup with shared memory implementations. It exploits both spatial and temporal coherency to minimize network communication. It also uses a ray storage mechanism to hide latency altogether.

We have used the distributed directory scheme for animating volumes by incrementally changing the view parameters. Future extensions of the method include allowing changes in the volume, and implementation for ray tracing of polygonal objects. Changes in the volume can be incorporated by invalidating all the copies of the effected cells, except the one at the processor that updates the cell.

We implemented our approach on the Cray T3D. Nevertheless, we make the claim that this approach is rather general. First, it is obvious that it can be implemented on any distributed memory architecture, including distributed shared memory, and NOWs (Network Of Workstations). We plan to demonstrate that the

unique ability of our methodology to avoid communication penalty by latency hiding has an even greater impact on these architectures when rendering high resolution volumes.

The extra hop used by the algorithm for requesting cells may prove quite detrimental when ActiveRay is implemented on NOWs connected by a shared medium (like the Ethernet or FDDI). The shared interconnection precludes any distance relationship between workstations, and thus the 3-hop cell request mechanism becomes meaningless. On the brighter side, the 3-hop system will be highly beneficial in non-shared interconnection medium as evidenced in distributed computing across WANs. This will also be advantageous in ATM-connected workstations, where switches decouple the communication taking place in parallel in disjoint parts of the network. The difference between an Ethernet-connection environment and an ATM-connection environment is the same as that between SM (using a globally shared bus) and DSM (using a scalable interconnection, with routers at each node).

Although we have implemented a volumetric variation of the ray casting algorithm, our approach is certainly applicable to ray casting (and ray tracers as shown in Section 4.2) of polygon-based models. We plan to extend our algorithm and evaluate its performance in these scenarios.

## **4.6 Conclusion**

In this chapter, we have described a distributed memory scheme for rendering animations of 3D scenes. The method 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 spatial, temporal, and network coherency in an animation sequence to reduce communication between nodes. The main advantages of the method are that the data are not statically distributed among the processors, leading to better local memory utilization. Effective latency hiding, reducing network congestion, and static load balancing are some other factors contributing to the good speedup. The method is easily extendible to ray tracing of polygonal objects also.

## CHAPTER 5

### EXPLOITING VOLUME COHERENCY: THE *RayFront* ALGORITHM

Having optimized the memory utilization and scalability of our algorithms (Chapter 4), we turned our attention to rendering colossal volumes: datasets too large to fit in the total memory of the ensemble of processors. Sometimes such volumes are even stored on remote disks in a compressed form. *Thrashing* is a phenomenon which is often encountered in such situations, where the same objects are fetched multiple times (either from remote processors or from disk) during the generation of a single frame. In the third phase of our research, we devised a method which is not only thrashless for a single frame generation, but retains its thrashless property across a number of slowly changing frame positions.

The RayFront scheme proposed here is an IP method implemented with object-dataflow. The algorithm capitalizes on the advantages of both the image-order and the object-order traversal schemes. The method is basically image-order, and thus all the advantages of the image-order scheme are preserved. In addition, the proposed voxel fetching mechanism totally eliminates thrashing, and thus exploits object-space coherency as in the object-order methods. We have also been able to avoid thrashing across a number of images generated for several screen positions. The combination of these attributes results in a system with most coherent screen traversal so much so that voxels once fetched and used are not needed again for a number of frames. Our scheme eliminates one of the burning problems encountered in parallel rendering of huge datasets, viz., thrashing. For colossal volumes, this provides significant advantage over existing methods. The algorithm also predetermines the order of the non-local cells to be processed, which facilitates effective latency hiding for even minimal cache sizes. Finally, the data structures we employ circumvent the need to search for the rays that should be traversed next.



## 5.1 Exploiting Coherency for Efficient Rendering

In their classic paper, Sutherland, et al., [108] have described coherency as the extent to which the environment, or the picture of it, is locally constant. In the present context, coherency refers to the degree to which an object required for one ray is used again for other rays, or objects used for the generation of one image (frame) are used again for another frame. In object-dataflow parallel ray-tracing systems, non-local objects are fetched from other processors on demand and are cached locally. With a coherent screen traversal, these objects are likely to be used again for subsequent rays in the current frame. If the cache is large enough, then the system can even take advantage of frame-to-frame coherency.

If the cache is not large enough, then it begins to *thrash*. Thrashing is manifested as the repeated transfer of the same data to the same processing node [22]. If a processor's cache cannot hold the number of blocks that it needs to render a single ray, then a cyclic refill of the cache will occur for each ray. As the size of the database increases, the effect of thrashing becomes more visible. In this respect, databases acquired from scientific sources, like sampled data from MRI, CT-scan, or CFD, are enormous, and rendering such scenes on a uniprocessor machine becomes extremely time consuming. To make the visualization of such databases more feasible, efficient parallel algorithms are being designed and implemented. The communication overheads depend on how much coherency is exploited by the algorithm, and thus how much thrashing is avoided.

Thrashing of objects in cache influences the performance in uniprocessor ray casting systems as well since the same objects can be cached a number of times. For each cache miss, a penalty has to be paid for fetching the required object from main memory into the cache. The situation is even more aggravating when processing very large datasets that do not fit into the processor's main memory, and has to be fetched from disk. One would like to avoid thrashing to improve the performance of the renderer. This penalty also becomes prominent when objects are fetched from remote memories in parallel rendering systems. Additional factors crop in when objects travel across the network. The latency is effected by network speed, which is particularly detrimental in case of distributed implementations where communication between computers are sometimes carried over slow links (e.g., Ethernet). In addition, network contention and size of the fetched data all play a combined role to increase latency and decrease performance, and therefore the scalability of the algorithm.

In view of this, it becomes particularly important to exploit object-space coherency so that objects once fetched are maximally utilized, and to ensure that objects once replaced in the cache will not be required

again, thus avoiding *thrashing*. Replacement in this context, is not due to different virtual addresses mapping to the same cache location (*conflict replacements*), but due to unavailability of space in the cache (*capacity replacements*).

Visualizing colossal databases, as in the case of scientific visualization, on limited memory multiprocessor systems are prone to thrashing. Equivalently, the performance of shared memory systems with very small caches also degrade for the same reason. These databases sometimes become so huge that they have to be stored in a compressed form on remote disks. Objects needed are decompressed and fetched on demand using the bottleneck I/O channels. Finally, there is an increasing demand for rendering multiple databases simultaneously. In all these cases, thrashing becomes unavoidable, thereby warranting a need for a thrash-less visualization system.

## 5.2 Method

The RayFront visualization method is a distributed memory implementation of parallel volume rendering. The scene is initially distributed among all the participating processors. Each processor employs the image-order scheme (forward projection) for casting rays through each pixel assigned to it. Voxels that are needed in the process but are not available locally, are fetched from other processors using explicit message passing - no data is shared among the processors. We chose this distributed memory paradigm as it provides control to the programmer for mapping the fetched data in local memory, so that conflict replacements are avoided and only capacity replacements take place. By restricting the size of locally available memory, we can demonstrate the effect of capacity misses even with smaller databases.

Although our algorithm is essentially an image-order method, it exploits the advantages of both the image-order algorithm (like opacity clipping and adaptive sampling) and object-order algorithm (like object-space coherency and no thrashing due to capacity replacements) to implement a caching system which totally eliminates thrashing across a number of frames by efficiently traversing the image plane and caching data in an efficient manner. In the method discussed below, we first describe how thrashing is avoided for a single frame, and then extend it to preserve its thrash-free property across multiple frames.

### 5.2.1 Screen and Scene Subdivision

The volume is subdivided into cubical cells (Figure 5.1a) similar to [17][22][63]. The cells are statically distributed to the processors in a pseudo-random manner to avoid hot-spotting. A cell is assigned to exactly one processor, referred to as the cell's *home node*. The screen is subdivided into tiles that are distributed cyclically to the processors (Figure 5.1b) to accomplish partial static load-balancing among processors.

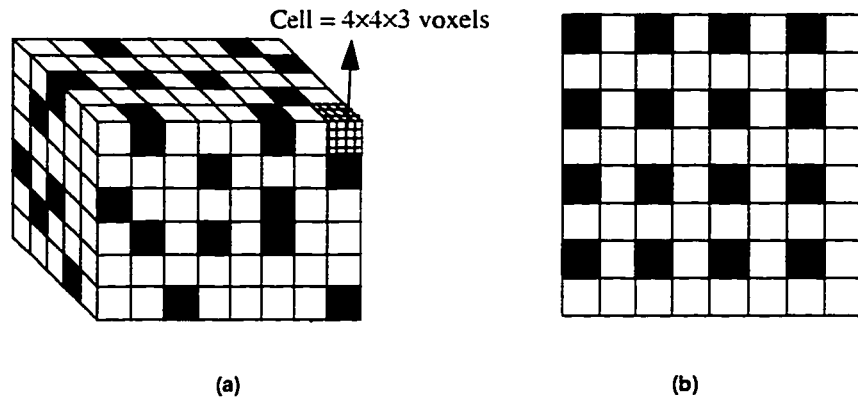


Figure 5.1. (a) A volume made up of  $32 \times 24 \times 15$  voxels is divided into  $8 \times 6 \times 5$  cells each of size  $4 \times 4 \times 3$  voxels. Each processor is home to 60 random cells in a 4-processor system. (b) A screen divided into 64 tiles of equal size and distributed cyclically to 4 processors, P1, P2, P3, and P4. For example, the black cells and the dark tiles are assigned to P1.

### 5.2.2 Preprocessing

The first step in the preprocessing stage is to divide the volume and image among the processors as described above. The local memory is partitioned into two segments: the first segment is used to store the home cells, while the other segment is used as a cache [11]. The size of the home memory in number of cells equals the total number of cells in the volume divided by the number of processors. The home region of the memory is static as cells residing in this region (home cells) are never replaced. The rest of the memory, in number of cells is denoted by  $C$ , and is used as cache.

### 5.2.3 Ray Casting

For generating an image the following procedure is executed. Each processor computes which cells lie inside the view frustum defined by its image segment(s). The list of these cells is then ordered in a front-to-

back manner depending on the position and orientation of the screen. This list is referred to as FTBL (Front-To-Back-List). Each processor also determines the first cell entered by each ray assigned to it. Next, each processor sends a request to fetch the first  $C$  non-home cells in its FTBL.

The ray casting algorithm with advancing ray-front is given in Figure 5.2. All the rays are initially marked as unfinished. A ray is finished if it had either accumulated enough opacity or if it exited the volume. Each ray is also marked with the cell it initially enters and is linked to a list of rays waiting for the same cell as we describe later. The algorithm traverses all the cells in FTB order, but has only one cell active at a time. All rays waiting for the active cell are advanced until they exit the cell. The active cell is then removed from the cache memory (if it is not a home cell) and a request is sent for the next non-home cell in the FTBL.

If there is space for exactly one cell in the cache (i.e.,  $C = 1$ ), then the latency of the requested cell may not be completely hidden. But if there is enough space for a few cells, then the latency of fetching non-home cells can be hidden, except for the first few cells. This is done by sending requests for the next few cells in the FTBL, while working on the currently active cell.

After advancing each ray through a cell, the buffers are polled for messages with a non-blocking probe. A software handler is provided for each kind of message [29]. Depending on the type of message a corresponding action is taken. For example, if the message contains cell information, it is read from the buffers and directly put in a proper place in memory (software-cache). If the message is a request for a cell, it is immediately serviced by sending the requested cell to the requesting processor using a non-blocking send. The interleaving of these non-blocking sends and receives provides ample latency hiding of data in the network. At the same time, it prevents deadlock due to filling up of communication buffers.

The method described above traverses the screen in the most coherent manner, as all the rays entering a cell are advanced before any other rays are processed. This implies that cells once used will not be required again for the current frame. The raw algorithm given in Figure 5.2 requires one to traverse the complete list of rays and advance only those rays which are waiting for the currently active cell. This gives the search complexity of the rendering process as  $O(NumCells \times NumRays)$ , where  $NumCells$  is the total number of cells in the volume and  $NumRays$  is the total number of rays to be traced by a processor. The traversal of the complete set of rays can be partially avoided by limiting the search to the bounding box of the cell's projection on the screen. A simpler and more efficient scheme is used here.

**Preprocessing:**

Divide the volume into cells.  
Randomly distribute the cells to the processors.  
Divide the screen into tiles.  
Distribute the tiles to the processors in a cyclic manner.

**Rendering on each processor:**

*/\* Initialization..... \*/*

Determine the FTBL of all the cells in the volume.

**for each ray r do**

    determine the first cell r enters - call this Ray[r].entering\_cell  
    **if** r does not hit the volume **then** Ray[r].finished = true  
    **else** Ray[r].finished = false

*/\* Rendering.... \*/*

**for each cell c in FTBL order do**

**for each ray r do**

**if** Ray[r].finished = false **then**

**if** Ray[r].entering\_cell = c **then**

                advance ray r through cell c

                update Ray[r].entering\_cell

**if** ray r exits volume **or** Ray[r].opacity > thresh\_opacity **then**

                    Ray[r].finished = true

Figure 5.2. Ray-casting algorithm with advancing ray-front.

We use a linked list to keep track of all the rays entering a cell (Figure 5.3). Initially, rays are inserted into the list of cells they enter. Whenever a particular cell becomes active (i.e., a cell brought into the cache), its list is traversed, and all the rays present in its list are advanced until they exit the cell. Rays exiting the cell are inserted into the list for the cells they enter next. This data structure precludes the necessity for traversing all the rays in the screen for determining the ones entering a cell. If no rays enter a cell, its list is of size zero, as shown for cell [21.8,13] in Figure 5.3. This reduces the search complexity of the algorithm to  $O(\text{Num-Rays})$ . The alternate algorithm is given in Figure 5.2.

The complete image generation process can be viewed as being composed of several *passes*, as shown in Figure 5.5. In the first pass, all the rays will proceed (advance) approximately the same distance from the screen. In the next pass, the rays continue approximately the same distance again. In this manner, a “ray-front” moves through the volume like a wave, refining the image in each pass. The image converges to the final image with each pass. The first few passes of the thrashless advancing ray-front method are exemplified in Figure 5.5 and in Table 5.1.

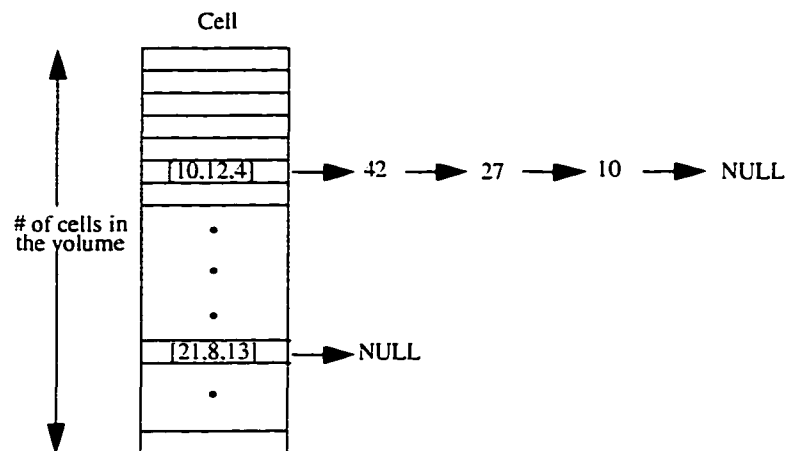


Figure 5.3. Illustration of the linked list data structure used for efficiently advancing only certain rays through a cell. The example shows that there are 3 rays entering cell [10.12.4], they are 42, 27, and 10.

**Preprocessing:**

Divide the volume into cells.  
 Randomly distribute the cells to the processors.  
 Divide the screen into tiles.  
 Distribute the tiles to the processors in a cyclic manner.

**Rendering on each processor:**

```

/* Initialization..... */
Determine the FTBL of all the cells in the volume.

for each ray r do
  determine the first cell c that r enters
  if r hits the volume then
    insert r in list of c

/* Rendering.... */
for each cell c in FTBL order do
  for each ray r in list of c do
    advance ray r through cell c
    if ray r exits volume or Ray[r].opacity > thresh_opacity then
      do nothing
    else
      determine cell ce that ray r enters next
      insert r in list of ce
  
```

Figure 5.4. The modified ray-casting algorithm with advancing ray-front.

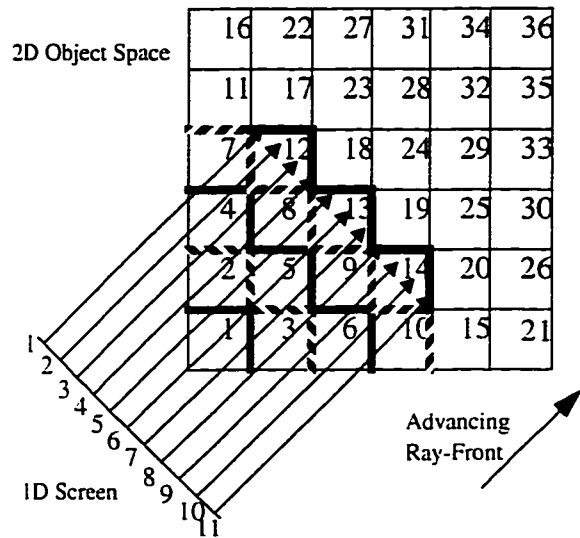


Figure 5.5. An example of advancing ray-front with 11 rays. The figure shows the advancement of the ray-front for the first 5 passes only. The 2D object space is divided into cells, and the numbers in each cell indicate its position in the FTBL.

Pass #	Active cells in pass (in order)	Rays advanced in pass	Ray-Front at line
1	1	5 - 7	————
2	2 - 3	3 - 9	////
3	4 - 6	1 - 11	————
4	7 - 10	1 - 11	////
5	12 - 14	1 - 11	————

Table 5.1. The first 5 passes of the ray-casting algorithm with advancing ray-front for the example shown in Figure 5.5.



The algorithm as described so far is thrashless within a single frame. Now we extend the method so that such coherency can be exploited across a number of similar frames also. By similar frames, we mean those screen positions for which the FTBL order remains the same. For example, in Figure 5.6, if the screen position remains in region I while viewing towards the center of the volume, then the FTBL order remains the same for all the frames, and thus the cells will be processed in the same order. The algorithm can now be followed for updating all such frames in the same pass. A frame number is attached to each ray to be processed. All the rays for all frames in a region are advanced simultaneously before the currently active cell is given up. This provides an algorithm which is thrashless across several frames. All the frames with the same FTBL is referred to as a *phase*.

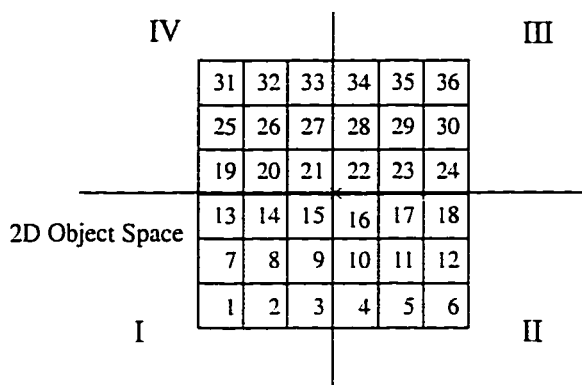


Figure 5.6. For all viewing positions in region I, and when viewed towards the center of the volume, the FTB order of the cells are as shown. X denotes the center of the volume.

## 5.3 Results

### 5.3.1 Number of Frames per Phase

Figure 5.7 shows the times and number of cells received as the number of frames in a phase increases. Timings are taken for generating 30 incrementally changing frames for the simple128 volume. *Frames/phase* indicates the number of frames across which thrash-free operation is preserved. Figure 5.7a shows a consistent decrease in total animation time with increase in frames/phase. This is mainly due to the savings in the communication required to fetch the drastically reduced number of cells, along with other associated overheads, like less frequent updating of local memory with the fetched cells, and reduced network contention.

The improvement in timing performance is not significant as the Cray T3D uses extremely fast and efficient communication channels for transferring data. The communication of extra cells has minimal effect of the performance of the system. We expect the savings in time to be much more significant when the communication is not as efficient, especially on a network of workstations with slower Ethernet links.

Although the improvement in time is not considerable, the total number of cells fetched decreases drastically. When all 30 frames in the animation process are processed all in the same phase, then a cell is fetched only once during the whole process. From Figure 5.7b, we can see that when only a single frame is processed in a pass (phase), then about 4000 cells are fetched from distant memory. In contrast, if all 30 frames can be processed in the same pass, then this figure drops down to the minimum required (about 200). The algorithm is much more effective when it is used with compression caches or when data is being fetched from disk on demand. In such cases, thrash-free operation across a number of frames will have an enormous impact on the performance of the system. For example, for rendering using compression caches, 4000 cells will undergo decompression in the case of 1 frame/phase as opposed to 200 cells in the case of 30 frames/phase (see Chapter 6).

### 5.3.2 Comparison

Figure 5.8 compares the performance of the RayFront algorithm with three of the most common screen-traversal algorithms: scan-line, spiral, and Hilbert [7]. In each of these, the screen regions were distributed to the processors in exactly the same manner as in our algorithm. The only difference was the way in which the pixels in each region were traversed by each algorithm. Out of these, the Hilbert is believed to be the most coherent screen-traversal scheme [136]. It is evident from this graph that the screen traversal used for the RayFront algorithm outclasses the others for all cache sizes for parallel projection ray casting. The performance gain at lower cache sizes is particularly noteworthy. The primary advantage is the algorithm's thrash-free property even for minimal cache sizes.

The consistent improvement in the timings at all cache sizes can be attributed to two main reasons. First, for parallel projection ray casting, the RayFront algorithm predetermines the order in which the cells should be processed. It further culls all the cells which do not fall within its view frustum. A processor thus fetches exactly those cells that are needed, and in the correct order. Second, the predetermination of the cells facilitates latency hiding - an attribute which cannot be exploited advantageously by the other algorithms.

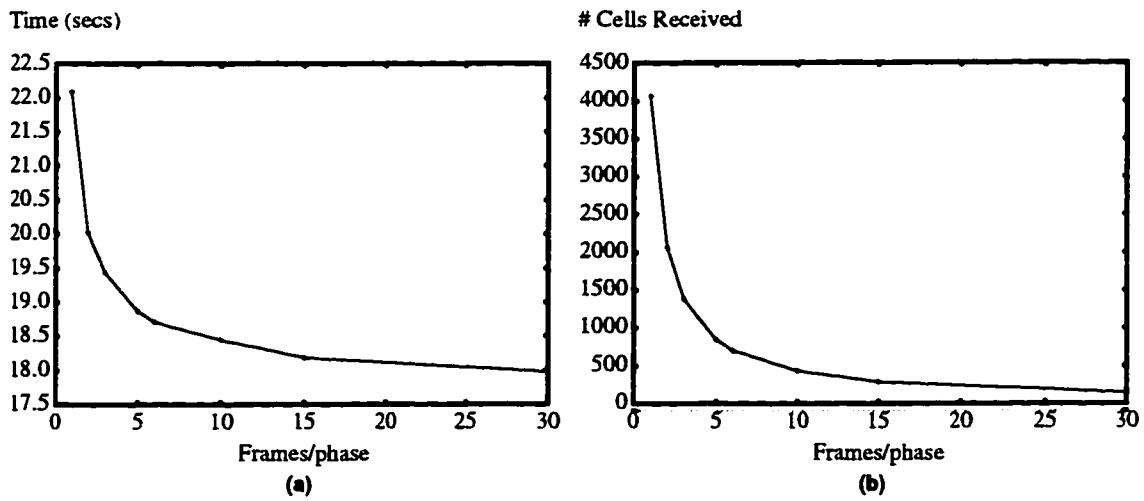


Figure 5.7. (a) Times and (b) Number of cells received with number of frames generated in each phase of the algorithm for generating 30 frames.

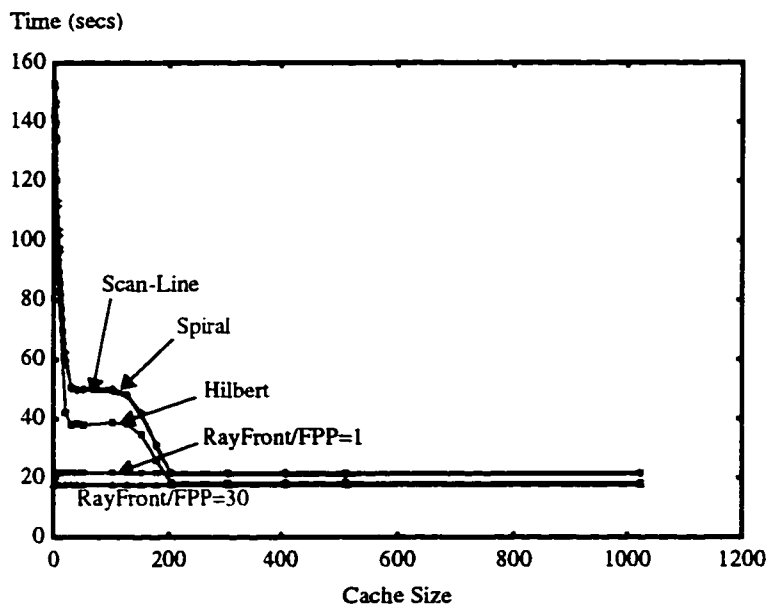


Figure 5.8. Comparison of four different screen traversal schemes - scan-line, spiral, hilbert, and rayfront. The graph shows the times taken for generating 30 frames in the animation.

That the latency is significantly hidden is manifested by number of cells fetched. The pattern of the number of transferred cells as a function of cache size is similar to what is shown in Figure 5.8. Previous algorithms, like scan-line, spiral, and Hilbert, fetch cells only if and when needed. The RayFront algorithm, on the other hand, makes a conservative estimate of the cells which may be required in the future. For higher cache sizes, the number of cells fetched becomes more than that of the other algorithms. This is because it is difficult to pre-determine the nature of the object's transparency properties, making it impossible to predict if a back-cell will be traversed by a ray or not. As a result, some unneeded back-cells are also fetched. In spite of fetching these extra cells, the total animation time remains constant, implying total latency hiding of these cells.

### 5.3.3 Load Balance

Figure 5.9 shows the times spent by each processor for rendering the simple128 and capsid256 volumes on 8 processors. Severe load imbalance exists for larger tile sizes, while it gets equally distributed as tile size becomes smaller. Load balance also helps in decreasing the average time required to generate each frame. From these figures we see that tile size of 8x8 is sufficient to attain ample load balancing. As tile size decreases further, other overheads start to adversely affect the frame times. One reason for this degradation is the loss of coherency introduced for generating images of several different portions of the screen, as opposed to one contiguous region. Thus, there is a compromise between load balance and the amount of coherency that can be exploited.

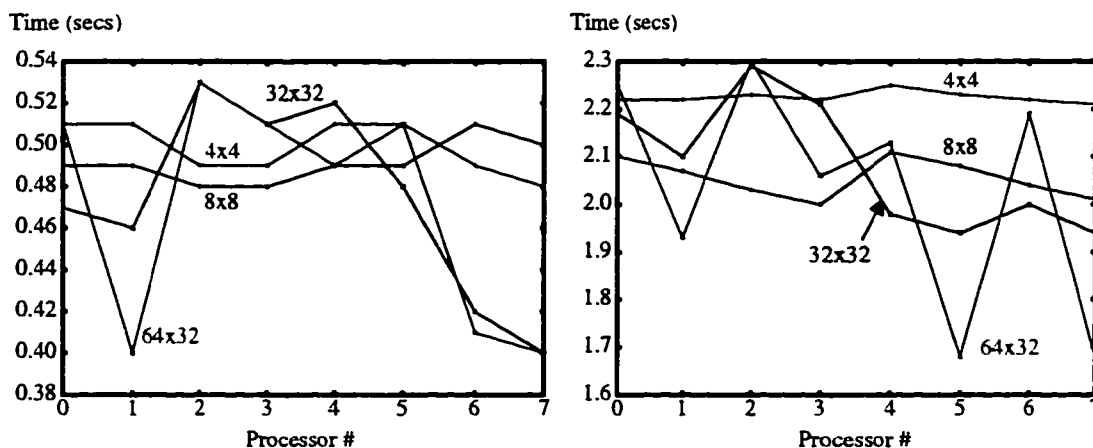


Figure 5.9. Time spent by each processor for rendering 30 frames as a function of tile size. (a) for simple128, and (b) for capsid256 volumes.

### 5.3.4 Scalability

Table 5.2 shows the frame times for rendering several volumes on different number of processors. The timings are averaged over 30 frames rotating around the object. The FPP was set to 5, signifying that the system was thrashless over 5 consecutive frames. Almost real-time speeds of 20 frames/sec. and interactive speeds of 5 frames/sec. were achieved for simple128 and capsid256 volumes respectively on 128 processors. The speedup graphs of the RayFront algorithm for different volumes are shown in Figure 5.10. The algorithm demonstrates about 80% efficiency for 32 processors for all these test volumes. The good speedup also suggests that considerable load-balancing has been achieved using the static block-cyclic scheme as described in Section 5.2.1.

## 5.4 Discussion and Future Work

The RayFront algorithm provides the most coherent screen traversal scheme to avoid thrashing in object dataflow parallel ray casting systems. Its main advantage lies in the arena of rendering colossal databases, where thrashing is bound to occur due to shortage of main memory. Thrashing manifests itself in the degraded turn-around time of the rendering system. Our method is particularly applicable in such cases, and shows significant advantages over existing screen traversal schemes. In the next chapter we show that the RayFront method is advantageous for rendering colossal volumes even on uniprocessor systems. This is because with the proposed advancing RayFront scheme, the cache efficiency improves, and thrashing is avoided even in uniprocessor machines. This can be asserted by verifying that the improvement gained by efficient caching of cells is not offset by the traversal of the data structures employed by our algorithm.

We have retained the thrash-free property across a number of frames also. Our efficient data structures optimize the complexity of the ray search, and our cell ordering scheme facilitates effective latency hiding making the algorithm scalable. Finally, we have brought the two classes of volume rendering algorithm, image-order and object-order, together, and successfully exploited the advantages of both these approaches.

The main disadvantage of this method is the additional memory expended for maintaining the data structures. Also, as the rays are not traversed to completion, the attributes of all the rays have to be stored. This is not required in traditional approaches as the ray currently being processed traverses to completion before starting the next ray. The parallel-projection system developed here should also be extended to include per-

spective projection. It will not be trivial to determine the FTBL of cells when viewed in perspective, making it more difficult to hide the latency for non-local fetches.

With this space-time trade-off, we want to extend the proposed parallel projection ray casting method to ray tracing also. In this sense, we suggest a breadth-first processing of rays instead of the commonly used depth-first approach. In existing parallel ray-tracing systems, the primary ray and all its secondary rays are processed before proceeding to the next pixel. For huge databases, or with sufficient depth of the ray-tree, this method is prone to thrashing. If a breadth-first approach is adopted instead, all the primary rays entering a cell can be processed before moving on to the next level of secondary rays. Data structures similar to the one used here can be employed to efficiently keep track of all the primary and secondary rays entering a cell. All the rays waiting for a particular cell should be advanced once this cell has been fetched. Of course, this method is not free from thrashing, but the number of thrashing instances will reduce. The determination of which cell to bring next is an open issue, as for ray-tracing systems, a front-to-back order cannot be assigned to the cells.

## **5.5 Conclusion**

We have presented a distributed memory ray-casting scheme which incorporates the advantages of both object-order (no thrashing, regularity of access, object-space coherency) and image-order (opacity clipping - avoiding extraneous calculations, higher image quality, simplicity, and usage of other acceleration techniques) algorithms. We have shown that our most coherent screen traversal method exploits coherency far more efficiently than the traditional counterparts. For rendering colossal databases, this provides significant improvement by making the system thrashless. The algorithm has been extended to avoid thrashing for a number of frames also. Efficient data structures have been suggested to improve the time complexity, and to facilitate effective latency hiding. This makes the method scalable to a number of processors. In the future, we would like to extend the algorithm to view in perspective and to use a similar scheme for ray tracing also.

Number of Processors	Simple128	SOD128	Head256	Capsid256	Capsid512
1	3.59	5.03	12.38	14.74	115.71
2	1.89	3.12	6.64	7.94	58.79
4	1.01	1.86	3.57	4.28	31.86
8	0.53	0.96	1.90	2.17	15.83
16	0.28	0.50	1.03	1.25	8.48
32	0.15	0.29	0.57	0.69	4.30
64	0.08	0.16	0.30	0.36	2.32
128	0.05	0.09	0.18	0.20	1.31

Table 5.2. Frame rendering times for various volumes as a function of number of processors. All times are in seconds.

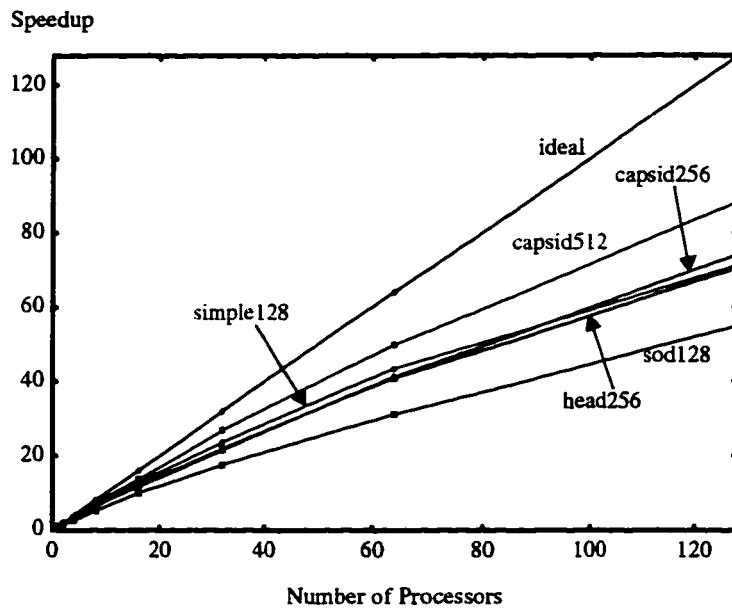


Figure 5.10. Speedups exhibited of the ray-front algorithm for different volumes.

## CHAPTER 6

### THE UNIPROCESSOR RayFront ALGORITHM FOR VISUALIZING COLOSSAL MEDICAL VOLUMES

In the last chapter, we presented the RayFront thrashless multiprocessor rendering system that maintains its thrashless behavior across a number of frames. In this chapter, we provide a classical application of this algorithm, which is used for thrashless uniprocessor rendering of colossal volumes, possibly in compressed form.

Modern computers are unable to store in main memory the complete data of high resolution medical images. Even on secondary memory (disk), such large datasets are sometimes stored in a compressed form. At rendering time, parts of the volume are requested by the rendering algorithm and are loaded from disk. If one is not careful, the same regions may be (decompressed and) loaded to memory several times. In this chapter we present a coherent algorithm that minimizes this thrashing and optimizes the time and effort spent to (uncompress and) load the volume. The volume is divided into cubic cells, each (compressed and) stored on disk, in contrast to the more common slice-based storage. At rendering time, each cell is allocated a queue of rays. For a sequence of images, all rays are spawned and queued at the cells they intersect first. Cells are loaded, one at a time, in front-to-back (FTB) order. A loaded cell is rendered by all the rays found in its queue. We analyze the algorithm in detail and demonstrate its advantages over existing ray casting volume rendering methods.

#### 6.1 Introduction

There has been a growing interest in visualizing extremely large medical databases, one classic example being the Visible Human, comprising of more than 30 gigabytes. This database was created by the National Library of Medicine's Visible Human Project [1], with the intention of creating anatomical atlases. This voxelized human provides a new level of educational value to anatomical visualization. Real time or interactive software rendering of such large databases still looks like an unattainable goal. A more immediate require-



ment is to reduce the rendering time as far as possible, so that the frame generation time decreases. Several attempts have been made to “conquer” the visible human, some notable ones been described in [48][56][80][93][109]. In [56], an attempt has been made to develop a comprehensive virtual environment, including efficient segmentation and realistic ray tracing of the volume. In [109] the Visible Human is used as a basis for a comprehensive medical atlas. Palmer, et al., [93] have gained speed by implementing a parallel volume renderer on clusters of shared-memory multiprocessors. Apart from using parallel computers, some researchers have taken a different route to improve the performance of the rendering algorithm by suggesting coherent algorithms, as in [7][42]. Similarly, direct rendering of compressed volumes have also gained attention [35][57][91][114].

In the method discussed below, the volume is divided into cubic cells, each (compressed and) stored on disk. This is in contrast to the more common slice-based storage. The uniprocessor RayFront method proceeds in exactly the same manner as the multiprocessor algorithm, except that cells required by a processor are not fetched from other processors; they are (decompressed and) read from disk as and when needed. At rendering time, each cell is allocated a queue of rays. For a sequence of images, the front-to-back (FTB) order of cells is determined. For all frames that share the same FTB order, all rays are spawned and queued at the cells they intersect first. Cells are loaded, one at a time, in FTB order. A loaded cell is rendered by all rays found in its queue. The end result is that the costly operation of decompressing and loading a cell into memory is done once for all the frames that share the same FTB order.

The coherent ray casting method proposed here can be used in three ways. First, it can be used to render compressed volumes by explicitly decompressing small blocks of the volume (cells in our case). The images produced in this manner are exactly the same as would be produced by a direct volume renderer. The algorithm is independent of the compression technique used, thus allowing higher compression ratios. The only assumption is that the compression is lossless [32] and that the corresponding decompression routine is available at render-time. Second, it can be used in conjunction with the methods for direct rendering of compressed volumes (e.g., [91]) to exploit coherency to a far higher degree. The image quality will be the same as guaranteed by the respective algorithms. Finally, the algorithm can simply be used advantageously to improve cache efficiency, for volumes that do fit in main memory.

In the next section, we describe our method in more detail and some optimizations we implemented (Section 6.2.2). We also propose an extension (Section 6.2.3) that will enable us to render a cell exactly once even

when the images do not share the same FTB order. In Section 6.3, we analyze the uniprocessor implementation with several compressed and uncompressed datasets, and suggest our conclusions in Section 6.4.

## 6.2 Method

Volumes are compressed because they do not fit in secondary or primary memory. Efficient compression techniques [11-4] and direct volume visualization of compressed volumes [35][57][91] have been suggested. These algorithms generally compromise with accuracy, but with an *a priori* knowledge of the behavior of the data, very efficient schemes can be developed. We have taken a different approach to visualize compressed volumes. We believe that the pain and effort taken in acquiring high resolution and high quality data, such as in the Visible Human Project, cannot be compromised at rendering time by employing lossy compression techniques. Our algorithm preserves the accuracy of the direct volume renderer by employing lossless compression [32] (or no compression) while trying to optimize rendering time.

The idea of multi-frame thrashless volume rendering was introduced in the last chapter. Here we extend the scope of this novel approach to the visualization of compressed or colossal volumes on uniprocessor machines. The algorithm described in [68] preserves the thrashless property across all the frames that share the same FTB order. In the next section, we briefly explain the idea of multi-frame thrashless volume rendering, followed by a proposal (in Section 6.2.3) to extend the method to work for arbitrary sets of frames, i.e., frames that do not share the same FTB order of cells. The net effect of this proposed feature will make the algorithm limited only by the memory required to store the frames since every cell will be decompressed and fetched exactly once for *all* images.

The only requirement of our algorithm is that volume is divided into cells that can fit in the machine's main memory. Each cell is optionally compressed and stored onto a disk. We assume that the corresponding decompression routine is available at rendering time. Compressing cells instead of the whole volume may lead to lower compression ratios [32]. Alternatively, very large volumes can also be divided and stored, as uncompressed cells, on distributed remote disks. The algorithm basically renders the original data, but uses an efficient ray traversal scheme to ensure that each cell is decompressed exactly once for a number of frames. It takes advantage of coherency between slowly changing frames to eliminate thrashing. In Section 6.2.3, we propose an extension to this method, so that the thrashless property is maintained even for arbitrary jumps of the screen during the animation.

### 6.2.1 Multi-Frame Thrashless Volume Rendering Revisited

In this section, we briefly describe the idea of multi-frame thrashless ray-casting. Refer to Chapter 5 for a detailed discussion. During preprocessing, the volume is divided into equal-sized cells and stored on a remote disk, possibly in a compressed form. The size of a cell is fixed at the size of the main memory. Before rendering begins, the list of cells is ordered in a front-to-back (FTB) manner depending on the position and orientation of the screen. This list is referred to as FTBL (*Front-To-Back-List*). For example, Figure 6.1 shows a  $24^2$  2D raster divided into 36 2D cells of size  $4^2$  voxels each. The number in each cell indicates its position in the FTBL for any screen position in region I, when viewing towards the center of the volume. We also determine the first cell entered by each ray and push the ray into the queue associated with that cell.

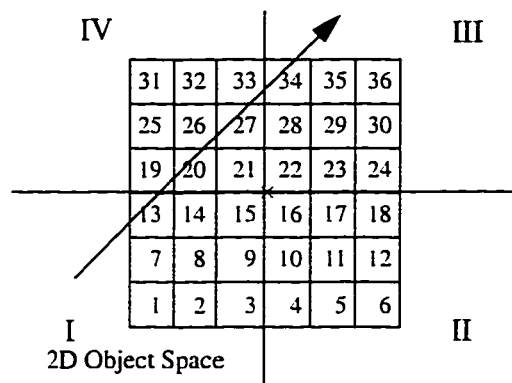


Figure 6.1. For all viewing positions in region I, and when viewed towards the center of the volume, the FTB order of the cells are as shown. X denotes the center of the volume. A ray is also shown which traverses cells 13, 19, 20, 26, 27, 33, 34, in order.

When rendering begins, the first cell in FTB order is (decompressed and) fetched from disk. This cell is referred to as the *active cell*. Since the cell size is the same as the size of main memory, only one cell can be active at a time. Also, as the cells are processed in FTB order, a cell becomes active at most once during the generation of a frame. All the rays associated with the queue of this active cell are advanced until they exit the cell. These rays are then dequeued from the currently active cell and, in turn, are queued into the list of the respective cells they enter. Figure 6.1 shows an example of a ray which is initially queued into the list of cell 13, then queued and dequeued from cells 19, 20, 26, 27, 33, 34 (in order) as these cells become active and inactive. Rays which either exit the volume or accumulate enough opacity during their traversal, are considered done, and are not queued into the list of any cell. Once all the rays in a cell's queue are traced, the active cell is removed from main memory, and the next cell in FTB order is brought in. In this manner, the

rest of the cells in the volume are (decompressed and) fetched from disk one at a time and all the rays associated with the fetched cell (now the active cell) are advanced accordingly.

The algorithm as described so far is thrashless within a single frame. We can extend the method so that such coherency can be exploited across a number of *similar* frames also. By similar frames, we mean those screen positions for which the FTB order of cells remains the same. The algorithm can now be followed for updating all such frames in the same pass. A frame number is attached to each ray to be processed. All the rays for all frames which are associated with the currently active cell, are advanced through the cell before the active cell is given up. This provides an algorithm which is thrashless across several frames. The set of all the frames with the same FTBL is referred to as a *phase*.

## 6.2.2 Enhancements

Before proposing a couple of major extensions to the original algorithm, we describe a few minor enhancements we have incorporated for tuning it to perform well with compressed volumes. As a preprocessing step, we break the volume into cells, and each cell is marked empty if none of the voxels in it are occupied. This allows us to totally skip all the empty cells (not even decompress or read) during rendering. For even better performance, the empty cells can be combined during pre-processing and the whole volume may be put into a hierarchical structure (e.g., octree). In addition, early ray-termination for opaque volumes saves the decompression times of cells lying totally behind an opaque object.

In order to save ray and queue storage space, we allocate queues only for non-empty cells. Therefore, when a ray intersects (enters) an empty cell, we look for the next cell rather than queue it there. This saves some time in pushing and popping rays from queues but, more importantly, rays that eventually hit the background go through a series of ray-cell intersection calculations without ever being allocated memory and without ever being queued.

## 6.2.3 Extension to Arbitrary Frame Animation

The method of multi-frame thrashless ray-casting described above can be extended to work for any arbitrary frame sequence with a minor extension. Firstly, we realize that the FTB order of cells, which remains unaltered for a set of frames, is also the BTF (back-to-front) order for a different set of frames. As a 2D example, the order of cells shown in Figure 6.1 conforms with the FTB order for all screen positions in region I, while

looking towards the center of the volume. Similarly, the same order is maintained as a BTF order for all screen positions in region III, while looking towards the center of the volume. The drawback of the latter is that early-ray termination can no longer be applied, and so all the cells in the volume have to be traversed. Working with colossal or compressed volumes, it may be worthwhile to stick to this drawback than to decompress a cell more than once.

For parallel viewing of a 3D volume, there are eight FTB orders of cells, one for each octant. With the above extension, we can divide the stream of frames into four sets, frames with same FTB or BTF falling in the same set. This reduces the method to just four phases, implying that a cell in the volume would be decompressed at most four times for generating all the frames in any animation sequence.

Now, we propose another extension which reduces the number of times a cell is decompressed to exactly once, irrespective of the order of frames. It is evident from the description of the original rayfront method that, at any time, each ray maintains exactly one segment of traversed volume, starting from the entering point in the volume to the current sampling point along the ray. This claim holds as long as cells are processed in an FTB or a BTF order.

We claim that if two segments of each ray are maintained, then there is a thrashless order for any arbitrary set of frames. This implies that for a single order of cells (which is not necessarily FTB), all the frames in the animation can be generated by decompressing a cell exactly once in the whole process.

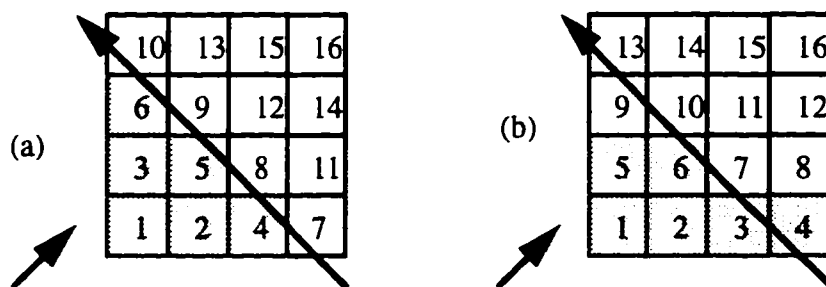


Figure 6.2. (a) An FTB order for eye in the left bottom side (red arrow) that requires the maintenance of arbitrary number of ray segments for some other eye positions (black arrow). (b) An FTB order that requires at most two ray segments for any ray orientation.

Let us consider the an FTB order as shown in Figure 6.2a. When some cells (i.e. 1-6 in Figure 6.2a) have been rendered, the image from some other orientation will have to save the information of several rays segments (e.g., in Figure 6.2a, for cells 4, 5, and 6) so that when other cells (8 and 9) are rendered, their results could be properly composited in FTB order. However, there is one order, the one that scans row by row and plane by plane, as shown in Figure 6.2b, that will require maintaining at most two ray segments for each ray. We call this method *ZZ-buffer* to portray the correspondence between Z-buffer and ray-casting. In normal ray casting, a single segment of the ray is maintained, and so is in Z-buffer. In our method, two segments are maintained corresponding to two Z values in a ZZ-buffer.

## 6.3 Results

The algorithm was implemented on a single 90 MHz R8000 processor of Silicon Graphics Power Challenge. Our machine, located in The Ohio Supercomputer Center, has 16 processors, 2 Gbytes main memory (8-way interleaved), and 4 Mbytes secondary cache. In this section, we investigate some of the important aspects that influence the performance and behavior of our algorithm.

### 6.3.1 Timings for Non-Compressed Volumes

We experimented with several volumes described in Chapter 3, and gathered timings by varying a number of parameters, e.g., cell size and the number of frames across which the thrashless property was maintained. As means of comparison, we have also shown the raw rendering times for each of these volumes in Table 6.1. In addition to the volumes described in Chapter 3, a subset of the Visible Human dataset was also used in our experiments. "VH" is the cryosection of the head, taken from the Visible Human dataset. We took only 250 slices of the head which originally took a total of 1.3 Gbytes. The images were cropped to size 600×700, and quantized to one color band (gray-scale), yielding a 113Mb dataset. Since we save our data in a compressed form large data sets such as Capsid512 and VH occupy fraction of disk space – 1MB and 16Mb, respectively.

Table 6.1 shows the raw times for rendering non-compressed volumes, as signified by the almost insignificant time taken for reading in the cells

	Render Time (secs.)	Read Time (secs.)	Total Time (secs.)
Simple128	3.57	0.11	3.68
SOD128	3.08	0.19	3.27
Capsid256	17.84	0.16	18.00
Head256	12.01	0.10	12.11
Capsid512	49.74	0.07	49.81
VH	78.00	1.12	79.12

Table 6.1. The volumes, along with reading, rendering, and total times (in seconds).

### 6.3.2 Effect of Cell Size

As a preprocessing step, the volume is divided into cells. Each cell is marked empty if none of the voxels in it is occupied. Needless to say, a smaller cell size will result in more empty cells in the volume and vice versa. The advantage of having smaller cell size is that a better portion of the empty space can be skipped. On the other hand, extraneous computation is involved with smaller cells, offsetting some of its advantages. Entry and exit points have to be calculated for each cell along with the extra time needed for reading a larger number of cells one at a time. Most of the volumes show optimal timings in the vicinity of cell size  $16^3$  or  $32^3$ .

Table 6.2 shows the timings obtained for various cell sizes. In Figure 6.3, the times are normalized with respect to the time taken to render the complete volume as a single cell, i.e., a cell with the size of the volume itself. For large datasets we did not measure for small cell sizes due the immense number of files required.

### 6.3.3 Cost of Overheads

If one has access to memory that can store the whole dataset, then there is no need for the maintenance and management of cells. To verify the additional cost of this overhead, we compared the performance of our algorithm with a ray tracer that stores all the data in the memory. Table 6.3 shows the cost of our algorithm compared with a normal ray caster. The increase in times will be compensated by the amount of time saved

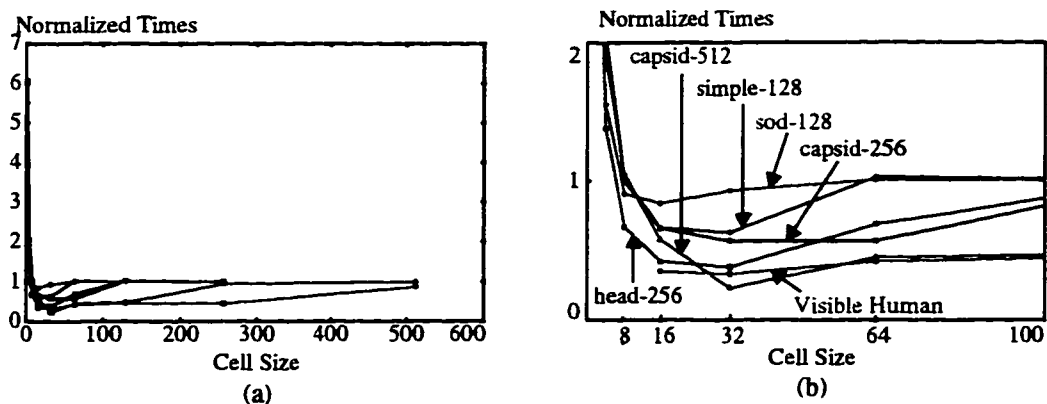


Figure 6.3. (a) Effect of cell size for different datasets. (b) A closer look at the lower left part of the graph.

when a number of frames are generated in each phase (see Section 6.3.4). A phase includes all the frames for which a cell is decompressed only once. The times change when decompression times are also included in the total animation time. With decreasing size of memory, or equivalently, with increasing volume size, our method will show much better speedup than normal ray casting of compressed volumes. The average frame generation times are also comparable. The number of cells used for timings in the second column of Table 6.3 are the same as used in Table 3.1 (Chapter 3). From Table 6.3, we see that our algorithm closes on the performance of a normal ray-caster for larger volumes or for larger cells.



	2 <sup>3</sup>	4 <sup>3</sup>	8 <sup>3</sup>	16 <sup>3</sup>	32 <sup>3</sup>	64 <sup>3</sup>	128 <sup>3</sup>	256 <sup>3</sup>	512 <sup>3</sup>
Simple128	20.3	7.1	3.7	2.4	<b>2.3</b>	3.8	3.7	--	--
SOD128	14.5	5.5	3.3	<b>3.0</b>	3.3	3.6	3.6	--	--
Capsid256	164.5	55.4	27.6	18.0	<b>15.4</b>	15.4	27.6	27.1	--
Head256	112.5	39.7	19.3	12.1	<b>10.8</b>	19.9	29.5	28.8	--
Capsid512	--	--	230.5	127.6	<b>49.8</b>	97.6	106	208	218.7
VH	--	--	--	65.9	<b>62.9</b>	79.1	86.1	85.2	163.5

Table 6.2. Total times (reading + rendering) in seconds, as a function of Cell Size. The minimum times are shown in bold.

	Time with volume as a single cell (normal ray-casting)	Time with several cells (our algorithm)	Degradation Factor
Simple128	3.69	6.32	1.71
SOD128	3.58	5.43	1.52
Capsid256	27.10	36.00	1.33
Head256	28.78	37.66	1.31
Capsid512	218.70	242.56	1.11
VH	85.18	87.36	1.03

Table 6.3. Degradation factor of our algorithm as compared to a normal ray-caster

	1	2	4	5	10	20
Simple128	33.31	19.02	11.85	10.41	7.55	6.13
SOD128	89.67	48.19	27.24	23.03	14.48	10.25
Capsid256	69.63	46.89	35.34	33.02	28.40	25.90
Head256	43.74	29.60	22.53	21.09	18.23	16.72
Capsid512	92.60	76.10	67.38	65.47	61.33	58.87
VH	262.40	175.36	131.50	122.65	104.68	95.52

Table 6.4. Average rendering times (decompressing + reading + rendering) in seconds, as a function of number of frames in a phase. Total number of frames = 20

### 6.3.4 Simultaneous Multi-Frame Rendering of Compressed Volumes

The term *Phase* is used to denote a collection of frames for which the thrashless property is preserved. Table 6.4 shows the variation in total rendering time for generating 20 frames. All the frames had the same FTBL of cells. Most of the time was spent in decompressing the cells to be read in. This time is amortized over all the frames as the number of frames in a phase (FPP) increases. On the extremes, each cell is decompressed 20 times when FPP is 1, while a cell is decompressed only once when FPP is 20. The graph in Figure 6.4 shows the speedup gained for different volumes for various FPPs between 1 and 20. The occupancy of the SOD volume is higher than the other volumes, signifying a higher number of non-empty cells, thus leading to more time spent for decompressing. This is the reason for the higher gain for this densely occupied volume. For higher density volumes, even higher speedup can be expected. For sparser volumes, the speedups is lower. The speedups shown in Figure 6.4 are relative to the case where no thrashing happens whatsoever, i.e., for the case when FPP equals 20. For example, the rendering for sod128 is about 9 times slower with an FPP of 1 compared to an FPP of 20. It must be stressed here that even with FPP of 1, the thrashless property is preserved across individual frames. Normal ray casting algorithms, when applied to such large datasets, are unable to maintain the thrashless property even across rays. It will be highly impractical to use normal ray casting with these datasets. Our algorithm will show speedups of enormous magnitude when compared with normal ray casting, especially in cases when main memory (cache) is extremely limited compared with the size of the complete volume.

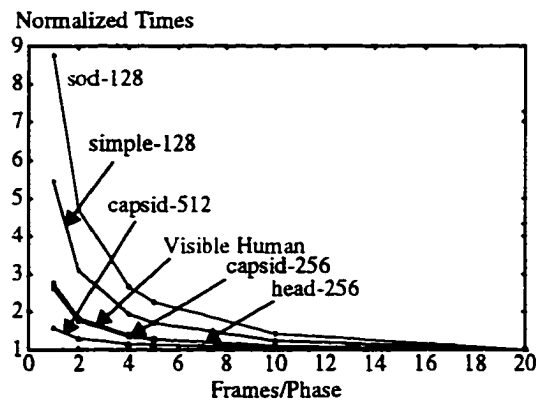


Figure 6.4. Speedups achieved for different volumes as FPP is varied between 1 and 20. Total number of frames remains the same at 20.

## 6.4 Discussion

In this chapter we have given a classical application of the RayFront algorithm for efficiently handling the rendering of colossal (possibly compressed) datasets. Our method is based on subdivision into cells. At rendering time, cells are brought, one at a time, to main memory and rendered from several viewpoints, so that a sequence of images can be generated without loading the same cell more than once. The only limit is the memory required to save the multiple frames. This is quite demanding since each frame is a 2D array of rays, each ray requesting ~40 bytes rather than an RGB $\alpha$  tuple (4 bytes).

The main advantages of our approach is that it exploits the benefits of both object-order methods (for the FTB order) and image-order methods (for early ray termination and other optimizations). It also provides a completely thrashless method for rendering from an arbitrary set of views.

Although we didn't measure cache utilization, we believe that the emphasized localization of memory accesses typical to our method also enhances cache hit ratio. The timings compare well with those reported for direct rendering of compressed volumes; in addition our algorithm does not compromise with accuracy.

## **CHAPTER 7**

### **CONCLUSIONS**

In the last few chapters, we proposed three PVR algorithms and presented some results of their implementation on Cray T3D. In order to validate our claim about the coherent nature of the algorithms, we tested their scalability on another scalable massively parallel machine, the Convex SPP, and also on a cluster of DEC-alpha workstations. Comparable performance and scalability in these environments will consolidate the latency hiding, load balance, and other positive features of our algorithms.

In the next section, we first provide a summary and comparative analysis of the CellFlow, ActiveRay, and RayFront algorithms. In Section 7.2, we compare the performance of our algorithms on Cray T3D with that on the Convex SPP. Finally, in Section 7.4, we provide some directions to future research, and some possible extensions to our work.

	<b>CellFlow</b>	<b>ActiveRay</b>	<b>RayFront</b>
Type of Parallelism	image partition	image partition	image partition
Rendering Algorithm	object-order or image-order	image-order only	hybrid
Data Partitioning			
1. Volume	1. cells	1. cells	1. cells
2. Screen	2. stripes	2. tiles	2. tiles
Partition Distribution			
1. Volume	1. dynamic	1. dynamic	1. static
2. Screen	2. static/interleaved	2. static/interleaved	2. static/interleaved
Load balancing	static - interleaved IP distribution	static - interleaved IP distribution	static - interleaved IP distribution
Coherency	temporal, network	spatial, temporal, network	spatial, temporal, volume
Latency Hiding	pre-fetching	postponing ray processing	pre-fetching
Communication	nearest-neighbor only	close-neighbor	global
Embedding Topology	1D ring	any	any
Architecture	MIMD	MIMD	MIMD
Programming Model	SPMD	SPMD	SPMD
Portability	to MIMD machines	to MIMD machines	to MIMD machines
Portability	most scalable network topologies	most scalable network topologies	most scalable network topologies
Current Limitations	1. not good for perspective projection 2. not good for dynamic load balancing		1. only for ray-casting 2. only for parallel projection 3. not too good latency hiding
Disadvantages	1. not suited for ray-tracing 2. not suitable for arbitrary jumps in screen positions	1. large memory requirements for holding ray data structures 2. extra message required by the algorithm for requesting cells 3. some extra overheads for management of cells	1. large memory requirements for holding ray data structures 2. global communication
Advantages	1. good latency hiding 2. nearest-neighbor comm	1. good latency hiding 2. good for reasonable sized memory machines 3. near-neighbor comm 4. good for dynamic load balancing	1. totally thrashless across multiple frames 2. best for extremely small memory sizes or colossal datasets 3. capitalizes on the advantages of both object-order and image-order rendering methods

Table 7.1. Summary of the features and performance of the CellFlow, ActiveRay, and RayFront algorithms described in Chapter 3, Chapter 4, and Chapter 5, respectively. (continued on next page)

Table 7.1 (contd.)

	<b>CellFlow</b>	<b>ActiveRay</b>	<b>RayFront</b>
<b>Applications</b>	<ol style="list-style-type: none"> <li>1. incremental rotation of screen around the volume</li> </ol>	<ol style="list-style-type: none"> <li>1. general purpose animation</li> <li>2. animations that involve changes only in the classification function</li> <li>3. generating images from TVVDs from the same view point</li> <li>4. easily extendible to ray-tracing</li> </ol>	<ol style="list-style-type: none"> <li>1. rendering of compressed volumes</li> <li>2. rendering colossal datasets</li> <li>3. improving cache efficiency for uniprocessor rendering</li> </ol>
<b>Future Research</b>	<ol style="list-style-type: none"> <li>1. extend to polygon-based scenes</li> </ol>	<ol style="list-style-type: none"> <li>1. extend to polygon-based scenes</li> <li>2. dynamic load balancing</li> <li>3. extend to ray-tracing</li> </ol>	<ol style="list-style-type: none"> <li>1. extend to polygon-based scenes</li> <li>2. dynamic load balancing</li> <li>3. extend to perspective</li> <li>4. extend to ray-tracing</li> <li>5. SM implementation</li> </ol>

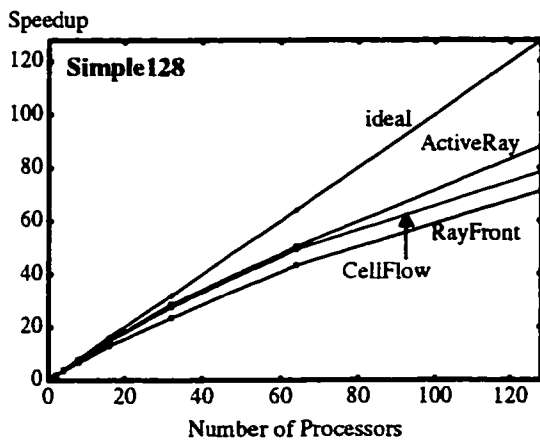
## 7.1 CellFlow vs ActiveRay vs RayFront

It is quite evident from the discussions in previous chapters that all three algorithms exploit one form of coherency or the other. Effective latency hiding, considerable load balancing, and proper data and computation distribution have combined to make them quite scalable on MPP machines like Cray T3D. Table 7.1 summarizes some of the important features of these algorithms

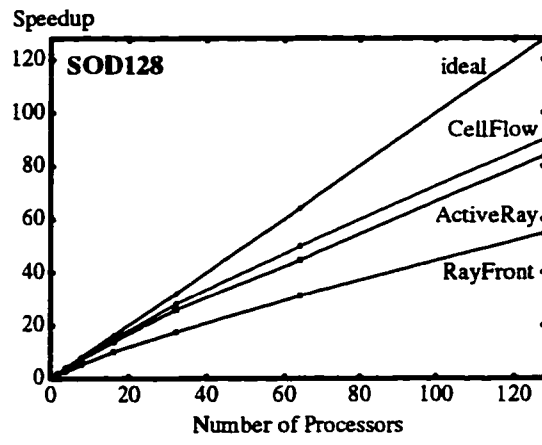
From the above table, we realize that the CellFlow algorithm is the most restrictive in its current form. It is limited to pure rotations only, and may not be extendible to ray-tracing with ease. Dynamic load balancing, extension to perspective, and allowance for non-smooth animation sequence cannot be handled by this incremental rotation method. The ActiveRay algorithm attempts to provide the much desired generalization, it is the most suitable for general purpose animation, it is quite amenable to dynamic load balancing, and can be extended to perspective and ray-tracing. On the other hand, it starts to thrash when local memory size becomes small compared to the volume size. The RayFront method provides the solution in such situations. It employs a screen traversal scheme that guarantees that a cell is needed only once for a set of frames.

A common thread that ties these three algorithms together is the coherency exploitation. Temporal, spatial, and network coherency have been the primary driving force in their design. All these methods try to optimize the use of local memory in one way or the other. To achieve this CellFlow restricts the animation to pure rotations only, ActiveRay uses its object migration and directory scheme, and RayFront resorts to a coherent screen traversal mechanism. Hiding latency, avoiding network congestion, and restricting communication to near neighbors are common features to each of these systems. As mentioned in the respective chapters, each of these algorithms is capable of outclassing the other if the assumptions for their designs apply. In order to optimally make use of all these methods, a scheme may be adopted so that one can switch from one method to the other depending on the situation. For example, if a part of the animation takes place with pure rotations, the CellFlow routine can be invoked; as the animator makes a non-smooth jump from one viewing position to the other, it can automatically switch over to the ActiveRay method, that handles such situations more efficiently; as the size of the dataset increases, thrashing causes the performance of the ActiveRay algorithm to deteriorate; RayFront turns out to be the most suitable choice in these cases.

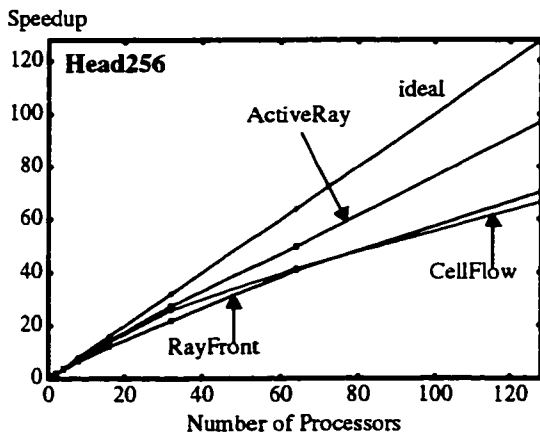
Figure 7.1 compares the efficiencies of the three algorithms, CellFlow, ActiveRay, and RayFront for four datasets, simple128, sod128, head256, and capsid256. With respect to volume size, larger volumes, like capsid256 and head256, are expected to provide higher efficiencies for larger number of processors. This is



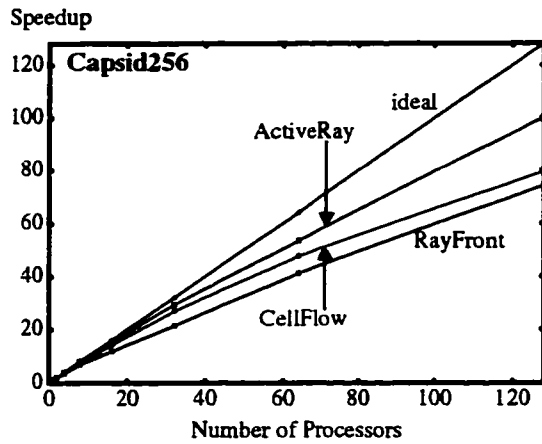
(a)



(b)



(c)



(d)

Figure 7.1. Graphs comparing the speedups of the CellFlow, ActiveRay, and RayFront algorithms on four datasets. (a) simple128, (b) sod128, (c) head256, and (d) capsid256.



because of the increase in the computation time with respect to communication. ActiveRay consistently demonstrates the most superior scalability. This may be attributed to its efficient latency hiding mechanism using the ray stack. This may also be the reason for RayFront's poor scalability, as it does not employ any latency hiding scheme in its current implementation. CellFlow shows mediocre scalability for all the datasets.

## 7.2 Cray T3D vs Convex SPP

Table 7.2 and Table 7.3 compares the frame times of the ActiveRay and the RayFront algorithms respectively on Cray T3D and Convex SPP. Comparison results are shown only for 8 processors, as only these many processors were available for users at a time on the Convex SPP located at the Ohio Supercomputer Center. These tables reveal a few interesting features of the algorithms. First, we see that due to the faster speed of the Convex processors, it consistently provides better frame times for all the volumes. Frame times on the Convex decrease almost by a factor of 2 when compared to those on the Cray T3D. One exception to this is the time taken for rendering capsid256 volume on a single processor of the SPP. For the ActiveRay algorithm, this time is almost as much as that taken on the T3D, but the timings using the RayFront method follows the earlier trend. The reason for this may be the cache thrashing. Capsid256 is a transparent volume, and rays have to traverse through the entire volume along its path. As the complete volume is unable to fit in a single processor's cache at the same time, it starts to thrash. Similar behavior is not present for the head256 dataset, as this volume is completely opaque, where the rays halt as soon as the first occupied voxel is encountered. Thus, in most cases, only one cell has to be brought in, i.e., the one containing an occupied voxel. Most other cells are marked empty and therefore are not read into the cache. The corresponding numbers for the RayFront algorithm also delineates its advantages for rendering larger datasets. As this is a thrashless system, the timings are consistent with the others.

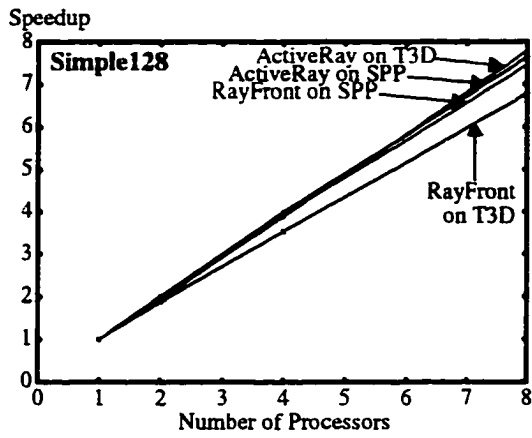
The speedup graphs in Figure 7.2 shows that Convex has the potential to provide comparable or better speedup than on the Cray T3D for the datasets tested. Convex SPP's cluster architecture is quite useful for utilizing network coherency. In the machine available at the Ohio Supercomputer Center, 8 processors are grouped together in each hypernode. Message passing takes place via shared memory reads and writes between processors in the same hypernode, as opposed to through sockets for processors across hypernodes. Use of shared memory data transmission circumvents the expensive traversal by the messages through several network layers, and thus helps to bring down the communication costs considerably. While ActiveRay scales better on the T3D, RayFront seems to be more efficient on the SPP. One caveat in these graphs is the superscalar curve for the capsid256 volume with the ActiveRay algorithm on the SPP. The superscalability can be attributed to the abnormally high time taken on a single processor, as shown in Table 7.2

	Simple128			SOD128			Head256			Capsid256		
	T3D	SPP	$T_{T3D}/T_{SPP}$	T3D	SPP	$T_{T3D}/T_{SPP}$	T3D	SPP	$T_{T3D}/T_{SPP}$	T3D	SPP	$T_{T3D}/T_{SPP}$
1	3.47	2.01	1.73	5.86	2.48	2.36	16.25	6.90	2.36	15.62	15.30	1.02
2	1.76	1.00	1.76	2.92	1.25	2.34	7.98	3.50	2.28	7.66	4.33	1.77
4	0.89	0.50	1.78	1.51	0.65	2.32	4.07	1.84	2.21	3.89	2.21	1.76
8	0.45	0.26	1.73	0.80	0.36	2.22	2.18	1.01	2.10	1.94	1.15	1.69

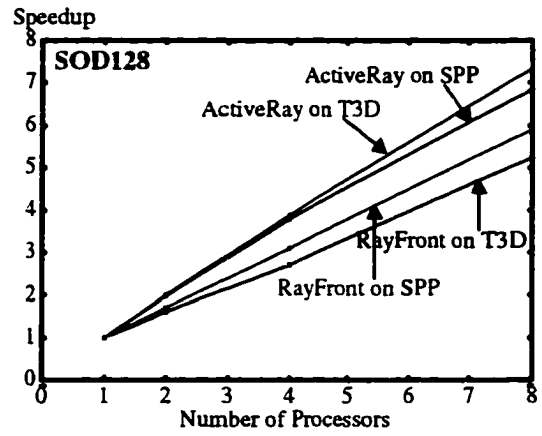
Table 7.2. Comparison of frame times (in seconds) for the ActiveRay algorithm on Cray T3D and Convex SPP.

	Simple128			SOD128			Head256			Capsid256		
	T3D	SPP	$T_{T3D}/T_{SPP}$	T3D	SPP	$T_{T3D}/T_{SPP}$	T3D	SPP	$T_{T3D}/T_{SPP}$	T3D	SPP	$T_{T3D}/T_{SPP}$
1	3.59	2.18	1.65	5.03	2.44	2.06	12.38	7.45	1.66	14.74	9.31	1.58
2	1.89	1.13	1.67	3.12	1.44	2.17	6.64	4.02	1.65	7.94	4.95	1.60
4	1.01	0.55	1.84	1.86	0.79	2.35	3.57	2.18	1.64	4.28	2.69	1.59
8	0.53	0.29	1.83	0.96	0.41	2.34	1.90	1.13	1.68	2.17	1.43	1.52

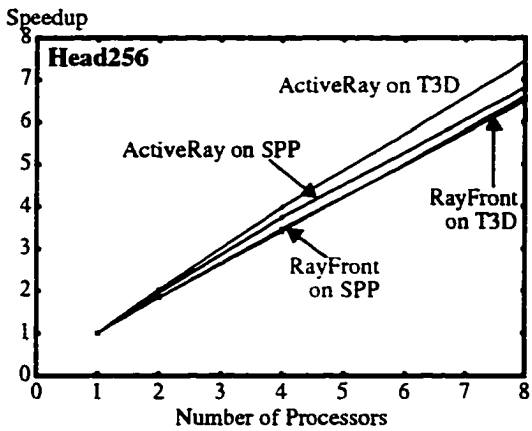
Table 7.3. Comparison of frame times (in seconds) for the RayFront algorithm on Cray T3D and Convex SPP.



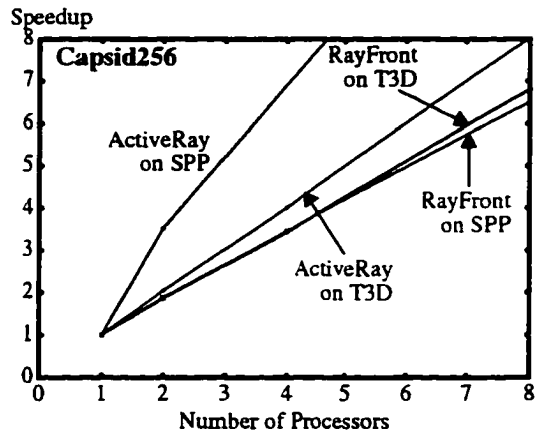
(a)



(b)



(c)



(d)

Figure 7.2. Graphs comparing the speedups of four datasets on Cray T3D and Convex SPP, (a) simple128, (b) sod128, (c) head256, and (d) capsid256.

### 7.3 Cluster of Workstations (COWs)

The ActiveRay and RayFront algorithms were also ported to a cluster of DEC-alpha workstations (model 3000/300), located at the Ohio Supercomputer Center. A group of 8 such workstations are connected using Ethernet (10 Mbits/s) and FDDI (100 Mbits/s). Neither network is switched although the FDDI network is completely isolated, and the ethernet is bridged from the rest of the OVL (Ohio Visualization Lab) network. The FDDI is token ring which results in slowdown in its communication rates. The performance of the above algorithms were tested for both these interconnections to verify their coherent nature and latency hiding capabilities. Table 7.2 and Table 7.2 give the frame times for these algorithms on the COWs.

	Simple128		SOD128	
	Ethernet	FDDI	Ethernet	FDDI
1	2.5	2.5	3.8	3.8
2	1.4	1.5	2.8	3.0
4	1.0	1.0	2.9	2.9
8	0.6	0.6	1.9	1.9

Table 7.4. Frame times (in seconds) for the ActiveRay algorithm on a cluster of DEC-alpha workstations, with Ethernet and FDDI connections.

	Simple128		SOD128	
	Ethernet	FDDI	Ethernet	FDDI
1	3.6	3.6	4.0	4.0
2	3.2	3.1	4.8	4.7
4	2.6	2.5	4.5	4.4
8	1.9	1.9	3.1	2.9

Table 7.5. Frame times (in seconds) for the RayFront algorithm on a cluster of DEC-alpha workstations, with Ethernet and FDDI connections.

The above tables demonstrate comparable performance for both Ethernet and FDDI interconnections. From the ActiveRay results, it seemed that our claim about coherent nature and the latency hiding capabilities of our algorithms were justified, as the speed of the underlying network had little influence on their performance. On the other hand, the RayFront algorithm, which employs no latency hiding, also showed only marginal improvement on the FDDI network. The slight drop in timings for FDDI (for the RayFront algorithm) is due to faster communication, and may be influenced by several factors. First, the message start-up latency

may be dominating the communication times, when compared to actual wire latency, nullifying the effect of network speed. Second, these timings may also imply that the communication is minimized to such an extent that it does not have an effect on the performance, signifying the coherent nature of our algorithms.

The tables also show that the scalability is poor on a cluster of 8 workstations, primarily because of the absence of switches. As such, sending and receiving between different processors cannot be carried on concurrently. The speedups are particularly poor for rendering the SOD128 dataset with the RayFront algorithm, mainly due to lack of latency hiding. Frame times rise when two or four workstations are employed in place of one.

## 7.4 Extensions and Future Research

Parallel rendering as a whole is by no means a mature field. Efficient parallel rendering is an elusive goal, with more questions raised than answered [119]. As parallel computers become more powerful and more affordable, integrating graphics with parallel applications will be a central issue for the visualization community. Efficient parallel rendering algorithms are a prerequisite to this process. Although we have provided effective solutions to some of the open issues, much more remains to be done. Below we propose directions to future research both with respect to this dissertation and to parallel rendering in general. Future extensions to the work presented in this dissertation may be classified in the following areas:

- alternative data types
- alternative volume rendering algorithms
- other architectures
- load balancing
- extension to ray tracing
- hardware implementation

*In general, the following research areas in parallel rendering deserve special attention:*

- dynamic load balancing
- minimizing network congestion and latency hiding

- heterogenous rendering
- algorithm embedding
- other architectures
- hardware implementation

In this dissertation, we have implemented three algorithms for rendering volume datasets, although they are not limited to rendering these models only. The general description of the parallel systems equally applies to rendering polygon-based scenes, with changes just in the rendering portion of the algorithm. Polygonal scenes may be partitioned into cells, each cell consisting of all the polygons residing within it. In contrast to volume-cells, polygon-cells may be of different sizes and thus may need a more sophisticated memory mapping strategy. Some research may be devoted in this area.

In addition to being independent of the scene modeling method, CellFlow and RayFront methods are also independent of whether the rendering is image-order or object-order. For example, either voxel projection, z-buffer, shear-warp, or ray-casting may alternatively be used to render each cell with these parallel algorithms. The current system, which uses ray casting, can be extended to incorporate these rendering techniques as well.

Coherent algorithms minimize communication overheads by exploiting spatial and temporal coherency. This, combined with latency hiding and congestion control, have made our methods extremely powerful in providing almost real-time animations and linear scalability on MPPs (Massively Parallel Processor) like the Cray T3D. The high-bandwidth network available on such MPPs also helps to keep latency low. This is not the case for other parallel environments such as a NOW (Network of Workstations) or a COW (Cluster of Workstations), which are known for their limited bandwidth network connections. To make matters worse, the workstations and network links are not exclusively reserved for a single application, other users may be using the links to add congestion to the network. The performance of our algorithms is bound to deteriorate in such environments (as seen in the previous section). In the future, we wish to study the effect of our algorithms, evaluate the bottlenecks, and device alternate methods to work efficiently in such environments. It will be interesting to evaluate the effect (if any) of different networks like Ethernet, FDDI, and the newly emerging network technology, ATM (Asynchronous Transfer Mode). Using different network specific APIs

(Application Programming Interface) like PVM, MPI (Message Passing Interface), and ATM-API, we expect to improve the performance of the parallel renderer on NOWs and COWs.

Results shown by several authors and those shown in Chapter 3, Chapter 4, and Chapter 5 verify the fact that block IP distribution schemes are prone to severe load imbalance. Interleaved or cyclic screen segment assignments improves the situation to a certain extent, but for certain scenes, even this distribution may offer modest performance improvement. Dynamic methods provide a versatile solution to the load balancing problem. Some authors have adopted these methods for SM machine implementations, but the high message passing overheads associated with DM machines may prove disadvantageous if dynamic methods are adopted. Prospective solutions to dynamic load balancing on DM parallel systems is thus another area of research. While ActiveRay can be extended in this direction with sufficient ease, CellFlow and RayFront algorithms are more rigid in nature.

The final goal of a graphics renderer is to incorporate all the features for realistic imagery. Volume visualization is an arena where the intent is to study and analyze scientific and medical databases. Although transparent objects are easily comprehensible, reflections and refractions are generally avoided due to the enormity and complexity of the datasets. Ray tracing algorithms, on the other hand, can handle these phenomena very conveniently by using simplified versions of the light-transport phenomenon. Introducing these features in parallel rendering is yet to be explored by us. The simplified assumption that the ray-paths are predictable in ray-casting algorithms (as in volume visualization) does not hold for ray tracing. Parallel algorithm design has to be made much more intricate. Several other factors, like load imbalance and severe network congestion start dominating the performance of the algorithm. In our research so far, we have strongly established that our algorithms work very efficiently for a special case of ray-tracing, i.e., ray-casting. We are exploring ways that will provide similar speedups and performance for ray-tracing using the concepts developed here, like memory optimization, maintenance of the thrashless property, minimization of network congestion, latency hiding, and optimal screen traversal scheme. Achieving similar speedups and performance for ray-tracing is a challenging proposition needing further research.

A hardware realization of the object-dataflow parallel algorithms is distinctly another area of future work. In their current form, it is difficult to determine which approach is the stronger candidate for hardware implementation. While ActiveRay is more general in nature, RayFront and CellFlow provide efficient solutions



for more restrictive applications. The easiness with which these software parallel implementations can be transformed to hardware remains to be seen.

Among the general drawbacks in parallel rendering systems, fruitful schemes for dynamic load balancing for IP methods on DM machines, and for OP methods are still lacking. It has almost been confirmed that none of the static load balancing schemes will be able to balance the load in all situations. Dynamic load balancing schemes offer a more versatile solution, but is infested with associated overheads. Several proposals have been made that try to reduce these overheads and make dynamic schemes more beneficial. Dynamic load balancing is simpler to implement on DSM machines like DASH [90], but is more difficult on message-passing machines like Cray T3D and CM-5. Similarly, OP methods have not been too compliant with dynamic load balancing schemes, so some research in this area is in order.

Only a handful of researchers have dealt with the problem of controlling network congestion, some of the notable ones being Neumann [89], Wittenbrink and Somani [127], Vezina, et al. [113], Cohen [19], and the algorithms described in Chapter 3 and Chapter 4. On a similar note, latency hiding for IP approaches have not gained much attention. In Chapter 3 and Chapter 4, we suggest two ways to hide latency, but much remains to be done to verify the validity and consistency of these methods.

With fast networks now becoming available for NOWs and COWs (like FDDI and ATM), the parallel processing platform is slowly but steadily shifting to these environments. Of course, these loosely coupled networks may never provide as high bandwidths as can be found with closely coupled connections on MPPs, but the cost-performance ratio of NOWs and MPPs suggest that despite the slower network, NOWs will be the choice of future massively parallel computing [4]. Much effort is being diverted to make these environments more acceptable within the PVR community.

Similarly, heterogenous implementation of PVR algorithms may prove beneficial. This will need a closer look at these algorithms to extract some level of functional parallelism. For example, Stredney, et al. [106] and Machiraju and Yagel [83] has already demonstrated the use of vector computers in doing incremental transformations of voxels in OP methods. By applying a functional decomposition to the rendering pipeline, different parts of the pipeline can be executed on machines that are most suitable for the function. Yoo, et al. [135] are the only ones to take this bold step. They have utilized both the SIMD and the MIMD portions of Pixel Planes-5 to implement their DM parallel volume renderer. Ma, et al. [81] has also reported the imple-

mentation of their binary-swap algorithm on a heterogenous environment, but without any functional decomposition.

Even when functional decomposition is not the goal, heterogenous environments can be utilized when similar computing resources are scarce. In certain cases, each machine can be used to generate some frames in an animation sequence. More popularly, the rendering job is decomposed and distributed to individual machines, which cooperate to generate a final image. Although the same operations are carried out on each machine, differences in hardware specifications and floating point precisions will result in slight differences in the generated sub-images. For example, in case of an IP decomposition, differences will be evident along the partition boundaries (producing seams along the boundaries). The situation is even more aggravated when one attempts to apply different rendering algorithms on different machines, depending on their capabilities. For example, a machine equipped with rendering hardware may use a slicing based approach, while a vector computer may be utilized for its vector calculation capabilities to perform shearing or splatting. Such heterogenous implementations will definitely give rise to differences in the generated sub-images. An even pressing problem is the load imbalance that arises in these situations. Seamless multi-platform computing is now becoming popular in distributed database queries, but the sophistication of our visual system makes such artifacts unacceptable in visualization applications, and more research is called for removing these artifacts.

The embedding of these algorithms in the underlying architecture has largely been neglected. Independence of the underlying architecture is definitely advantageous. Some algorithms designed for certain topologies may also be ported to other topologies with proper embedding. For example, algorithms designed for ring networks can be conveniently embedded in a hypercube using Gray codes. Efficient methods are also available to embed trees and meshes in other topologies.

One architecture that is slowly gaining popularity is the cluster. Several processors are clustered in the same node, while clusters are connected using some scalable interconnection network. Due to their improved locality characteristics, cluster architectures are being used in current MPPs, like Cray T3D, Convex SPP, and Stanford DASH. For example, the cluster architecture of Convex SPP uses fast shared-memory reads and writes for communicating between processors within the same hypernode, while the more expensive sockets are used for sending messages across nodes. It will be interesting to see how coherent PVR systems can exploit the good locality feature as envisaged in this class of machines.

It is clear that hardware implementation of volume rendering like Cube, Voxel Processor, and Pixel-Planes 5, using multiple processors offers the fastest solution for achieving real-time frame rates. These architectures sacrifice implementation flexibility for gaining rendering speed. A dedicated parallel processing environment for volume rendering, including the desired features in hardware is a challenging proposition. Such specialized hardware will find immense usage in the fields of medical imaging and for virtual reality applications in medicine. As it stands now, providing complete flexibility in hardware seems to be an unattainable task. A consolidation of the most compute-intensive but general operations like compositing and texture mapping in hardware with additional desired features in software is a more realizable goal.

## 7.5 Conclusion

In conclusion, this research has primarily focussed on some of the as yet unexplored problems in parallel volume rendering, e.g., latency hiding, optimizing local memory utilization, optimal screen traversal, reducing network congestion, and portability, and has suggested exclusive ways to eliminate some or all of these. Our coherent algorithms have demonstrated scalability to very high degrees, with potential to improve even further. The accomplishments and contributions to the field of parallel volume rendering is summarized below.

- An incremental rotation scheme [Chapter 3] that exploits temporal coherency to minimize communication, accomplishes effective latency hiding, and utilizes local memory optimally.
- A method with the most efficient local memory utilization [Chapter 4], among the ones proposed in literature. This “cache-only” scheme minimizes the communication overheads, and achieves effective latency hiding, providing a general purpose animation algorithm at the same time.
- A thrashless method for visualizing colossal datasets [Chapter 5], which also maintains its thrashless property across a number of similar frames. This method has shown considerable promise for visualizing compressed volumes as well [Chapter 6], i.e., volumes so big that they have to be stored on disk in a compressed form.
- The thrashless property of the above method also inherently provides an optimal screen traversal method, and has outclassed the most efficient screen traversal scheme proposed so far, i.e., the Hilbert transform [136].
- The scalability demonstrated by the above algorithms are far superior to those reported in the literature.
- The algorithms are transparent to the underlying network topology and thus are portable to parallel machines with any topology.
- Unlike previous methods, our algorithms are primarily designed to perform efficiently on DM architectures, so that the widely available and unused computing resources, like the NOWs and COWs, may be effectively utilized for such enduring tasks like volume rendering.

## List of References

1. M.J. Ackerman, "The Visible Human Project", National Library of Medicine, [http://www.nlm.nih.gov/extramural\\_research.dir/visible\\_human.html](http://www.nlm.nih.gov/extramural_research.dir/visible_human.html).
2. K. Akeley, "Reality Engine Graphics", *Computer Graphics*, 1993. pp. 109-116.
3. M.B. Amin, A. Grama, V. Singh, "Fast Volume Rendering Using an Efficient, Scalable Parallel Formulation of the Shear-Warp Algorithm", *Proceedings 1995 Parallel Rendering Symposium*, October 1995. pp. 15-22.
4. T. Anderson, D.E. Culler, D.A. Patterson, "A Case for NOW (Networks of Workstations)", *IEEE Micro*, February 1995. pp. 54-64.
5. C. Bajaj, V. Anupam, D. Schikore, M. Schikore, "Distributed and Collaborative Volume Visualization", *IEEE Computer*, 27, 7, (July, 1994) 37-43.
6. E. Artzy, G. Frieder, G. Herman, "The Theory, Design, Implementation, and Evaluation of a Three-Dimensional Surface Detection Algorithm", *Computer Graphics and Image Processing*, 15, January 1981, pp. 1-24.
7. J. Arvo, "Space-Filling Curves and a Measure of Coherence", *Graphics Gems II*, Chapter 1.8. pp. 26-30.
8. R. Avila, L.M. Sobierajski, A.E. Kaufman, "Towards a Comprehensive Volume Visualization System", *Proceedings Visualization 92*, October 1992, pp. 13-20.
9. D. Badouel, K. Bouatouch, T. Priol, "Ray Tracing on Distributed Memory Parallel Computers: Strategies for Distributing Computations and Data", in *SIGGRAPH '90 Parallel Algorithms and Architecture for 3D Image Generation Course Notes*, pp. 185-198.
10. D. Badouel, T. Priol, "An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures". pp. 93-106.
11. D. Badouel, K. Bouatouch, T. Priol, "Distributing Data and Control for Ray Tracing in Parallel", *IEEE Computer Graphics & Applications*, July 1994, pp. 69-77.
12. S. Badt Jr., "Two Algorithms for taking Advantage of Temporal Coherence in Ray Tracing", *The Visual Computer*, April 1988, pp. 123-132.
13. L. Bergman, H. Fuchs, E. Grant, S. Spach, "Image Rendering by Adaptive Refinement", *Computer Graphics*, 20, 4, November 1986, pp. 29-37.
14. 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.
15. B. Cabral, N. Cam, J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware", *Symposium of Volume Visualization 94*, pp. .

16. E. Camahort, I. Chakravarty. "Integrating Volume Data Analysis and Rendering on Distributed Memory Architectures", *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp.89-96.
17. J. Challinger, "Parallel Volume Rendering on a Shared-Memory Multiprocessor". Department of Computer and Information Science, University of California at Santa Cruz technical report # UCSC-CRL-91-23 (revised March 1992).
18. J.G. Cleary, B. Wyvill, G.M. Birtwistle, R. Vatti, "Multiprocessor Ray Tracing", Research Report 83/128/17, University of Calgary, October 1983.
19. D. Cohen, S. Fleishman, "An Incremental Alignment Algorithm for Parallel Volume Rendering". *Proceedings Eurographics 95*, September 1995, pp. 123-133.
20. D. Cohen, Z. Shefer, "Proximity Clouds - An Acceleration Technique for 3D Grid Traversal". Tech Report FC 93-01, Math & Computer Science, Ben Gurion University, Beer-Sheva, 1993.
21. R.L. Cook, T. Porter, L. Carpenter, "Distributed Ray Tracing". *Computer Graphics*, 1984, pp. 137-145.
22. B. Corrie, P. Mackerras, "Parallel Volume Rendering and Data Coherence", *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 23-26.
23. T.W. Crockett, "Parallel Rendering", NASA CR-195080 ICASE Report No. 95-31.
24. T. W. Crockett, T. Orloff, "A MIMD Rendering Algorithm for Distributed Memory Architectures". *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 35-42.
25. T.J. Cullip, U. Neumann, "Accelerating Volume Reconstruction with 3D Texture Hardware". Technical Report #TR93-027, Department of Computer Science, University of North Carolina at Chapel Hill, 1993.
26. H Deguchi., H. Nishimura, H. Yoshimura, T. Kawata, I. Shirakawa, K. Omura. "A Parallel Processing Scheme for Three-Dimensional Image Generation", *ISCAS*, 1984, pp. 1285-1288.
27. M. Dippe, J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", *Computer Graphics*, 18, 3, July 1984, pp.149-158.
28. R. A. Drebin, L. Carpenter, P. Hanrahan, "Volume Rendering", *Computer Graphics*, 22, 4, August 1988, pp. 65-74.
29. T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauer, "Active Messages: a Mechanism for Integrated Communication and Computation", *Proceedings 19th International Symposium on Computer Architecture*, May 1992, pp. 256-266.
30. T. T. Elvins, "Volume Rendering on a Distributed Memory Parallel Computer". *Proceedings Visualization 92*, October 1992, pp. 93-98.
31. J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, "*Computer Graphics, Principles and Practice*". second edition. Addison Wesley, 1992.
32. J.E. Fowler, R. Yagel, "Lossless Compression of Volume Data", *Proceedings 1994 Symposium on Volume Visualization*, October 1994, pp. 43-50.
33. G. Frieder, D. Gordon, R.A. Reynolds, "Back-to-Front Display of Voxel-Based Objects", *IEEE Computer Graphics and Applications*, 5, 1, January 1985, pp. 52-60.
34. H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, "Pixel-Planes 5: A Heterogenous Multiprocessor Graphics System Using Processor Enhanced Memories", *Computer Graphics*, 23, 4, July 1989, pp. 79-88.
35. M.H. Ghavamnia, X.D. Yang, "Direct Rendering of Laplacian Pyramid Compressed Volume Data", *Proceedings Visualization 95*, pp. 192-199.
36. A.S. Glassner. "An Introduction to Ray Tracing", Academic Press, 1989.

37. V. Goel, A. Mukherjee, "An Optimal Parallel Algorithm for Volume Ray Casting". *Proceedings International Parallel Processing Symposium*, 1995, pp. 707-711.
38. S.M. Goldwasser, "A Generalized Object Display Processor Architecture". *Proceedings 11th Annual International Symposium on Computer Architecture*, May 1984, pp. 38-47.
39. S.M. Goldwasser, "Rapid Techniques for the Display and Manipulation of 3D Biomedical Data". *Proceedings NCGA '86 Conference, II*, May 1986, pp. 115-149.
40. D. Gordon, R.A. Reynolds, "Image Space Shading of 3-Dimensional Objects". *Computer Graphics and Image Processing*, 29, 3, March 1985, pp. 196-214.
41. S.A. Green, D.J. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing". *The Visual Computer*, 1990, pp. 62-73.
42. S.A. Green, D.J. Paddon, "Exploiting Coherence for Multiprocessor Ray Tracing". *IEEE Computer Graphics and Applications*, 9, 6, November 1989, pp. 12-26.
43. E. Hagersten, S. Haridi, D.H.D. Warren, "The Cache-Coherence Protocol of the Data Diffusion Machine". Michel Dubois and Shreekanth Thakkar, eds., *Cache and Interconnect Architectures in Multiprocessors*. Kluwer Academic Publishers, 1990.
44. P. Hanrahan, "Three-Pass Affine Transforms for Volume Rendering". *Computer Graphics*, 24, 4, November 1990, pp. 71-78.
45. C. Hansen, M. Krogh, J. Painter, C. Verdieri, R. Troutman, "Binary-Swap Volumetric Rendering on the T3D". *Proceedings Cray User Group Meeting*, March 1995, pp. 61-69.
46. C. Hansen, M. Krogh, W. White, "Massively Parallel Visualization: Parallel Rendering". *Proceedings 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995, pp. 790-795..
47. Herman, Liu.
48. L. Hong, A. Kaufman, Y.C. Wei, A. Viswambharan, M. Wax, Z. Liang, "3D Virtual Colonoscopy". *Proceedings Biomedical Visualization 95*, October 1995, pp. 26-32.
49. W.M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering". *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 7-14.
50. T. Itoh, K. Koyamada, "Isosurface Generation by Using Extrema Graphs". *Proceedings Visualization 94*, pp. 77-83.
51. T. Joe, J.L. Hennessy, "Evaluating the Memory Overhead Required for COMA Architectures". *IEEE Computer*, September 1994, pp. 82-93.
52. A. Kaufman, "The Cube Workstation: A 3D Voxel Based Graphics Workstation". *The Visual Computer*, 4, 4, October 1988, pp. 210-221.
53. A. Kaufman. "Volume Visualization", IEEE Computer Society Press, 1991.
54. A. Kaufman, R. Bakalash, D. Cohen, R. Yagel, "A Survey of Architectures for Volume Rendering". *IEEE Engineering in Medicine and Biology*, 9, 4, December 1990, pp. 18-23.
55. M.J. Keates, R.J. Hubbold, "Interactive Ray Tracing on a Virtual Shared-Memory Parallel Computer". *Computer Graphics Forum*, 14, 4, 1995, pp. 189-202.
56. J. Kerr, P. Ratiu, M. Sellberg, "Volume Rendering of Visible Human Data for an Anatomical Virtual Environment". Chapter 44 in *Health Care in the Information Age*, H. Sieburg, S. Weghorst, K. Morgan (eds.), IOS Press and Ohmsha, 1996.
57. R. Knittel, "High-Speed Volume Rendering Using Redundant Block Compression". *Proceedings Visualization 95*, October 1995, pp. 176-183.
58. H. Kobayashi, T. Nakamura, Y. Shigei, "Parallel Processing of an Object Space for Image Synthesis Using Ray Tracing". *The Visual Computer*, 1987, pp. 13-22.

59. H. Kobayashi, H. Kubota, S. Horiguchi, T. Nakamura, "Effective Parallel Processing for Synthesizing Continuous Images". *New Advances in Computer Graphics, Proceedings CGI '89*, pp. 343-352.
60. 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.
61. M.F. Krogh, C.D. Hansen, "Visualization on Massively Parallel Computers using CM/AVS". *Proceedings AVS Users Conference*, May 1993..
62. P. Lacroute, M. Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation". *Computer Graphics*, 1994, pp. 451-458.
63. P. Lacroute. "Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization". *Proceedings 1995 Parallel Rendering Symposium*, October 1995, pp. 15-22.
64. D. Laur, P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering". *Computer Graphics*, 25, 4, July 1991, pp. 285-288.
65. A. Law, R. Yagel, D. N. Jayasimha, "VoxelFlow: A Parallel Volume Rendering Method for Scientific Visualization". *Proceedings ISCA International Conference*, March 1995. pp. 260-264.
66. A. Law, R. Yagel, D. N. Jayasimha, "Parallel Volume Rendering for Scientific Visualization". *ISCA Journal of Computers and Their Applications*, 3, 3, December 1996.
67. A. Law, R. Yagel, "CellFlow: A Parallel Rendering Scheme for Distributed Memory Architectures". *Proceedings International Symposium on Parallel and Distributed Processing Techniques and Applications*, November 1995. pp. 3-12.
68. A. Law, R. Yagel, "Multi-Frame Thrashless Ray Casting with Advancing Ray-Front". *Proceedings Graphics Interface 96*, May 1996, pp. 70-77.
69. A. Law, R. Yagel, "An Active-Ray Approach to Parallel Rendering on Distributed Memory Multiprocessors". *Proceedings Eighth Symposium of Parallel and Distributed Processing, SPDP 96*, October 1996, pp. 414-421.
70. A. Law, R. Yagel, "Exploiting Spatial, Ray, and Frame Coherency for Efficient Parallel Volume Rendering". *Proceedings 6th International Conference on Computer Graphics and Visualization in Russia, GraphiCon 96*, July 1996, pp. 93-101.
71. A. Law, R. Yagel. "An Optimal Ray Traversal Scheme for Visualizing Colossal Medical Volumes". *Proceedings Visualization in Biomedical Computing '96*, September 1996, pp. 33-42, Karl H. Hoehne, Ron Kikinis (ed.). Springer, 1996. Lecture notes in computer Science Vol. 1131.
72. A. Law, R. Yagel, "Parallel Ray Tracing Algorithms: A Survey", in preparation.
73. T.Y. Lee, C.S. Raghavendra, J.B. Nicholas, "Parallel Implementation of Ray-Tracing Algorithm on the Intel Delta Parallel Computer". *Proceedings International Parallel Processing Symposium*, 1995. pp. 688-692.
74. W. Lefer, "An Efficient Parallel Ray Tracing Scheme for Distributed Memory Parallel Computers". *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 77-80.
75. M. Levoy, "Display of Surfaces from Volume Data", *IEEE Computer Graphics and Applications*, 8, 5, May 1988, pp. 29-37.
76. M. Levoy, "Efficient Ray Tracing of Volume Data". *ACM Transactions on Graphics*, 9, 3, July 1990, pp. 245-261.
77. H.K. Liu, "Two- and Three-Dimensional Boundary Detection", *Computer Graphics and Image Processing*, 6, 1977, pp. 123-134.
78. Y. Livnat, H.W. Shen, C.R. Johnson, "A Near Optimal Isosurface Extraction Algorithm Using The Span Space", *IEEE Visualization and Computer Graphics*, 2, 1, March 1996, pp. 73-84.



79. W.E. Lorenson, H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", *Computer Graphics*, 21, 4, July 1987, pp. 163 - 169.
80. W.E. Lorenson, "Marching Through the Visible Human", *Proceedings Visualization 95*, pp. 368-373.
81. K.L. Ma, J.S. Painter, C.D. Hansen, M.F. Krogh, "A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering", *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 15-22.
82. K.L. Ma, J.S. Painter, C.D. Hansen, M.F. Krogh, "Parallel Volume Rendering Using Binary-Swap Compositing", *IEEE Computer Graphics & Applications*, July 1994, pp. 59-68.
83. R. Machiraju, R. Yagel, "Efficient Feed-Forward Volume Rendering Techniques for Vector and Parallel Processors", *Proceedings Supercomputing '93*, pp. 699-708.
84. S. Molnar, J. Eyles, J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Computer Graphics*, 26, 2, 1992, pp. 231-240.
85. C. Montani, R. Perego, R. Scopigno, "Parallel Volume Visualization on a Hypercube Architecture", *Proceedings 1992 Workshop on Volume Visualization*, October 1992, pp. 9-15.
86. K. Nemoto, T. Omachi, "An adaptive subdivision by sliding boundary surfaces for fast ray tracing", *Proceedings Graphics Interface*, 1986, pp. 43-48.
87. U. Neumann, "Volume Reconstruction and Parallel Rendering Algorithms: A Comparative Analysis", Doctoral Dissertation, 1993.
88. U. Neumann, "Parallel Volume-Rendering Algorithm Performance on Mesh-Connected Multicomputers", *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 97-104.
89. U. Neumann, "Communication Costs for Parallel Volume-Rendering Algorithms", *IEEE Computer Graphics & Applications*, July 1994, pp. 49-58.
90. J. Nieh, M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architecture", *Proceedings 1992 Workshop on Volume Visualization*, pp. 17-24.
91. P. Ning, L. Hesselink, "Fast Volume Rendering of Compressed Data", *Proceedings Visualization 93*, October 1993, pp. 11-18.
92. T. Ohashi, T. Uchiki, M. Tokoro, "A Three-Dimensional Shaded Display Method for Voxel-Based Representation", *Proceedings Eurographics 85*, 1985, pp. 221-232.
93. M.E. Palmer, S. Taylor, B. Totty, "Interactive Volume Rendering on Clusters of Shared-Memory Multiprocessors", *Proceedings Parallel Computational Fluid Dynamics*, Elsevier Science Publishers B.V., 1995.
94. T. Priol, K. Bouatouch, "Static load balancing for a parallel ray tracing on a MIMD hypercube", *The Visual Computer*, 1989, pp. 109-119.
95. D. Rogers, "Procedural Elements for Computer Graphics", McGraw-Hill, 1985.
96. J.S. Rowlan, G.E. Lent, N. Gokhale, S. Bradshaw, "A Distributed, Parallel, Interactive Volume Rendering Package", *Proceedings Visualization 94*, October 1994, pp. 21-30.
97. P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields", *Computer Graphics*, 22, 4, August 1988, pp. 51-58.
98. I.D. Scherson, E. Caspary, "Multiprocessing for Ray Tracing: a Hierarchical Self-Balancing Approach", *The Visual Computer*, 1988, pp. 188-196.
99. P. Schroder, J.B. Salem, "Fast Rotation of Volume Data on Data Parallel Architecture", *Proceedings Visualization 91*, pp. 50-57.
100. P. Schroder, G. Stoll, "Data Parallel Volume Rendering as Line Drawing", *Proceedings 1992 Workshop on Volume Visualization*, October 1992, pp. 25-32.

101. H. Shen, C.R. Johnson, "Sweeping Simplicies: A Fast Iso-Surface Extraction Algorithm for Unstructured Grids", *Proceedings Visualization 1995*, October 1995, pp. 143-150.
102. C.T. Silva, A.E. Kaufman, "Parallel Performance Measures for Volume Ray Casting", *Proceedings Visualization 94*, October 1994, pp. 196-204.
103. J.P. Singh, A. Gupta, M. Levoy, "Parallel Visualization Algorithms: Performance and Architectural Implications", *IEEE Computer*, 27, 7, July 1994, pp. 45-55.
104. M. Sramek, "Fast Surface Rendering from Raster Data by Voxel Traversal Using Chessboard Distance", *Computer Graphics*, 25, 4, July 1991, pp. 188-195.
105. P. Stenstrom, T. Joe, A. Gupta, "Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures", *Proceedings 19th Annual International Symposium on Computer Architecture*, 20, 2, May 1992, pp. 80-91.
106. D. Stredney, R. Yagel, S. F. May, M. Torello, "Supercomputer Assisted Brain Visualization with an Extended Ray Tracer", *Proceedings 1992 Workshop on Volume Visualization*, pp. 33-38.
107. M.R. Stytz, G. Frieder, O. Frieder, "Three-Dimensional Medical Imaging: Algorithms and Computer Systems", *ACM Computing Surveys*, 23, 4, December 1991, pp. 421-499.
108. I.E. Sutherland, R.F. Sproull, R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms", *ACM Computing Surveys*, 6, 1, March 1974, pp. 1-55.
109. U. Tiede, T. Schiemann, K.H. Hohne, "Visualizing the Visible Human", *IEEE Computer Graphics and Applications*, 16, 1, January 1996, pp. 7-9.
110. H.K. Tuy, L.T. Tuy, "Direct 2D Display of 3D Objects", *IEEE Computer Graphics and Applications*, 4, 10, November 1984, pp. 29-33.
111. J.K. Udupa D. Odhner, "Interactive Surgical Planning: High-Speed Object Rendition and Manipulation Without Specialized Hardware", *Proceedings First Conference on Visualization in Biomedical Computing*, May 1990, pp. 330-335.
112. C. Upson, M. Keeler, "V-BUFFER: Visible Volume Rendering", *Computer Graphics*, 22, 4, August 1988, pp. 59-64.
113. G. Vezina, P.A. Fletcher, P.K. Robertson, "Volume Rendering on the MasPar MP-1", *Proceedings 1992 Workshop on Volume Visualization*, October 1992, pp. 3-8.
114. R. Westermann, "Compression Domain Rendering of Time-Resolved Volume Data", *Proceedings Visualization 95*, October 1995, pp. 168-175.
115. R. Westermann, "Parallel Volume Rendering", *Proceedings 1995 International Parallel Processing Symposium*, October 1995, pp. 693-699.
116. L. Westover, "Footprint Evaluation for Volume Rendering", *Computer Graphics*, 24, 1990, pp. 367-376.
117. L. Westover, "*Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm*", Doctoral Dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, July 1991.
118. S. Whitman, "*Utilizing Scalable Shared Memory Multiprocessors for Computer Graphics Rendering*", Doctoral Dissertation, The Ohio State University, 1991.
119. S. Whitman, C.D. Hansen, T.W. Crockett, "Recent Developments in Parallel Rendering", *IEEE Computer Graphics and Applications*, July 1994, pp. 21-22.
120. S. Whitman, "*Multiprocessor Methods for Computer Graphics Rendering*", A.K. Peters, Ltd. Wellesley, MA, 1992.
121. S. Whitman, P. Li, J. Tsiao, "Volume Rendering of Scientific Data on the T3D", personal communications.

122. J. Wilhelms, A. Van Gelder, "A Coherent Projection Approach for Direct Volume Rendering", *Computer Graphics*, 25, 4, 1991, pp. 275-284.
123. J. Wilhelms, A. Van Gelder, "Octrees for Faster Isosurface Generation", *ACM Transaction on Graphics*, 11, 3, July 1992, pp. 201-227.
124. C. Wittenbrink, "Designing Optimal Parallel Rendering Algorithms", Doctoral Dissertation, University of Washington, 1993.
125. C. Wittenbrink, M. Harrington, "A Scalable MIMD Volume Rendering Algorithm", *Proceedings Eighth International Parallel Processing Symposium*, April 1994, pp. 916-920.
126. C. Wittenbrink, A. Somani, "Permutation Warping for Data Parallel Volume Rendering", *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 57-60.
127. C. Wittenbrink, A. Somani, "2D and 3D Optimal Parallel Image Warping", *Journal of Parallel and Distributed Computing*, 25, 1995, pp. 197-208.
128. T. K. Wu, M. L. Brady, "Parallel Approximate Computation of Projections for Animated Volume Rendered Displays", *Proceedings 1993 Parallel Rendering Symposium*, October 1993, pp. 61-66.
129. R. Yagel, "Efficient Methods for Volume Graphics", Doctoral Dissertation, Department of Computer Science, SUNY at Stony Brook, December 1991.
130. R. Yagel, D. Cohen, A. Kaufman, "Discrete Ray Tracing", *IEEE Computer Graphics & Applications*, 12, 5, September 1992, pp. 19-28.
131. R. Yagel, A. Kaufman, "Template Based Volume Viewing", *Computer Graphics Forum*, 11, 3, September 1992, pp. 153-157.
132. R. Yagel, Z. Shi, "Accelerating Volume Animation by Space-Leaping", *Proceedings Visualization 93*, October 1993, pp. 62-69.
133. R. Yagel, D. Stredney, G. Wiet, A. Law, "PARAVOL: Parallel Volume Rendering for Virtual Medicine", *Proceedings Cray User Group Meeting*, September 1995, pp. 131-138.
134. R. Yagel, D. Stredney, G.J. Wiet, P. Schmalbrock, L. Rosenberg, D.J. Sessanna, Y. Kurzion, S. King, "Multisensory Platform for Surgical Simulation", *Proceedings IEEE Virtual Reality Annual International Symposium 96*, March 1996, pp. 72-78.
135. T.S. Yoo, U. Neumann, H. Fuchs, S.M. Pizer, T. Cullip, J. Rhoades, R. Whitaker, "Achieving Direct Volume Visualization with Interactive Semantic Region Selection", *Proceedings IEEE Visualization 91*, pp. 58-65.
136. H. Zhang, S. Liu, "Order of Pixel Traversal and Parallel Volume Ray Tracing on the Distributed Volume Buffer", Presented at the *Eurographics Workshop on Volume Visualization*, 1995.
137. K. Zuiderveld, A. Koning, M. Viergever, "Acceleration of Ray Casting Using 3D Distance Transforms", *Proceedings Visualization in Biomedical Computing*, SPIE Vol. 1808, October 1992, pp. 324-335.