

© Copyright 1993
Craig Michael Wittenbrink

Designing Optimal Parallel Volume Rendering Algorithms

by

Craig Michael Wittenbrink

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

University of Washington

1993

Approved by _____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree _____

Date _____

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microfilm."

Signature_____

Date_____

University of Washington

Abstract

Designing Optimal Parallel Volume Rendering
Algorithms

by Craig Michael Wittenbrink

Chairperson of the Supervisory Committee: *Professor Arun K. Somani*
Department of Electrical Engineering
and Department of Computer Science and Engineering

Volume rendering is a method for visualizing volumes of sampled data such as CT, MRI, and finite element simulations. Visualization of medical and simulation data improves understanding and interpretation, but volume rendering is expensive and each frame takes from minutes to hours to calculate. Parallel computers provide the potential for interactive volume rendering, but parallel algorithms have not matched sequential algorithm's features, nor have they provided the speedup possible.

I introduce a methodology to control the complexity in designing parallel algorithms, and apply this methodology to volume rendering. The result is parallel algorithms with all of the features of sequential ones that deliver the promise of parallelism. My algorithms are sufficiently general to run on single instruction multiple data (SIMD) computers and multiple instruction multiple data (MIMD) computers. Through complexity analysis and performance measurements I show that volume rendering is ideally parallelizable with linear speedup and low memory overhead.

Chapter I

Overview 11

Motivation	11
<i>Overview of Dissertation</i>	11
Volume Rendering	11
Problem Statement	12
Research Contributions	13
<i>Computer Aided Research</i>	14
<i>Image Warping Algorithms</i>	14
<i>Volume Rendering Algorithms</i>	15
<i>Fourier Volume Rendering</i>	16
Summary	17

Chapter II

Framework 18

Background	18
<i>Development of Applications</i>	18
<i>Promise and Reality of Parallel Computing</i>	19
Speedup Through Slowdown	24
<i>Bridging the PRAM to Real Machines</i>	29
<i>Slowdown Compiler Techniques</i>	29
<i>Existing System Software And Parallel Languages</i>	32
Algorithm Design On Transition Graphs	34
Automated Choices In Transform Graphs	36
Digression on Optimal Algorithms	38
Summary and Discussion	39

Chapter III

Spatial Warping 40

Background	40
Possible Image Warping Approaches	41
Warping Filters	43
Error Derivation Of Filtering Approaches	48
Optimal RAM Image Warping Algorithm	51
Optimal PRAM Image Warping Algorithms	54
<i>Optimal CREW PRAM Backwards Direct Warp Algorithm</i>	55
<i>Optimal EREW Forward Direct Warp Algorithm</i>	56
<i>Nonlinear Mapping Rules For Forward Algorithms</i>	59
<i>Sequences of Nonscaling Transforms</i>	61
<i>Optimal MCCM 3D Equiareal Algorithm</i>	63
<i>Comparison to Previous 3D Techniques</i>	65
Scaling and Perspective	67
Virtualization	69

MasPar Performance Results	73
<i>Initial Forward and Backward Algorithms</i>	73
<i>Interpolation and Overlapping Optimizations</i>	76
<i>Filter Complexity, Zero Order Hold</i>	78
<i>Optimization By Power of 2 Virtualization, and Register Optimization</i>	80
<i>Optimization Improvements</i>	82
<i>3D Rotation Performance and Implementation Results</i>	84
Summary and Discussion	90

Chapter IV

Spatial Volume Rendering 93

Background	93
<i>Volume Rendering Lighting and Shading Models</i>	94
<i>Surface Lighting Models</i>	95
<i>Particle Lighting Model</i>	96
Algorithm Development Methodology and Existing Approaches	101
<i>Backward Warping Algorithms-Ray Tracing</i>	103
<i>Forward Algorithms-Compositing</i>	106
<i>Surface Fitting</i>	107
<i>Reprojection and Fourier Volume Rendering</i>	107
<i>Existing Methods Performance Summary</i>	107
Optimal RAM Volume Rendering Algorithm	111
Optimal PRAM Volume Rendering Algorithm	111
Permutation Warping For Parallel Volume Rendering	118
<i>Data Parallel Virtualization</i>	122
<i>High Granularity Virtualization</i>	125
MasPar and Proteus Performance Results	127
<i>MasPar Implementation</i>	134
<i>Proteus Implementation</i>	140
<i>Comparison of Proteus With Existing Methods</i>	142
Summary and Discussion	143

Chapter V

Fourier Volume Rendering 145

Background	145
Possible Fourier Volume Rendering Approaches	147
Summary and Discussion	148

Chapter VI

Conclusions 149

Applying the Framework to Other Algorithms	149
Designing Parallel Warping Algorithms	149

Designing Parallel Volume Rendering Algorithms 150
Future Research 150

Bibliography 152

Appendix A
Glossary 164

Appendix B
Derivation of Compositing Complexity 167
Background 167
Back To Front Compositing 168
Front To Back Compositing 169
Parallel Binary Tree Compositing 171
Front-To- Back Binary Tree Compositing 173
Sum of Attenuated Emittances Approach 175
Summary and Discussion 177

FIGURE 1	Mental Processes Used In Research.	18
FIGURE 2	Critical Processes	18
FIGURE 3	Cost Performance Comparison	20
FIGURE 4	Cost vs. Performance	22
FIGURE 5	Slowdown By Reducing Parallelism (Similar to [HENN90])	24
FIGURE 6	Classes of Algorithms	26
FIGURE 7	MCCM Mixed Cost Communication Machine	28
FIGURE 8	Compilation Process By Virtualization and Communication Refinement	28
FIGURE 9	Multigrid Adaptation between Supersteps	31
FIGURE 10	Filtering Directed Graph Representation	34
FIGURE 11	Volume Rendering Transform Graph	36
FIGURE 12	Transform Graph	37
FIGURE 13	Spatial Image Warping	40
FIGURE 14	Image Warping Classification Tree, (*) with new algorithms: Backwards, Forwards, and Overlapped Forwards	41
FIGURE 15	n^{th} order polynomial interpolation by Neville's form of Aitken's algorithm	44
FIGURE 16	Tensor product 2D interpolation by Aitken's algorithm	45
FIGURE 17	Filter Quality Comparison (upper left: zero order hold, upper right: first order hold, lower left: quadratic interpolation, lower right: cubic interpolation)	46
FIGURE 18	Linear interpolation As Affine Combination	47
FIGURE 19	Bilinear interpolation done in horizontal direction first and then vertical direction	47
FIGURE 20	Complete Image Processing System	48
FIGURE 21	Block Diagram of Operations In 2D Warping Algorithm	48
FIGURE 22	Linearized 2D Warp Systems	49
FIGURE 23	3D Linearized Warp Systems	50
FIGURE 24	Simple to Code RAM Backwards Algorithm, , , (RAMB-Simple)	52
FIGURE 25	Clipping To Upright Rectangle	52
FIGURE 26	Optimal RAM Backwards Algorithm, , , (RAMB)	54
FIGURE 27	Backwards Algorithm (CREWB= , MCCMB= for and)	55
FIGURE 28	Nonlinear Mapping	57
FIGURE 29	Near Neighbors In Mesh	57
FIGURE 30	Forward Algorithm (EREWF= , MCCMF=)	58
FIGURE 31	512x512 35 and 45 image otation performed on the MasPar MP-1.	58

FIGURE 32	Processor assignments in a 9x9 mesh to calculate 35 (left) and 45 (right) rotation	59
FIGURE 33	Distance of Interpolation Point in and .	61
FIGURE 34	Processor assignments in a 5x5x5 volume to calculate 25/2, 25, 0 and 35/2,35,0 (x,y,z) rotations	65
FIGURE 35	3D Perspective Volume Distortion	67
FIGURE 36	Scaling Of Data	68
FIGURE 37	Trade-off curve of trading jobs versus communication	68
FIGURE 38	Spreading To Distribute Data	69
FIGURE 39	Striped Allocation of Volume Warping Jobs	69
FIGURE 40	Virtualization Showing Overlapping Boundaries of Subimages	70
FIGURE 41	Volume Virtualization Techniques on a 2D Mesh	71
FIGURE 42	3D Tile Notation	71
FIGURE 43	Nearly Constant Run Time Versus Angle For 2D Image Rotations, Bilinear Filter, Forward and Backward All Sizes	74
FIGURE 44	Run Time Linear In The Number of Pixels, 2D Rotation, Bilinear Filter	75
FIGURE 45	Run Times for 2D Rotation, Bilinear Interpolation on Unit Interval, with Backward, Forward, and Overlapped Forward	78
FIGURE 46	2D Rotations with Zero Order Holds, and Rule (Me) Variant	79
FIGURE 47	2D Rotation, Power of 2 Addresses and Register Optimization, Bilinear Interpolation Forward/Backward, and Zero Order Hold Backward	81
FIGURE 48	Improvement of Each Program Variant for 512 x512 Image Rotation, Seconds Versus Optimization Step	82
FIGURE 49	All 2D Rotation Variants Over All Image Sizes	83
FIGURE 50	Column Virtualization on 1024 PE MP-1 Warping a 128x128x128 Volume	86
FIGURE 51	Slice and Dice Virtualization on 16,384 PE MP-1 warping a 128x128x128 volume	87
FIGURE 52	16k MP-1 MasPar Performance on 128x128x128 Volume Rotation, Slice and Dice compared to Column Virtualization	89
FIGURE 53	Volume Visualization	95
FIGURE 54	Single Level Scattering Particle Model	96
FIGURE 55	Intensity calculation for one point in the volume	99
FIGURE 56	Volumetric compositing calculations	100
FIGURE 57	Data Parallel Volume Rendering Algorithm	102
FIGURE 58	Volume Rendering Transform Graph	103
FIGURE 59	Viewing Frustum For Ray Tracing	104
FIGURE 60	Octree Space and Graph Representation	105

FIGURE 61	Forward Mapping of Voxels into Pixels	106
FIGURE 62	speedup as the number of processors is increased from to for an order interpolation, .	112
FIGURE 63	Fully Parallel Compositing	115
FIGURE 64	Halving of Frames During Parallel Product for Compositing	116
FIGURE 65	Overall Volume Rendering Complexity	117
FIGURE 66	Permutation Warping Parallel Volume Rendering Algorithm	118
FIGURE 67	Transformations and Communications in Permutation Warping for a Single Voxel	119
FIGURE 68	Volume Transformations in Parallel	120
FIGURE 69	Transformation with OS and SS Merged	121
FIGURE 70	Three Dimensional Tiling To Calculate Processor Identification and Subvolume Addresses from Coordinates.	123
FIGURE 71	Spatial Volume Virtualization For a Variety of Architectures	124
FIGURE 72	Steps of Virtual_Permutation_Volume_Render, Virtualized SubVolumes to SubFrames to Final Image	125
FIGURE 73	High Granularity Permutation Algorithm for , Image order resampling storage .	126
FIGURE 74	High Granularity Rounds of Permutation Sends	127
FIGURE 75	Maximum Error in Reconstruction of Cube	129
FIGURE 76	Maximum Error in Reconstruction of Sphere	130
FIGURE 77	OMAX Error for 45x45x45 rotations, Top: Zero Order Hold, Middle: Multipass, Bottom Trilinear	131
FIGURE 78	Data with Ramp to Show Noise	133
FIGURE 79	8X magnification, Zero Order Hold/ Trilinear	133
FIGURE 80	Nearly Constant Run Time Versus Angle	135
FIGURE 81	Run Times Versus Volume Size for the 16384 processor MP-1	136
FIGURE 82	Spatial Volume Virtualization For Proteus	141
FIGURE 83	Run Time Versus Volume Size for Proteus and 16k processor MP-1	141
FIGURE 84	Fourier Slice Theorem, projection top, spectra bottom	145
FIGURE 85	Fourier Volume Rendering	146
FIGURE 86	Volume Rendering Transform Graph	147
FIGURE 87	Polar coordinates	148
FIGURE 88	Back-To-Front Compositing Tree	168
FIGURE 89	Back-To-Front Compositing Calculations	169

FIGURE 90	Front To Back Compositing Tree	170
FIGURE 91	Front-To-Back Compositing Calculations	170
FIGURE 92	Front-To-Back Compositing with Transparency	171
FIGURE 93	Binary Tree Compositing Associative Alternatives	172
FIGURE 94	Front-To-Back Parallel Compositing	173
FIGURE 95	Update Node Problem	174
FIGURE 96	Sum of Attenuated Emittances Sequential Calculations	176
FIGURE 97	Sum of Attenuated Emittances Parallel Calculation	176
FIGURE 98	Compositing Methods	179

TABLE 1	Cost Performance Data for Peak Performance [ZORP92][CYBE92][BELL92]	21
TABLE 2	2D Interpolation error and resolution error for separable interpolation functions (Reproduced from [PRAT78])	51
TABLE 3	Sequential algorithm alternatives	53
TABLE 4	Terms Used in Algorithm Alternatives Table	53
TABLE 5	Algorithms Inner Loop Cost	54
TABLE 6	Performance Constants for Algorithms and filters with restricted rotations	66
TABLE 7	MasPar 2D Rotations (times in seconds) with interpolation not mapped to unit interval, Bilinear Filter	74
TABLE 8	Overlapped Forward Rotation Subroutine Timings, 45 degree rotation	76
TABLE 9	% Improvement and Run Times 2D Rotations (Run times in seconds)	77
TABLE 10	MasPar 2D Rotations (times in seconds) with Zero Order Hold Filters and Rule (Me) Variant	79
TABLE 11	MasPar 2D Rotations (times in seconds) Power of 2 and Register Optimized Versions	81
TABLE 12	Improvement of Each Program Variant for 512x512 Image Rotation, Seconds Versus Optimization Step	83
TABLE 13	Column Virtualization 3D Image Rotation 1k MP-1 Performance in Seconds	84
TABLE 14	16K Processor MP-1 Slice And Dice Timings For Warping, Seconds	87
TABLE 15	Percent Improvement for 3D Slice and Dice Algorithms on 16k Processor MP-1	88
TABLE 16	Rotation Only, From [VEZI92][SCHR91] Milliseconds	89
TABLE 17	16k MP-1 Column Virtualization 3D Image Warping Performance in Seconds	90
TABLE 18	Terms in algorithm	102
TABLE 19	Opaque Voxel Algorithm Architecture Performance	108
TABLE 20	Transparency Voxel Algorithm Architecture Performance	109
TABLE 21	Mean of the Measured Absolute Summed Error over all rays for 45 degree rotation about all axes.	128
TABLE 22	Absolute summed error on rays for 45, 45, 45 degree rotation (See FIGURE 77)	132
TABLE 23	16k Processor MP-1 128x128x128 Volume Rendering Times in Milliseconds	135
TABLE 24	16K Processor MP-1 Slice And Dice Timings For Warping Only, Milliseconds. Reconstruction to align and resample byte voxels with orthographic view.	137

TABLE 25	Rotation Only, From [VEZI92][SCHR91] Milliseconds	138
TABLE 26	Percent Performance Improvement for Different filters using Using Permutation Warping on 16k Processor MP- 1	138
TABLE 27	Volume Rendering Times For 1K MP-1, Seconds	139
TABLE 28	4K MP-2 Column Virtualization Timings for 128x128x128 Volume, Seconds	139
TABLE 29	Proteus Run Times, all output images are 256x256, Seconds	139
TABLE 30	4K MP-2 Slice and Dice Timings for 128x128x128 Volume, Seconds	140
TABLE 31	Speedup Versus for 32 Processors	142
TABLE 32	Initialization costs	177
TABLE 33	Number of Intensity Compositing Steps	177
TABLE 34	Compute Cost, Update Cost	177
TABLE 35	Number of Composites for Updates to transparency/ opacity	178
TABLE 36	Multiplications for All Methods	178
TABLE 37	Additions for All Methods	178

Acknowledgments

I thank Professor Arun K. Somani for his guidance and complete energy in assisting me. I am grateful to my committee Professors Linda G. Shapiro, Robert M. Haralick, Anthony DeRose, and Mark M. Ganter for their interest and help. I am indebted to Professor Steven Tanimoto for his suggestions, inspiration, and example. I received immeasurable help from others at the University of Washington including, Michael Harrington, Srinivas Tri-dandapani, Chung-Ho Chen, Eric Koldinger, M. Y. Jaisimha, and Jonathan Unger. The students and researchers of the GRAIL laboratory in the Department of Computer Science and Engineering have always been helpful and insightful including Stephen Mann, Hugues Hoppe, David Meyers, and Professor David Salesin. Support from the NASA graduate students researcher's program as well as from the Navy through the Proteus project was instrumental in allowing me to complete my research. Professors Arun K. Somani, Robert M. Haralick, and Thomas Seliga provided me with the avenues to obtain research funding.

I was lucky to have contact (mostly e-mail) with many researchers in my field including Professor Marc Levoy, Rachael Brady, Jonathan Becher, Professor Arie Kaufman, Professor Jane Wilhems, Dr. Donald J. Meagher, Professor Bill Lorensen, Claudio Silva, Bill VanZandt, Professor Roni Yagel, Professor Ira Kalet, P. Schroeder, and Tom Malzbender. By both providing references and feedback these graphics researchers have improved my work. I hope to continue my interaction and collaboration with them.

Most importantly of all I thank my wife, Debra, for her support and encouragement.

Chapter I

Overview

1.1 Motivation

Researchers developing parallel applications encounter many difficulties, because there is greater flexibility and complexity than in sequential applications. I propose a methodology to simplify and assist parallel algorithm research. Parts of the methodology can be automated for computer aided research. My initial experiments in the area of graphics and image processing show that not only are there paradigms for algorithms, but there are paradigms for algorithm development. In this dissertation I develop parallel volume rendering algorithms with the methodology. My contribution is superior parallel volume rendering algorithms and a framework for parallel algorithm development. This chapter gives a brief overview of the dissertation in Section 1.1.1. Volume Rendering is introduced in Section 1.2. Then in Section 1.3 the research questions addressed in this dissertation are enumerated and explained. My research contributions, and answers to the questions raised are highlighted in Section 1.4.

1.1.1 Overview of Dissertation

Chapter II covers the scientific research process, reviews the state of the art in parallel algorithm and application development, and then develops a framework for parallel algorithm design. Parallel models of computation are introduced, including my bridging model, the mixed cost communication machine (MCCM). Examples of applying the framework are given. Chapter III covers spatial warping algorithms. Spatial warping is a geometric transform of an image important in volume rendering and image processing algorithms. I present my parallel warping algorithms in Chapter III. Chapter IV covers volume rendering algorithms. Both a survey of existing methods and my parallel algorithms are discussed. Chapter V covers Fourier volume rendering algorithms. Chapter VI concludes the dissertation and addresses future work and generalization of the research to other algorithms and applications.

1.2 Volume Rendering

Volume rendering is an algorithm to visualize sampled three dimensional data. Applications that create sampled data include medical imaging, finite element modelling, photorealistic graphics, molecular microscopy, and nondestructive testing. A collection of point samples is called a scalar field, volumetric data, or voxels (for volume elements). The appropriate visualization technique depends on the application, and surface fitting and rendering is often adequate [FOLE90]. The rendering fidelity is lower when using

intermediate surface representations; therefore, researchers have argued in favor of direct methods that do not convert to surfaces [LEVO90][WEST91].

I call the direct methods *transparency volume rendering* [BLIN82][KAJI86][LEVO90][SABE88]. The physical interaction of light is solved by evaluating particle transport equations, a computationally expensive process. The primary disadvantage of transparency volume rendering is the large amount of computer time and memory required. Using transparency at least doubles the memory requirements over surface methods. The largest volumes that can be processed are about 512x512x512 voxels, and the highest performance is several (1-3) frames a second (on smaller 128x128x128 volumes) [KAUF88][SCHR90][YOO91][NIEH92]. Volume rendering will be more useful with interactive update rates (10-30 frames/second). Animation by changing viewpoints, data, and lighting allows steering simulations [MARS90] and creating internalized visualizations [LAUB90]. Recently, many parallel volume rendering implementations have been published, but they have left open important research questions which I discuss in the next section.

1.3 Problem Statement

There are four primary questions addressed in this dissertation:

- 1 What is the best algorithm for parallel volume rendering?
- 2 What is the best architecture for parallel volume rendering?
- 3 How can trade-offs be made between resources, quality, and time?
- 4 How can questions 1, 2 and 3 be determined for other parallel algorithms and parallel machines?

These questions embody many hypotheses. I have investigated several resulting hypotheses and answered them. Section 1.4 gives an outline of my results. The goal of the dissertation is to determine the best algorithms and architectures for parallel volume rendering. A companion goal is to understand and improve the methods used to develop parallel algorithms.

I investigated techniques for general parallelism: single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD). This approach contrasts special purpose architecture research. Systems for volume rendering have proliferated [GOLD85][JACK88][KAUF88][GEME90][KAUF90][KAUF91b][MALZ90][MEAG91][MOLN92], but are eclipsed by general parallel computers in speed, cost, and programmability. Special purpose architectures such as the Pixar [LEVI84][DREB88], Kaufman-Cube [KAUF88], LMO-2 [MEAG91], PARCUM, Voxel Processor, SCOPE, and 3DP⁴ [OHAS85][KAUF90] achieve only limited improvements over general supercomputers such as the Connection Machine [SCHR90][THIN89], MasPar MP-1 [BLAN90], and MP-2. Also, the adaptability of special purpose architectures to different algorithms, data

sets, and modelling is limited. For example, the Kaufman-Cube [KAUF88] achieves update rates of 16-35 frames/second but uses binary voxel classification, an algorithm simpler than transparency volume rendering. The Kaufman-Cube can not use more advanced shading algorithms such as compositing or numerical integration, and scientists are using ever more advanced visualization techniques. Special purpose graphics machines [KAUF90] (Stellar GS, Ardent Titan, AT&T Pixel Machine [POTM89], Silicon Graphics 4D, HP Turbo SRX, SUN TAAC-1, Pixar, Pixel Planes-5 [FUCH89]) show good performance with heterogeneous (SIMD and multiple types of MIMD) processors, but have limited availability and lag general computing technology.

From each question, 1 through 4, I have investigated the following hypotheses:

- Hypothesis 1 There is an optimal algorithm for parallel volume rendering.
- Hypothesis 2 There are optimal parallel volume rendering algorithms on weak models of computation, such as SIMD.
- Hypothesis 3 There is a continuum of choices between image quality and compute time.
- Hypothesis 4 The optimal parallel volume rendering algorithms and architecture mappings are adaptable to similar image and graphics applications.

These hypotheses are statements that ask: does SIMD or MIMD have an inherent advantage for volume rendering, and what are the different algorithms that are most appropriate for each? To investigate the hypotheses I used both analytical and experimental results. For example, complexity analysis and performance measurement are necessary tools for comparing algorithms. I was careful to correlate my complexity analysis with initial performance measurements, so that subsequent choices in algorithms could be more confidently evaluated. After validation, only algorithms with a clear complexity advantage were implemented. Performance measurements were used to validate and compare to other researcher's results.

1.4 Research Contributions

My research contributions are a new methodology for developing parallel algorithms in Chapter II, new algorithms for parallel image warping in Chapter III, and new algorithms for parallel volume rendering in Chapters IV and V. My methodology for developing parallel algorithms can be automated to simplify and accelerate parallel algorithm design. My algorithms are general and adaptable to shared memory, distributed memory, SIMD, and MIMD machines. They are also efficient in their space and run time complexity. Features and limitations of these algorithms are briefly described in Sections 1.4.2 to 1.4.4. My empirical measurements support these efficiency claims and allow comparison to following work on parallel image warping and volume rendering. Timing and filter results are given in Chapters III and IV, with notable performance results.

1.4.1 Computer Aided Research

Many algorithms exhibit similar characteristics. For example, I have studied morphology, warping, volume rendering, and free-form deformations that all use geometric transformations. Geometric transform based algorithms can be calculated with multiple processor assignments and transform directions. Using a classification of possible transform approaches helps an algorithm designer understand the many possibilities. I have also investigated the use of a dependency flow graph for knowledge representation in algorithm design. I have developed classifications and applied them to warping and volume rendering.

The graph representation allows search methods to optimize algorithms. Stated as a shortest paths problem, the most efficient algorithm altering a representation (vertex) to another representation by subroutines (edges) is selected by calculating the single source shortest paths. Because arcs are missing, the knowledge of the minimum time and space complexity bounds is used to conjecture that the algorithm is optimal or another may be found. I have taken such an approach, and shown how to build the dependency flow graph representation for morphology, warping, and volume rendering. The algorithms that I developed are new arcs which were likely to exist because previous arcs did not achieve the optimal bounds.

An important part of the methodology is use of an abstract machine, the mixed cost communication machine (MCCM), that I developed. This machine is a parallel random access machine (PRAM) that assigns costs to memory accesses. The communication costs help compare algorithms from different PRAMs to determine their relative efficiency.

This methodology can be extended to other image processing and graphics applications, and perhaps to more general domains as well. The algorithm representations, and the algorithm design process have helped me to understand how I create algorithms, and kept me from ignoring important alternatives.

1.4.2 Image Warping Algorithms

An invertible warp is one where the transform has an inverse. A simple algorithm to calculate invertible warps is $O(1)$ on the concurrent-read exclusive-write (CREW) PRAM with optimal storage efficiency. I assume there is a processor for each sample point, and the filter evaluation cost is constant. Because of the concurrent read capability the data is easily accessed, but concurrent reads are hard to emulate in real hardware.

A restricted transform domain allowed me to develop a permutation warp which calculates equiareal warps¹ [MESE83], such as shears, rotations, reflections, and translations, on the exclusive-read exclusive-write (EREW) PRAM. My permutation warp has run time complexity $O(1)$ assuming constant filter complexity, and $P = S$, where P is the

number of processors and S is the number of sample points. The EREW and CREW algorithms have linear speedup when there are fewer processors than sample points. The parallel run time on P processors is therefore $\Theta(t_s/P)$ where t_s is the sequential run time and $P < S$.

On my bridging model of computation, the MCCM, the EREW PRAM algorithm is superior to the CREW PRAM algorithm because of network congestion when using first order or larger filters. Performance measurements on the MasPar MP-1 show that CREW algorithm is superior for zero order filters, but the EREW PRAM algorithm is up to 58% faster for two dimensional warps and up to 100% faster for three dimensional warps with first order filters. The gap widens for higher order filters. The MCCM assumes a general interconnection network, so that these algorithm results generalize to hypercube networks (iPSC Cube), hypercubic networks (butterfly, benes, etc.), reconfigurable meshes (Intel Paragon), and shared memory machines. The use of exclusive reads on shared memory machines such as the Sequent Symmetry S-81 reduces shared bus congestion for higher performance.

1.4.3 Volume Rendering Algorithms

Spatial volume rendering using parallel product and general viewing transforms has run time complexity $O(\log W)$ for W sample points along a view ray and storage complexity $O(S)$, $S = \text{rows} \times \text{cols} \times \text{slices}$, on the CREW PRAM. Using my new permutation warp, volume rendering algorithms using equiareal viewing transforms have run time complexity $O(\log W)$ and storage complexity $O(S)$ on the EREW PRAM. I show how to achieve optimal speedup for both the CREW and EREW algorithms with fewer than $P = O(S/\log S)$ processors for S samples maintaining the same storage efficiency. For $S/\log S < P$ additional speedup is achieved but efficiency falls off. Optimal speedup is linear speedup, and machines with fewer processors than the stated bound achieve linear speedup using my volume rendering algorithm. For example, any machine with fewer than 3,025,551 processors rendering a $(256 \times 256 \times 256)$ volume achieves linear speedup. Extensions to more general viewing transforms are straight forward, more efficient than other methods, but are not optimal.

I have proven Hypothesis 1, because my EREW and CREW algorithms are optimal for processors bounded to $P = O(S/\log S)$. Because available parallel machines have few processors relative to input sizes, volume rendering is ideally parallelizable with run time $\Theta(t_s/P)$ on P processors. This significant result is discussed in Chapter IV.

1. The set of affine transformations that preserve the numerical values of the measures of triangles, and the determinant is equal to ± 1 .

Using both the EREW and CREW algorithms I have developed volume rendering algorithms for high and low granularity. Performance measurements exhibit linear speedup in the problem size, supporting Hypothesis 1. The SIMD implementations on the MasPar MP-1 proves Hypothesis 2. The algorithms require general interconnections, but are efficient on SIMD machines. Weaker interconnection structures require restricting viewing transforms and filter quality. Linear speedup is achieved without communication overhead on SIMD or MIMD computers. And neither architecture has an inherent advantage. In fact, dynamic load balancing techniques can be used on both SIMD and MIMD machines.

The future bottleneck for parallel volume rendering is compositing. Machines will reach this bottleneck only when there are millions of processors, and at that time compositing can be implemented in the interconnection network. In all machines, constant factor speedups can be achieved using data dependent optimizations and dynamic load balancing. I did not implement these techniques as they will only gain constant amounts depending on the data. Data optimizations and load balancing are interesting future work.

I have proven Hypothesis 3 for filter quality. I show that by changing the filter quality you can vary the run time, and the same quality filters used in sequential algorithms can be used in my parallel algorithms without communication congestion. Because the communication and resampling by my techniques in Chapters III and IV solve for multiple order filters, more involved shading models or data preprocessing can be efficiently added. My techniques allow shading and preprocessing to be changed without communication congestion making general purpose parallel machines efficient and extensible for volume rendering. As faster machines become available more sophisticated visualizations will be interactive.

1.4.4 Fourier Volume Rendering

Parallel Fourier volume rendering has complexity $O(\log R)$ so its complexity appears to be similar to the spatial volume rendering algorithms, but the fact that it works with R data elements to calculate a projection, gives it a significant advantage over spatial volume rendering. The time for P processors for spatial volume rendering is $O(S/P)$ while the time for Fourier volume rendering is $O(R \log R / P)$ showing immediately the advantage ($S > R \log R$). I have looked into developing a polar coordinate Fourier transform which would allow picking arbitrary viewing directions without spectral resampling. I also looked into developing ways to incorporate shading into Fourier volume rendering, but I have not solved this problem. Levoy [LEVO92] recently published Fourier directional shading that partially works, but is expensive and may likely be improved.

1.5 Summary

In this dissertation I present a methodology for developing efficient parallel volume rendering algorithms. I present the spatial warping and spatial volume rendering algorithms and implementations that show: there is an optimal parallel volume rendering algorithm, that there are no inherent advantages for SIMD or MIMD, and also that there are clear quality/time trade-offs that can be made. Applying permutation warping to solid modeling through Free-Form-Deformations (FFD's) [SEDE86], or to ray tracing of surface scenes [GLAS89], seems possible. This dissertation will enable others to apply the development methodology to other applications as I show in Chapter II supporting Hypothesis 4.

Chapter II Framework

2.1 Background

2.1.1 Development of Applications

Consider research at the highest level. Ideas are intuitively generated and then are analyzed by critical thought to further develop, validate, or falsify them. The mental processes occurring are shown in FIGURE 1 [STOC85]. As shown in FIGURE 2 creative ideas enter our minds, and we must form hypotheses to test them.

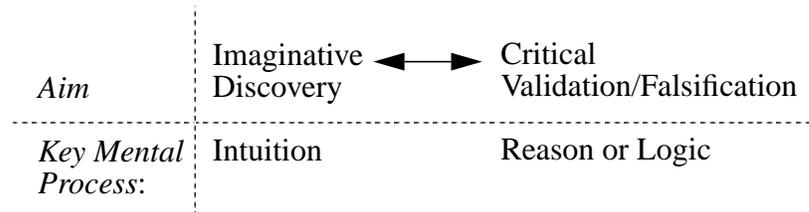


FIGURE 1 Mental Processes Used In Research.

For example, creative idea: projection of volumetric data may be done fastest through direct resampling; hypothesis, a comparison of existing techniques may reveal a superior approach. The hypothesis is further and further quantified as critical thinking progresses, and ideas may be thrown out or deemed useful for some other area. This occurs in the validate/test thought process. Computer applications are an important tool in evaluating hypotheses, and form much of the research done in computer graphics and image processing.

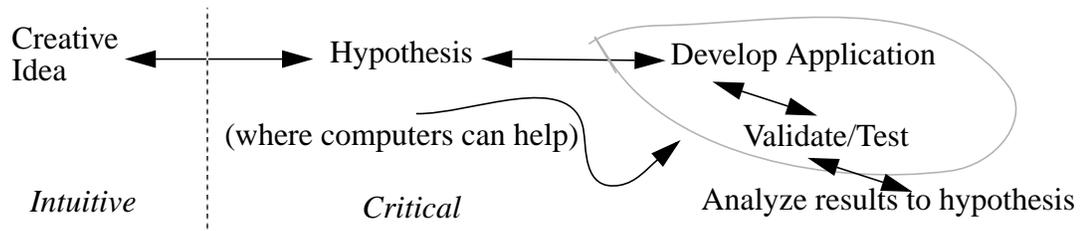


FIGURE 2 Critical Processes

In research, application development has several purposes. Applications are used to calculate intermediate forms of scientifically collected data, make decisions on that data, and control external devices. In my example, volume rendering may be used by medical researchers to interpret tissues. Volume rendering is one part of a larger application, and hypotheses about direct resampling affect the whole application.

A performance study of volume rendering approaches can be done by complexity analysis alone, but application development and measurement give empirical evidence to help support any conclusions from the analysis. If a researcher wishes to try such an approach they need to develop software for each step, integrate each module, and evaluate the performance. The performance can vary due to user input, quality, and the correctness of the approach. Any factors that can be automatically calculated allow the researcher to focus on unproven aspects of the application. The goal is to have applications work as quickly as possible and then investigate the performance. Application development difficulties delay important feedback, and if the effort is too great for implementation, analytical analysis is done rather than building a functional system. Any insight from a prototype system is lost.

The process of developing an application has several steps and goals. The steps are: (1) problem formulation, (2) specification of a solution, (3) means to achieve a solution, (4) development of system, (5) testing and analysis of results, and (6) further refinements or restart in new directions. The goals often used are accuracy, efficiency, correctness, understanding, speed, alternative viewpoints, and validation or hypothesis testing.

Parallel computers provide more potential for demanding and previously intractable calculations, but have also added to the complexity of the development phase. Parallelism requires more sophisticated techniques than sequential computing to develop and analyze applications. If one takes a step back from the research in parallel computing and looks at the goal, it is utilization of parallelism in applications. Parallel control, partitioning, scheduling, and communication are building blocks for developing parallel applications. At a higher level, above the parallelism, application research is done. Problem formulation, theorizing, and system prototyping are what computers are used for, and my proposed framework can assist in application development on parallel computers. The difficulty in using parallelism demands developing tools for parallel application development, as I will show in the next section.

2.1.2 Promise and Reality of Parallel Computing

Parallelism has given a 1000 fold increase in performance over single processors in the last 10 years, but the scientific community has not completely adopted parallelism because of the difficulty in harnessing it. Parallelism has not been widely successful. Scores of applications have been developed for parallel computers, but they are not portable and took

considerable effort to develop [HATC91]. The primary differences in single and multiprocessor development are synchronization, partitioning, communication, deadlock avoidance, standards, and availability.

Parallelism has a lower cost to peak-performance ratio than sequential processing, and its use is the next logical step for high end applications. But, the cost performance improvement is for tuned applications. Because of overhead (synchronization, communication, and scheduling) the peak performance is very difficult to attain, and 1% to 25% of peak is typical [CYBE92]. Today's cost performance ratio is in flux. FIGURE 3 and TABLE 1 [BELL92][CYBE92][ZORP92] show that the cost to peak-performance ratio is not monotonic.

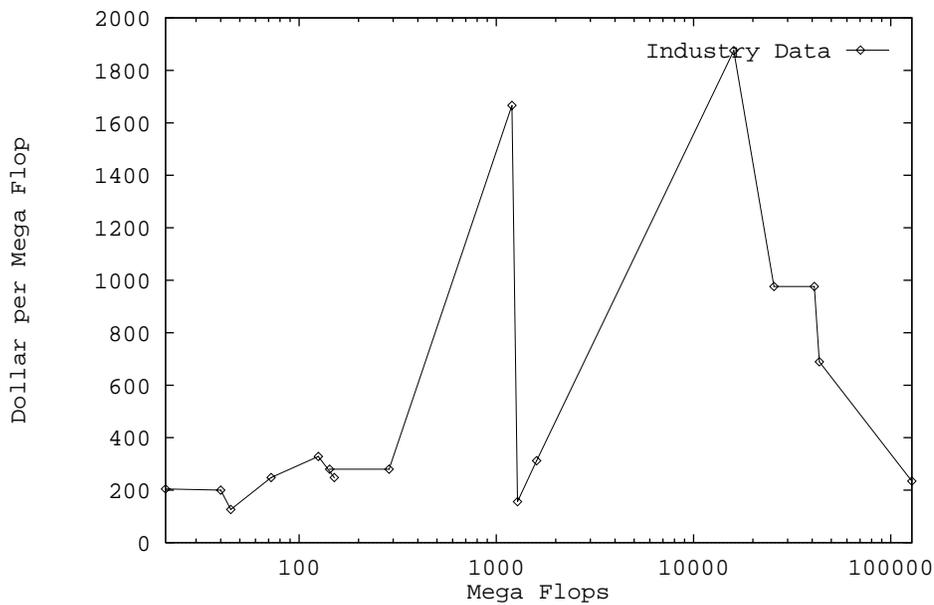


FIGURE 3

Cost Performance Comparison

TABLE 1

Cost Performance Data for Peak Performance
[ZORP92][CYBE92][BELL92]

#Computer	proc.	MFlops	cost dollar \$	Flop/\$	\$/MFlop
SparcClassic	1	21	4295	4889.4	204.52
SunSparLX	1	40	7995	5003.1	199.875
HP715/33	1	45	5695	7901	126.55
HP725/50	1	72.1	17895	4029	248.1969
Decalpha	1	125.1	41195	3036.7	328.285
HP735___	1	150.6	37395	4027.2	248.3
SunSparc10	2	142.8	40000	3570.0	280.11
SunSparc10	4	285.6	80000	3570.0	280.11
MP-1	16k	1200	2000000	600	1666.6
Proteus	16	1280	200000	6400	156.25
MasPar_MP-2	4096	1600	500000	3200	312.5
Cray_C90	16	16000	30000000	533.33	1875
NEC_SX3	4	25600	25000000	1024	976
Intel_Paragon	512	40960	40000000	1024	976
KSR_1	1088	43500	30000000	1450	689
TMC_CM5	1024	128000	30000000	4266.6	234.375

In FIGURE 4 the cost versus peak performance is shown for all of the systems in TABLE 1. The added peak processing power is achieved through using more processors. See for example the jump from the NEC SX3 (4 processors) to the Intel Paragon (512 processors). The improvement from new processors is seen in the gap between the KSR-1 and the CM-5. This cost performance survey shows that machines are not gaining much more parallelism by using off the shelf processors in specialized networks. Machines such as the KSR-1, CM-5, Intel Touchstone, etc. are gaining more performance by using more power-

ful processors. But one cannot use peak power for accurate comparisons. I note only that parallelism remains modest with ever increasing costs.

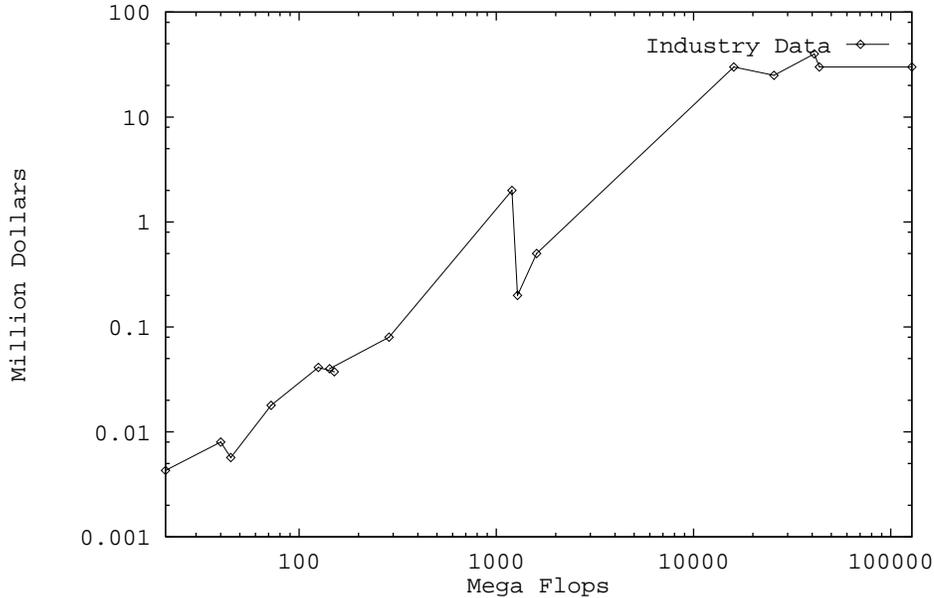


FIGURE 4 Cost vs. Performance

Existing approaches have been expressed as the road to Eldorado [HENN90], a fabled city of riches. Either one opts for millions of simple processors, known as the high road, or one uses thousands of more complex processors, the low road. In both the low and high roads sequential programs are turned into parallel ones.

The speedup of an application is the sequential time T_1 over the parallel time T_p ,

$$\text{Speedup} = \frac{T_1(A_1)}{T_p(A_p)}. \quad (\text{EQ 1})$$

The parallel time is a function of the number of processors and how much work per processor. If an algorithm can be virtualized arbitrarily the relationship of run time while varying the number of processors is,

$$T_p(A_v) = (v/P)T_v(A_v). \quad (\text{EQ 2})$$

T_v is defined as the run time of a virtual processor's work on the algorithm A_v . For algorithms/applications of this type the speedup can be compared on different architectures.

As an extreme example consider using the Cray Y-MP and the Connection Machine CM-2. The time for a virtual processor's work on each processor is different,

$$T_{v_{CM-2}}(A_v) > T_{v_{Y-MP}}(A_v), \quad (\text{EQ 3})$$

related to the performance per processor (MFLOPS, MIPS). There are also different numbers of processors, $P_{CM-2} = 65536$, $P_{Y-MP} = 8$. If the application can be ideally speeded up on both architectures, then the most speedup is attained by the algorithm-architecture pair that minimizes the parallel run time,

$$T_{P_\gamma}(A_v) = \min\left(\frac{v}{P_\gamma} T_{v_\gamma}(A_v)\right), \quad \gamma \in \text{machines} . \quad (\text{EQ 4})$$

So the machine's run times are in a greater or less than relation to each other depending upon the number of processors p , the power per processor T_v , and the ability to efficiently virtualize (EQ 2),

$$\frac{v}{P_{CM-2}} T_{v_{CM-2}}(A_v) ? \frac{v}{P_{Y-MP}} T_{v_{Y-MP}}(A_v) . \quad (\text{EQ 5})$$

In this example the CM-2 achieves greater speedup for $T_{v_{CM-2}} < 8192 T_{v_{Y-MP}}$, or the CM-2's processors can be 8K times slower than the Y-MP's processors and still achieve more speedup.

In practice peak performance and linear speedup are difficult to achieve. As mentioned earlier, most parallel applications achieve only 1% to 25% of peak performance. Existing approaches to achieve speedup are new programming styles (data parallel [HATC91] and functional) and parallelizing compilers (convert sequential to parallel programs) [GELE90][WOLF89][CANN92][BELL92]. The low road allows automated compiling of existing code while the high road requires rewriting applications in parallel form. But those approaches are the paths to El Dorado. Given that parallel hardware has improved while the acceptance of parallelism has not, how can software technology improve? I believe that efficient parallel applications adapt to parallel architectures. I conjecture that reversing the road to Eldorado will provide greater portability and freedom from these lower level issues.

I call the approach *speedup through slowdown*, which means developing applications with as much parallelism as possible. See FIGURE 5.

Definition 1: Speedup through slowdown. A program with fully specified parallelism for v processors can be modified to run on $P < v$ processors by grouping work and slowing it down by emulating v processors with P Processors. The speedup attained is the run time of the P processors over a single processor (EQ 1).

Using higher level algorithm representations allows not only an algorithm designer to crystallize the important information, but provides hooks for automating the design and

compilation of algorithms. Use of slowdown compilers, bridging machines, my transition graph design approach, and transition graph optimizations may help further the success of parallel processing.

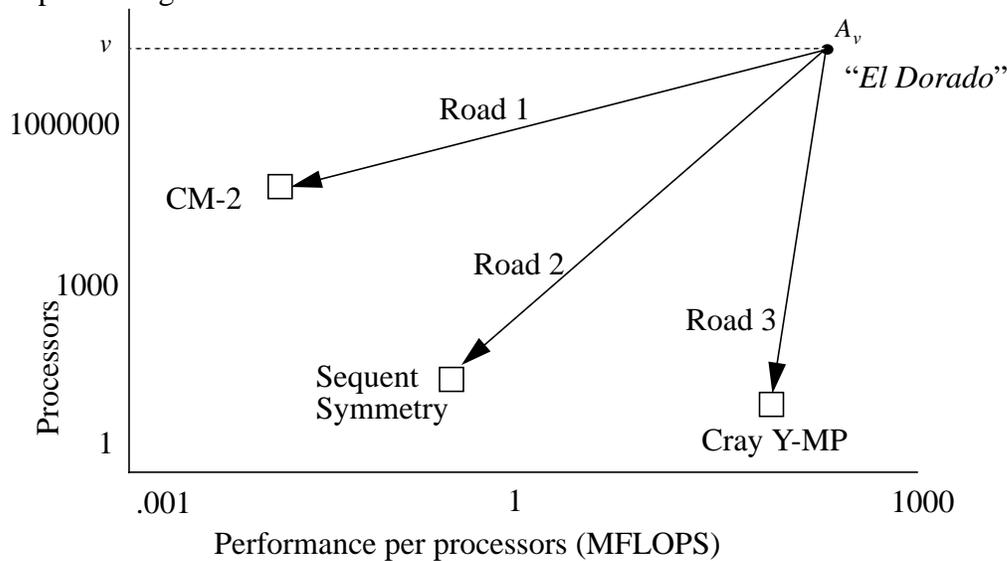


FIGURE 5 Slowdown By Reducing Parallelism (Similar to [HENN90])

In the next section I review efficient parallel algorithm classes and relate them to slowdown compilers. My bridging model, the MCCM, is introduced as a tool for analyzing parallel algorithms. The techniques useful in slow down compilers are reviewed. Then I present a paradigm for transform based algorithms with several important examples.

2.2 Speedup Through Slowdown

The key difference in speedup through slowdown is to ignore the number of processors P of existing machines, and concentrate on developing algorithms with as many virtual (v) processors as needed. v is bounded to be polynomial in the problem size n to be realistic. FIGURE 5 shows algorithm A_v traveling the reverse road to three architectures. This is the opposite approach from [HENN90]. For an introduction to speedup refer to that source. Valiant and others have discussed similar approaches [CANN92][GIBB88][VALI90][VALI90b]. By using explicit parallelism in the algorithm, subsequent transforms of the algorithm maximize parallelism. Whether one takes the high road or the low road back, the difficulty of creating parallelism is gone. Compilers can preserve the efficiency. Slowdown creates portable parallel algorithms.

Hypothesis 5: Slowdown is the methodology that will create parallel program portability and efficiency necessary for the success of parallel processing.

Because slowdown abandons existing parallel software its rewards must be great for the community to investigate and adopt it. The advantages of developing applications with full parallelism are: (1) they can be automatically mapped into machines with fewer processors, (2) portability between different architectures, and (3) generation portability because they benefit from new architectures with more processors. For single processor code portability results from hardware abstraction. Compilers adapt source code to nearly all processors efficiently. Slowdown allows the same approach for parallel portability.

The CM-2/Cray Y-MP example illustrates that parallelism contributes a great deal to speedup for fully parallel algorithms. I assumed that the example application was mappable onto the CM-2. What this means formally is the algorithm is SIMD-transformable. Having examined the price performance and how to get around parallelizing single processor code I now examine the classes of parallel algorithms amenable to slowdown. On the highest level, to achieve portability, algorithms must be developed for abstract machines. This approach has been famously successful for the random access machine (RAM) used for single processor models. The parallel random access machine (PRAM) and its variants are widely used abstract parallel models [GIBB88][CORM90]. A PRAM is a shared memory machine where each processor can randomly access any memory element with unit cost. The processors are assumed to be tightly synchronized. Restrictions are placed upon reads and writes such as: concurrent reads, exclusive reads, and concurrent writes and exclusive writes. The exclusive read and write (EREW) PRAM while the most restrictive is also most efficiently simulated on existing machines.

2.2.1 Efficiently parallelizable algorithms

Nick's Class (NC) is the class of computable and efficiently parallelized algorithms shown in FIGURE 6. NC is defined as parallel algorithms that use a polynomial number of processors, $O(n^{k_1})$, and take polylogarithmic time, $O(\log^{k_2} n)$ [GIBB88], where the input size is n and k_1 and k_2 are constants. P is defined as sequential RAM algorithms whose time is polynomial in the problem size $O(n^{k_3})$. FIGURE 6 is partitioned into parallel (sequential) abstract machine and parallel (sequential) actual machine space. I am interested in these algorithms because they are efficient by definition.

Parallelizing compilers adapt \wp -class algorithms to PRAMs, which is an inherently hard problem because $\wp \not\subset NC$ so efficient parallel algorithms may not exist for some code. I bypass this problem by starting with efficient parallel algorithms in NC. This is done using a slowdown compiler, and sidesteps the difficulty of creating parallelism.

The simulation of a theoretical machine by a real machine is made practical by compilation. Once automated, the compilation of a program for an abstract machine to an actual machine allows one to develop efficient code at a more abstract level. And, the abstract machine hides many details of the hardware from compilers and system software as

well. This approach follows from the methodology of partitioning design work to reduce complexity.

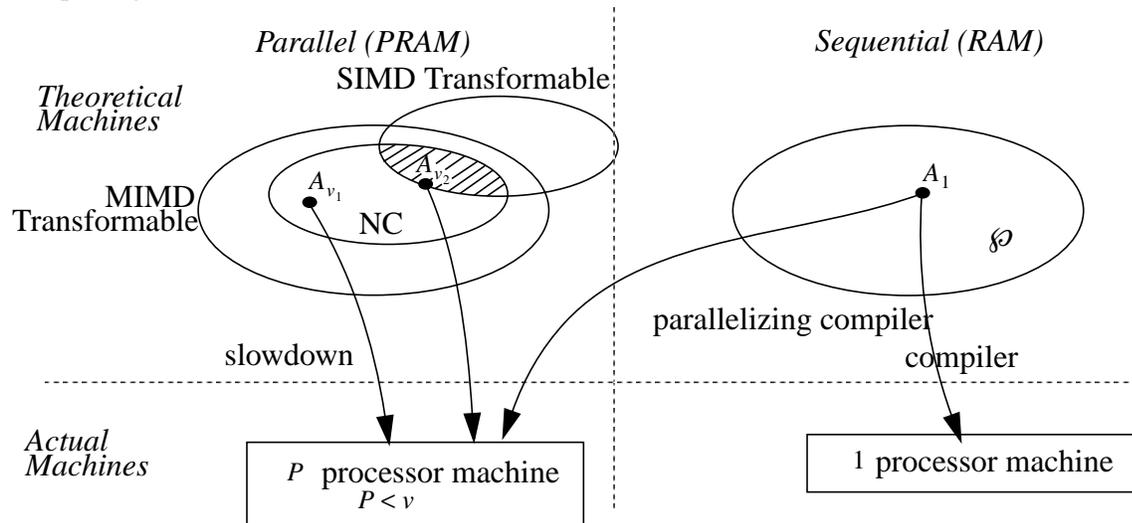


FIGURE 6 Classes of Algorithms

There are four main tasks for developing parallel computing: languages, applications, hardware, and system software development. Slowdown helps by separating these tasks (even with blurry lines). Application development is severed from the platform. Algorithms that are destined for greater parallelism and parallel cost effectiveness are noted by defining classes of algorithms according to their characteristics. I define these algorithms by the following two classes:

Definition 2: MIMD Transformable algorithm is a fully parallel algorithm A_v that can be converted to a machine with P processors $P < v$ with efficiency to within a constant factor, $T_P(A_{P_{\text{MIMD}}}) = O(v/PT_v(A_v))$.

Definition 3: SIMD-Transformable, A fully parallel algorithm A_v developed for the PRAM is SIMD-transformable when it may be converted to an algorithm $A_{P_{\text{SIMD}}}$ that runs on a SIMD PRAM with P processors $P < v$ whose efficiency is $T_P(A_{P_{\text{SIMD}}}) = O(v/PT_v(A_v))$.

A SIMD PRAM is a machine where each processor uses the same instruction stream but for an appointed controller who both controls and determines the instructions the rest of the processors execute.

The slowdown to either MIMD or SIMD is automated through optimizations of an intermediate form of the program. The same source code generates radically different algorithms for the two machines, extracting all parallelism available to the machine. My postulated relation of MIMD-transformable and SIMD-transformable classes to other

classes is shown in FIGURE 6. If the overlap of NC and SIMD-transformable is large than the following hypothesis may be true.

Hypothesis 6: A significant number of parallelizable algorithms achieve greater absolute speedup on SIMD machines than on MIMD machines, because $P_{\text{SIMD}} \gg P_{\text{MIMD}}$, SIMD is more efficiently synchronized, and many SIMD algorithms have already been developed.

The facts that $P_{\text{SIMD}} \gg P_{\text{MIMD}}$ and SIMD is tightly synchronized makes SIMD-transformable algorithms and SIMD machines achieve more speedup. The size of overlap, the cross hatched area where A_{v_2} is in FIGURE 6, is one critical question to investigate for Hypothesis 6. Similar approaches have been discussed for data parallelism such as in [HILL85], but I believe the class of algorithms to contain more than just data parallel algorithms. The two extremes, MIMD and SIMD represent the high and low reverse roads shown by FIGURE 5 (road 1 and road 3): (1) make the machine so powerful and general it can implement any algorithm, and (2) make the mapping and assignment sophisticated to exploit the greater number of processors available to SIMD machines. Because algorithm efficiency on each machine is so critical I define the following bridging models whose characteristics are closer than the PRAM to physically realizable machines. SIMD models in the literature such as [RICE88] are more detailed than I require for the discussion. I propose the *mixed cost communication machine* (MCCM) for the development of fully parallel algorithms. Fully parallel means that the algorithm is written for the problem size and not the machine size. The MCCM is defined in the next Section.

2.2.2 A Bridging Model, Mixed Cost Communication Machine (MCCM)

By ignoring the specific machine topology, algorithms can be designed independently of network specifics but they are ignorant of network costs. My solution to this is to design algorithms on the PRAM, then evaluate their efficiency on a bridging model that takes into account network costs. A 3 level cost model is used with self, local, and global costs. The abstract machine is called a mixed cost communication machine or MCCM shown in FIGURE 7.

Definition 4: Mixed cost communication machine (MCCM): has PRAM execution constructs, and communication constructs: *self* < *local* < *global*. The time complexity for any algorithm is a function of both the computation and the mixed communication costs. Global communication is cost G for a permutation, and any number of requests from or to a destination is N for NG global communication cost, where N is the congestion. Local communication cost is L , and is serialized if there is contention for the same neighbors. Local connections are through a multidimensional toroidally connected mesh. Self communication has unit cost, and is equivalent to the PRAM's memory cost.

Definition 5: $\text{MCCM}_{\text{SIMD}}$ A synchronous MCCM that has the same network power, but uses a single instruction stream. There is also a controller with a separate instruction stream that broadcasts instructions and data to the rest of the machine.

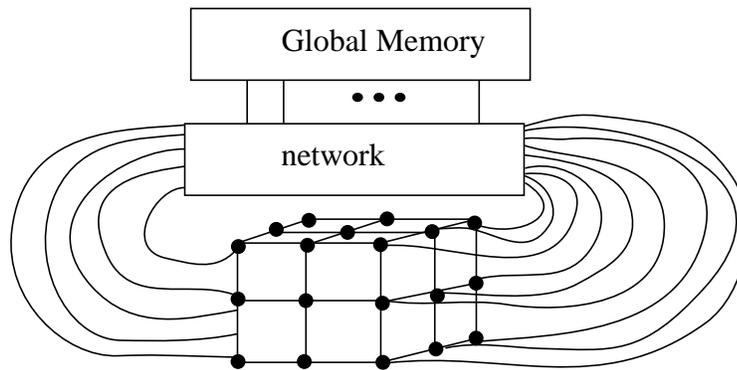


FIGURE 7 MCCM Mixed Cost Communication Machine

The MCCM not only allows evaluating the approximate network costs of algorithms, but also provides an abstract machine that can insure portability. If languages, compilers, and systems software are cognizant of the MCCM model, computer architects can provide MCCM features in computers making the simulation of the machine efficient. FIGURE 8 shows the compilation or evaluation of EREW PRAM and CREW PRAM algorithms on the MCCM. The transitions represent the same transitions given in FIGURE 6, and FIGURE 8 makes explicit the bridging of abstract machines to real machine like the CM-5, Intel Paragon, etc.

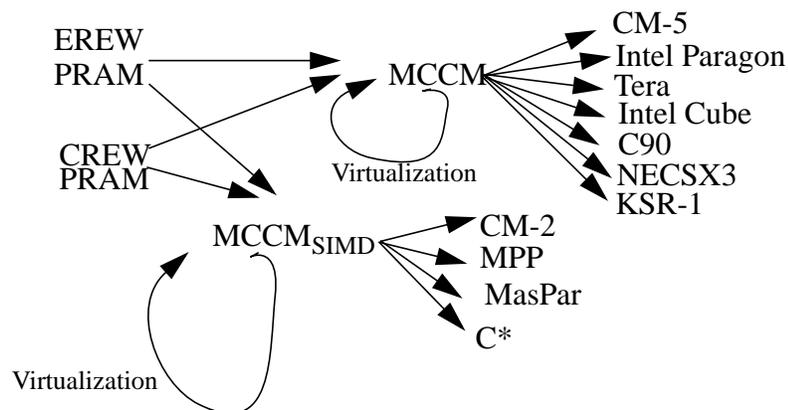


FIGURE 8 Compilation Process By Virtualization and Communication Refinement

If the key difficulty is parallelizing applications and not slowing them down (virtualization techniques, scheduling decisions and so on), then investment in slowdown com-

pilers technology may yield greater gains for both MIMD and SIMD algorithms (Hypothesis 5). In the next section I discuss some of my work on slowdown transform techniques and those in the literature.

2.2.3 Bridging the PRAM to Real Machines

My proposed algorithm development process is to develop a PRAM algorithm, then develop an MCCM version. My claim is that the MCCM algorithm runs with the same efficiency on a wide class of machines. Until efficient slowdown compilers are available, the process is manual, where the MCCM layout, communication, and run time are calculated by the algorithm designer. Through familiarity of required layouts and efficiency on the MCCM I hope the conversion will be automated.

There are a variety of languages designed to perform a similar job such as Spot [SOCH91], Data Parallel C [HATC91], Linda [CARR90], and Ensembles [GRIS90][ALV90] but these are low level languages. The PRAM is the ideal means to develop parallel algorithms, and the MCCM provides a way of more accurately accessing their cost. Many machines are roughly equivalent to the MCCM because they provide a general interconnection network, and neighboring interconnections. Examples of SIMD machines include the Connection Machine CM-2, CM-200, and the MasPar MP-1 and MP-2. Examples of MIMD machines with powerful interconnection networks and neighboring connections include the Connection CM-5, Intel Touchstone Gamma (iPSC/860 Hypercube), Intel Touchstone Delta, and Intel Paragon (Mesh machines).

2.2.4 Slowdown Compiler Techniques

The conversion from a PRAM algorithm to machine code requires compilation of the source language into intermediate code, and then intermediate code is used for global optimization to assign P physical processors to the v virtual processor's work. Parallel compiler research has focused on discerning parallelism from sequential code [WOLF89][GELE90]. Others have worked on necessary techniques in isolation. What I attempt to do here is bring them all together and mention how they are interrelated.

The choices in contracting a program to a machine are: processor assignment, superstep sizes, communication globbing, communication patterns, sequential subroutines, data replication, load balancing, data copy elimination, interprocedural flow analysis, update analysis, graph transforms, and operation optimizations. The compiler can consider each choice in turn, or in multiple passes. The most important choice is *processor assignment*, which affects the communication and load balancing. Ideally one could analyze communication in isolation as proposed by Li and Chen [LI91a], but the relationship between computation and communication requires an iterative analysis.

Virtualizing from v to $P < v$ processors is best chosen by using the program's communication and computation characteristics. In data parallel single point algorithms this is done by looking at the stencil [SOCH90] and picking from the best virtualization to match that stencil and computation. In more complicated programs it requires analyzing the amount of communication between each virtual processor and combining jobs for those with a great deal of communication. This is *communication globbing* where communication is removed by assigning communicating virtual processors to the same physical processors. Once the processor assignments have been made, global communication is improved by decomposition and optimization, through techniques such as dynamic programming [LI91a]. For algorithms calculated in pipeline fashion embedding of the algorithm graph into the architecture graph can be done [GREE92][LEIG92]. For further communication savings the compiler can use *data replication*.

Data replication is most useful for read shared data. One example of data replication that I have found to be useful is in saving boundaries of virtual arrays on processors to eliminate local communication with near neighbors. The replication step requires knowing how much memory is available so as not to overflow caches and/or local memories. The knowledge of the memory hierarchy and data placement improves the compiler's decisions. An example of manual data replication is in [YOO91].

Virtualized parallel calculation is often not as efficient as sequential calculation, therefore the compiler can invoke *sequential subroutines*. For example in parallel prefix [LADN80][KRUS85] the most efficient evaluation for each processor i assigned sub-nodes $x_i \dots x_{i+k}$ is to calculate their prefix in serial fashion rather than by the parallel method.

Load balancing choices are partially fixed in the processor assignment but the key to the contraction of v to $P < v$ is the supersteps of work. Each superstep of the algorithm may use a different processor assignment to both allow optimizations, and to improve performance, a technique more critical for SIMD code. SIMD efficiency can be achieved through supersteps of work. For example in multigrid [BRIGG87][LEIG92] a numerical technique to calculate boundary value, finite difference, and algebraic multigrid problems, the grid spacings change throughout the program, and are dependent upon the progress of the program. For SIMD this requires simply spreading the appropriate data and rerunning the grid with the new assignment. For small grids, the opportunity to work on multiple grid problems simultaneously is possible. See FIGURE 9 below, that shows how the cal-

ulation proceeds to a fine grid where data must be communicated for all processors to be used.

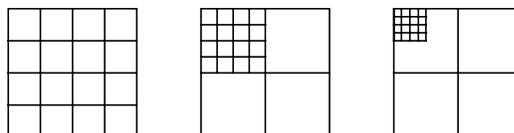


FIGURE 9

Multigrid Adaptation between Supersteps

Another example of load balancing is shown by [MASP91] in the calculation of fractals. By straight forward assignment of pixels to processors the run time is the worst case, because all processors must wait for the slowest to complete its iterations. By randomly assigning pixels the slowest processor has less work by checking during iterations if the pixel has finished and relying on the worst case pixels to be distributed across the processors. Load balancing can be achieved through judicious intermixing of supersteps and processor assignments even reassigning processors after partial calculation to redistribute the load. This dynamic superstep assignment is useful for both MIMD and SIMD processing.

Another compiler optimization method is *data copy elimination*. A result of functional programming and interprocedural effects, data copying can be extremely inefficient [CANN92]. Interprocedural flow analysis helps the compiler to achieve data copy elimination. Update analysis allows different types of data areas to be declared, as whether they are read shared or exclusive areas. Special memory areas can be more efficiently used if located in closer levels of the memory hierarchy. Additionally, restrictions on the development of the program can greatly improve the chances of efficient compilation, for example using EREW PRAM abstraction removes concurrent reads. Improvements such as hashing [VALI90] have been proven to distribute arbitrary communication to avoid contention.

Graph transforms can be done by library search [LI91b] and decomposition [LI91a], but they can also be used to alter the calculation itself Section 2.3. By examining the dependency graphs in image processing algorithms Li and Jamieson [LI91b] collapse the graph into a hypergraph that may be more quickly matched to the architecture embedding library. The library provides a solution using graph matching with heuristics to reduce the graph isomorphism complexity.

Li and Chen [LI91a] use a similar approach. A shared memory parallel algorithm is compiled to distribute it. Communication is broken up, changing it into canonical form and matching a library of cost parametrized routine using dynamic programming [CORM90] to minimize communication cost.

More generally, *graph transforms* are used to reduce communication, reduce the depth of the dependency graph, and load balance the computation. The data dependency flow graph is used in traditional supercompilers, as well as systolic array designs [KUNG88]. By transforming the full dimensional dependency flow graph into lower dimensions efficient virtualizations can be found. Also general communication graphs can be embedded into hypercubes and butterfly graphs for more efficient communications [VALI90][LEIG92]. Load balancing communication can also be done by not changing the program graph, but by hashing either the memory addresses [VALI90b], or the processor assignments themselves.

The final task is *operation optimizations*, where reordering of calculations makes code more efficient. The 12 compiler tasks discussed are some of the techniques necessary for effective slowdown compilers. Others are to be found, and the relative effectiveness and importance of each needs to be fully researched.

2.2.5 Existing System Software And Parallel Languages

System software is important to the success of slowdown. Because effective use of parallelism requires high average throughput, the system load and user requirements, affect how best to run programs. This is apparent in the fastest throughput machines today, such as the Cray Y-MP, that serves thousands of users with transparent process migration and switching [BELL92]. The ability to serve a community of users requires rapid task switching, and also examination of the true performance of applications on machines. By requiring full parallelism in the algorithms, flexibility is enforced so they can be run on machines of any size.

Programming languages are also necessary for writing fully parallel programs. Functional programming, parallel Fortran-D [HIRA92], PRAM languages (as suggested in [VALI90b]), and message passing languages such as OCCAM [INMO84] and Ensembles [GRIS90][ALVE90] are all attempts to provide user control of parallelism. There are architecture independent languages (PICL [GEIS90], Data Parallel C [HATC91]), proposed parallel programming environments (C-Linda [CARR90], C++ and Presto [BERS88]), Amber [CHAS89]), new types of approaches for parallelism (functional programming), and parallelizing compilers to make parallel code more efficient. There has been mixed success such as Harrison III and Ammarguella's study [HARR90] where parallelizing compilers out performed parallel languages and compilers and vice-versa. Another example of the difficulty is Data parallel C where Hatcher et al. report that they must tune matrix multiplication for each architecture, "However, the result of tuning may well improve performance on one architecture at the expense of execution speed on another architecture," ([HATC91] p381).

Another example of an architecture independent programming language is the PICL extension to C programs for image and vision processing. Geist et al. [GEIS90]

claim the same source code executes upon the ipsc/2, ipsc/860, NCUBE 3200, and NCUBE 6400 without change. The libraries are limiting though, because they use explicit message passing forcing the user to program in that fashion.

Carriero and Gelernter approach architecture independence by using coordination languages [CARR90]. A coordination language is a programming language together with a coordination language—constructs used to synchronize, communicate, and schedule. A coordination language is the organizing strategy for parallel programs. Their implementation of the Linda coordination, called C-Linda, allows the expression of programs in what they believe are the three important types of parallelism: result, agenda and specialist. The ones that they don't consider are data parallelism and speculative parallelism or logic programming.

Yet, the tendency to link the programming model to the hardware is typical as shown in Li and Chen's communication optimization discussed earlier [LI91a]. They approached algorithm development by automated compilation, claiming the shared memory programming paradigm "shields the user from many such low level concerns." But, to optimize difficult communication they propose to add user directives to the compiler. To do this the user must somehow realize that the original coded algorithm is inefficient and add compiler directives to make the program more efficient.

All of these systems have been fielded with mixed success. Architecture independent programming languages are more difficult to develop for parallel computers than for sequential computers. The problems with scheduling, synchronization, and communication overhead are exacerbated by languages designed for different parallel computers. A program may be very efficient on one system, and not nearly as efficient on another. That is why I propose slowdown as a new methodology to develop architecture independent parallel languages. It is important to allow description of parallelism at a high level, natural to the programmer, and have tools/compilers perform the conversion to the most efficient implementation. I feel high level PRAM languages combined with sophisticated compiler and software technology are an ideal and practical approach.

Compiler transforms must create efficient code without tweaking and omniscience by the user, or slowdown will not work. Also application libraries cannot be relied upon, because efficient routines may not exist. Available application library routines can also be improved. Alternative means to solve the algorithm may be more efficient than the first coded approach, but if the algorithm is optimal on an abstract model of computation, such as the PRAM, then the algorithm should be efficiently mapped to the hardware. Using more restrictive models of PRAMs can of course improve efficiency, but should not be required.

System software and high level languages must be developed to support slowdown compilers. System software and language research should be done in parallel with slowdown compiler and architecture work. In fact one of the best reasons to pursue slowdown,

is that it finalizes the intermediate parallel code and algorithms, so that refinements in operating systems, compilers, profilers, and languages can be worked upon in isolation from future platforms, and used to improve existing platforms.

2.3 Algorithm Design On Transition Graphs

As there are paradigms for algorithms, there are also paradigms for algorithm development. For the majority of image and graphics processing a transform based technique may be used to develop, validate, and analyze research. More importantly the large number of data representations and means to generate them allows high level applications to take advantage of untried transitions, and the assisted computational investigation of these transitions can speed the application assembly and validation.

Algorithms are represented as a directed graph $G = (V, E)$ where data representations are vertices $v \in V$ and transforms to intermediate representations are edges $e \in E$. In geometric transform algorithms there are many edges going from one representation to another. The edges represent alternative ways to calculate the same result. FIGURE 10 shows sequence filtering. Filtering can be calculated by convolving a spatial filter with samples to create an output sample. This is shown by the backward edge in the graph. The same output can also be calculated by taking each input and convolving them with a filter that is summed in the output. This is the forward edge in the graph.

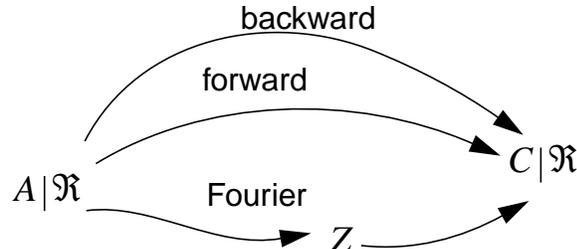


FIGURE 10 Filtering Directed Graph Representation

The backward operation convolves sequence A with B to get sequence C , $C = A \text{Operator} B$. Pseudo code for backward processing is:

```

Initialize C
  for  $c \in C$ 
    for  $k \in B \cap A$ 
       $C[c] = \text{Operator}(C[c], A[c, k], B[c, k])$ 
  
```

The forward algorithm is:

```

Initialize C
  for k ∈ B ∩ A
    for c ∈ C
      C[c] = Operator(C[c], A[c, k], B[c, k])

```

The outer loop defines forwards or backwards calculation and each direction calculates the same result. Also one can calculate the filtered image by first transforming into the frequency domain, multiplying by a filter, and then transforming into the spatial domain. This is the Fourier edge transitions in FIGURE 10. Examples of operations that can be calculated forwards or backwards include matrix product, grey scale dilation, convolution, and grey scale erosion. Below I define the example operators:

Matrix multiply

$$\text{Operator}(C[c], A[c, k], B[c, k]) = C[i, j] + A[i, k] \times B[k, j] \quad , \quad \text{(EQ 6)}$$

Convolution

$$= C[i] + A[i - k] \times B[k] \quad , \quad \text{(EQ 7)}$$

Grey scale dilation

$$= \text{Max}(C[i], A[i - k] + B[k]) \quad , \quad \text{(EQ 8)}$$

Grey scale erosion

$$= \text{Min}(C[i], A[i + k] - B[k]) \quad . \quad \text{(EQ 9)}$$

In fact, matrix multiply can be calculated on the Fourier transition, and perhaps dilation and erosion can as well. The operators generalize to higher dimensions. The existence of multiple ways to calculate the same result arise again and again. Other examples include spatial warping Chapter III, volume rendering Chapters IV-V, ray tracing, and radiosity [FOLE90]. If a new application has been implemented with an algorithm that uses such transforms, then most likely there will be a variety of ways to calculate an equivalent output. Examining an algorithm at this level of detail allows quick classification of approaches, identification of untried approaches, and analysis of the most effective approach for the application at hand.

FIGURE 11 gives volume rendering's transform graph. Each approach will be discussed in Chapters III-V, and application of this graph helped me to develop the algorithms in this dissertation. The basic transitions, forward, backward, Fourier, are applicable to many graphics and image processing algorithms. When designing algorithms, each alternative should be investigated because memory cost, computer resources, and module integration affect the best choice. The transition graph allows research with a view of the entire application. Further, resource, time, and quality goals can guide an opti-

mization tool. In the next section, 2.4, I describe how to verse algorithm design as an optimization problem.

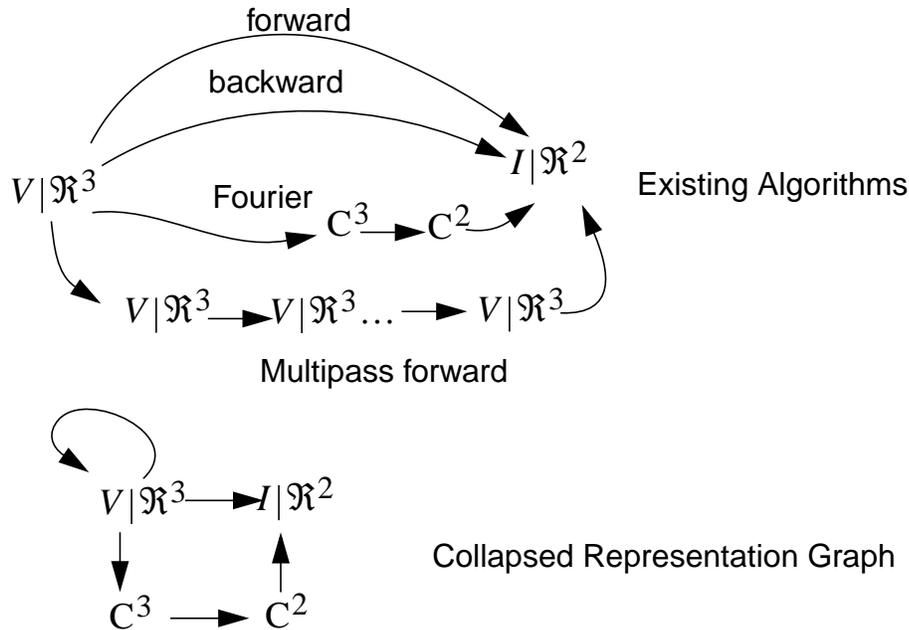


FIGURE 11 Volume Rendering Transform Graph

2.4 Automated Choices In Transform Graphs

The obvious approach is to solve an optimization problem with the application described in terms of source destination paths labelled with complexity costs and empirical results. This methodology allows graphics and image processing research to be assisted by specifying important factors to minimize and the available resources. Available resources include

- computer systems performance, memory, I/O, and storage.
- time: interactive 30 frame/sec. or days to weeks available.

In a research environment applications are in development, and transition costs are unknown. The output of the minimization includes the unknown costs in a meaningful fashion and provides alternative transitions if available. Such an interactive tool formulates its own information, prompts for the most useful information and speeds development. Because graphics and image processing algorithms are edges, the novelty of this approach is that it provides a framework within which to develop and validate/test computer applications, a computer aided research (CAR) tool. The researcher enlists the com-

puter to aid the critical thought process to prove or disprove hypotheses making parallel computers more useful.

Using the dependency flow graph representation of the algorithm a single source shortest paths can find the most efficient algorithms. Once the best path is chosen, efficient machine code can be created. The best approach to maintain parallel performance is through a slowdown compiler because of the difficulty in parallelizing sequential code as discussed earlier.

Consider a transform graph with edges, $e \in E$, and vertices $v \in V$. FIGURE 12 shows an example graph. Each vertex is a data representation, and each edge is the cost of the transition from a vertex to another vertex. There is a full vector complement of costs along each edge. Costs are time complexity, space complexity, empirical compute time, etc. Example representations are spatial, frequency, Hough space, boundary representation, NURBS, URBS, triangles, and the multiple dimensions of these spaces. Example costs are the fast Fourier transform, resampling, Radon transform, rendering, interpolating, surface fitting, volume rendering.

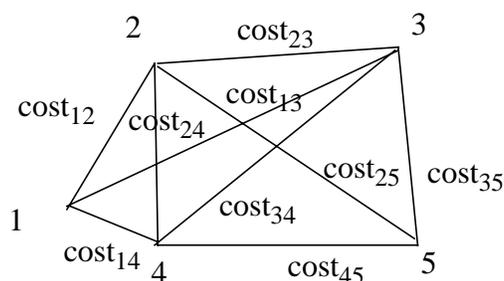


FIGURE 12 Transform Graph

The vectors $e = (u, v)$ are used to determine a minimum cost path from one vertex to another. The transition from one vertex to another represents a software/hardware module. From an earlier example, convolution, the representations are digital images. For a serial algorithm the transition is the complexity of convolving which depends on the filter. The researcher decides the filter type, image and filter sizes, and representations. But the most efficient calculation edge may change when the compute platform, representation, or filter, changes. The choice of the most efficient edge is left to an assisting tool in the framework.

The transition graphs used will vary with cost type, transition direction, and compute platforms involved. Because each compute platform will have different capabilities, the problem changes simply to an optimization in a transition graph for each system. The

comparison factor, true speedup, is calculated using the fastest sequential algorithm available.

2.5 Digression on Optimal Algorithms

For the PRAM optimality is the minimum amount of work required [GIBB88]. This includes a measure of the execution time, and the number of processors required. The cost of the algorithm is the number of time steps required for the algorithm to complete $T(n)$. The efficiency is given as utilization = #of PE's used in each time step, or work, the processor time product $P(n)T(n)$. An algorithm may be time optimal, but work inefficient.

Definition 6: Optimal speedup. Linear speedup of the parallel program over the fastest known sequential program.

Definition 7: Optimal run time. A lower bound dependent upon the model of computation.

Definition 8: Optimal space complexity. $O(n)$ on the order of the input size.

Definition 9: Optimal efficiency. Work efficiency, or time for the parallel algorithm times the number of processors equals the time for the fastest sequential algorithm.

Optimality is often determined by achieving lower bounds for combining data. For example, if a calculation is associative and pair wise the lower bound is $\log n$ [LEIG92]. For fixed network algorithms optimality can be measured by the bisection width, I/O bandwidth, and diameter of the network. For example a mesh without toroidal connections has a lower bound on communication of data from one PE to another that is $O(n)$ which results from the diameter of the network. These measures are only rough guides for optimality. Obviously you can do worse than optimal. For example, a mesh algorithm for connected components is $O(n^2)$ because the path of a fully snaking component [TANI90]. And you may also do better such as a $O(\log n)$ time sorting of binary numbers on a full binary tree with bisection width of 1 [LEIG92]. Algorithms that beat the optimality criterion given by the graph connectivity are the exception.

Optimality for the MCCM is determined by the network also. If communication is one-to-one then the diameter of the network is G , the global communication cost. A minimum for any algorithm that must combine global info is $O(G)$. The communication cost of the algorithm is expressible by $C(n) = a + bL + cG$ where a is self communications, b local communications and c non conflicting global communications. An optimal MCCM algorithm will use $1G$ for each global value that must be exchanged, and $1L$ for each local value that must be accessed, plus the computational requirement which may be no less than the PRAM's $T(n)$. Reads to common locations may be done, but they are serialized.

2.6 Summary and Discussion

By understanding how parallel algorithms are created, partial automation of the algorithm design process is possible. First the most efficient approach is chosen from a dependency flow graph representation of the algorithm which changes with platforms and resource importance. Then the subroutines are compiled from a high level parallel language using speedup through slowdown. These abstractions provide greater freedom and knowledge in adapting parallel programs to machines. Because high level fully parallel programs are scalable and generation portable, I can motivate the necessary software development of languages and compilers. The long term goal of speedup through slowdown allows efficient speedup of parallel algorithms and applications, and the dependency flow graph automated search will assist development of algorithms for applications, and improve algorithm design.

The success of software tools for refining applications into parallel systems will determine the success of the machine models. By using bridging models, algorithm designers can quickly determine the efficiency of their algorithm, and hardware designers understand the features that must be provided. My bridging model the mixed cost communication machine (MCCM) is useful in comparing PRAM algorithm's actual implementation performance. In Chapters III and IV I use the MCCM to assess the network cost of my PRAM algorithms and validate the results with performance measurements.

Both MIMD and SIMD can make general dramatic improvement through development of slowdown software technologies. Through examination of the number of processors in parallel machines I demonstrated that parallelism has not greatly increased. I advocate developing parallel machines with many more processors, high level algorithms, and slowdown software technologies to help make parallelism more cost effective and useful.

Chapter III

Spatial Warping

In this chapter I present two new parallel image warping algorithms that are optimally efficient with low and high order filters on the parallel random access machine (PRAM). The simpler algorithm has concurrent reads and exclusive writes (for the CREW PRAM), and the non-obvious algorithm has exclusive reads and exclusive writes (for the EREW PRAM). I use the MCCM to provide a more accurate prediction of algorithm performance. I show that the algorithms are optimal on the PRAM, and that the EREW PRAM algorithm has optimal communication on the MCCM. MasPar MP-1 and MP-2 performance measurements correlate with the MCCM 2D and 3D spatial warping predictions.

I review warping and classify algorithms into forwards and backwards methods in Sections 3.1 and 3.2. Filters are discussed in Sections 3.3 and 3.4. My algorithms are discussed in Sections 3.5 to 3.8. Section 3.9 gives performance measurements and optimization details. The chapter concludes in Section 3.10.

3.1 Background

Image warping is a spatial transform of an image called texture mapping, rubber sheeting [WEIN90], coordinate transforms, and geometric correction [JAIN89]. Image warping can be quite complex, such as quadratic and cubic transforms [SMIT87] [WEIN90] [WOLB89]. I define two geometric spaces: the object space (OS), where the input image resides, and the screen space (SS). The algorithm relates a point p in OS to p' in SS by a transform T or $p' = T(p)$. The inverse transform is $p = T^{-1}(p')$. FIGURE 13 shows image I being warped to image J . In each space there is a rectangular coordinate system where the images are represented by discrete samples: $I[x, y]$ and $J[x, y]$. The extent of valid points is defined by a bounding box for each image, B_I and B_J .

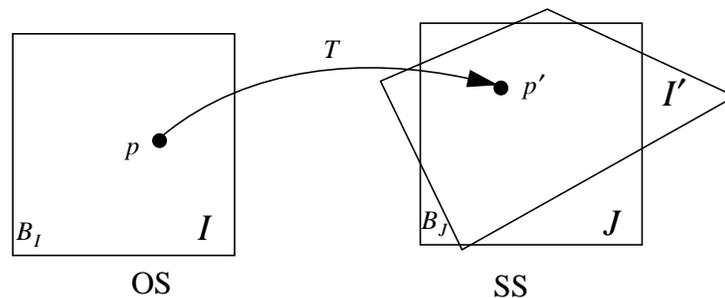


FIGURE 13

Spatial Image Warping

To calculate the discrete samples of $J[x, y]$ requires reconstruction of I at points p , which do not in general lie at I 's samples. Quality versus cost trade-offs have resulted in many approximations of reconstruction [BARR81] [BENN84] [CATM80] [FRASE85] [PAET86] [SCHR91] [SMIT87] [TANA86] [WEIM80] [WEIN90]. Image warping is done in a one pass transforms, *direct warps*, and multiple pass transforms, *multipass warps*. Direct warps and multipass warps may be performed by serial or parallel algorithms.

Image warping has many applications in computer graphics and image processing. Computer graphics applications include texture mapping [HECK86], ray tracing [GLAS89], graphics design, and volume rendering [DREB88] [SCHR91]. I am interested in warping primarily for volume rendering in Chapter IV but my algorithms may also be adapted to image processing applications such as correcting optical aberrations, image registration [OWCZ89], image restoration [GOSH89] [YOKO86], and cartography.

3.2 Possible Image Warping Approaches

It is easy to get lost in the details of specific approaches. FIGURE 14 shows a classification using the directed graph representation of warping. Each leaf is a different edge from the warping flow graph representation, and algorithms are differentiated by four factors.

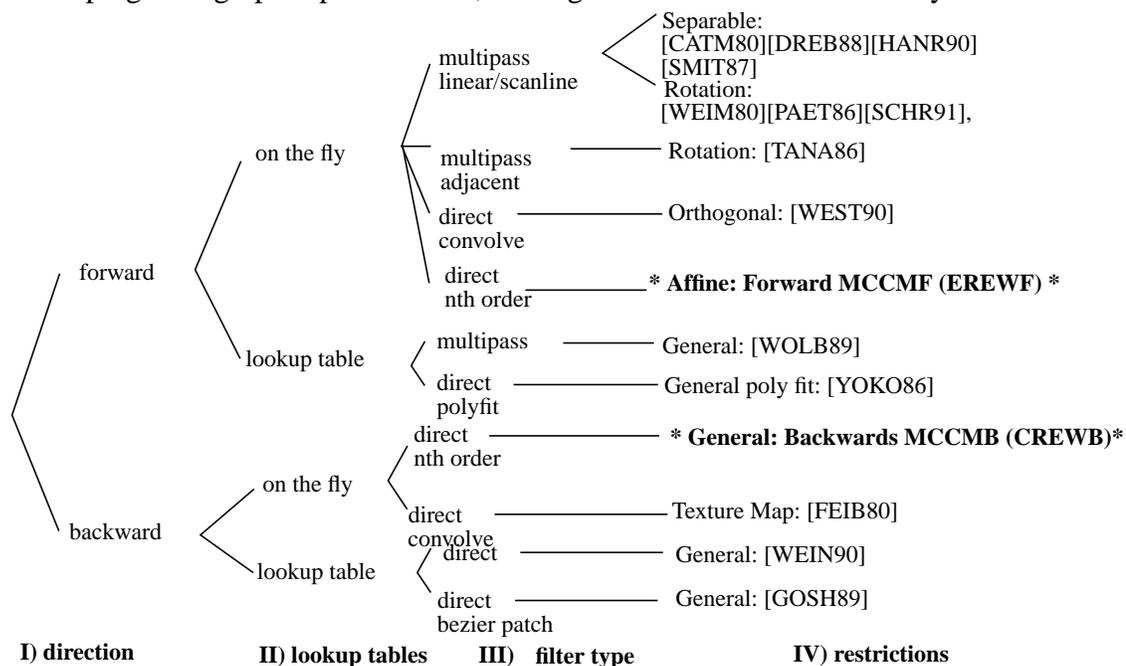


FIGURE 14

Image Warping Classification Tree, (*) with new algorithms: Backwards, Forwards, and Overlapped Forwards

The first factor, **the transform direction**, is the data flow of the program. An algorithm is forward mapping if data are passed by T to the output [CATM80][DREB88][HANR90][SMIT87][WEIM80][PAET86][SCHR91][TANA86][WEST90], or backward mapping if output locations are inverted by T^{-1} to the input where an appropriate value is reconstructed [FEIB80][GOSH89][WEIN90]. Forward or backward mapping is better depending on the size and organization of the input and output, the transform, and the filter.

The second factor, **lookup tables**, saves time by pre-calculating transform coordinates, filter coefficients, shading functions, and other values. A speed versus memory trade-off, as well as added programming complexity, are the key issues in using a lookup table. Image warping by lookup table is very general and efficient. A good description is in [WEIN90]. See also [WOLB89][WOLB90][GOSH89][YOKO86].

The third factor, **the filter type**, has the largest effect on efficiency, because more accurate reconstruction requires more calculation. The most common computational optimization for forward transforms is to decompose the transform, allowing regular scanline access and, more efficiency for some architectures. See the multipass branches in FIGURE 14. Multipass warps are applied to decomposable transforms including rotations [PAET86][SCHR91][TANA86][WEIM80], bicubic, and biquadratic warps [SMIT87]. But, multipass warps cannot perform higher order interpolation well.

The fourth factor, **transform restrictions**, allows optimizations such as coordinate calculations by differencing, efficient partitioning and job assignment. For example, restricting 2D transforms to rotation allowed researchers to optimize by decomposing into multiple passes of 2 or 3 matrices. A nonscaling sequence of shears [PAET86][SCHR91][TANA86] is,

$$T = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \sin\theta & 1 \end{bmatrix} \begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix}. \quad (\text{EQ 10})$$

A shear is a transform that operates on only one coordinate. Shears may scale (stretch or shrink axes) or not scale (distances are preserved). Nonscaling shears have 1's along their diagonals such as in (EQ 10). The scaling factor by which an affine transform changes area is the magnitude of the determinant of T [BARN88]. I use the decomposition of (EQ 10) in my forward warp algorithm's processor assignments of Section 3.6.2 and 3.6.3. Additional decompositions include the scaling shear sequences in (EQ 11) and (EQ 12).

[CATM80][PAET86]:

$$T = \begin{bmatrix} 1 & 0 \\ \tan\theta & \sec\theta \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta \\ 0 & 1 \end{bmatrix} \quad (\text{EQ 11})$$

[TANA86]:

$$= \begin{bmatrix} 1 & 0 \\ \tan\theta & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 \\ 0 & \sec\theta \end{bmatrix} \begin{bmatrix} 1 & -\tan\theta \\ 0 & 1 \end{bmatrix} \quad (\text{EQ 12})$$

I have derived a deterministic formula to calculate a three pass sequence of pure shears for both two dimensional and three dimensional transforms that are equiareal [MESE83]. Rotation turns out to be a special case of my symbolic decomposition in Section 3.6.4.

I show that direct warps are nearly as efficient as multipass warps, generalize to higher order filters, and have less aliasing (error). Direct parallel warps provide a trade-off of speed versus filter quality. In Section 3.6 I describe the straightforward approach for the CREW PRAM, and predict real machine cost by placing it on the MCCM of Chapter II. Then an EREW PRAM algorithm is presented in Section 3.6.2 that is optimal on the MCCM. Sections 3.6.3 through 3.8 detail algorithm variants and implementation issues. Performance measurements are presented in Section 3.9 and the chapter concludes in Section 3.10.

3.3 Warping Filters

I use polynomial interpolation (n^{th} order) filters which include the zero-order hold (zoh) and first-order hold (foh). The implementation and performance of these filters is widely discussed [JAIN89][PRESS88][FARI88]. I derive a polynomial interpolation algorithm of Aitken's using Neville's organization [FARI88][PRESS88], for calculating a low order piecewise polynomial fit. My algorithm is faster and more robust than the implementation given by [PRESS88]. I also show that the most efficient higher dimensional expressions of these filters is the tensor product approach.

When interpolating to a sequence (1D), image (2D), or surface (3D) several approaches can be taken. One approach is to fit a polynomial to the given points. A second approach is to create a system of control points that determines a spline or bezier surface that interpolates the points. A third approach is to create a least squares fit not attempting to exactly interpolate the points because there may be significant error in them. A polynomial fit avoids the overhead of creating control points and is reasonable for applications where there is high confidence in the data. Aitken's algorithm as presented by Farin [FARI88] is repeated linear interpolation to compute a higher order polynomial interpolation. The recursive evaluation is more efficient than the direct evaluation. A larger and larger neighborhood of points calculates a higher order polynomial fit. The 1D polynomial interpolation fits an order n polynomial to $n + 1$ points. Starting with Farin's expression [FARI88]¹,

1. page 61, Equation 6.2 in Chapter 6, with u replacing parameter t , l replacing r , and I replacing p because I work almost exclusively with intensities.

$$I_i^l(u) = \frac{u_{i+1}-u}{u_{i+1}-u_i} I_i^{l-1}(u) + \frac{u-u_i}{u_{i+1}-u_i} I_{i+1}^{l-1}(u); \quad \begin{cases} l = 1, \dots, n; \\ i = 0, \dots, n-l \end{cases} \quad (\text{EQ 13})$$

l is the level of recursion, n is the order of interpolation, and i is the index of interpolated points. $I_0^n(u)$ is the intensity at location u on the n^{th} order interpolation. For example a 3rd order filter requires 4 points. One may reorganize the calculation by transforming the interval $(u_{i+1}-u_i)$ to the unit interval by the substitution $u' = \frac{u-u_i}{u_{i+1}-u_i}$ [FARI88]². Then simply calculate

$$\begin{aligned} I_i^l(u) &= (1-u')I_i^{l-1}(u) + u'I_{i+1}^{l-1}(u); \\ &= I_i^{l-1}(u) + u'(I_{i+1}^{l-1}(u) - I_i^{l-1}(u)) \end{aligned} \quad (\text{EQ 14})$$

A further simplification if the point coordinate system has unit spacing ($u_{i+1}-u_i = 1$) is $u' = \frac{u-u_i}{l}$. As a final form calculate,

$$I_i^l(u) = I_i^{l-1}(u) + \frac{u-u_i}{l}(I_{i+1}^{l-1}(u) - I_i^{l-1}(u)); \quad \begin{cases} l = 1, \dots, n; \\ i = 0, \dots, n-l \end{cases}, \quad (\text{EQ 15})$$

as a concise optimized solution for the interpolated point. FIGURE 15 shows the pseudocode polynomial interpolation procedure *polyint*.

```

polyint(float ua[], float y[], int n, float u)
{
    int i,l;
    float I[N];

    for i = 0 to n
        I[i] = y[i] ;
    for l = 1 to n
        for i = 0 to n-l
            I[i] = I[i] +  $\frac{u-ua[i]}{l}$ (I[i+1] - I[i])
    return I[0] ;

```

FIGURE 15

n^{th} order polynomial interpolation by Neville's form of Aitken's algorithm

In FIGURE 15 ua is the array of parameter locations, y is the function values at those locations, n is the order of interpolation, and u is the parameter location at which to interpolate. If the coordinate system is not unit spacing replace l with $ua[i+1] - ua[i]$ as shown in (EQ 14). For multidimensional interpolation a de Casteljau approach (multidimensional recursion), while mathematically elegant, is less efficient than a tensor product

approach (operating on each dimension in turn). If each intermediate point is an evaluation, the quadratic 2D interpolation takes 13 evaluations by deCasteljau formulation, and 12 by tensor product. For bicubic deCasteljau takes 34 and tensor product 30. For 3D quadratic deCasteljau takes 45 and tensor product 39 evaluations. FIGURE 16 shows the pseudo code tensor product polynomial interpolation procedure polyint2d. Polyint (FIGURE 15) is the subroutine used for 1D interpolation.

```

polyint2d(float * Ix, float * Iy, float * I, int m, int n, int offset, float u, float v)
{
    int j;
    float Itmp[n+1];

    for(j=0;j<=n;j++){
        Itmp[j]= polyint(Ix,I,m,u);
        I=I+offset; /* wrap around to the next row of interp*/
    }
    return(polyint(Iy,Itmp,n,v); /* column to get final value*/
}

```

FIGURE 16 Tensor product 2D interpolation by Aitken's algorithm

In FIGURE 16 I_x and I_y are the parameter locations in the x and y directions, I is a pointer to a 2D array whose rows have offset number of elements. The order of interpolation is m in the x direction and n in the y direction. The parameter locations at which to interpolate are u in the x direction and v in the y direction. As a comparison of efficiency, I compared [PRESS88] interpolation routine, `polint`, that took 0.21 milliseconds per call while `polyint` (FIGURE 15) took 0.05 milliseconds per call a 320% improvement. I ran the 2D functions with the Numerical Recipes Example routine (`xpolin2.c`) [VETT88] on a Sun Sparc 2 using optimized gnu c compiled code. Because the iterations calculate the same points that [PRESS88] calculates, a crude error estimate may be provided as they do by giving the difference between the last point and the previous point ($\text{error estimate} = I[0] - I[1]$). 3D filters are developed analogously by using 2D for each slice and a final 1D interpolation of the remaining dimensions. In general, tensor product polynomial interpolation has

$$\# \text{int} = 1/2(n+1)[(n+1)^d - 1] \quad , \quad (\text{EQ 16})$$

interpolations for an n^{th} order filter in d dimensions.

As an example of filter quality a 4x4 array of points is interpolated using a zero order hold, a first order hold, a quadratic interpolation, and a cubic interpolation shown in FIGURE 17.

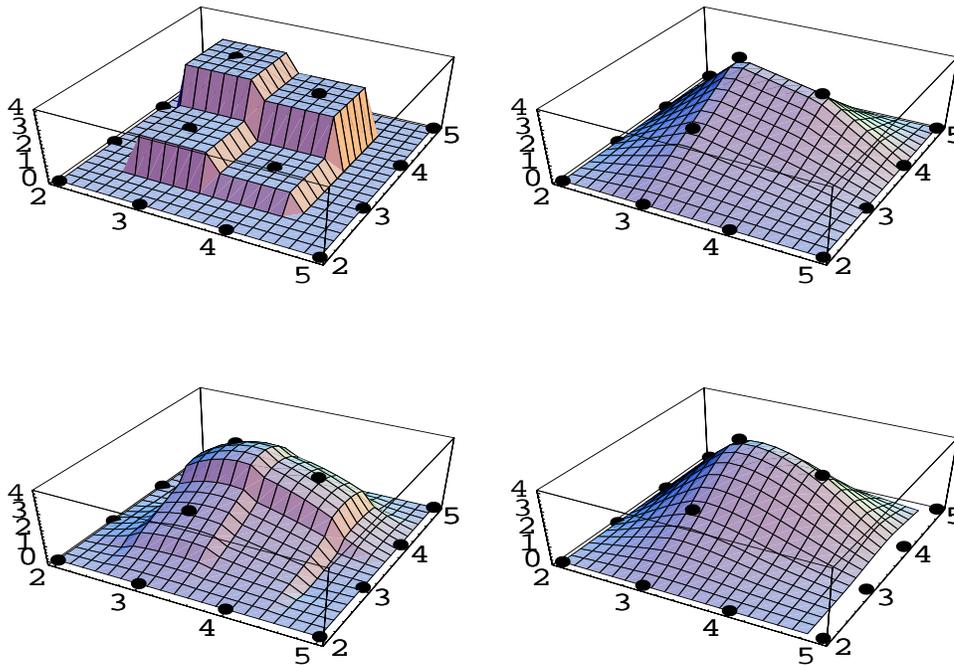


FIGURE 17

Filter Quality Comparison (upper left: zero order hold, upper right: first order hold, lower left: quadratic interpolation, lower right: cubic interpolation)

For parallel implementation, polynomial interpolation is calculated at each processor. Processors access a neighborhood of points whose size depends on the order of interpolation. One can look at linear interpolation as an affine combination of point intensities. The intensities I_1 and I_2 are combined according to their relative distances from each point. The relative distances are ratios u and $(1-u)$ which multiply I_1 and I_2 as shown in (EQ 17) and FIGURE 18.

$$I_a = (1-u)I_1 + uI_2 \quad (\text{EQ 17})$$

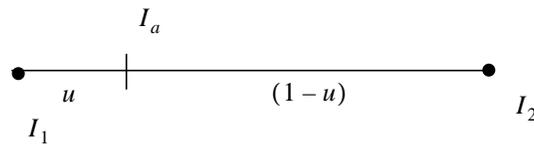


FIGURE 18

Linear interpolation As Affine Combination

As an example consider bilinear interpolation, which is a 2D first order hold equivalent to polyint2d in FIGURE 16 with $m = n = 1$. FIGURE 19, (EQ 18), and (EQ 19) show the arrangement of the points and intermediate interpolated values I_a and I_b . Here I interpolate in the x direction first getting I_a and I_b . Then I interpolate in y to get the final value $I(x, y)$. The interpolation is calculated on the unit interval as described earlier. In three dimensional reconstruction interpolation takes place in three orthogonal directions. For higher order reconstruction, more intermediate points are calculated to create a single point along each row of interpolation, and moving to the next orthogonal direction, the same approach is followed. The derivation provides for varying the filter quality with varying compute costs, and is simple and highly efficient for working with point sampled images.

$$I_a = I_1 + u(I_2 - I_1)$$

$$I_b = I_4 + u(I_3 - I_4) \quad (\text{EQ 18})$$

$$I(x, y) = I_b + v(I_a - I_b)$$

$$u = \frac{x - x_j}{x_{j+1} - x_j} \quad v = \frac{y - y_k}{y_{k+1} - y_k} \quad (\text{EQ 19})$$

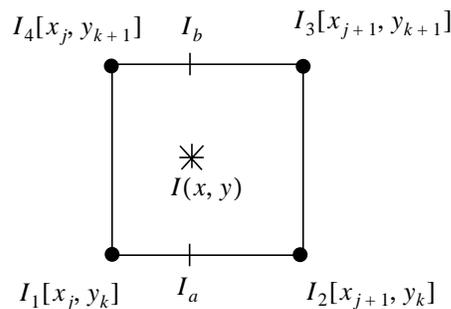


FIGURE 19

Bilinear interpolation done in horizontal direction first and then vertical direction

3.4 Error Derivation Of Filtering Approaches

Computer algorithms and their filter characteristics can be modeled by the discrete time system block in FIGURE 20 [DUDG84][JAIN89][CAST79]. The continuous to discrete (C/D) module digitizes images. Examples are scanners, computed tomography machines, and MRI scanners. The discrete to continuous (D/C) block is a computer display, printer, or film recorder, and reconstructs a continuous image that one views. Because the D/C block and one's visual system can filter out differences between images quantitative analysis is restricted to the discrete time system block.

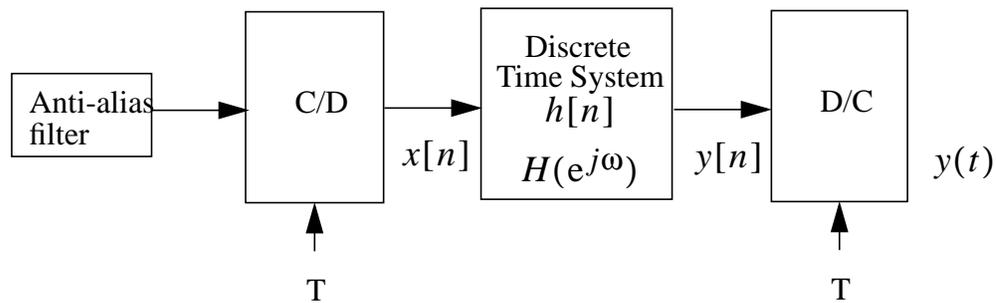
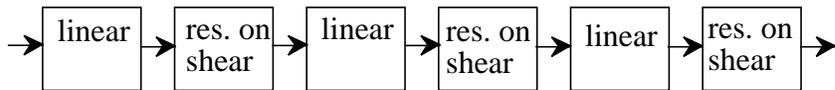
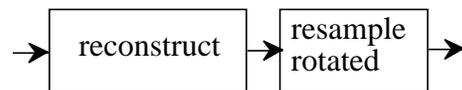


FIGURE 20 Complete Image Processing System



PAET86, SCH90, TANA86 Multipass Warp Rotation



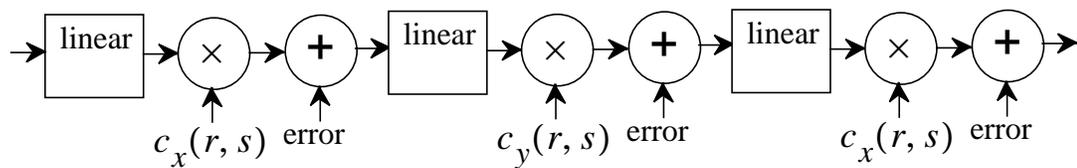
Forward and Backward Direct Warp

FIGURE 21 Block Diagram of Operations In 2D Warping Algorithm

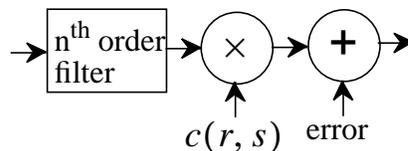
Direct warps are superior filters to multipass warps, because of less aliasing and less arithmetic error as shown in FIGURE 21. The multipass warp operation first reconstructs or interpolates the data in a scanline direction using a linear interpolation filter. Then the data is resampled on the shear coordinates. There are 3 passes of linear recon-

struction followed by resampling on shear coordinates, with the shear coordinates and the direction of linear reconstruction varying each time according to the angle and order of shears. The direct filtering approach makes one pass reconstruction and resamples once. Its block diagram system is also shown in FIGURE 21.

Ignoring aliasing, both filters are linear systems, but because the piecewise linear reconstruction spreads frequencies, Nyquist's criterion will be violated in resampling. This happens because the reconstruction filter spreads the frequencies to an infinite range. A linear system may be created by assuming the aliasing noise is additive error. If the error is exactly known, the linearized system and the nonlinear system are equivalent. FIGURE 22 gives a linearized view of the multipass and direct rotation.



PAET86, SCHR91, TANA86 Multipass Warp Rotation



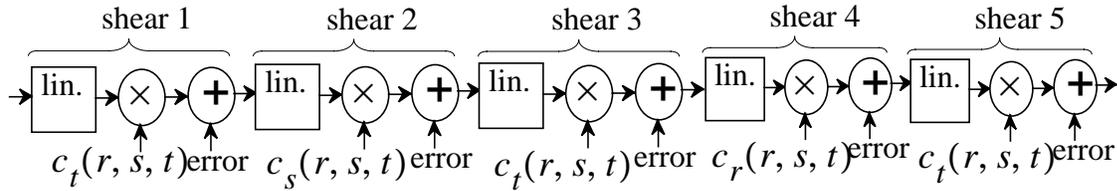
Wittenbrink Direct Warp (MCCMF or MCCMB)

FIGURE 22

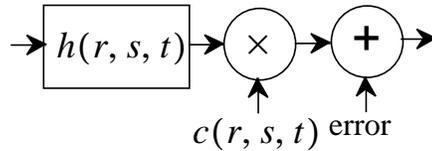
Linearized 2D Warp Systems

My derivation assumes that the original signal is bandlimited, and the first reconstruction avoids aliasing. By reconstructing a bandlimited image the direct warp filter does not alias, but additional amounts of noise/error will be introduced by the quantization effects and by the finite precision arithmetic. I will ignore quantization and precision errors

in analysis noting that the multipass warp will suffer more than direct warping. FIGURE 23 gives 3D filters are (assuming [SCHR90] reduced freedom rotation of 5 passes.)



[SCHR90] Multipass Warp Rotation



(n^{th} order hold)

Direct Warp (MCCMF or MCCMB)

FIGURE 23 3D Linearized Warp Systems

The direct warp algorithm has less aliasing for any n^{th} order hold. Assuming the initial data was bandlimited the error is computable by the energy in the sidelobes of the frequency response. The amount of error may be controlled by supersampling the original data or by prefiltering the images or volume to further restrict the bandwidth of the data.

Assuming the power spectral density of the ideal image is

$$\begin{aligned}
 W_{FI}(w) &= \sqrt{\left(\frac{w_s}{2}\right)^2 - w^2} & w^2 \leq \left(\frac{w_s}{2}\right)^2 \\
 &= 0 & w^2 > \left(\frac{w_s}{2}\right)^2
 \end{aligned}
 \tag{EQ 20}$$

[PRAT78] derived the interpolation and resolution error for the n^{th} order and gaussian filters given in TABLE 2. In (EQ 20) w_s is the sampling frequency. The sinc is an ideal low pass filter. The square is a zero order hold, the triangle is a first order hold and the bell is a second order hold. The cubic b-spline is a 3rd order hold, and all of these filters can be calculated by the *polyint2d* algorithm in the previous section (FIGURE 16). The multipass warp algorithm uses a triangle filter in three passes, and ignoring the effect of repeated aliasing the filter error is the sum of three passes of triangle filters. This is a conservative

estimate of the error because resampling after reconstruction in each pass severely aliases the data. The “% Interpolation” error places the multipass filter between the square (zoh) and the triangle (foh) which is supported by empirical filter comparisons in Chapter IV.

TABLE 2 2D Interpolation error and resolution error for separable interpolation functions (Reproduced from [PRAT78])

Function	% Resolution Error	% Interpolation Error
Sinc	0.0	0.0
Square	26.9	15.7
Triangle	44.0	3.7
Bell	55.4	1.1
Cubic B-spline	63.2	0.3
Gaussian $\sigma_w = 3T/8$	38.6	10.3
Gaussian $\sigma_w = T/2$	54.6	2.0
Gaussian $\sigma_w = 5T/8$	66.7	0.3
Multipass (3)	132.0	11.1

As interpolation error is reduced the resolution error grows. The amount of calculation required also goes up. Note that the amount of interpolation error in the direct filters is more (zoh) or less (foh, soh, etc.) than the multipass approach, and the resolution error of the multipass is greater than all of the direct filters.

3.5 Optimal RAM Image Warping Algorithm

The simplest algorithm to code is to iterate in SS and clip in OS, doing everything at the point granularity. FIGURE 24 shows a simple algorithm that does not use differencing, computes every point's transform, and clips to upright rectangles in OS.

```

J ← RAMB-Simple(I, BI, BJ, T-1) {
  1) for all p' ∈ BJ {
  2)   p = T-1(p')
  3)   If (p ∈ BI) /* Upright Rectangle Clip FIGURE 25*/
  4)     J[p'] = Reconstruction(p, I)
  5) Else
        J[p'] = Background
  }}

```

FIGURE 24 Simple to Code RAM Backwards Algorithm, $O(n^2)$,
 $T = (7M + 10A + 4\text{Comp.})n^2$, (RAMB-Simple)

Clipping is done by four comparisons of a point's coordinates to the bounding box of the image. I represent the bounding box as two points: the upper right point pur and the lower left point pll . FIGURE 25 shows code for clipping to an upright rectangle.

```

Clip(p, pll, pur) {
  1) inside = TRUE
  2) if  $p_x < pll_x$  inside = FALSE
  3)   else if  $p_x > pur_x$  inside = FALSE
  4)   else if  $p_y < pll_y$  inside = FALSE
  5)   else if  $p_y > pur_y$  inside = FALSE
  6) return (inside)}

```

FIGURE 25 Clipping To Upright Rectangle

The optimal sequential warping algorithm is a basis for comparing parallel algorithms. I argue that the following three observations can be used to determine an optimal 2D warping algorithm. The iteration can be performed at different granularity: point, scanlines, and polygons. And the clipping can be performed at different granularity: point, scanlines, and polygons. Additionally the clipping can be performed in OS or SS.

Observation 1: Clipping in object space is cheaper than clipping in screen space.

This observation holds, because screen space clipping requires clipping against hyperplanes which is more expensive than clipping against an upright rectangle.

Observation 2: Transforming points is more costly than differencing³ them.

This optimization is restricted, because differencing can only be done for affine transforms.

3. differencing points calculates transforms by using several transformed points and offsets for all other points.

Observation 3: Clipping by polygons is cheaper than clipping by lines which is cheaper than clipping by points.

Clipping becomes an insignificant part of the processing when clipping at higher granularity, because the overhead goes from clipping every point to clipping each scan-line, to clipping one bounding polygon of the entire image.

Therefore by Observations 1, 2, and 3 the best alternative is to clip in object space, use differencing for point calculations, and clip polygons. For parallel algorithms it depends on the granularity or amount of parallelism available. I summarize and the costs of clipping, transforms, and reconstruction below in TABLE 3 and TABLE 4.

TABLE 3

Sequential algorithm alternatives

	Clipping How	Clipping Where
	Screen Space	Object Space
Point	$n^2(\text{Rec.} + T + \text{HC})$	$n^2(\text{Rec.} + T + \text{UC})$
Line	$n^2(\text{Rec.} + T) + n\text{HLC}$	$n^2(\text{Rec.} + T) + n\text{ULC}$
Polygon	$n^2(\text{Rec.} + T) + \text{HPC}$	$n^2(\text{Rec.} + T) + \text{UPC}$

TABLE 4

Terms Used in Algorithm Alternatives Table

Term	Definition	Cost
Rec.	reconstruction	Bilinear is $3M + 6A$
T	transform	$4M + 4A$
HC	hyperplane clipping	$12M + 16A + 4\text{Comp.}$
UC	upright clipping	4Comp.
HLC	hyperplane line clip	
ULC	upright line clip	
HPC	hyperplane polygon clip	
UPC	upright polygon clip	

The difference in clipping cost is clipping each point n^2 , each line n , or the bounding box, 1. Line and polygon clipping require a similar amount of work in SS and OS but object space is slightly more efficient. For line and polygon clipping see [FOLE90] [MAIL92].

The optimal sequential algorithm computes all object space point coordinates by differencing, 2 additions, and the bilinear filter is calculated recursively with 3 multiplications and 6 additions ([CAST79]). FIGURE 26 gives the RAM backwards algorithm (RAMB). Higher order filters are calculated using polyint2d (FIGURE 16). If the trans-

form is not possible by using differences the algorithm will be slightly less efficient. Polygonal clipping precalculates the bounding boxes, without affecting the n^2 processing constants.

```

J ← RAMB(I, BI, BJ, T-1) {
1) BJ' = T-1(BJ) /* Inverse x-form the output image bounding box into the object space OS */
2) PJ' = Clip(BJ', BI) /* Clip BJ' to the box of input BI */
3) for all p' ∈ PJ' /* Compute by differencing to correspondingly, integer indexed p' */
    J[p'] = Reconstruction(p, I)
4) for all p' ∉ PJ'
    J[p'] = Background }

```

FIGURE 26 Optimal RAM Backwards Algorithm, $O(n^2)$,
 $T = (3M + 8A)n^2$, (RAMB)

An examination of the sequential complexity shows that the higher quality filters are available for a small cost. On real machines, if memory access is efficient then the RAM algorithms will accurately indicate the performance, and one should use the higher quality filter of the RAMB algorithm instead of a lower quality shear approach such as [PAET86] or [CATM80]. See TABLE 5 for the sequential algorithm comparisons.

TABLE 5 Algorithms Inner Loop Cost

	RAMB	RAMB, Rotation	[PAET86]	[CATM80]
Transform	General 4M, 4A	Differencing	3 Pass Shear	2 Pass Shear
Filter	Bilinear 3M, 6A	Bilinear 3M, 6A	Linear 3M, 6A	Linear 2M, 4A
Cost of Inner Loop	$(7M, 10A)n^2$	$(3M, 8A)n^2$	$(3M, 3A)n$ $(3M, 6A)n^2$	$(4M, 2A)n$ $(2M, 4A)n^2$
Quality of Filter	High	High	Lower	Lowest

3.6 Optimal PRAM Image Warping Algorithms

Optimal parallel algorithms were defined in Chapter II. I present here optimal parallel algorithms for the CREW and EREW PRAMs. They have time $O(1)$ using $P = S$ processors where either $S = n^2$ for an $n \times n$ image or $S = n^3$ for a volume of $n \times n \times n$. By

optimality Definitions 1 and 2 the algorithms are optimal. The EREW algorithm also exhibits optimal speedup on the MCCM, a network model with less power than the PRAM.

3.6.1 Optimal CREW PRAM Backwards Direct Warp Algorithm

For parallel image warping, one can partition the input object space (OS) for forward mapping or the output screen space (SS) for backward mapping approaches as discussed in Chapter II. The simpler algorithm is a SS backward mapping algorithm, called the CREW PRAM backwards algorithm (CREWB). The CREWB algorithm is the backward direct mapping branch in FIGURE 14. The algorithm is given below in FIGURE 27.

```

J ← CREWB(I, BI, BJ, T-1) {
1) for all output pixels (p') in J, p' ∈ BJ, where BJ is the bounding box of J,
   do {
2)   find the inverse transformed pixel, p = T-1(p') (Time = 4M + 4A )
3)   if p ∈ BI then (clip to the input image): (Time = 4Comp )
4)     J[p'] = Reconstruction(p, I) (Time = 3M + 6A + 2Rnd + 4GN )
5)     else set J[p'] to a background value.}

```

FIGURE 27

Backwards Algorithm (CREWB= Time = 23, MCCMB=
Time = 23 + 4GN for $d = 2$ and $n = 1$)

This algorithm uses any T where the inverse transform T^{-1} exists. For general inputs of T , numerical software can estimate the invertibility of T to caution the user if the transform is nearly singular. A singular matrix does not have an inverse. A condition number estimate could be calculated for the matrix [KAHA89], similar to what Matlab or Mathematica [WOLF88] uses for matrix calculations. The inverse is analytically solvable and typically stable for affine and orthogonal transforms without projections [FOLE90]. To compare filters the number of linear interpolations required was given in (EQ 13).

Assigning one processor per pixel, the filter complexity is $O(n^{d+1})$. If $n(n+1)^{d-1}$ processors per pixel are used the filter complexity is just $O(nd)$ where both a de Casteljau [FARI88] and a tensor product approach have nd steps. Using small order filters and more samples than processors there is one or fewer processors per sample. Assuming the PRAM has a processor per sample, the asymptotic complexity of the CREWB algorithm is $O(n^d)$, and $O(1)$ when n and d are small constants. The remaining constants are also small for this algorithm. If additions (A), multiplications (M), rounding (Rnd), and comparisons (Cmp) have the same cost a 2D image warping with a foh ($n = 1$) has 23 operations. FIGURE 27 shows the cost of each step. The complexity of the CREWB algorithm on the MCCM using a foh and $d = 2$ is Time = 23 + 4GN. N , the congestion, varies with the transform. The backwards algorithm with an n^{th} order filter uses,

$$\#pts = (n + 1)^d, \quad (\text{EQ 21})$$

points which requires $\text{Time} = \#ptsNG$ to fetch. The MCCM shows the realistic high run time, and if $(\#pts)$ is considered constant the run time complexity is $O(N)$ on the MCCM. In the next section I present a non-obvious EREW PRAM algorithm with congestion $N = 1$ for an optimal MCCM run time of $O(1)$. Because the MCCM is an abstract machine N , G , and L will vary for real machines.

3.6.2 Optimal EREW Forward Direct Warp Algorithm

The CREW PRAM backwards algorithm in FIGURE 27 is general. It can be used to warp an image using any invertible transform, but suffers from inefficiency on actual machines because of the concurrent reads illustrated by the MCCM costs. Restricting T results in a more efficient MCCM algorithm. In this section I present an optimal MCCM algorithm for nonscaling affine transforms. Optimal MCCM complexity is defined as communication efficiency as small as the diameter of the network. My algorithm is optimal on the MCCM, and requires exactly one global communication, or $1G$. This efficiency results from a clever processor assignment.

Processors choose output samples with a rule, then use local data to interpolate, and reorder with an efficient one-to-one global communication. The rule is an assignment of OS processors to SS samples. Reconstruction of SS samples is done in OS. Then the SS samples are sent to their proper locations. Processors and their points are π in OS and π' in SS. There is a duplicity of processor spaces, as processors are both in OS (π) and SS (π'). Obviously $\pi[x, y] = \pi'[x, y]$, but they are differentiated to more clearly describe the assignments. Processor π chooses a processor to work for by the nonlinear mapping or rule, $M: \pi \rightarrow \pi'$. For example in 2D rotation M is

$$M = \text{round}(\text{SH}_x \text{round}(\text{SH}_y \text{round}(\text{SH}_x \pi))) \quad , \quad (\text{EQ 22})$$

where the first shear to be applied is (from (EQ 10)),

$$\text{SH}_x \pi = \begin{bmatrix} 1 & -\tan \theta / 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \pi_x \\ \pi_y \end{bmatrix} = \begin{bmatrix} \pi_x - \pi_y \tan \theta / 2 \\ \pi_y \end{bmatrix} . \quad (\text{EQ 23})$$

SH_x denotes a shear in the x direction. Further details on the processor mapping M are in Section 3.6.3. FIGURE 28 shows π in object space (OS) and π' in the screen space (SS). After choosing the processor to work for, processor π calculates processor π' 's inversed

point position by mapping between spaces, $p_{\pi'} = T^{-1}(\pi')$. Point $p_{\pi'}$ and mapping T^{-1} are also shown in FIGURE 28.

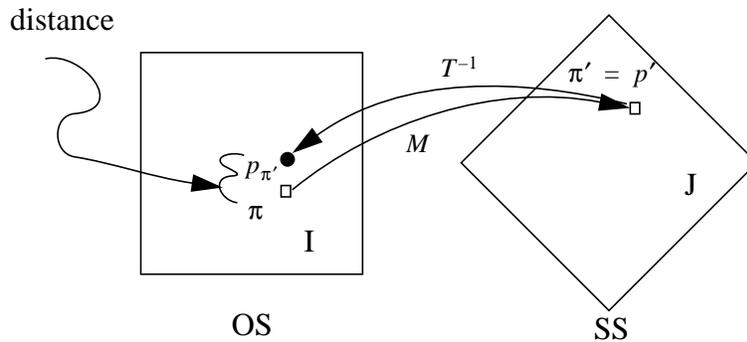


FIGURE 28 Nonlinear Mapping

Now, by using point $p_{\pi'}$'s coordinates, processor π retrieves the neighboring intensities of point $p_{\pi'}$, and interpolates them. For example in 2D bilinear interpolation the four points surrounding $p_{\pi'}$ are used in (EQ 15) with *polyint2d* of FIGURE 16. The points surrounding $p_{\pi'}$ are guaranteed by my rule to be near neighbors in the mesh as shown in FIGURE 29. For higher order reconstruction a larger neighborhood is used. To insure exclusive reads the neighboring values are read in directional phases to avoid conflicts, such as (1) north, (2) northwest, (3) west, etc.

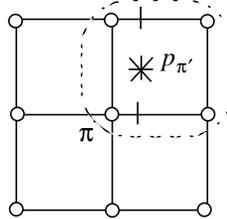


FIGURE 29 Near Neighbors In Mesh

The reconstruction creates $J[p_{\pi'}]$ that lies at processor π . In a final step, π sends $J[p_{\pi'}]$ to π' by a global send. Every processor π is given a unique π' to work for and the global send is a one-to-one send. This permutation of writes guarantees the exclusive write property of the algorithm. This algorithm is called the forward algorithm and is summarized in FIGURE 30. Theorem 3.1 proves one-to-one properties of M and the neighborhood properties of $T^{-1}M$ in Section 3.6.3.

```

J ← EREWF(I, BI, BJ, T-1, M) {
1) Each OS processor calculates output processor to work for
   If  $\pi \in B_I$  {  $\pi' = \text{rule}(\pi) = M\pi$  (Time = 3Rnds + 3M + 3A )
   2) If  $\pi'$  lies within processor array, or clip to output image,
      If  $\pi' \in B_J$  (Time = 4Comp) {
      3) Calculate the inversed location of the output pixel/processor in the input
          $p = T^{-1}(\pi')$  (Time = 4M + 2A )
      4) Get local neighbors reading in directional phases to avoid conflicts
         and reconstruct  $J[p'] = J[\pi']$  reconstruction(neighbors( $\pi$ ), p)
         (by bilinear for 2D Time = 3L + 3M + 6A + 2Rnd )
      5) Processor  $\pi$  sends the  $J[\pi']$  pixel value to processor  $\pi'$  (Time = 1G)}}}

```

FIGURE 30 Forward Algorithm (EREFW= 30, MCCMF=
Time = (30 + G + 3L))

A 512x512 image rotated by 35° and 45° is shown in FIGURE 31, and corresponding π to π' assignments for a 9x9 mesh are shown in FIGURE 32. The image is rotated clockwise to the grey image orientation. Black dots indicate π' and a line is drawn from π to π' . Clear dots with no lines from them are processors where $\pi = \pi'$. Gray dots are processors who have been clipped.

FIGURE 31 512x512 35 and 45 image rotation performed on the MasPar MP-1.

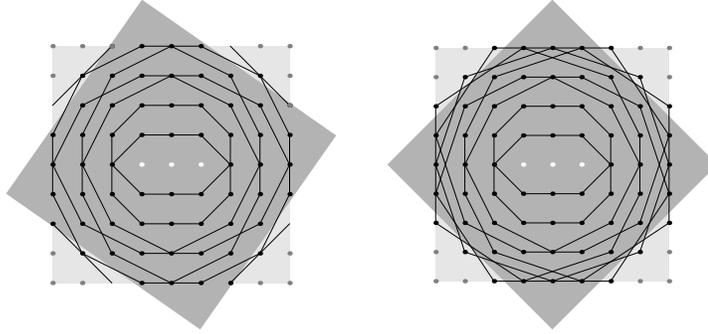


FIGURE 32 Processor assignments in a 9x9 mesh to calculate 35 (left) and 45 (right) rotation

The MCCM complexity for the forward algorithm shows slightly more calculations but optimal communication. For example, rotation complexity is given in FIGURE 30 for the 2D foh after each step. Assuming each operation (M , A , Rnd , $Comp$) is the same cost, the time is $Time = (30 + G + 3L)$ where G and L are the communication costs described earlier. The forward algorithm is better if $30 + G + 3L < 23 + 4GN$, because global communication is expensive, and/or there is congestion. The local reconstruction neighborhood is enlarged without changing the global $1G$ cost. The overhead is the rule calculation to assign processors. The forward algorithm is better if the following inequality holds,

$$(RULE) < \#pts(NG - L) - (G - L) \quad . \quad (EQ 24)$$

$RULE$ is the cost to evaluate the mapping (M). The transform cost and reconstruction costs are the same, and ($\#pts$) is the number of points used in the reconstruction. This inequality proves to be true for parallel machines with up to a 59% improvement exhibited on the MasPar MP-1 (Section 3.9) for 2D images with foh and a 100% improvement with 3D using a foh.

The forward algorithm is $O(1)$ on the MCCM, and therefore the forward image warping algorithm is optimal.

3.6.3 Nonlinear Mapping Rules For Forward Algorithms

I prove that nonscaling affine transforms are calculable by the forward algorithm. Transforms include translation, shearing, and rotation, (EQ 22). For translation, I round to integer coordinates by mapping $M:\pi \rightarrow \pi'$, $\pi'_x = \text{round}(\pi_x + T_x)$, $\pi'_y = \text{round}(\pi_y + T_y)$, $\pi'_z = \text{round}(\pi_z + T_z)$ etc. The proof that translation and rounding gives a one-to-one assignment is Lemma 3.1.

Lemma 3.1: Arbitrary translation and rounding is one-to-one.

Proof: Each whole number grid location is unique before the mapping, so two points p and q have coordinates $p_x \neq q_x$. If each point is translated by the same amount, then the integral and fractional amount of both translations is the same. Therefore assignments remain unique after the translation and rounding. Higher dimensional translations are proven by induction using the same argument for each dimension. ■

Next I show that equiareal (including nonscaling affine) warps allow a one-to-one nonlinear processor assignment, and further, using this mapping insures that filtering takes place in local neighborhoods.

Theorem 3.1: Equiareal warps T , $|\det(T)| = \pm 1$, can be decomposed into pure shears, and shearing followed by rounding is both one-to-one and results in a point whose inversed position is always within distance 1, in orthogonal directions.

Proof: There are 2 points to prove the correctness (1) that the mapping M is one to one, and (2) neighbors of $p_{\pi'}$ are near neighbors of π . Point (1): Nonscaling affine maps are decomposable into a sequence of translations, shears, and rotations, because they are invertible. Translations are one-to-one by Lemma 3.1. A pure shear (no scaling) translates a single coordinate. Each row (column/slice) is translated by a fixed amount, and therefore a shear and round is one-to-one transform by Lemma 3.1 also. Rotation is decomposable into a multistep shear, and because each shear is one-to-one rotation is one-to-one.

$$(M:\pi[x, y] \rightarrow \pi'[r, s]) \Rightarrow (\pi[x, y] \xrightarrow{M_1} \pi_1[x_1, y_1] \xrightarrow{M_2} \pi_2[x_2, y_2] \xrightarrow{M_3} \pi'[r, s])$$

$(M_i \quad i \in 1, 2, 3)$ is one-to-one $\Rightarrow M$ is one-to-one.

Point (2): The interpolation point is within the local neighborhood of the processor if $|(p_{\pi'})_x - \pi_x| < 1$ and $|(p_{\pi'})_y - \pi_y| < 1$. OS processor coordinates pass through two maps ($T^{-1}M$) to arrive at the interpolation point (see FIGURE 28). Translation is trivially within distance one because the rule rounds to the nearest point. For rotation calculate the coordinates of the interpolation point in terms of the OS coordinates, the rotation angle, and the rounding errors. The distances between the processor and the point calculated in the x and y directions are nonlinear trigonometric equations. The distance for any angle is less than 1 for up to around 100° , as calculated by,

$$(p_{\pi'})_x - \pi_x = \left(\pm \tan \frac{\theta}{2} \text{mod} 0.5 \right) (\cos \theta + 1) + (\pm \sin \theta \text{mod} 0.5) (1 - 2 \cos \theta) \quad (\text{EQ 25})$$

$$(p_{\pi'})_y - \pi_y = \left(\pm \tan \frac{\theta}{2} \text{mod} 0.5 \right) (\cos^2 \theta - \sin^2 \theta + \cos \theta) + (\pm \sin \theta \text{mod} 0.5) \left(\sin \theta + \tan \frac{\theta}{2} \cos \theta \right) \quad (\text{EQ 26})$$

A series of translations, shears, reflections, and rotations achieves all equiareal maps, therefore the map M is one-to-one, and the distance properties of $T^{-1}M$ do not violate the neighborhood property for all equiareal maps. ■

Because of the complicated nature of the distance functions, a plot of distance helps to convince. The functions are plotted from 0 to 180°. Two values of the x distance (with different signs of second term), and the y distance are superimposed in FIGURE 33. This shows that the distance is strictly less than 1 for angles up to about 100°. The functions explode at 180° because of the singularity in $\tan\theta/2$. Decomposition using $\cot\theta/2$ is stable from $\pi/2$ to $3\pi/2$ complementing the tangent. A reflection about both axes results in a decomposition with $\cot\theta/2$.

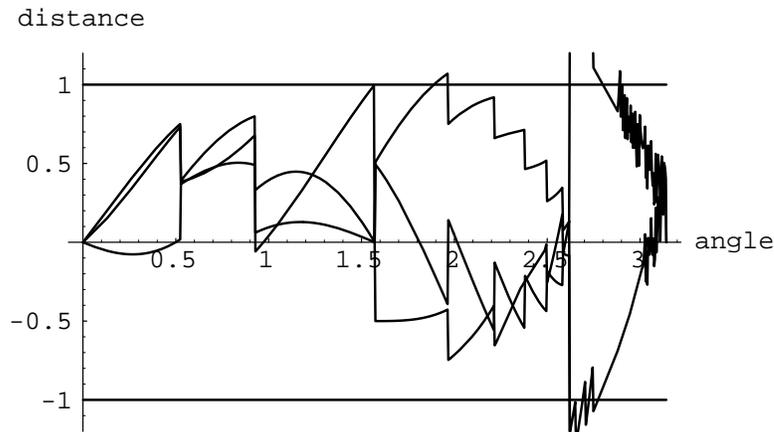


FIGURE 33

Distance of Interpolation Point in x and y .

3.6.4 Sequences of Nonscaling Transforms

The forwards algorithm can do any sequence of translations, rotations, or pure shears. The rule is optimized by combining transforms where possible. For example in arbitrary centering of rotation the first two and last two matrices may be combined. The transform to rotate about the point (r_x, r_y) and center the rotation in the output about (c_x, c_y) is

$$T = T(c_x, c_y)R(\theta)T(-r_x, -r_y) \quad (\text{EQ 27})$$

Translations are $T(t_x, t_y)$, and the rotation is $R(\theta)$. To use the transform for a mapping M , round after each pass denoted by \bar{T} with the bar above the transform pass. The transform after replacing $R(\theta)$ by the pure shear matrices is $M = \bar{T}(c_x, c_y)\bar{S}H_y\bar{S}H_x\bar{S}H_y\bar{T}(-r_x, -r_y)$. I premultiply the first translation into the first shear $\bar{S}H_y$ and the final translation into the last shear $\bar{S}H_x$. Recall that a point is postmultiplied with the transform matrix so the first transform is $\bar{T}(-r_x, -r_y)$ [FOLE90]. The mapping becomes $M = \bar{M}_3\bar{S}H_x\bar{M}_1$. This gives a three pass rule to calculate the processor mapping. Transform \bar{M}_1 is the rounding of values produced by

$$M_1 = SH_y T(-r_x, -r_y) = \begin{bmatrix} 1 & 0 & 0 \\ -\alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -r_x \\ 0 & 1 & -r_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -r_x \\ -\alpha & 1 & \alpha r_x - r_y \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{EQ 28})$$

and similarly,

$$M_3 = \begin{bmatrix} 1 & 0 & c_x \\ -\alpha & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix}. \quad (\text{EQ 29})$$

Each of these matrices is sparse so many terms (0 or 1) do not require multiplication. My example of arbitrary centered rotation is calculated by,

$$\pi' = \text{round}(M_3 \text{round}(SH_x \text{round}(M_1 \pi))) \quad (\text{EQ 30})$$

Concatenation of rotations, translations, shears, and scalings is possible with one-to-one processor assignments, and therefore optimal communication. Scalings do not have a one-to-one assignment, but if the scaling is decomposed from the general transform then the remaining transforms are optimal. All of the manipulation can be done symbolically. I have also derived a decomposition for any two dimensional or there dimensional equiareal transform.

The general solution to a 2 dimensional equiareal transform is calculated by solving a system of 5 equations with 3 unknowns. The unknowns are the coefficients in the 3 pass shearing operation. An equiareal transformation by definition has

$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = a_{11}a_{22} - a_{12}a_{21} = \pm 1. \quad (\text{EQ 31})$$

The other four equations are found by setting

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & b_1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ b_2 & 1 \end{bmatrix} \begin{bmatrix} 1 & b_3 \\ 0 & 1 \end{bmatrix} \quad (\text{EQ 32})$$

The result is $b_1 = (a_{11} - 1)/a_{21}$, $b_2 = a_{21}$, and $b_3 = (a_{22} - 1)/a_{21} = (a_{12}a_{21} - a_{11} + 1)/(a_{11}a_{21})$. A special case is rotation, where

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \quad (\text{EQ 33})$$

and $b_1 = (\cos\theta - 1)/(\sin\theta) = -\tan\theta/2$ by insertion and reduction by the half angle formula, $b_2 = \sin\theta$, and $b_3 = (\cos\theta - 1)/(\sin\theta) = b_1$. This shows how to calculate the result given by [PAET86] [TANA86].

3.6.5 Optimal MCCM 3D Equiareal Algorithm

The beauty of the forward algorithm is that in 3D there is still only 1 global communication instead of (#pts) or more global communications by the backwards method (8 for trilinear interpolation, 27 for soh, and 256 for toh). With the first order hold there are 8 corners in a cube whose intensities are trilinearly interpolated. By using local neighbors for the interpolation step the forward algorithm does 7 local MCCM communications. The rotation algorithm is the same as given in FIGURE 30, where the constants differ because of 3D transforms, fetching of additional local neighbors and 3D reconstruction.

The same decomposition approach used for 2D is used for 3D, and there are 10 equations with 9 unknowns,

$$\det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \pm 1 \quad (\text{EQ 34})$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & b_{12} & b_{13} \\ 0 & 1 & b_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ c_{21} & 1 & 0 \\ c_{31} & c_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & d_{12} & d_{13} \\ 0 & 1 & d_{23} \\ 0 & 0 & 1 \end{bmatrix}. \quad (\text{EQ 35})$$

The solution from Mathematica(TM) is,

$$\begin{aligned} c_{31} &= a_{31}, & c_{32} &= \frac{a_{31} - a_{22}a_{31} + a_{21}a_{32}}{c_{21}}, & d_{12} &= \frac{a_{22} - 1}{c_{21}} + \frac{a_{32}}{a_{31}} - \frac{a_{21}a_{32}}{c_{21}a_{31}}, \\ d_{23} &= \frac{c_{21} + a_{23}a_{31} - a_{21}a_{33}}{a_{22}a_{31} - a_{21}a_{32}}, & d_{13} &= \frac{a_{23}}{c_{21}} - \frac{a_{23}a_{31} + a_{21}a_{33} - c_{21}}{c_{21}(a_{22}a_{31} - a_{21}a_{32})} + \frac{a_{33}}{a_{31}} - \frac{a_{21}a_{33}}{c_{21}a_{31}}, & b_{23} &= \frac{a_{21} - c_{21}}{a_{31}}, \\ b_{13} &= \frac{a_{11}}{a_{31}} - \frac{c_{21}a_{11}a_{32} + c_{21}a_{31}a_{12} - a_{31}}{a_{31}(a_{22}a_{31} - a_{21}a_{32})}, & b_{12} &= -\frac{1}{c_{21}} + \frac{a_{31} + c_{21}(a_{12}a_{31} - a_{11}a_{32})}{c_{21}(a_{22}a_{31} - a_{21}a_{32})}. \end{aligned} \quad (\text{EQ 36})$$

This allows direct solution for a three pass nonscaling transform, which I use in the rule calculation. It could also be used for a multipass warping provided the data could move in 2 directions operating on “scanframes” if you will.

Most viewing transforms are rigid body transformations. I show how arbitrary rotation, and then arbitrary translation and rotation are decomposed into \bar{M} matrices. Reflection can be easily added. (EQ 37) shows 3D rotation as a concatenation of rotation about each axis x , y , and z [FOLE90] p. 215.

$$R_z(\psi)R_y(\phi)R_x(\theta) = \begin{bmatrix} \cos \psi & -\sin \psi & 0 \\ \sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \phi & 0 & \sin \phi \\ 0 & 1 & 0 \\ -\sin \phi & 0 & \cos \phi \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}. \quad (\text{EQ 37})$$

This transformation is decomposed into pure shears. (EQ 38) gives a decomposition of $R_x(\theta)$, or rotation about x by θ , into pure shear matrices. Rotation about y and z are done likewise with the 2D decomposition developed by [PAET86] and [TANA86], and 9 matrices result.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -\tan\theta/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & \sin\theta & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -\tan\theta/2 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{EQ 38})$$

You can use all 8 shears, and also concatenate prior translations to provide arbitrary centering and flybys, or you can use just 6 shears from the decomposition in (EQ 35) and (EQ 36).

Note that the decomposed matrices are not used for a multipass resampling, only to calculate the permutation, M . The order of the transformations does not result in poor filtering or the bottleneck problem [SMIT87]. Any angle of rotation, or translation can be performed in one pass without transposing the data. Different decompositions are used for rotation because of the discontinuity in $\tan\theta/2$. In fact 90 degree rotations are done in one pass, by a permutation of unsampled voxel values. After each shear operation the point coordinate being operated upon is rounded to an integer coordinate maintaining the one-to-one assignment. The operation for the right most matrix in (EQ 38) results in $\bar{M}_\eta = \text{Round}(y - z \tan\theta/2)$. Because only one coordinate is affected, and no scaling is used, rounding chooses a unique coordinate.

The inverse used in determining the reconstruction point is numerically stable. In fact equiareal transformations are by definition invertible. For arbitrary centered rotation the transform is a product of translation matrices, $T(x, y, z)$, and the rotation matrix, $R(\psi, \phi, \theta)$. Rotate about the point (r_x, r_y, r_z) and center the rotation in the output about (c_x, c_y, c_z) . The transformation given in (EQ 39) is decomposed and contracted into operations on single coordinates, and used to calculate M .

$$T = T(c_x, c_y, c_z)R(\psi, \phi, \theta)T(-r_x, -r_y, -r_z) \quad (\text{EQ 39})$$

For arbitrary centered rotations the inverse T^{-1} is easily calculated because rotation $R(\psi, \phi, \theta)$ is orthogonal, meaning $T^{-1} = T^T$, and translations are inverted by negating their values,

$$\begin{aligned} T^{-1} &= (T(c_x, c_y, c_z)R(\psi, \phi, \theta)T(-r_x, -r_y, -r_z))^{-1} \\ &= (T(-r_x, -r_y, -r_z))^{-1}(R(\psi, \phi, \theta))^{-1}(T(c_x, c_y, c_z))^{-1} \quad (\text{EQ 40}) \\ &= T(r_x, r_y, r_z)(R(\psi, \phi, \theta))^T(T(-c_x, -c_y, -c_z)) \end{aligned}$$

The rotation matrix and a translation matrix are given in (EQ 41) and (EQ 42), and the transpose of (EQ 41) is composed with the translations for calculating the inverse with the minimum number of calculations.

$$R(\psi, \phi, \theta) = \begin{bmatrix} (\cos \phi \cos \psi) & (-\cos \theta \sin \psi + \cos \psi \sin \phi \sin \theta) & (\cos \psi \cos \theta \sin \phi + \sin \psi \sin \theta) & 0 \\ (\cos \phi \sin \psi) & (\cos \psi \cos \theta + \sin \phi \sin \psi \sin \theta) & \cos \theta \sin \phi \sin \psi - \cos \psi \sin \theta & 0 \\ (-\sin \phi) & (\cos \phi \sin \theta) & (\cos \phi \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{EQ 41})$$

$$T(c_x, c_y, c_z) = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{EQ 42})$$

In FIGURE 34 are 2 volume rotations showing the π to π' processor assignments with π' covered by a black dot and connected to π by a line. The cube is rotated by 25° about y $25/2^\circ$ about x (left) and a 35° about y $35/2^\circ$ about x (right) with z axis up, y to the left, and x to the right.

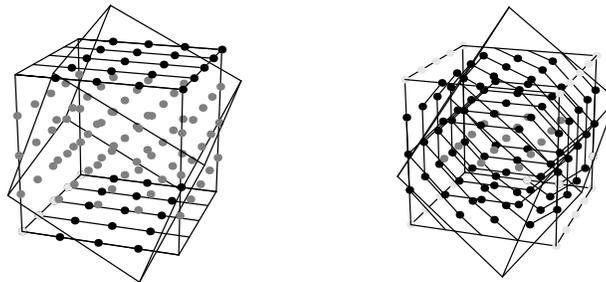


FIGURE 34

Processor assignments in a 5x5x5 volume to calculate $25/2, 25, 0$ and $35/2, 35, 0$ (x,y,z) rotations

3.6.6 Comparison to Previous 3D Techniques

There are several previous parallel 3D warping techniques [SCHR91][DREB88][HANR90]. Schroeder and Salem use a multipass warp with 5 scale shear passes on the Thinking Machines CM-2 [SCHR91]. By restricting the 3D rotation to two axes of freedom and combining two adjacent direction shears only 5 shears are needed. Neighboring voxel data locations are calculated through look up tables and resampling occurs in each

pass. Because of multiple resampling steps more error is introduced, and the angles of rotation are limited to 0-45 degrees to limit error. My algorithms do not have this limitation.

In TABLE 6 I have calculated roughly the MCCM complexity of Schroeder and Salem's algorithm and my algorithm with the restricted two axes rotation. Clipping was ignored in TABLE 6 because Schroeder does not discuss it. The complexity of the algorithms is essentially the same. Schroeder and Salem's multipass warp has $24 + 1G \cong 18 + 1G$ or $29 + 1G$ of my direct warp. See rows one, two, and three of TABLE 6. My algorithm uses more accurate filters with greater cost or less accurate filters for less cost, showing the added complexity is essentially for the more accurate filters. The zoh has the least cost and is simpler to program. Next in run time complexity is Schroeder and Salem's algorithm, but very nearly the same run time is the more accurate foh filter using the MCCMF algorithm. The MCCMB algorithm is penalized by the congestion N , but is the simplest to implement. The forwards algorithm is slightly more complicated, but is the most efficient with general filters.

TABLE 6

Performance Constants for Algorithms and filters with restricted rotations

	Filter	2D	3D
[SCHR91]	Multipass linear	24+1G	40+1G
MCCMF	zero order hold	18+1G	34+1G
	first order hold	29+1G	61+1G
	second order hold	61+1G	176+1G
	third order hold	122+1G	666+1G
MCCMB	zero order hold	10+1NG	18+1NG
	first order hold	19+4NG	39+8NG
	second order hold	46+9NG	137+27NG
	third order hold	100+16NG	436+256NG

Drebin et al.'s [DREB88] techniques have been generalized in Hanrahan's [HANR90] three pass decomposition for 3D affine transforms, using scale shear matrices. This is an extension of Smith's [SMIT87] 2D approach. Vezina et al. [VEZI92] have implemented Hanrahan's methods on the MasPar. This approach suffers from the same problems as Schroeder and Salem's, multistep filter error and more work than directly resampling. But Hanrahan's method covers a larger number of warpings. Nonscaling warpings are important, though, because combined with scaling algorithms they can achieve more general transforms more efficiently.

3.7 Scaling and Perspective

To perform 3D perspective and scaling I use spreads to achieve optimal MCCM algorithms. See FIGURE 35 below. The viewing frustum delineates the edges of a new volume. Distortion of the volume is scaling in each column. The perspective view rays are distorted to an orthogonal view volume. The orthogonal view volume is a highly efficient distribution of data for z-buffering, max intensity, or compositing calculations performed through parallel product evaluation (For parallel product and prefix see [LEIG92][GIBB88][CORM90][KRUS85] and Chapter IV.

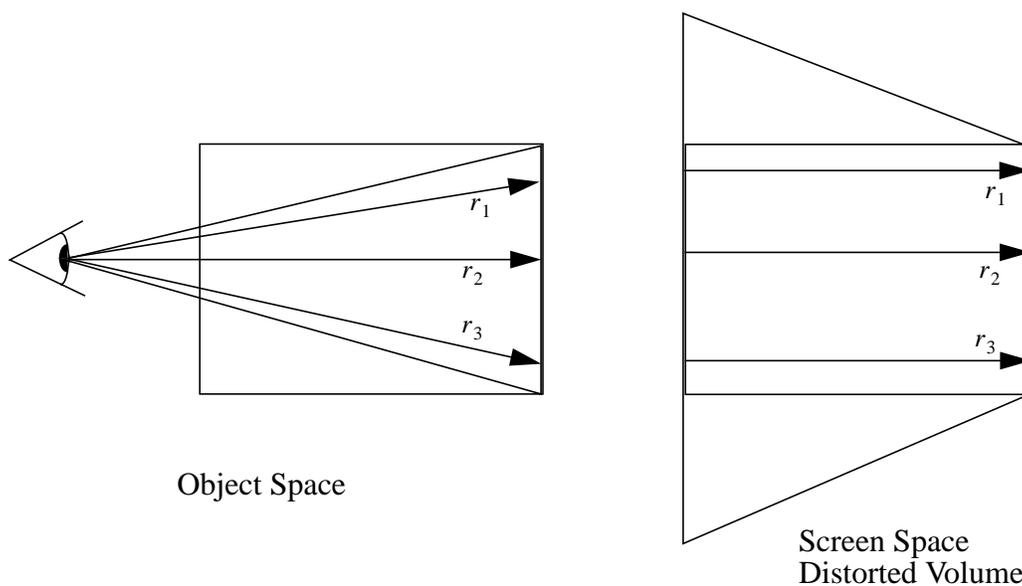


FIGURE 35

3D Perspective Volume Distortion

In each column a spread communicates after which processors reconstruct. Consider expansion. Given PE's π_a through π_f , the data I_1 , I_2 , and I_3 are scaled up, or expand-

ed. Data values I_1 and I_2 determine the intensities for all PE's. There are two ways the outputs are calculated.

$$\begin{array}{ccccccc}
 \pi_a & \pi_b & \pi_c & \pi_d & \pi_e & \pi_f & \\
 I_1 & I_2 & I_3 & \downarrow & & & \text{scaled up} \\
 \pi_a & \pi_b & \pi_c & \pi_d & \pi_e & \pi_f & \\
 I_1 & & & & & & I_2
 \end{array}$$

FIGURE 36 Scaling Of Data

1) Processors get this data directly and interpolate or 2) processors get the data from someone else and interpolate for someone else. There is a continuum of load balancing with varying performance as shown in FIGURE 37. When the data is interpolated locally the PE's have a lot of work. When all of the data is first sent, then the PE's will incur a large communication overhead.

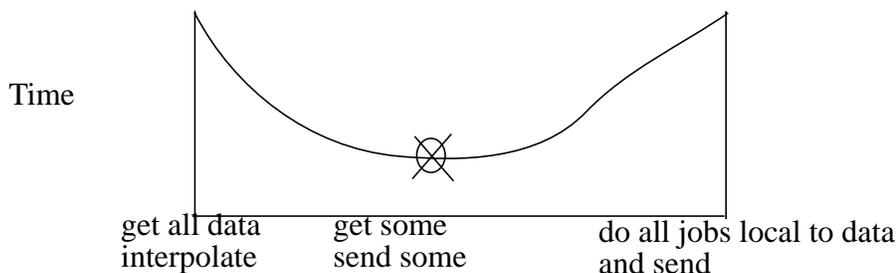


FIGURE 37 Trade-off curve of trading jobs versus communication

The optimal approach lies within these two extremes. Processors can get the data to be interpolated on the MCCM mesh by either sending on local connections or through the interconnection network (ICN). The use of a mixture of the two can also be advantageous, and a decomposition of the communication into ICN and local sends is the optimal approach.

The processor job assignment, or who sends, interpolates, receives, etc. is determined by a choice of the optimal decomposition of the communication patterns. Adjacent columns of data communicate their values to allow interpolation. For the extreme expansion of data in FIGURE 36 the data is sent along the mesh interconnections by a spread operation. This broadcast is efficiently built into the MasPar [BLAN90] and the CM-2

[THIN89]. The spread occurs in an aligned dimension of the multidimensional mesh of the MCCM.

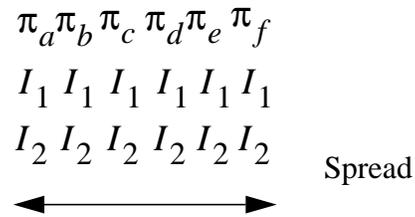


FIGURE 38 Spreading To Distribute Data

Job assignment is done efficiently on algorithms on machines with fewer physical processors than virtual processors. $MCCM_{SIMD}$ takes advantage of the coherence of each region's scaling and matches processors communication and computation in a tiled coverage of the total job. FIGURE 39 shows tiled trapezoidal distortion with vertical tiles insuring that communication and scaling are similar in a widely varying format. This load balancing allows SIMD to achieve good performance with diverse requirements.

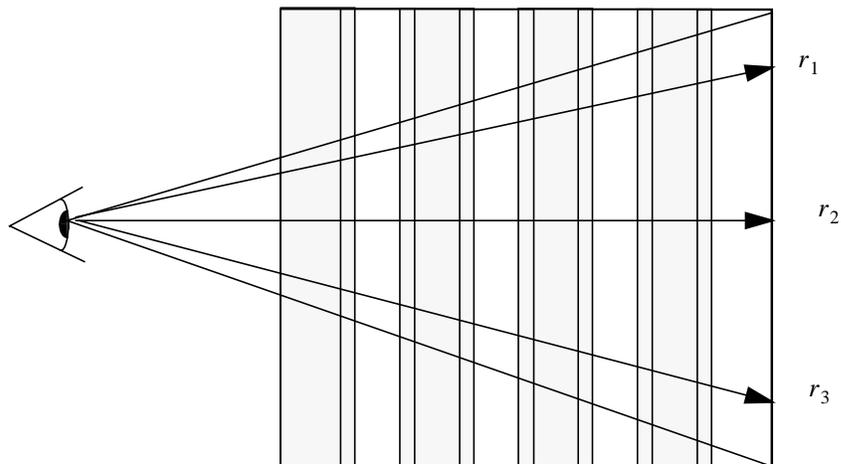


FIGURE 39 Striped Allocation of Volume Warping Jobs

3.8 Virtualization

I have found that virtualization significantly affects performance. Virtualization is running an algorithm written for v processors on v processors. FIGURE 40 and FIGURE 41 show space subdivision for virtualization. In the forward transform algorithm, local communication is removed by overlapping boundaries of the virtual subimages stored on each processor. FIGURE 40 shows 2D virtualization with data in one processor highlighted. Only the

global communication is performed. This is the EREW PRAM algorithm, but because of the virtualization the final global send is no longer 1-to-1. I call this algorithm the MCCM-SIMD overlapped forward algorithm ($MCCM_{SIMD}OF$).

I achieved further savings by not only processing at virtual subimage levels, but by communicating at the virtual subimage level, and only transferring large amounts of data in each global communication. Using this improvement an upright SS rectangle is calculated at each processor that lies near the original data. It's as if each processor is warping his own small image. The data required for the calculation of the subimage is local and the subimage is an upright rectangle that fills in the appropriate tile in the output. There are a variable number of messages depending on the overlap of a processor's data in the output. This technique was used in the Proteus large granularity algorithm in Chapter IV. For SIMD data parallel control is easier, and I found the subimage approach is more practical for large granularity MIMD processing because only large messages are efficient.

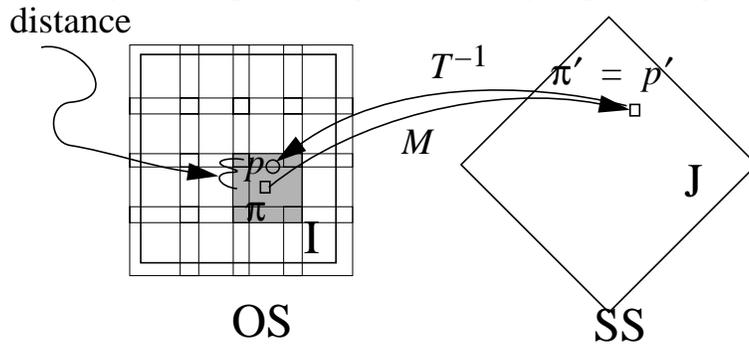


FIGURE 40

Virtualization Showing Overlapping Boundaries of Subimages

I present two 3D virtualization techniques in FIGURE 41 illustrating with 16 processors. Column virtualization is a natural extension of 2D virtualization, and increases each processor's storage in the z dimension. This allows the 2D routines to be used iteratively for input and output, and for the overlapping routines. Slice and dice virtualization assigns processors across all 3 dimensions. A factoring of the cube size is used, as is done in FIGURE 41 where column is $2^4 = 2^2 2^2$ and slice and dice is $2^4 = 2^1 2^1 2^2$ processors. For the MasPar MP-1 a balanced 3D assignment would be $2^{10} = 2^4 2^3 2^3$. Slice and dice removes dependency of run time on rotation angle, but complicates processor to volume assignments, and also makes file I/O more complicated. The application, preceding and following procedures, and the volume dimensions determine the best virtualization. I

show in the results that slice and dice is important for maintaining constant run time with arbitrary view angle freedom

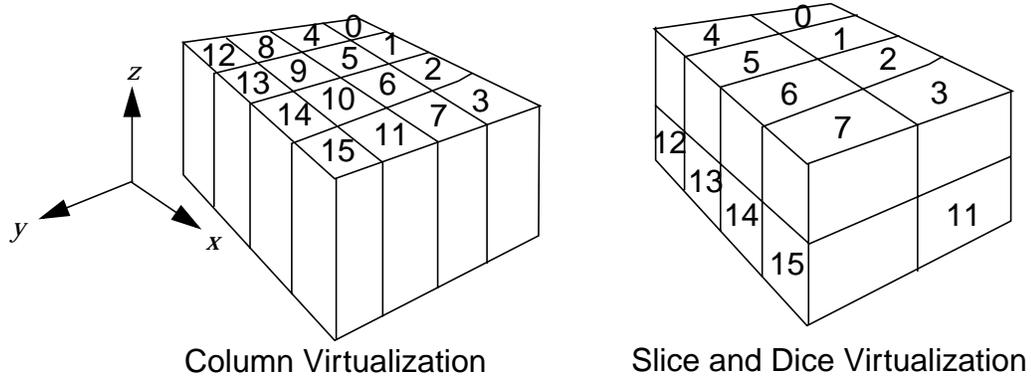


FIGURE 41 Volume Virtualization Techniques on a 2D Mesh

To virtualize with either of the above schemes, I tile the volume similar to tiling an image [WITT91]. Given a voxel address $\langle z|y|x \rangle$ where the row, col, and slice coordinates are y , x , and z respectively, it may be decomposed into tiles. The address in the tile is (i, j, k) . The dimensions of the tile are (m, n, o) . FIGURE 42 shows a schematic of the voxel space.

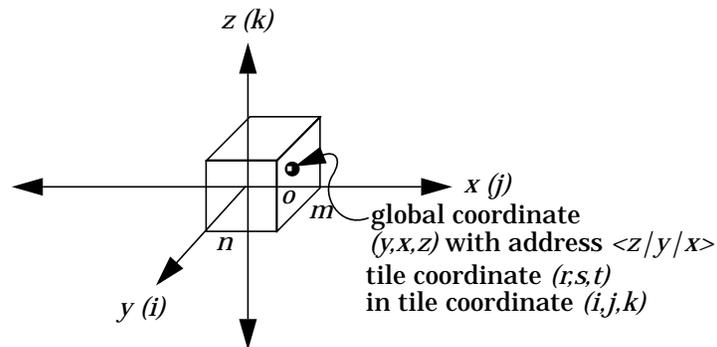


FIGURE 42 3D Tile Notation

A $\text{rows} \times \text{cols} \times \text{slices}$ volume consists of $\text{voxels} = \text{rows} \cdot \text{cols} \cdot \text{slices}$ elements. The volume is addressed by a (row, column, slice) address. The addressing is by slice, then row major order. (y, x, z) denotes any voxel $0 \leq y \leq \text{rows} - 1$, $0 \leq x \leq \text{cols} - 1$, $0 \leq z \leq \text{slices} - 1$. Tiles are also referenced in slice, row major order with m being the height, n the width, and o the depth of the tile.

The address calculations can be very efficient if $rows$, $cols$, $slices$, m , n , and o are powers of two. The Thinking Machines CM-2 [THIN89] provides built in virtualization with power of 2 shapes, but the MasPar doesn't. A voxel at (y, x, z) is stored at address $z \cdot rows \cdot cols + y \cdot cols + x$, and if they are powers of 2 the address $address = z \cdot rows \cdot cols + y \cdot cols + x = \langle z|y|x \rangle$ is a concatenation of binary strings representing x , y and z .

The volume consists of $(rows/m) \cdot (cols/n) \cdot (slices/o)$ tiles. Thus a given voxel (y, x, z) in an volume belongs to tile (r, s, t) , where $r = \lfloor y/m \rfloor$, $s = \lfloor x/n \rfloor$, and $t = \lfloor z/o \rfloor$, and its address in the tile is given by (i, j, k) , where $i = y \bmod m$, $j = x \bmod n$, and $k = z \bmod o$. The address $\langle z|y|x \rangle$ can be viewed as consisting of six parts $\langle t|k|r|i|s|j \rangle$ where bits representing y are a concatenation of bits representing r and i , x is a concatenation of s and j , and z is a concatenation t and k . $\langle t|r|s \rangle$ gives the tile address and $\langle k|i|j \rangle$ gives the address in the tile. Finally, if an image is stored in a tiled form, then the address in the tiled form is given by

$$\text{tiled_address} = t \cdot rows \cdot cols \cdot o + r \cdot cols \cdot m \cdot o + s \cdot m \cdot n \cdot o + k \cdot m \cdot n + i \cdot n + j = \langle t|r|s|k|i|j \rangle. \quad (\text{EQ 43})$$

To create a tile address from a row major address the k , i , and j bit fields are gathered to the right,

$$\langle t|k|r|i|s|j \rangle \rightarrow \langle t|r|s|k|i|j \rangle. \quad (\text{EQ 44})$$

To implement column virtualization with the notation introduced, the tile sizes are chosen to cover the processor numbers. The tile address is the processor's address, and the address in a tile is the position within the virtual array. The virtualization in depth is chosen to be completely virtualized, or $o = slices$. Because of this the t field is 0 bits, and the k field is the number of bits needed to represent z . The transform is the following,

$$\langle z|r|i|s|j \rangle \rightarrow \langle r|s|z|i|j \rangle. \quad (\text{EQ 45})$$

Processor $\langle r|s \rangle$ has an array of values $\langle z|i|j \rangle$ as shown in FIGURE 42.

For slice and dice virtualization m , n , and o are chosen to equally subdivide the number of physical processors. In this case, the t field is a non zero number of bits and the transform given in (EQ 44) is used.

With the transforms represented in (EQ 44) and (EQ 45) processor numbers and virtual array coordinates can be calculated from voxel addresses and vice versa. For example in 2D, the processor and virtual array coordinates can be calculated from a pixel coordinates. I first define the number of virtual rows by $v_rows = rows/m$ and virtual columns by $v_cols = cols/n$. Given pixel coordinate (y, x, z) with address $= \langle z|y|x \rangle$ the processor and virtual array coordinate can be calculated by,

$$\text{proc} = (y/v_rows) \times \text{nxproc} + x/v_cols \quad (\text{EQ 46})$$

$$\text{vir_array_add} = (\text{y mod } v_rows) \times v_cols + \text{x mod } v_cols \quad (\text{EQ 47})$$

But if all of these sizes are powers of 2 the same processor number and virtual array address can be calculated much more efficiently replacing multiplications by left shifts divisions by right shifts, and remainders by masking. The field lengths are represented with a symbol such as b_r , meaning b bits for field r . A mask with b_r bits in the least significant positions and zeros otherwise is given by $\text{mask}(b_j)$. The calculations are,

$$\text{proc} = (\text{y} \gg b_i) \ll b_s + \text{x} \gg b_j \quad (\text{EQ 48})$$

$$\text{vir_array_add} = (\text{y} \& \text{mask}(b_i)) \ll b_j + \text{x} \& \text{mask}(b_i) \quad (\text{EQ 49})$$

Such optimizations help by a constant amount, and the result of address and virtualization optimizations are given in the next Section.

3.9 MasPar Performance Results

In this section I detail the decisions and procedures necessary for optimizing the code on the MasPar MP-1 [BLAN90]. Four groups of programs were created for 2D rotations. I attempted to make the implementation as efficient as the MasPar could allow. 3D implementation performance is also presented, and discusses optimization issues related to virtualization. I implemented the $\text{MCCM}_{\text{SIMD}B}$ backward algorithm, the $\text{MCCM}_{\text{SIMD}F}$ forward algorithm, the $\text{MCCM}_{\text{SIMD}OF}$ overlapped forward algorithms, and some variants on the MasPar. Timings for all of the variants are presented and discussed.

Performance measurements were taken on either a 1024 or 16384 SIMD processor MP-1 whose peak performance is 26,000 MIPS (32 bit integer) and 1,200 MFLOPS (32 bit floating point). The architecture supports frame buffers through VME frame grabbers, HIPPI connection, or through MasPar's frame buffer (not available yet). Image display in the current implementation is done on the X host. The processors are interconnected through both a toroidally connected mesh with 23,000 Mbytes/sec peak bandwidth, and through a general multistage crossbar router with 1,300 Mbytes/sec peak bandwidth. The array controller provides a software accessible hardware timer that accurately captures the elapsed run time.

3.9.1 Initial Forward and Backward Algorithms

The performance of the MasPar MP-1 algorithms correlates well with predicted MCCM performance. I did not implement the multipass shear or the nonmoving multipass shear [SCHR91] and refer the reader to the MCCM complexity comparison of Section 3.6.5. MP-1s with 1024 and 16384 processors were used for performance measurements by reading from the SIMD array controller's timer. 100 timings for each angle were used, and then 10 timings when confidence in the measurements increased. The forward and backward algorithms were implemented with a first order hold and zero order hold.

TABLE 7 gives timings for the initial implementation of 2D rotation. The programs for these timings used single precision (float) variables for interpolation fractions and image coordinates. The pixels are stored as unsigned characters of 8 bits. Angles 5 to 85 degrees were run with 100 trials of each angle. The mean of the 100 trials was calculated during the measurement run. For 2D rotation, performance varies little with different angles. FIGURE 43 shows the time versus angle for the forward and backward programs using all image sizes from 32x32 to 2048x2048. In TABLE 7 the mean of the time for all angles is given.

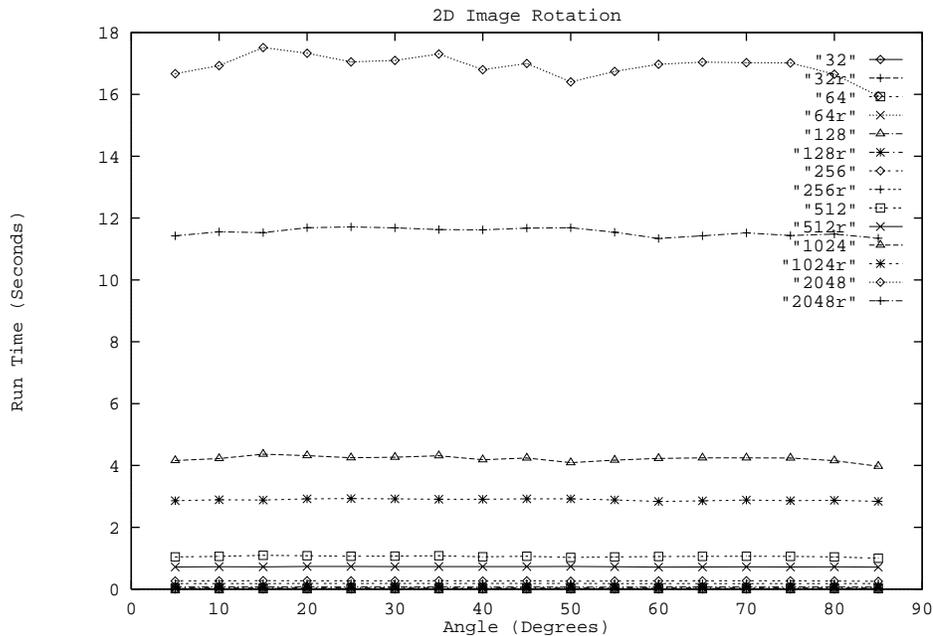


FIGURE 43 Nearly Constant Run Time Versus Angle For 2D Image Rotations, Bilinear Filter, Forward and Backward All Sizes

TABLE 7 MasPar 2D Rotations (times in seconds) with interpolation not mapped to unit interval, Bilinear Filter

size	Backward	Forward	improvement
32x32	0.005723	0.004932	16.03%
64x64	0.01800	0.01316	36.77%
128x128	0.06735	0.04686	43.72%

TABLE 7

MasPar 2D Rotations (times in seconds) with interpolation not mapped to unit interval, Bilinear Filter

size	Backward	Forward	improvement
256x256	0.265	0.1819	45.68%
512x512	1.055	0.7228	45.69%
1024x1024	4.216	2.887	46.03%
2048x2048	16.91	11.55	46.40%

The implemented algorithms are also linear in the problem size. As the number of pixels is increased the time to process them increases by a linear amount. FIGURE 44 shows the average run times from TABLE 7 on a logarithmic time scale to remove the pixels squared term. The run times are clearly linear.

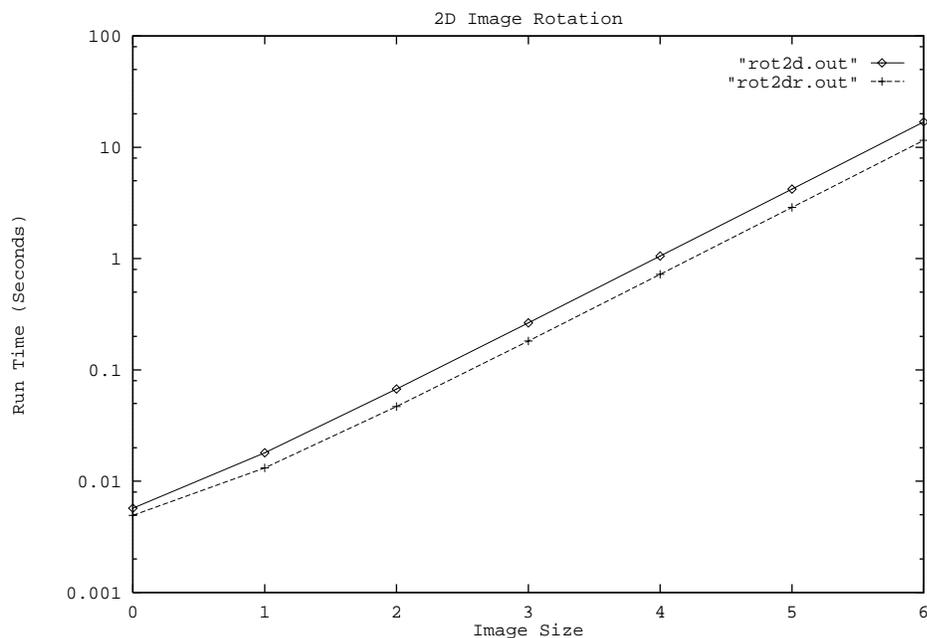


FIGURE 44

Run Time Linear In The Number of Pixels, 2D Rotation, Bilinear Filter

3.9.2 Interpolation and Overlapping Optimizations

Further optimizations in the reconstruction filter yielded improvement for both algorithms. The use of interpolation on the unit interval reduces the work for each linear interpolation. Mapping to the unit interval was described in Section 3.3. An additional algorithmic improvement was removal of a large switch statement required in SIMD processing because of the possible locations of the pixel being worked on. Because the current pixel in the forward algorithm could be on any of four edges, in any of four corners, or in the center of the virtual array, a switch statement was required to decide which communication to perform. Every processor was required to perform the entire switch statement. By storing an overlapped amount of image on each processor, the pixel was guaranteed to be contained within the processor's virtual array. The added cost was copying the neighbors data before hand, but this is very efficient because of the near neighbor connections, and the added storage is negligible because of the dynamic storage overhead already increases the used size to the power of 2 larger than the image. The overlapped image sizes allow the same size images to be processed.

TABLE 8 gives the timings for each subroutine in the overlapped forward program. The parser reads the user's file names, image size, and rotation angle. The subroutine *pl_createimage* is the parallel malloc of data on all of the processors, two images, one for the input, and one for the output. The subroutine *pl_readimage* is the slow process of reading the image data from disk, across the VMEbus, to the processors. The subroutine *pl_image_o*, takes the input data in unoverlapped form, and copies it to an array with overlapped storage, reading neighbors data as necessary. The final subroutine shown is *rotate*, or timing to resample the image. The rotate algorithm is focused on in this section. To write the file to disk takes about the same amount of time as reading it in. The overlap cost is small and the resampling savings is considerable.

These timings are for running the algorithm once, and show the performance penalty of running code the first time. Look at the columns *pl_createimage 1* and *pl_createimage 2*. The timings show that the first time *pl_createimage* runs, it takes 40 times longer because of the penalty of loading the program into the array controller's memory. These factors were removed from the other timing results by running the subroutines 100 times and averaging, or 11 times throwing out the first time and averaging.

TABLE 8 Overlapped Forward Rotation Subroutine Timings, 45 degree rotation

Size	parser	pl_createimage 1	pl_createimage 2	pl_readim age	pl_image_ o	rotate
32x32	0.082735	0.00863552	0.0002072	0.0708901	0.0002784	0.004387

TABLE 8 Overlapped Forward Rotation Subroutine Timings, 45 degree rotation

Size	parser	pl_createimage 1	pl_createimage 2	pl_readim age	pl_image_ o	rotate
64x64	0.083327	0.0084488	0.00020752	0.11803	0.0004224	0.011879
128x128	0.0825923	0.00834048	0.00020688	0.181399	0.00073088	0.041491
256x256	0.0828725	0.00840512	0.00020656	0.506679	0.00143344	0.161324
512x512	0.0907654	0.00843712	0.000208	3.09354	0.0031824	0.639548
1024x1024	0.0847466	0.00844944	0.0002072	19.6275	0.00803344	2.552792

TABLE 9 gives timings for 2D centered rotations using the improved interpolation, and the overlapped and switched forward program timings. The improvement of the overlapped forward algorithm over the backward algorithm is 24% to 59% percent for all angles. There is a greater improvement for the larger images. As in TABLE 7 the results in TABLE 9 are the averages of each angle's average. Run time remains linear with the problem size as illustrated in FIGURE 45 which shows the three 2D alternatives on a log scale (lines, scale right vertical axis). The table shows the percent improvement of different alternatives is constant for images larger than 256x256 due to virtualization.

TABLE 9 % Improvement and Run Times 2D Rotations (Run times in seconds)

Image Size	% Imp.	O. For. over Bac.	O. For. over For.	Run Times		
	For. over Bac.			Backward	Forward	O. Forward
32x32	11.12%	23.94%	10.93%	0.00547059	0.00489659	0.00441376
64x64	33.41%	44.93%	8.63%	0.01707747	0.01279988	0.01178253
128x128	41.18%	54.18%	9.21%	0.06368441	0.04510853	0.04130276
256x256	43.37%	56.97%	9.48%	0.25039082	0.17464594	0.15951435
512x512	43.92%	57.94%	9.74%	0.99782029	0.69327718	0.63174306
1024x1024	43.86%	58.06%	9.86%	3.98377918	2.76904247	2.52040165
2048x2048	44.70%	58.89%	9.80%	16.1994028	11.194726	10.1949947

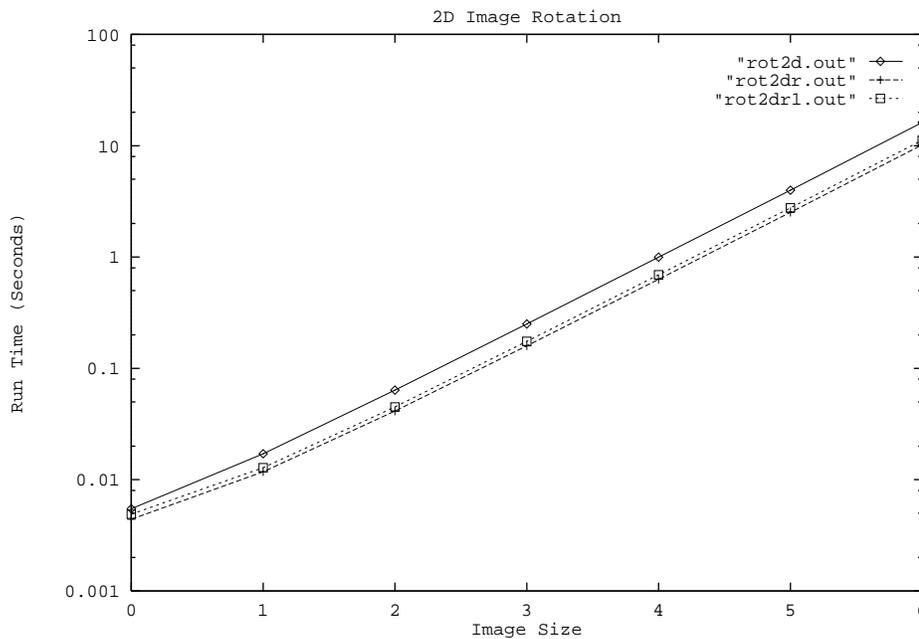


FIGURE 45 Run Times for 2D Rotation, Bilinear Interpolation on Unit Interval, with Backward, Forward, and Overlapped Forward

3.9.3 Filter Complexity, Zero Order Hold

Because the communication is efficient, I also examined how the filter impacts performance. A zero order hold was implemented. The number of interpolations is reduced from 3 to 0. FIGURE 46 shows a comparison of the zero order hold filters with several variations. Because of the rule overhead, a backwards zero order hold is fastest, as predicted with the MCCM complexity examination. The rule is overkill, because the algorithm needs to do overlapping, read a local value then send the value globally. The congestion of a single fetch is small for the backwards algorithm, as the MasPar efficiently supports small messages.

An interesting alternative sends the pixel value determined by the rule, M , and does not calculate the inverse transform. The image has nonlinear noise, and poor filter quality, but every pixel is in the output! For this approach, every processor calculates M and sends its pixel to that location. This Me variation is the most efficient when the virtualization ratio is very small. This is because the communication is more efficient than backwards, but the rule has an overhead of rounding after each step which dominates for images after virtualization of 4 to 1 (image of 64x64 or larger). Both the backwards zero

order hold, and the forwards zero order hold calculate the same value. FIGURE 46 and TABLE 10 give the run times.

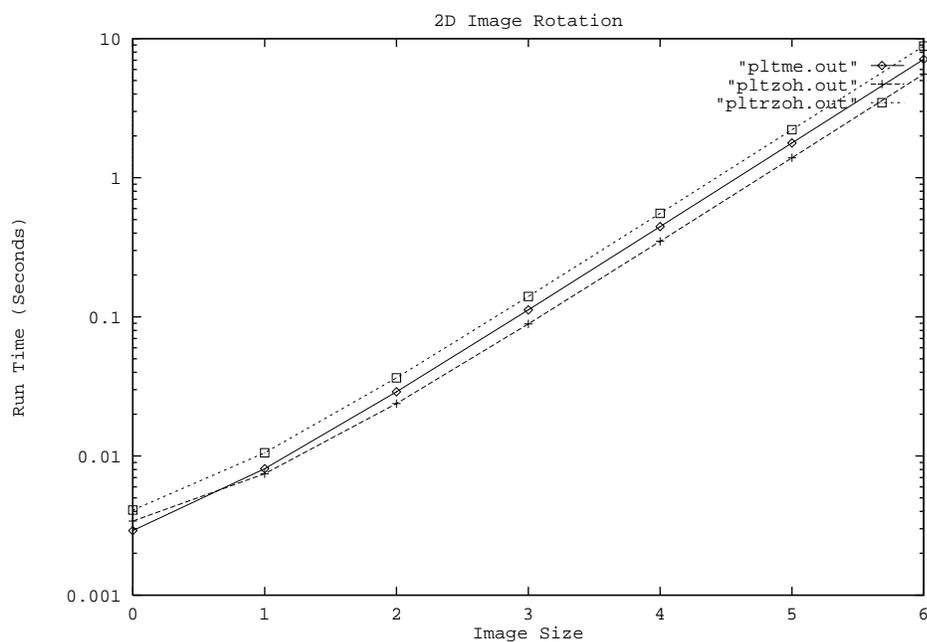


FIGURE 46

2D Rotations with Zero Order Holds, and Rule (Me) Variant

TABLE 10

MasPar 2D Rotations (times in seconds) with Zero Order Hold Filters and Rule (Me) Variant

size	Backward	Forward	Rule Only (Me)
32x32	0.003411	0.004089	0.002909
64x64	0.007454	0.01056	0.008107
128x128	0.02379	0.03647	0.02894
256x256	0.08893	0.1403	0.1124
512x512	0.34931	0.5549	0.4457
1024x1024	1.390	2.213	1.778
2048x2048	5.561	8.845	7.109

3.9.4 Optimization By Power of 2 Virtualization, and Register Optimization

Another improvement in run time was achieved by using power of 2 size images to make the address calculations in the virtualized images more efficient. For example, to calculate a processor coordinate from a pixel coordinate requires division and remainder. If the images are powers of 2 the calculation can be done by binary shifts and masks, which is much faster on the 4 bit simplified processor of the MasPar MP-1. See Section 3.8. I also used as many register declarations in these variants as possible so as to avoid loads and stores, which can become a significant overhead in the bit slice operation of the MasPar. The resulting programs are restricted to images with sizes equal to a power of 2. This is similar to the virtualization restrictions of the CM-2 [THIN89]. FIGURE 47 and TABLE 11 give the results for the address and register optimized programs. Their order of efficiency is, from high efficiency to low efficiency is backwards zero order hold, forward bilinear, and backwards bilinear. Therefore, the best bilinear approach remains the forwards algorithm. Higher order filters will be more efficient with the forward algorithm as well. The backwards algorithm is the most efficient approach with the simplest (zoh) filter. Through modification of the MasPar executable, the largest image size possible was increased to 4096x4096 for these programs so an additional row is included in TABLE 11.

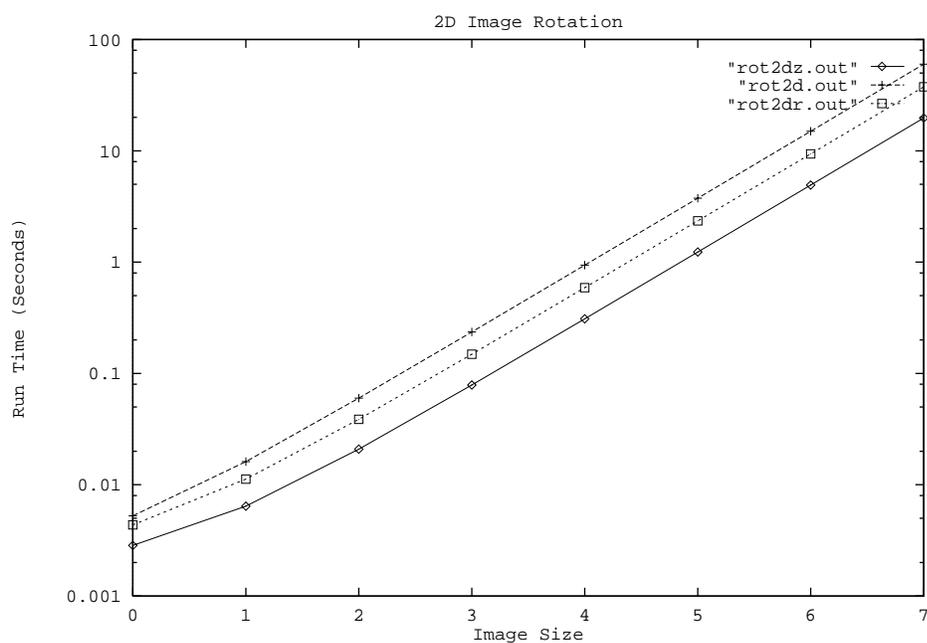


FIGURE 47

2D Rotation, Power of 2 Addresses and Register Optimization, Bilinear Interpolation Forward/Backward, and Zero Order Hold Backward

TABLE 11

MasPar 2D Rotations (times in seconds) Power of 2 and Register Optimized Versions

size	Backward, Bilinear	O. Forward, Bilinear	Backwards, Zero Order Hold
32x32	0.005253	0.004362	0.002847
64x64	0.016147	0.011228	0.006415
128x128	0.060026	0.038720	0.020897
256x256	0.235750	0.148841	0.078631
512x512	0.938320	0.588704	0.309550
1024x1024	3.749883	2.347833	1.233059
2048x2048	14.998778	9.383540	4.926640
4096x4096	60.001183	37.528438	19.702110

3.9.5 Optimization Improvements

Each program variant was improved through successive optimizations. FIGURE 48 illustrates the improvement from the initial programs, to the unit interval interpolation optimizations, to overlapping storage, and finally to addressing and register optimizations. FIGURE 48 shows average run times for rotation of a 512x512 image. The fastest program is the backwards zero order hold. The second fastest program is the forward bilinear program, and the slowest is the backwards bilinear program. The forward zero order hold and forward *Me* variation are included at their time of development, near the unit interval optimization step. They are faster than the bilinear filters, but not faster than the backwards zero order hold. This plot shows conclusively that the optimization steps significantly improved the programs, but that the relative efficiency of the programs was not affected. This supports using complexity models such as the MCCM to compare program's relative efficiencies.

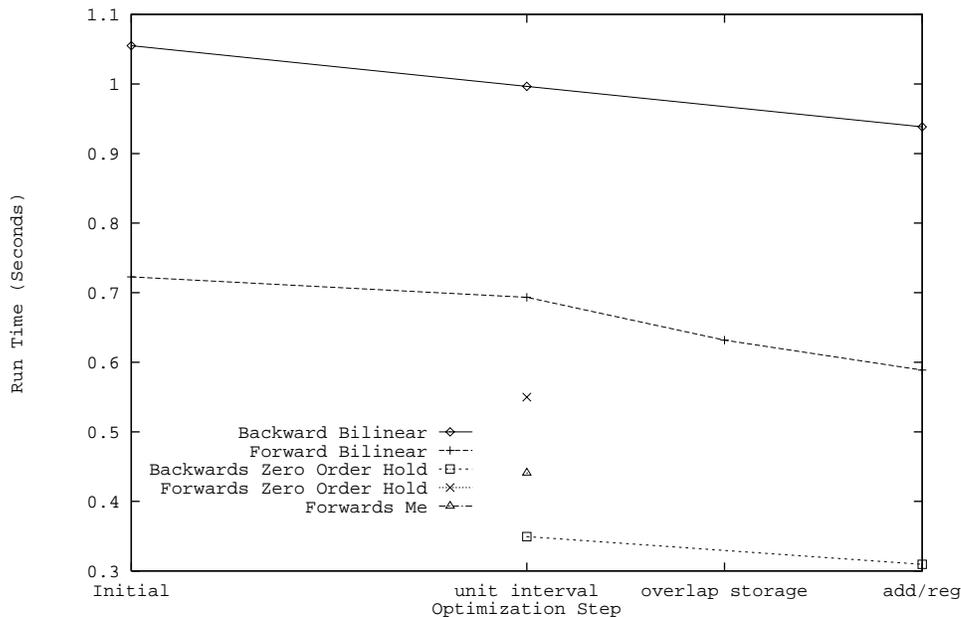


FIGURE 48

Improvement of Each Program Variant for 512 x512 Image Rotation, Seconds Versus Optimization Step

For completeness the timings shown in FIGURE 48 are given in TABLE 12. FIGURE 48 gives the ranking of all 2D rotation variants. All of the 2D rotations vary linearly with the number of pixels. There are eleven variants in all. FIGURE 49 gives their perfor-

mance, but because of the log scaling it is very hard to distinguish different programs. This figure highlights the fact that the optimizations are only constant improvements.

TABLE 12 Improvement of Each Program Variant for 512x512 Image Rotation, Seconds Versus Optimization Step

Variant	Optimization Steps			
	Initial	Unit Interval	Overlap Storage	Address and Register
Backwards Bilinear	1.055036	0.996531		0.938320
Forward Bilinear	0.722759	0.693277	0.631743	0.588704
Forward Zero Order Hold		0.549814		
Forward Me		0.440839		
Backwards Zero Order Hold		0.349311		0.309550

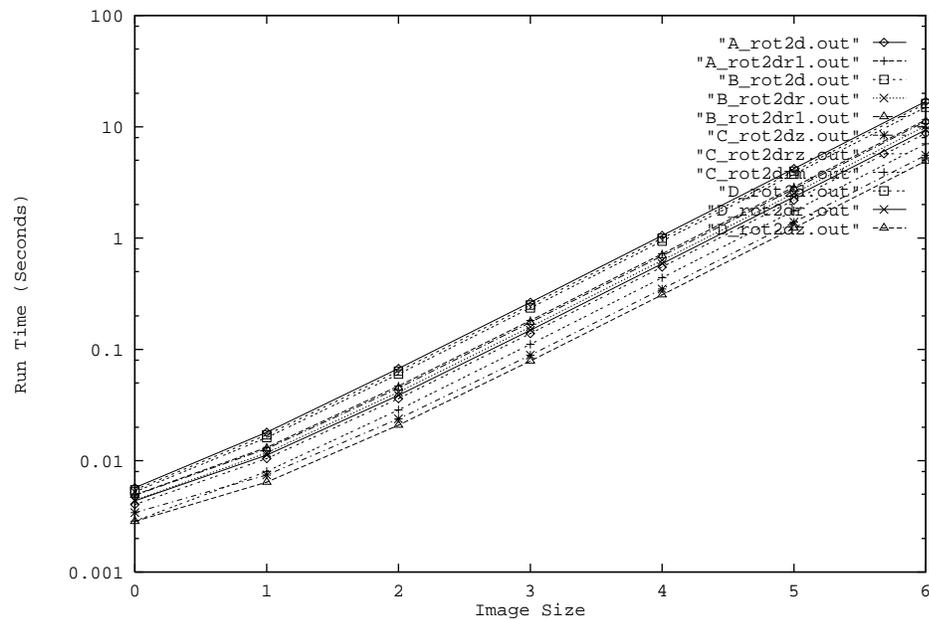


FIGURE 49

All 2D Rotation Variants Over All Image Sizes

3.9.6 3D Rotation Performance and Implementation Results

For 3D, I implemented both column virtualization and slice and dice virtualization (FIGURE 41). Column virtualization does not have constant run time with angle of rotation. Rotations of 0 to 45 degrees are reasonably efficient, but rotations of 45-85 degrees are not as efficient because of communication congestion. FIGURE 50 shows the run times for a volume size of 128x128x128 with the fastest variants of each filter: backwards zoh and forwards foh. A small number of these timings are also given in TABLE 13.

The advantages of column virtualization, are that the image slices may be read in with the same routines used for 2D processing, and the same overlap routines can be used. The performance penalty results because 3D processing is being done on a mesh machine. The performance penalty can be avoided by using rotations between 0 to 45 degrees and doing full 90 degree transpositions for other filters. I don't feel this is a good solution because it is similar to the multipass approaches, even if the filter quality is better.

Both the forward trilinear and backwards zoh timings are shown. In 3D the difference between a zoh and a foh is greater than in the 2D case, because there are 8 values instead of 4 values to interpolate. Therefore, the differences between the forward trilinear and the backward trilinear are exaggerated over the differences in the 2D bilinear filters. By altering the virtualization to slice and dice I removed the dependence on angle but programming and I/O are complicated.

TABLE 13 Column Virtualization 3D Image Rotation 1k MP-1 Performance in Seconds

Filter	Rotation Axes	Image Size	0	20	40	60	80
foh	About x	32x32x32	0.113783	0.120679	0.122798	0.127573	0.158763
		64x64x64	0.873327	0.925875	0.941115	0.962964	1.206774
		128x128x128	6.944496	7.365861	7.479546	7.626624	9.616525
		256x256x256	55.496932	58.839463	59.767284	60.697352	76.917359
foh	About y	32x32x32	0.113784	0.124002	0.127627	0.146191	0.198402
		64x64x64	0.873322	0.948982	0.983515	1.122743	1.496889
		128x128x128	6.944496	7.545647	7.817494	8.949651	11.948496
		256x256x256	55.496932	60.302461	62.466563	71.554257	95.339964
foh	About z	32x32x32	0.114852	0.124515	0.127035	0.125806	0.126379
		64x64x64	0.874561	0.963425	0.964367	0.948151	0.959466
		128x128x128	6.945591	7.713501	7.677307	7.541527	7.647887
		256x256x256	55.496937	61.628235	61.243510	60.157255	60.790894
foh	About x, y, and z	32x32x32	0.113784	0.130417	0.143873	0.189098	0.242284
		64x64x64	0.873321	0.986322	1.092952	1.454064	1.869899
		128x128x128	6.944496	7.822128	8.678686	11.556266	14.970083

TABLE 13 Column Virtualization 3D Image Rotation 1k MP-1 Performance in Seconds

Filter	Rotation Axes	Image Size	0	20	40	60	80
		256x256x256	55.496932	62.520676	69.326421	92.388327	119.45221
zoh	About x	32x32x32	0.048917	0.056515	0.057663	0.060113	0.088392
		64x64x64	0.367715	0.421055	0.432319	0.449363	0.680601
		128x128x128	2.931376	3.338181	3.435066	3.556403	5.419631
		256x256x256	23.499534	26.681865	27.493456	28.392862	43.382922
zoh	About y	32x32x32	0.048916	0.058983	0.062726	0.078388	0.119829
		64x64x64	0.367716	0.441973	0.472276	0.601324	0.940745
		128x128x128	2.931376	3.507766	3.757546	4.797153	7.533232
		256x256x256	23.499522	28.048093	30.038246	38.413253	60.420284
zoh	About z	32x32x32	0.049982	0.057913	0.055157	0.056251	0.056250
		64x64x64	0.368808	0.436837	0.431357	0.428977	0.442319
		128x128x128	2.932458	3.569854	3.506354	3.491875	3.474053
		256x256x256	23.499522	28.569808	28.116221	28.268908	28.102764
zoh	About x, y, and z	32x32x32	0.048916	0.062139	0.072289	0.100700	0.158865
		64x64x64	0.367717	0.469397	0.553640	0.791117	1.263828
		128x128x128	2.931376	3.743829	4.427272	6.354319	10.178167
		256x256x256	23.499532	30.015604	35.481798	51.011271	81.745668

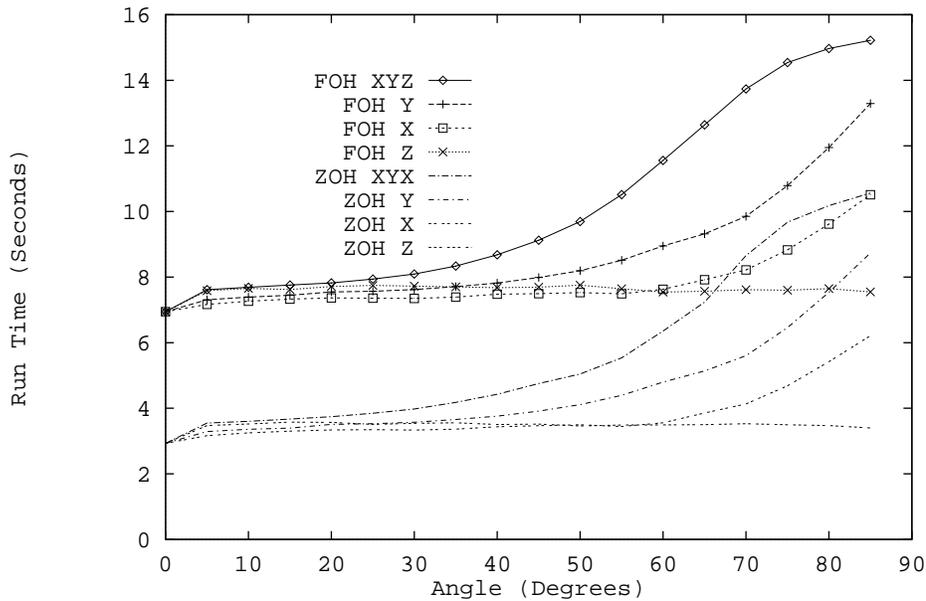


FIGURE 50

Column Virtualization on 1024 PE MP-1 Warping a 128x128x128 Volume

Slice and dice virtualization does indeed remove the dependence of run time on the rotation angle for a one pass, any angle approach. Both the forwards and backwards algorithms are more efficient for higher angles of rotation using slice and dice. FIGURE 51 and TABLE 14 show backwards first order hold, forwards first order hold, forwards zero order hold, and backwards zero order hold, respectively. TABLE 15 shows the relative improvements between slice and dice virtualized algorithms. The separation of forwards over backwards increases to 62% to 101%. The zoh is from 85% to 148% faster than the first order hold, comparing the forwards foh to the backwards zoh. MCCM predictions, TABLE 6, and the MasPar measurements correspond well, because the congestion for 2D is 7 and for 3D is 10 assuming normalized global communication costs, $G = 1$. The algorithms are ranked according to the congestion and rule overhead. The backwards zero or-

der hold is the fastest, followed by forwards first order hold, and as the number of points used goes up the congestion becomes even more important.

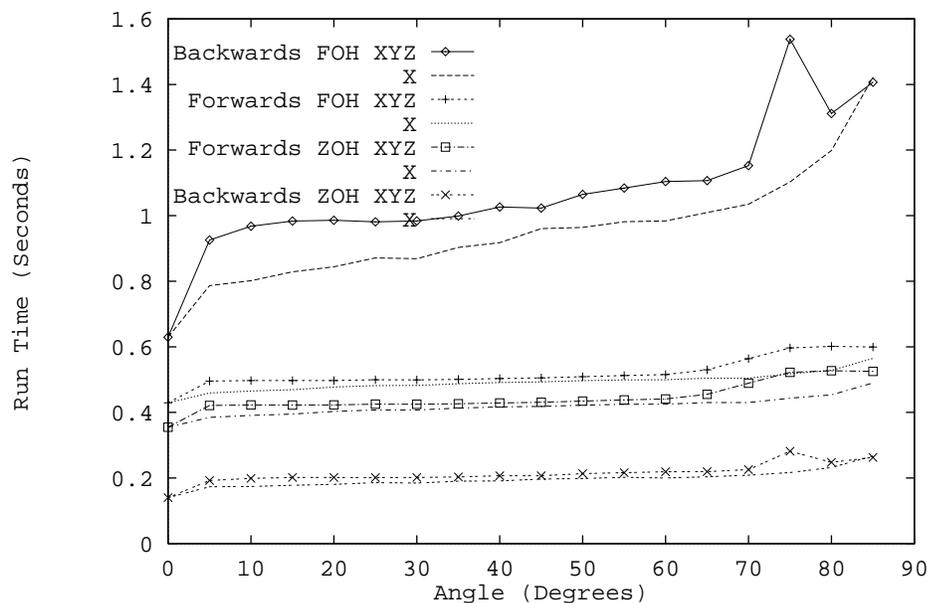


FIGURE 51

Slice and Dice Virtualization on 16,384 PE MP-1 warping a 128x128x128 volume

TABLE 14

16K Processor MP-1 Slice And Dice Timings For Warping, Seconds

	vol size	Mean	Min	Max
Back foh	32x32x32	0.019801	0.016054	0.028579
	64x64x64	0.130724	0.090662	0.198416
	128x128x128	1.006664	0.629505	1.537600
	256x256x256	7.976678	4.685603	12.314704
Forward foh	32x32x32	0.012223	0.010403	0.014923
	64x64x64	0.066698	0.056962	0.081431
	128x128x128	0.501673	0.429225	0.601604
	256x256x256	3.977112	3.407390	4.749763
Forward zoh	32x32x32	0.011060	0.009243	0.013762
	64x64x64	0.057393	0.047662	0.072114

TABLE 14 16K Processor MP-1 Slice And Dice Timings For Warping, Seconds

	vol size	Mean	Min	Max
	128x128x128	0.427223	0.354747	0.527172
	256x256x256	3.381126	2.810794	4.154157
Backward zoh	32x32x32	0.006591	0.005384	0.008076
	64x64x64	0.028422	0.020378	0.037909
	128x128x128	0.203063	0.140105	0.281743
	256x256x256	1.602002	1.096583	2.223762

TABLE 15 Percent Improvement for 3D Slice and Dice Algorithms on 16k Processor MP-1

Volume Size	Forward over Backward (FOH)	ZOH over FOH
32x32x32	61.99%	85.44%
64x64x64	95.99%	134.67%
128x128x128	100.66%	147.05%
256x256x256	100.56%	148.25%

FIGURE 52 compares slice and dice to the column virtualization. Slice and dice virtualization keeps the run time constant for any angle. Some of the column virtualization timings are also given in TABLE 17 to compare with those from TABLE 14.

Comparisons to [VEZI92] and [SCHR92] show that our resampling times are about a factor of 4 slower than [VEZI92] and 1.3 to 5 times faster than [SCHR91] for rotation only. See TABLE 16. The factor of 4 slowdown is clearly a result of the general router

and mesh router mismatch, recall 1300 Mbytes/s versus 23,000 Mbytes/sec. The router start-up penalty and/or the rule overhead accounts for the rest of the difference.

TABLE 16 Rotation Only, From [VEZI92][SCHR91] Milliseconds

	Computer	vol size	Time	Speedup vs. Permutation Warp
[VEZI92] zoh 4 pass	16k pe MP-1	128x128x128	49	0.241
	16k pe MP-1	256x256x256	390	0.243
[VEZI92] foh 4 pass	16k pe MP-1	128x128x128	139	0.277
	16-k pe MP-1	256x256x256	1107	0.278
[SCHR91] foh 5 pass	64k pe CM-200	128x128x128	268	1.320
	32k pe CM-200	128x128x128	511	2.516
	16k pe CM-200	128x128x128	1033	5.087

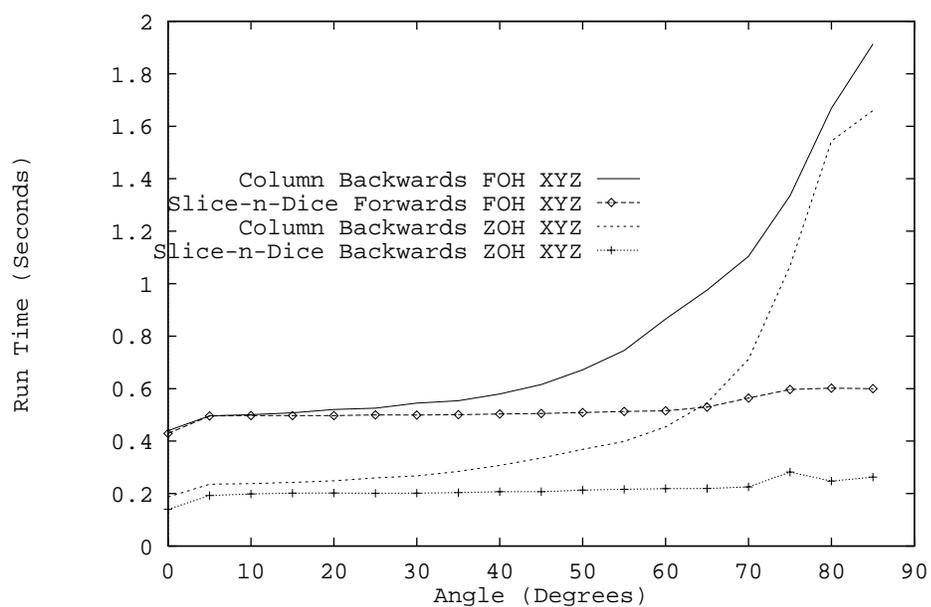


FIGURE 52

16k MP-1 MasPar Performance on 128x128x128 Volume Rotation, Slice and Dice compared to Column Virtualization

TABLE 17 16k MP-1 Column Virtualization 3D Image Warping Performance in Seconds

Filter	Rotation Axes	Volume Size	20	40	60	80
Back foh	About x	128x128x128	0.482388	0.495113	0.524356	0.675803
		256x256x256	3.827529	3.918777	4.057814	5.312173
	About y	128x128x128	0.482790	0.502875	0.571791	0.845904
		256x256x256	3.820866	3.984160	4.503583	6.745278
	About z	128x128x128	0.497712	0.503362	0.497712	0.485640
		256x256x256	3.901328	4.041634	3.930743	3.901326
	About x, y, and z	128x128x128	0.520609	0.579541	0.864906	1.668463
		256x256x256	4.047185	4.545814	6.880653	13.093213
Back zoh	About x	128x128x128	0.229737	0.240114	0.24969	0.391298
		256x256x256	1.801486	1.888355	1.962068	3.103868
	About y	128x128x128	0.228393	0.247116	0.304206	0.559010
		256x256x256	1.794471	1.943612	2.404857	4.450803
	About z	128x128x128	0.238539	0.245222	0.242995	0.238540
		256x256x256	1.867224	1.955556	1.927851	1.861124
	About x, y, and z	128x128x128	0.248391	0.307315	0.454092	1.543877
		256x256x256	1.957220	2.434717	3.599034	12.357759

3.10 Summary and Discussion

I presented new optimal direct warp algorithms for the CREW and EREW PRAMs. The EREW PRAM algorithm is restricted to equiareal transforms, but is more efficient than the CREW algorithm in practice and can be used in conjunction with the more general algorithm for other transforms. The parallel run time complexity of both algorithms is $O(nd)$ using $n(n+1)^{d-1}$ processors per sample, $O(n^{d+1})$ using 1 processor per sample, or $O(1)$ if filter complexity is considered constant (n is the order of the polynomial interpolation reconstruction filter, and d is the dimension of the image being warped.) The PRAM complexities are the same, but on different strength machines. The MCCM more accurately predicts performance of existing machines. The backwards algorithm is $O(N)$ on the MCCM where N is congestion that varies with the transform. The forward direct warp is $O(1)$ on the MCCM, and therefore optimal because of its efficient communication. The MCCM directly maps to most commercial parallel processing machines, and therefore the forward direct warp algorithm can achieve arbitrary image and volume rotations with low communication costs. The forward algorithm works because of clever job assignment. The nonlinear processor mapping assigns jobs to be close to where the original data lies, allow-

ing efficient gathering of the results, and optimal implementation of local neighborhood filters.

The forward algorithm works for all dimensions, and I presented results for 2D and 3D. Using a first order hold the EREW algorithm has up to a 59% improvement over the CREW algorithm for 2D. Improvements of up to 100% were measured for 3D. 3D volume rotations are important for scientific visualization of voxel data. My implementation of the forward algorithm on the MasPar MP-1 is linear in problem size and fast enough for interactive (<1 second) visualization. 2 million voxels (128x128x128) are rotated in about half of a second using 16384 processors and a first order hold. It takes only one fifth of a second with a zero order hold. I also showed how virtualization techniques can severely affect performance. Future research is necessary to generalize the forward algorithm to scaling affine and other transforms.

Chapter IV

Spatial Volume Rendering

In this chapter I present two new volume rendering algorithms that are optimally efficient on the PRAM. The first algorithm uses a data parallel approach and warping from Chapter III for massively parallel architectures. The second algorithm uses large granularity messages for architectures that do not support small messages. Empirical error analysis supporting the analysis in Chapter III illustrates the quality advantage over existing parallel methods, and the performance measurements show the new flexibility in view angles. The performance measurements also confirm the flexibility of my direct warp algorithms which can use many orders of filters efficiently.

I review volume rendering and develop the rendering equations in Sections 4.1. I survey and classify existing algorithms in Section 4.2, then I present two new permutation warping algorithms for volume rendering that achieve linear speedup in Sections 4.4 through 4.5. Empirical error analysis and Maspar and Proteus performance measurements are given to confirm the speedup and tunable accuracy characteristics of the algorithms in Section 4.6. The chapter concludes in Section 4.7.

4.1 Background

Volume rendering is transparency visualization of sampled three dimensional data [KAJI84] [LEVO88][KAUF91]. Samples, called voxels, are created by magnetic resonance imaging, finite element analysis, computed tomography, and other applications. Visualization techniques are application dependent, and many applications are well suited by traditional graphics using surface models [TIED90][UDUP90]. But, there are other applications that require semi-transparent, volume rendering. The transparency is used either for effect, for example special effects in movies, or because the data are aliased¹ or hard to segment². Applications that use transparency effects include photo realistic rendering of clouds [KAJI84], creating contextual clues for medical imaging [LEVO89], and viewing of multi-valued functions [KRUE90].

The lighting and shading calculation for transparency is solved by transport theory [CHAN60]. Those who have extended the solution to non-homogeneous media call it volume rendering [BLIN82][KAJI84][KRUE90][LEVO90][SABE88]. Because volume rendering is computationally expensive, special purpose architectures have been developed to improve performance [KAUF90][KAUF91b][GOLD85][KAUF88]. Recently researchers

1. undersampled data where multiple frequencies are seen as the same frequency
2. To separate regions in an image as to their membership in desired sets

have investigated algorithms for general purpose parallel computers, because they are more widely used, can use enhanced shading and illumination models, and provide generation scalable solutions. Volume rendering is an $O(S)$ run time algorithm where S equals the number of samples stored in the volume, $S = n \times n \times n$. Because S is very large, the run time is great. For example, $S = 2^{24}$ for a $256 \times 256 \times 256$ volume, and if each sample point is a red, green, blue, α (opacity) tuple, the source volume is 64 Megabytes. Volume rendering is memory and compute bound. My volume rendering algorithms use the warping techniques from Chapter III and new techniques developed here to achieve linear speedup on shared and distributed memory parallel machines. The error in direct volume rendering versus multipass approaches is empirically investigated, showing the improvement of my algorithm.

First, I review the particle lighting and shading models, Section 4.1.1. Then I survey existing volume rendering techniques classifying them by their viewing transform as introduced in Chapter II. The algorithms are described in Section 4.4. Complexity of the parallel algorithms is discussed in Section 4.4, and filter quality and run time measurements are given in Section 4.6.

4.1.1 Volume Rendering Lighting and Shading Models

If a three dimensional function $V(u, v, w)$ is prefiltered to avoid aliasing it can be sampled provided the Nyquist criterion is satisfied for spatial frequencies $f_{S_{i,j,k}} \geq 2f_v$, where f_v is the highest frequency contained in the volume,

$$V[x, y, z] = V\left(\frac{1}{f_{S_x}} x, \frac{1}{f_{S_y}} y, \frac{1}{f_{S_z}} z\right). \quad (\text{EQ 50})$$

f_{S_x} , f_{S_y} , and f_{S_z} are the sampling frequencies. Examples of $V(u, v, w)$ are X-ray attenuation, radio pharmaceutical concentration, and proton density.

To render, the samples $V[x, y, z]^3$ are classified and segmented to extract features of interest. Segmentation separates regions of the volume into its components [JAIN89]. For example, density ranges can identify bone, skin, and other tissues with a segmentation operator of $|V[x, y, z] - V_s| \leq \xi$, where V_s is the scalar value of the chosen surface and ξ is the tolerance [DREB88]. The results are opacities (α) and normals (\vec{N}) used to calculate the lighting and shading in the volume. Both surface and particle lighting models are used to render voxels.

3. [] brackets will be used to denote discrete functions and () parenthesis to denote continuous functions.

Then, samples are projected into a 2D image. A projection, T , is defined as a mapping from object space (OS) to screen space (SS), $SS = T(OS)$. Two common projections are perspective and orthogonal. The process of volume rendering is shown in FIGURE 53.

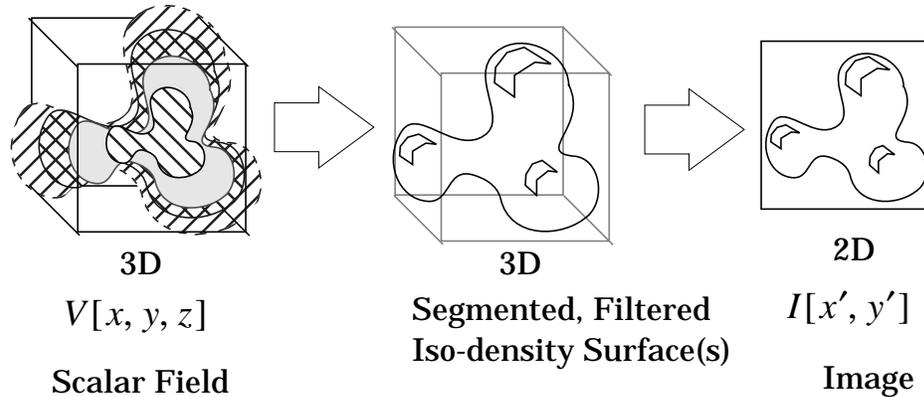


FIGURE 53 Volume Visualization

The difficulty of projection is that many voxels in OS contribute to each pixel in SS, therefore combining rules are defined on the volume. Also, the transformed voxels do not match up with the pixel locations, and reconstruction and resampling of the projected voxels are necessary. The combining operations are lighting and shading discussed in Section 4.1.3. Projections are warps from Chapter III and high granularity warps developed in this chapter.

4.1.2 Surface Lighting Models

One approach is to fit voxels with opaque surfaces. The surface normals (\vec{N}), material properties (k_a , k_d , k_s , and n), and colors ($O_{d\lambda}$ and $O_{s\lambda}$) are used to calculate the shading intensity. The lighting model sums scattered I_a (ambient), diffuse $I_{\lambda\gamma}(\vec{N} \cdot \vec{L})$ (Lambertian), and reflected $I_{\lambda\gamma}(\cos^n \theta_\gamma)$ (specular) light. A lighting model is [FOLE90]

$$I_{S\lambda} = I_a k_a O_{d\lambda} + f_{\text{att}} I_{\lambda\gamma} [k_d O_{d\lambda} (\vec{N} \cdot \vec{L}) + k_s O_{s\lambda} \cos^n \theta_\gamma]. \quad (\text{EQ 51})$$

The material properties are determined by trial and error in this empirical lighting model. f_{att} is the light source attenuation factor. θ_γ is the angle between the reflected light and the view direction. $I_{\lambda\gamma}$ is the intensity of the directional light sources, where the direction to the light source is specified as \vec{L} . The model is calculated for 3 primary specular wavelengths with $\lambda = \{\text{red, green, blue}\}$. A lighting model may incorporate ambient, diffuse, and specular or any combination. Also, depth cueing may be incorporated to divide down the intensity of light as a function of the distance to the viewer.

A shading model is the evaluation of this lighting equation. Rather than evaluate it at all points on the surface if one uses a triangulated surface they can interpolate the colors or normals to calculate colors between the vertices. Many variations on shading exist due to the different ways in which normals may be calculated from voxels (z-buffer shading, grey-level gradient shading, and adaptive grey-level shading [TIED90][KAUF91]) and the lighting models calculated (constant, Gouraud, Phong, and Torrance-Sparrow [FOLE90]).

4.1.3 Particle Lighting Model

Particle lighting models are used to render transparent materials [BLIN82][KAJI84] [LEVO89] [SABE88]. Transport theory of energy solves a system of interacting light sources and particles that absorb, reflect, emit, and transmit light. Initially applied to the modelling of stellar phenomena such as interstellar clouds and planetary rings [BLIN82][CHAN60], transport theory is now applied to voxel visualization [KAJI84]. Methods to solve for the final intensity that reaches the eye has been the focus of graphics researchers [BLIN82][DREB88] [GOLD88] [KAJI84] [KRUE90] [LEVO90] [SABE88] [UPSO88] [WEST90]. I define intensity, I , as the radiant intensity (Watts) or amount of measured light energy, not to be confused with luminance brightness (lumens), the perceived intensity. Consider point light sources, γ , of intensity I_γ illuminating a variable density volume of particles. FIGURE 54 shows a sketch of the system.

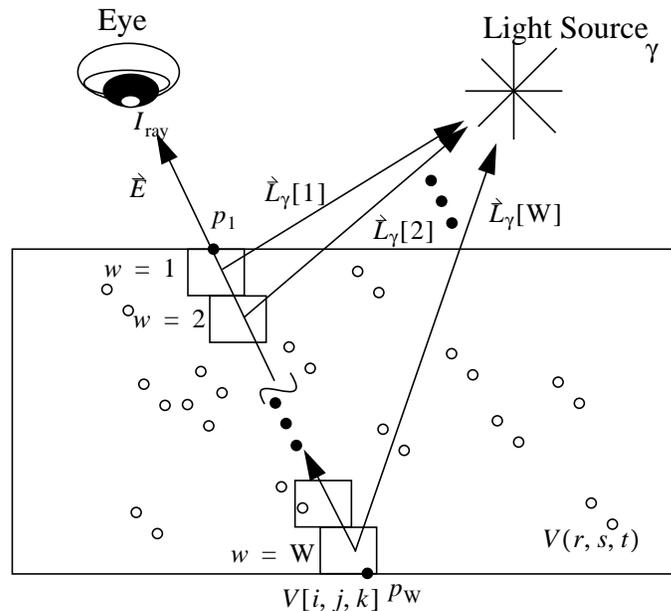


FIGURE 54

Single Level Scattering Particle Model

The densities, gradients, and lighting properties of particles determine the amount of light reflected, absorbed, and scattered. Particles are quantified through their phase

function, a function which determines the direction and amount of reflected light. If the particles have low albedo, or little reflection, then a single level of scattering is used to model the light in the volume.

Kajiya [KAJI84] and others assume that the density of particles is low, therefore the probability that there are no particles in a volume V is modelled by a poisson process [BLIN82][ESPO79],

$$P(0;V) = e^{-n_0V}. \quad (\text{EQ 52})$$

The parameter n_0 is the number of particles per unit volume, and n_0V is the expected number of particles in the volume. Higher density models are derived by [ESPO79] and [KAJI84]. At each point in the volume an intensity is emitted only if there are particles to reflect, transmit, or emit light. All points therefore are attenuated by the expected number of particles at that point in the volume. I use α to denote the expected number of particles, and the segmented function. Several researchers [BLIN82][KAJI84][SABE88][LEVO89] do some fudging with the transport theory in the expression of the optical depth. It is appropriate to look at the original sources such as [ESPO79] for clear understanding of the many approximations taking place. Optical depth is the dimensionless attenuation of light as it passes through the particles $\tau = n_0V$ of (EQ 52).

Define $\alpha(u, v, w)$ to be the probability density of an encounter at point (u, v, w) . Then for a poisson process the probability of encountering 0 particles along path l , allowing the ray to pass is [ESPO79],

$$t_l(u, v, w) = P(k=0, l) = e^{-\int_l \alpha(l')dl'}. \quad (\text{EQ 53})$$

Light from a source reaches a point in the volume if it doesn't encounter a particle to scatter it. The probability of the ray being scattered as it proceeds from the source into the volume is (EQ 53) integrated along the illuminated ray or transparency. I consider any modification of $\alpha(u, v, w)$ such as powering [SABE88], scaling [KAJI84], or mapping [LEVO89] part of the classification. Ignore the constants given by other researchers, such as absorption coefficient [KAJI84], particle volume [BLINN82][SABE88], and mean cross sectional area for extinction [ESPO79][SABE88]. These constants are used to ratio-nalize the poisson density that most researchers use.

The incident light energy (I_L) of a light source at any point in the volume is simply the product of the light source strength I_γ and the transparency $t_{l_\gamma}(u, v, w)$ along the path from the point to the light source,

$$I_{L_\gamma}(u, v, w) = I_\gamma t_{l_\gamma}(u, v, w). \quad (\text{EQ 54})$$

The intensity resulting from the interactions of light with the particles is the shading intensity (I_s). This interaction in surface graphics is described by the shading function. In transport theory, interaction is described by the phase function $\Phi(\vec{E} \cdot \vec{L})$, a function of the view

direction $(-\vec{E})$ and light source direction (\vec{L}) . See FIGURE 54. The volume rendering lighting model takes into account both the phase function of particle interaction and the oriented surface interaction. Normals (\vec{N}) within the volume are calculated as part of classification [LEVO89][KRUE90]. For multiple light sources the shading intensity is

$$I_S(u, v, w) = \sum_{\gamma} \text{Shading}(\vec{E}, \vec{N}, \vec{L}_{\gamma}, \alpha, I_{L\gamma}) . \quad (\text{EQ 55})$$

Kajiya [KAJI84] considers only the particle interaction (not surface) so the shading intensity resulting from all light sources with incident light intensity $I_{L\gamma}$ is

$$I_S(u, v, w) = \sum_{\gamma} I_{L\gamma}(u, v, w)(\Phi(\vec{E} \cdot \vec{L}_{\gamma})) , \quad (\text{EQ 56})$$

a function of the phase function $\Phi(\vec{E} \cdot \vec{L})$. The particles are chosen to be oriented along a plane whose normal is determined by the particle gradient. The gradient is approximated by central differences,

$$\nabla V[i, j, k] \approx$$

$$\frac{1}{2}(V[i+1, j, k] - V[i-1, j, k]), \frac{1}{2}(V[i, j+1, k] - V[i, j-1, k]), \frac{1}{2}(V[i, j, k+1] - V[i, j, k-1]) \quad (\text{EQ 57})$$

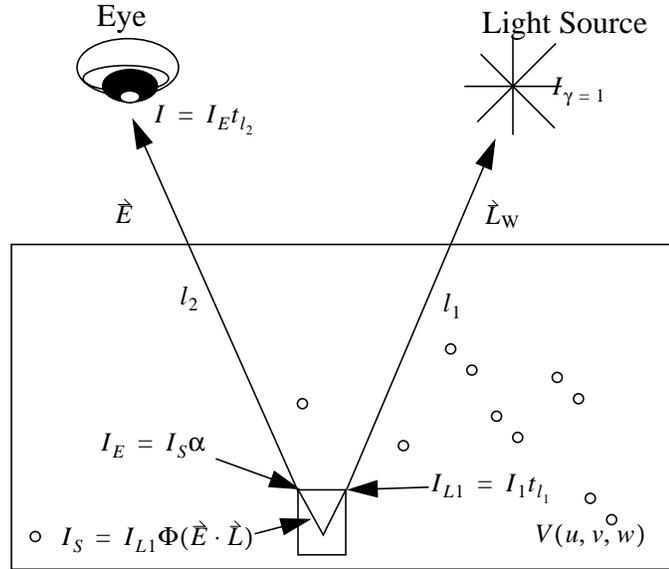
Or normals can be approximated in the two dimensional SS [KAUF91][TIED90].

For particle models this shading intensity is not what is emitted from that point in the volume. The emitted intensity from any point in the volume is the product of the shading intensities $I_{S\gamma}(u, v, w)$ and the probability that there is a particle there for the light to bounce off of. This probability as I defined earlier is α the probability of an encounter. A completely opaque surface emits all of the light, $\alpha = 1$, but a semitransparent voxel emits only the amount of light that will reflect or be transmitted by the particles,

$$I_E(u, v, w) = I_S(u, v, w)\alpha(u, v, w) . \quad (\text{EQ 58})$$

In FIGURE 55 I clarify the intensities I_{γ} , I_L , I_S , and I_E . Shown is the complete path of light from source to eye. The light source has strength intensity I_{γ} , that travels along path l_1 and is attenuated along that path giving an incident intensity of I_L . The incident energy interacts with the particles according to their shading or phase function and gives a shading intensity I_S . The shading intensity scaled by the probability of there being a particle at that position creates the emitted intensity I_E . The emitted intensity travels to the eye attenuated by the transparency t_{l_2} along the path l_2 . The intensity resulting from a single

point in the volume is $I_E t_{l_2}$. The ray intensity I_{ray} is the sum of all points on that ray through the volume.



For a specific point (r, s, t)

FIGURE 55 Intensity calculation for one point in the volume

To summarize, the intensity, I_{ray} , of a view ray is equal to the sum of the contributions at all points along the ray,

$$I_{\text{ray}} = \int_{p_1}^{p_w} t(l) I_S(l) \alpha(l) dl. \quad (\text{EQ 59})$$

This is a line integral along a path l from point p_1 to point p_w shown in FIGURE 54. The final intensity of the view ray is the product of the transparency, shaded intensity, and the volume density (expected number of particles). Kajiya [KAJI84] uses numerical Rydberg integration to calculate the intensities and transparencies; others use rectangular rule integration to be described shortly.

(EQ 59) defines the lighting and shading model in terms of a low albedo (reflectivity) shading or phase function. If higher albedo particles are modelled, the solution of the scattering equation becomes more involved. The solution depends on many paths of scattering and not the single path illustrated in FIGURE 54. A perturbation theory solution is derived by Kajiya to model high albedo particle clouds. Multiple scattering effects in a volume have been proposed [KAJI84], but at this time are not practical for interactive visualization. Kajiya shows that multiple scattering is useful for rendering clouds but simpler methods are often adequate for other applications.

The low albedo model, (EQ 59), is further simplified [LEVO90] by ignoring the inter-particle shadowing along lines of illumination. Calculate $I_{L\gamma}(u, v, w)$ directly at each point ignoring the attenuation of the light through the volume. (Let $I_{L\gamma}(u, v, w) = I_\gamma$.) This is how Drebin, Levoy, and Sabella perform volume rendering. The algebraic formulations they use are numerical integration of (EQ 59) by the rectangular rule. Consider samples along each view ray $w = 1$ to W where the shading intensities and opacities have already been computed. The discrete formulation of (EQ 59) is [SABE88]

$$I_{\text{ray}} = \sum_{w=1}^W e^{-\sum_{m=1}^{w-1} \alpha[m]} I_S[w] \alpha[w] . \quad (\text{EQ 60})$$

Which may be reformulated to

$$I_{\text{ray}} = \sum_{w=1}^W I_S[w] \alpha[w] \prod_{m=1}^{w-1} e^{-\alpha[m]} . \quad (\text{EQ 61})$$

Levoy gives an alternative development reaching (EQ 61) by starting with a discrete system. The models are the same but the discrete model derivation clarifies the results. Refer to FIGURE 56 which considers effects along the view ray. The volume is modelled as a varying density emitter with a single level of scattering. The light is attenuated in the view direction, where two effects are considered. Depth-cueing attenuates the intensity as an inverse of the distance from the viewer, and particles block rays from voxels behind them. Each sub slab is considered separately by using a conditional probability that slabs between the eye and the current slab will block the illumination from the slab.

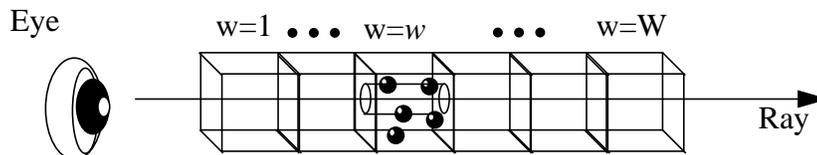


FIGURE 56 Volumetric compositing calculations

Consider the contribution of slab $w = w$. If all slabs in front of w have no particles, w has an intensity only if the view ray hits a particle in w . I can quantify the events: have-no-particles, and hits-a-particle, by probabilities. In FIGURE 56, the probability that slabs $\{1, w-1\}$ have no particles is evaluation of (EQ 52) for the cylindrical ray volume $V_{1\dots(w-1)}$. The probability that the ray hits a particle in cylindrical volume V_w is $1 - P(0; V_w)$. So the joint probability of these events, the probability the shading intensity reaches the eye, is $P(0; V_{1\dots(w-1)})(1 - P(0; V_w))$. The resulting intensity from slab w is then just the shading intensity times the probability that there are no particles in slabs $1\dots(w-1)$ and there is one particle in slab w .

The discrete transparency is often approximated by the Taylor series expansion of the exponential as $t[m] = e^{-\alpha[m]} \cong (1 - \alpha[m])$, where $t[m]$ and $\alpha[m]$ are probabilities in the range $[0, 1]$. Levoy's approximation is the expected number of particles in the cylindrical volume $\alpha[w] = (1 - P(0; V_w))$, which is summed over the entire unit volume of FIGURE 56. The volume terms drop out $e^{-n} = e^{-nV}$. Sabella, Blinn, and Kajjiya use the density as the expected number of particles (EQ 59). The sum of all slab's contributions is given in (EQ 61) derived earlier.

Compositing is evaluation of (EQ 61), and there are several alternatives. Images are composited from front to back (or back to front) carrying both the intensity and the opacity to each sub-slab. A running product and sum is calculated adding the contributing factor of each sub-slab until for each ray a final intensity is calculated. I express the combined results as emitted intensities, $I_E[w]$, the starting intensity as $I_S[w]$ (EQ 55) or (EQ 56), and the initial density or opacity as $\alpha[w]$. Note that the combined opacities are denoted as $\alpha_{i\dots j}$ and similarly for the transparencies. A front-to-back recursive solution of (EQ 61) in terms of the transparency, $t = 1 - \alpha$, is

$$I_E[w] = I_E[w-1] + I_S[w]\alpha[w]t_{1\dots(w-1)}[w-1] \quad , \quad (\text{EQ 62})$$

$$t_{1\dots w}[w] = t_{1\dots(w-1)}[w-1](1 - \alpha[w]) \quad . \quad (\text{EQ 63})$$

A substitution of $t = 1 - \alpha$ can be made to derive the equations in terms of the opacity, α ,

$$I_E[w] = I_E[w-1] + I_S[w]\alpha[w](1 - \alpha_{1\dots(w-1)}[w-1]) \quad , \quad (\text{EQ 64})$$

$$\alpha_{1\dots w}[w] = \alpha_{1\dots(w-1)}[w-1] + \alpha[w](1 - \alpha_{1\dots(w-1)}[w-1]) \quad . \quad (\text{EQ 65})$$

For all of the above $w = 1..W$. Such an algebraic evaluation rule is called compositing [FOLE90] and is used in other graphics applications such as animation. The transparency equations ((EQ 62) and (EQ 63)) are more efficient than the opacity equations ((EQ 64) and (EQ 65)). When compositing back-to-front the incremental transparencies or opacities do not need to be maintained,

$$I_E[w] = I_S[w]\alpha[w] + I_E[w+1](1 - \alpha[w]), \quad w = W..1 \quad . \quad (\text{EQ 66})$$

Other combining rules have proven useful for volume rendering such as the maximum intensity, the sum of the two highest intensities $I_{\max_1} + I_{\max_2}$, or the standard deviation of the intensities [LAUB90]. Other simplifications include binary voxel rendering where the voxels are opaque or transparent [FRIE85][KAUF88].

4.2 Algorithm Development Methodology and Existing Approaches

Volume rendering is a transform based algorithm. Because of this, all view transform alternatives can be investigated in the directed graph paradigm of Chapter II. I separate the algorithm into three separate steps to clarify design choices, then classify existing ap-

proaches. An optimal sequential algorithm is described, after which parallel algorithms and their optimal approaches are discussed.

Volume rendering is concisely represented as the data parallel algorithm in FIGURE 57 with terms defined in TABLE 18. For each step points lie in: (Step 1) object space (OS) p , (Step 2) 3D screen space (SS) p' , and (Step 3) 2D screen space (SS) p'_w . The domain of points in each space is defined by bounding hulls B_V , $B_{V'}$, and B_{SS} . The three steps are: (1) the preprocessing stage (PPS), (2) the volume warping stage (VWS), and (3) the compositing stage (CS).

$$\begin{aligned}
 I_{\text{ray}}[] &\leftarrow \text{Transparency_Volume_Render}(V, \Gamma, T, \text{classify}, \text{shading}) \{ \\
 &\quad \vec{N}_p = \text{normal}(V) \\
 \text{PPS} \quad &\alpha_p = \text{classify}(V, \vec{N}_p) \quad \forall (p \in B_V) \\
 &I_{Sp} = \text{shading}(\vec{N}_p, \vec{E}_p, \vec{L}_{p\gamma}, \alpha_p, I_\gamma) \\
 \text{VWS} \quad &\alpha_{p'}, I_{Sp'} = T(\alpha_p, I_{Sp}) \quad \forall (p' \in B_{V'}) \\
 \text{CS} \quad &I_{p'_w} = \int_{p'_1}^{p'_2} t(l) I_{Sp}(l) \alpha(l) dl \quad \forall (p'_w \in B_{SS}) \\
 &\}
 \end{aligned}$$

FIGURE 57

Data Parallel Volume Rendering Algorithm

TABLE 18

Terms in algorithm

p	point in original volume space OS
\vec{N}_p	normal at point p in OS
$\vec{L}_{p\gamma}$	direction to light source at point p
α_p	opacity at point p in OS
I_{Sp}	shading intensity at point p
p'	point in volume screen space SS_{final}
$\alpha_{p'}$	opacity at point p' in SS_{final}
$I_{Sp'}$	resampled shading intensity at point p' in SS_{final}
T	transformation $OS \rightarrow SS_{\text{final}}$
$I_{p'_w}$	intensity of ray at point p'_w in 2D image space SS created from intensities, and opacities along the W ray at all points $p'_w \in W$.
B_V	bounding hull of volume data
$B_{V'}$	bounding hull of transformed volume data
B_{SS}	bounding hull of 2D screen space image

The PPS calculates normals \vec{N}_p , opacities α_p , and initial shaded intensities I_{Sp} . The VWS transforms the initial shading intensities I_{Sp} and the opacities α_p to the 3D screen

space by resampling. The CS evaluates the view ray line integrals to get the 2D screen space pixel intensities, $I_{p'_w}$.

FIGURE 58 shows the VWS and CS alternatives intermixed as transitions from three dimensional OS to two dimensional SS. Essentially calculation may be done forward, backward, multipass forward, surface fitting, or through Fourier techniques. The CS has different freedoms for sequential algorithms such as back-to-front (EQ 66) or front to back (EQ 62) to (EQ 65), and in parallel there are multiple ways of compositing. I discuss parallel compositing in Section 4.4 and in Appendix B, but first I explain and discuss the existing volume rendering approaches. Other surveys of volume rendering are [GOLD85][KAUF91][WILH91][ELVI92].

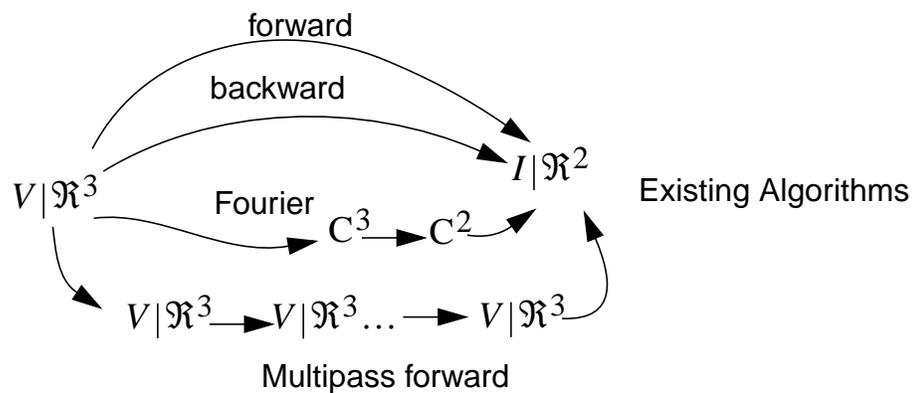


FIGURE 58

Volume Rendering Transform Graph

4.2.1 Backward Warping Algorithms-Ray Tracing

Ray tracing is a backwards transform algorithm. FIGURE 59 shows backwards warping, or ray tracing, where a ray passes from the eye through pixel p'_{xy} into the volume. The pixel intensity depends on the contributions of points p'_1 through p'_w [BLINN82][KAJI84][KRUE90][LEVO90][LEVO90d][SABE88]. The sample points $p'_1 \dots p'_w$ are reconstructed, typically by trilinear interpolation (such as in [LEVO89]). Reconstruction samples are

composed as described in the previous section in (EQ 62) and (EQ 63). Backward warping is also used in opaque voxel rendering [JACK88][KAUF88][WILH92].

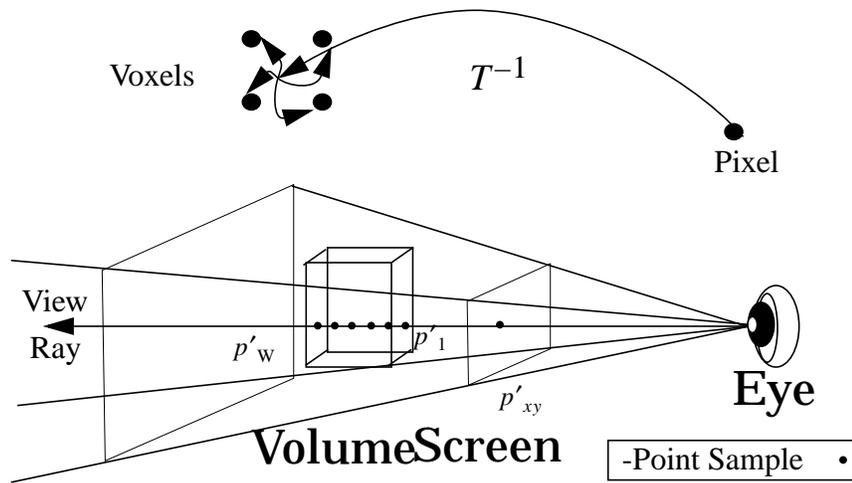


FIGURE 59 Viewing Frustum For Ray Tracing

Techniques to reduce the amount of work in ray tracing including hierarchical enumeration, adaptive termination, and bounding hulls. Hierarchical enumeration is the creation of multiple representations of the volume data so that empty or homogeneous areas are traversed quickly. If space is divided into octants, and the octants are subdivided into octants, a pyramidal structure is created. See FIGURE 60 below. Levoy uses this approach [LEVO90] and stores the multiple representations. See also [DANS92]. Meagher

[MEAG91] creates a single octree representation that holds all of the data, and uses forwards algorithms to display.

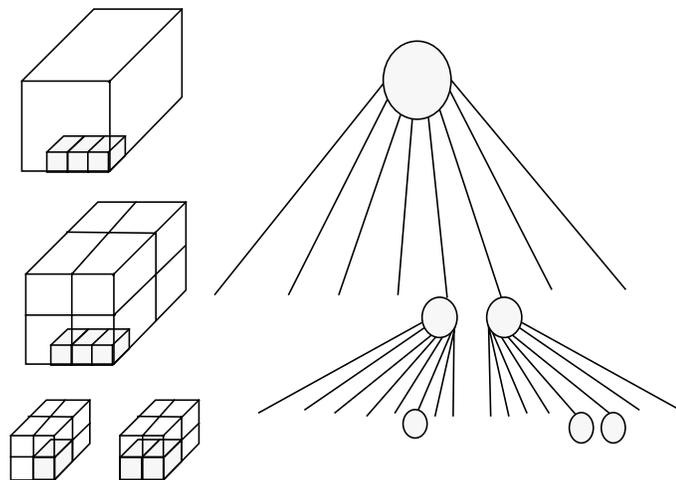


FIGURE 60

Octree Space and Graph Representation

Adaptive ray termination [LEVO90] stops processing a ray when the full opacity is reached. If you traverse the ray from front to back when the opacity of the ray has accumulated to a threshold you stop processing that ray. Danskin and Hanrahan have developed variance octrees and importance sampling for hierarchical enumeration and adaptive termination [DANS92]. Another speedup is bounding hulls that surround objects of interest. The hull is used to test ray intersections in deciding whether to continue processing the ray. For animation a space-time-hull further reduces rays [GLASS88]. Ray tracing provides for incorporating geometric primitives [LEVO90c] which are useful for three dimensional perception.

Parallelism has also been used to speedup backward mapping algorithms. Nieh and Levoy [NIEH92], Yoo et al. [YOO91], Challenger [CHAL92], and Montani et al. [MONT92] have developed parallel backwards algorithms. Nieh and Levoy use a shared memory machine (Stanford DASH) where arbitrary memory requests are satisfied by the system. Challenger [CHAL92] also uses a shared memory machine (BBN TC2000). Memory congestion and storage overhead are the primary disadvantages, but the architectural strength of the DASH gives nearly linear speedup. Yoo et al. implemented a backwards algorithm on Pixel Planes 5, a distributed memory machine, and because of network congestion elected to replicate the data set on every processor. This results in high performance (See TABLE 20), but limits the amount of data that can be rendered. Montani et al. encountered similar difficulties on the nCube, where clusters of processors get copies of the data set, and data must also be sent on request resulting in both memory limitations and network congestion. Backwards parallel algorithms require lots of storage or lots of random accesses of voxels. Yoo et al. and Nieh and Levoy reduce storage by using a compact-

ed 32 bit voxel representation, a grey scale 8 bit voxel value combined with 1 bit of octree information and a 2 byte compressed normal value.

4.2.2 Forward Algorithms-Compositing

Forward algorithms send volume elements into the screen. As discussed in Chapter II, forward algorithms are the opposite of the backwards algorithms, and one iterates over OS samples rather than SS samples. Approaches include the multipass forwards, forwards wavefront, and forwards splatting. Surface fitting transforms data forward also, but is discussed in 4.2.3.

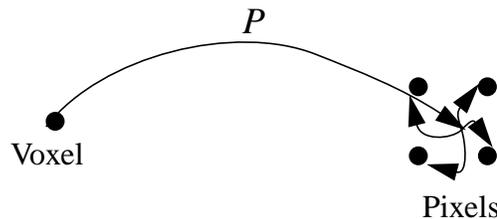


FIGURE 61 Forward Mapping of Voxels into Pixels

The *multipass forward* approach has been taken by [FRIE85][LENZ86][DREB88][HANR90][WEST90][SCHR91][CAME92][VEZI92][KABA92][WRIG92]. Parallel multipass forwards approach [DREBB88][SCHR91][VEZI92][KABA92][WRIG92] use a decomposition of the viewing transform into shears for low network congestion, but they suffer from lower filter quality and view angle restrictions. Essentially multipass algorithms cannot calculate the same quality or variety of images that backwards warping can.

The *forwards wavefront* approach [CAME92] works on SIMD machines with simple interconnection networks, such as a token ring [SCHR92], and gives better filter quality than the multipass methods. Limitations are similar to the multipass methods where view angles are restricted, and the filter quality is not as good as because of a post projection resampling. Perspective projection is not possible, and the technique suffers from network congestion as well [SCHR92].

Forwards splatting algorithms [WEST89][WEST90][LAUR91] have been described as easily parallelizeable. A voxel is splatted [WEST90] into the screen and a cumulative image is saved. Splatting techniques suffer from ordering noise because of the unavoidable overlap in the splatted kernels [WEST90][WILH91b][WEST92]. Similar to ray tracing, general viewpoints require random accessing of the screen, resulting in congested writes. An implementation without view angle freedom by Elvins [ELVI92] uses sequential compositing limiting speedup. Opaque forward algorithms do not have ordering problems, and are very efficient [MEAG84][MEAG91][GEME90].

4.2.3 Surface Fitting

Surface fitting matches a surface to voxels during segmentation and classification [GALL89][LORE87]. the technique is similar to the creation of contours from two dimensional data. Marching cubes [LORE87] creates a triangulated surface from an iso-density segmentation of the voxel data. The voxel's corners are inside or outside of the surface for 256 different possibilities (2^8). A look up table accessed by the 8 bit corner decision word specifies triangle(s) to generate. Due to the large number of triangles generated follow on processing may prune triangles from the representation [SCHR92][TURK92].

Saving previously calculated intersections as you march along, and even using reduced resolution volumes improves performance. The memory accessing is relatively efficient, because only 4 slices need be held in memory to compute the gradients. The resulting image triangles are rendered directly in a traditional graphics pipeline (such as a z-buffer and Phong shading pipeline.) Parallelizing marching cubes is straight forward by object space assignment. A recent study is [HANS92] which gives Thinking Machines CM-2 performance to create an isodensity surface of triangles from volumetric data. Hansen et al. do not discuss the rendering speeds. is classification of the point samples to isosurfaces. Surface fitting [WILH92][CLIN88] with points instead of triangles is more efficient (TABLE 19 [CLIN88]) but resolution is lost.

4.2.4 Reprojection and Fourier Volume Rendering

Reprojection, or Fourier volume rendering [MALZ91][DUNN90][LEVO92] creates 2D renderings from 3D frequency information. Through application of the Fourier slice theorem [KAK88] a plane of the volume's spectral (frequency) information creates a shadowgram of the entire 3D data set. This approach is fast because 2D frequency data creates 3D spatial information. There are similar techniques in the spatial domain [HARR78][JAFF82]. Chapter V more fully describes this approach.

4.2.5 Existing Methods Performance Summary

TABLE 19 lists special purpose architectures that use opaque voxel algorithms. Because these machines are determining isosurfaces, they compete directly with the surface fitting algorithms such as marching cubes [LORE87]. All of the special purpose machines, except the Insight system [MEAG85][MEAG91], use image space normal calculation which is inaccurate [TIED90]. High frame rates are achieved through volume size limitations and restricted voxel formats. The performance of each architecture is shown in frames per second. Performance of 10 f/s [OHAS85], 16f/s [GOLD85], and 35 f/s [KAUF88] are notable for parallel architectures, but higher quality shading with a uniprocessor architecture achieves 5 f/s [MEAG91] with newer technology. The results are hard to compare, but im-

proved algorithms have resulted in speedups while demand for more features has reduced performance.

TABLE 19 Opaque Voxel Algorithm Architecture Performance^a

	Transform	Shading	Perf.	Voxel	Prototype	Volume	Proc.
Cube [KAUF88]	backwards, orthogonal	image nor- mal	35 f/s	8bits	16 ³ mem- ory	512 ³	3
Insight II/ [MEAG84] [MEAG91] [GEME90]	forward	block and volume normal, depth cued	5 f/s, 250k pix- els/s	1-16 bits	Insight I, Insight II	512x512x 90, up to 80Mb octree	1
PARCUM II [JACK88]	backward	image nor- mal, diffuse and specular	1/38 to 1/110 f/s	1, or 8bit	MC68020 emulation	512 ³ 1bit, or 256 ³ 8bit	4
Voxel Proces- sor (GODPA) [GOLD85]	forward, arb. rotation and scaling	image nor- mal	16 f/s	4 bit	64 ³	256 ³	64 + 8+ 1 ^b
3DP ⁴ [OHAS85]	forward, per- spective	image nor- mal	10 f/s, estimated	app. depen- dent	software simulated	256 ³	256+ 255+1
iso-surface generation with octrees [WILH92]	backward/ forward to render	NA	(1/4.5 + polygon rend.) f/s	32 bits	Sparc 1	256x256x 113	1
point alg. [CLIN88]	forward/sur- face	volume normal, smooth	.2-.5 f/s	Unkn own	on GE 9800 scanner	64x64x93 to 256x256x 93	1
[MONT92]	backward	Unknown	0.1945f/s	Unkn own	NCube-2 Model 6410	97x97x11 6 to 350x250	128
[HANS92]	surface	NA	.14-6.8 conver- sions/sec- ond	NA	CM-2,	64 ³ -256 ³ to trian- gles	65536

a. Partially adapted from [KAUF90] which is also published as [KAUF91b]

b. Denotes multiple types of processors 64 of type 1, 8 of type 2, and 1 controller

With the need for more sophisticated shading and lighting models grew the need for more processing power. Performance studies have been almost entirely on general purpose and graphics machines. I list in TABLE 20 parallel and sequential performance numbers for transparency rendering. Direct comparison of results is difficult, because of the variation in shading, data set sizes, resolution of voxels and images, and generality of algorithms.

TABLE 20 Transparency Voxel Algorithm Architecture Performance

	Transform	Shading	Perf.	Voxel	Prototype	Volume	Proc.
[KAJI84]	Spherical Harmonic	High albedo	7×10^{-5} to 2.8×10^{-4} f/s	Unknown	VAX 11/780, IBM 4341	16^3 , $128 \times 128 \times 16$ to 512^2	1
[LEVO90]	backwards	phong	.008-.33 f/s	32 bits	Sun 4/280	$72 \times 60 \times 33$ to 256^2 , 256^3 to 512^2	1
[DREB88]	forward, multipass	phong	.017 f/s ([LEVO90d])	32 bit r,g,b, α	Pixar Computer	256^3	4 SIMD
[LEVO90d]	backwards	volume normal, phong	.17 -.017 f/s (1-10f/s)	8 bytes	DEC 3100/Sun 4/280	$256 \times 256 \times 128$, 512^2 image	1
[SCHR91]	forward, multipass (pure shear)	Unknown	.609-6.32 f/s	Unknown	CM-2	64^3 - 128^3 to, 64^2 - 256^2	16k - 64k SIMD
[NIEH92]	backward	Phong	1.18-11.1 f/s	32 bits	Dash	128^3 to 209^2 , 256^3 to 416^2	48 MIMD
[VEZI92]	forward, multipass (scale shear)	unknown	.288-11.6 f/s	Unknown	MasPar MP-1	32^3 to 32^2 , 256^3 to 256^2	16384 SIMD
[YOO91]	backward, trilinear	Phong	15 f/s	32 bits ^a	Pixel Planes-5	128^3 to 640×512 images	16 MIMD
[YOO91]	backward trilinear	Phong	1.4 f/s	64bit, color	Pixel Planes-5	128^3 to 640×512 images	16 MIMD + SIMD

TABLE 20

Transparency Voxel Algorithm Architecture Performance

	Transform	Shading	Perf.	Voxel	Prototype	Volume	Proc.
[CAME92]	forward wavefront	unknown	10 f/s	unknown	DAP 510	128x128x64	1024 SIMD
[CHAL92]	backward	unknown	.084 f/s	NA	BBN TC2000	100x120x16 to 512 ²	100 MIMD
[CHAL92]	forwards	unknown	0.38 f/s	NA	BBN TC2000	100x120x16 to 512 ²	100 MIMD
[SCHR92b]	wavefront	pre shaded	1.09-17.9 f/s	grey scale, color	CM-2	64 ³ and 128 ³	16384, or 32768 SIMD
[SCHR92b]	wavefront	pre shaded	2.58-35.7 f/s	grey scale, color	Princeton Engine	128 ³ and 256 ³	1024 SIMD
[STRE92]	forward	grey scale	42-.18f/s	unknown	Cray YMP-8	64 ³ -512 ³	8 MIMD
Chapter IV	permutation warping	max	.17 - 3.44 f/s ^b	8 bit	Proteus	32 ³ -256 ³ to 256 ²	16 MIMD
Chapter IV	permutation warping	max	.71 - 6.66 f/s ^c	8 bit	Proteus	32 ³ -256 ³ to 256 ²	32 MIMD
Chapter IV	permutation warping	max	.89-155 f/s	8 bit	MasPar MP-1	32 ³ -256 ³ to 32 ² -256 ²	16384 SIMD

a. 8 bit gradient magnitude, 13 bit normal, 1 bit octree

b. No preprocessing or data structure optimization

c. No preprocessing or data structure optimization

Even with parallelism, performance is only up to 15 f/s [YOO91] using restricted volume sizes and voxel formats. Higher rates can be achieved with restricted viewpoints [SCHR92b], but achieving higher performance requires simply more processing power. I show in Sections 4.5.1 and 4.5.2 algorithms that allow full linear speedup with the flexibility of the backwards warping algorithms. I call them permutation warping because they use non conflicting communication. Sequential algorithms are discussed briefly next.

4.3 Optimal RAM Volume Rendering Algorithm

The optimal RAM volume rendering algorithm has been shown by Malzbender to be Fourier volume rendering $O(R \log R)$ [MALZ91]. The output image is R rays, the volume is $S = \text{rows} \times \text{cols} \times \text{slices}$ samples, and the number of samples along each view ray is w . But Fourier volume rendering does not allow surface shading. The optimal spatial volume rendering algorithm is Levoy's $O(S)$. Assume that $S \approx RW$. In fact forward, backward, and surface fitting are all $O(S)$ because they require visiting all voxel points. The only additional savings are data dependent including the hierarchical data sets, adaptive termination, and bounding hulls mentioned earlier. So a simple straightforward RAM algorithm is given in FIGURE 57, iterating over each volume sample point to shade, warp, and composite. The forward or backward approach is better depending on the data set size. See Chapter 3 for clarification of the optimal RAM warping approaches. [WILH91] Suggests that ray tracing is easier to implement, but splatting is slightly more efficient. Of course splatting is inaccurate in the accumulation of opacities because of ordering problems.

4.4 Optimal PRAM Volume Rendering Algorithm

I examine each of the three steps of the data parallel algorithm to explain the PRAM efficiency. The *PPS* is the calculation of opacities and of the initial volume intensities, I_S . These operations take constant time and therefore are $O(S)$ for the RAM and for the PRAM opacities and shaded intensities are calculated in parallel once the normals are available. Since preprocessing consists of simple point operations (P) processors divide up the work to $O(S/P)$ for $1 \leq P \leq S$, which is $O(1)$ for $P = S$.

The complexity for warping is also $O(1)$ for constant factor sized neighborhoods, and $O(nd)$ for maximal sized windows where n is the order of reconstruction. Constant order filters are quite accurate and therefore the PRAM complexity is typically $O(1)$ to warp, but can reach $O(n^d)$ for the most accurate filters. Fast high order filtering requires $n(n+1)^{d-1}$ processors per voxel, where d is the dimension of the image, and n is the order of interpolation in each dimension. Further speedup by assigning more processors to each sample point does not result in further linear speedup, because of the mesh of trees

[LEIG92] structure of the filter. FIGURE 62 shows the diminishing speedup return as more processors are added.

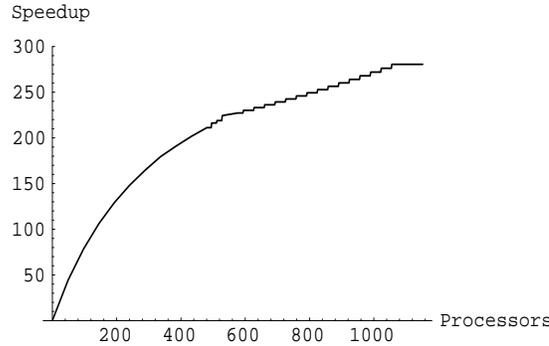


FIGURE 62 speedup as the number of processors is increased from $P = 1$ to 32×36 for an $n = 32$ order interpolation, $d = 2$.

The number of steps in the parallel evaluation of the filter is,

$$\# \text{par timesteps} = \sum_{j=0}^{d-1} \sum_{i=1}^n \left\lceil \frac{(n+1)^{j_i}}{P} \right\rceil \quad (\text{EQ 67})$$

Using the number of interpolations, Chapter III (EQ 16), and the number of time steps, the speedup can be calculated,

$$\text{Speedup} = \frac{\# \text{int}}{\# \text{par timesteps}} \quad (\text{EQ 68})$$

The amount of work being done is the number of interpolations. The efficiency is the work divided by the product of the time and number of processors,

$$\text{Efficiency} = \frac{\# \text{int}}{\# \text{par timesteps} P} \quad (\text{EQ 69})$$

The efficiency drops off as more and more processors are idled. The warping stage can be speeded up using more processors through mesh of trees calculation, but the returns decrease. Filters used in Chapter III and in this Chapter are the zero order hold and first order hold, and will be assumed constant complexity with one processor per sample.

The compositing stage, CS, is calculated by a general technique of *parallel product* [KRUS85][LEIG92]. The run time complexity for data independent (data locations are known before processing begins) is $O(\log W)$, and for $P_z = O(W/\log W)$ the product calculation is work efficient. To apply the parallel product algorithm to compositing, I prove that compositing is an associative operator in *Lemma 4.1*.

To simplify the notation in the proof I change briefly to subscripts, where emitted intensities are, $I_{E_w} = I_E[w]$, the initial shading intensities are $I_{S_w} = I_S[w]$, and the initial

opacities are $\alpha_w = \alpha[w]$. I use the classified opacity labelled at each level α_i . (EQ 62) and (EQ 63) in a more concise subscript notation to composite two images i and j are,

$$I_{Ej} = I_{Sj}\alpha_j \quad (\text{EQ 70})$$

$$I_{Eij} = I_{Ei} + I_{Ej}t_i \quad (\text{EQ 71})$$

$$t_{ij} = t_i(1 - \alpha_j). \quad (\text{EQ 72})$$

As before α is the opacity, t is the transparency, I_{Eij} is the combined emitted intensity, and I_{Sj} is the shading intensity at level j . I now prove that compositing by (EQ 70), (EQ 71), and (EQ 72) of three images by any associative groupings is the same. A short hand for compositing is $I_{ij} = I_i \text{over} I_j$ [DREB88].

Lemma 4.1: Compositing is associative $(I_{E1} \text{over} I_{E2}) \text{over} I_{E3} = I_{E1} \text{over} (I_{E2} \text{over} I_{E3})$.

Proof: Given initial opacities, α_i , and shading intensities, I_{Si} , the initial emitted intensities and transparencies are,

$$\begin{aligned} I_{E1} &= \alpha_1 I_{S1} & I_{E2} &= \alpha_2 I_{S2} & I_{E3} &= \alpha_3 I_{S3} \\ t_1 &= (1 - \alpha_1) & t_2 &= (1 - \alpha_2) \end{aligned} \quad (\text{EQ 73})$$

Two associative groupings for compositing 3 images are

$$\begin{aligned} (I_{E1} \text{over} I_{E2}) \text{over} I_{E3} & \quad ? \quad I_{E1} \text{over} (I_{E2} \text{over} I_{E3}) \\ A & \qquad \qquad \qquad B \end{aligned} \quad (\text{EQ 74})$$

Evaluation of the transparencies and intensities of terms A and B shows that they are equal,

A ,

$$I_{E12} = I_{E1 \text{over} 2} = I_{E1} + I_{E2}t_1$$

$$t_{12} = t_{1 \text{over} 2} = t_1 t_2$$

$$I_{E123} = I_{E12 \text{over} 3} = I_{E1} + I_{E2}t_1 + I_{E3}t_1 t_2$$

B ,

$$I_{E23} = I_{E2 \text{over} 3} = I_{E2} + I_{E3}t_2$$

$$I_{E123} = I_{E1 \text{over} 23} = I_{E1} + (I_{E2} + I_{E3}t_2)t_1$$

$$= I_{E1} + I_{E2}t_1 + I_{E3}t_1 t_2$$

$$\therefore I_{E(1 \text{over} 2) \text{over} 3} = I_{E1 \text{over} (2 \text{over} 3)} \quad \blacksquare$$

For more layers of images compositing is associative by induction on Lemma 4.1. Because compositing is not commutative it must be evaluated in a strict order, but it is an acceptable operation for parallel product. Last one tries to relax the order of evaluation (as

is done in some splatting techniques [WEST90] [WILH91b]) consider the following constraint,

Lemma 4.2: Compositing is not commutative $(I_1 \text{ over } I_2) \neq (I_2 \text{ over } I_1)$.

Proof:

$$I_{E12} = I_{E1 \text{ over } 2} = I_{E1} + I_{E2}t_1$$

$$I_{E21} = I_{E2 \text{ over } 1} = I_{E2} + I_{E1}t_2 = I_{E1}t_2 + I_{E2}$$

$$\therefore I_{E(1 \text{ over } 2)} \neq I_{E(2 \text{ over } 1)}$$

In $(I_{E1} \text{ over } I_{E2})$ I_{E2} is attenuated by t_1 and in $(I_{E2} \text{ over } I_{E1})$ I_{E1} is attenuated by t_2 . The results are not the same, therefore compositing is not commutative. ■

Theorem 4.1: Parallel compositing is $O(\log W)$ and sequential compositing is $O(W)$, where W is the number of sample points along a view ray.

Proof: $I_{\text{ray}} = I_{E1 \text{ over } I_{E2 \text{ over } I_{E3 \dots I_{EW}}}}$ by Lemma 4.1 can be combined through any associativity. Assign 2 sample points to each processor, composite, and the number of points is halved. Continue this process of halving the number of sample points until the final ray intensity is calculated. (See FIGURE 63.) The time complexity is the depth of the tree which is $\log W$. If done sequentially there are $W - 1$ compositing evaluations or $O(W)$. ■

Constant factors for additions and multiplications using binary tree compositing are $3W - 2 - \lceil \log W \rceil$ multiplications and $2W - 2$ additions to take α_i , and I_{Si} to I_{ray} when W is a power of 2. If there are not a power of 2 image levels, then the tree may be balanced to the back edge to reduce the number of incremental transparencies calculated. The most efficient sequential method, ignoring data dependent optimizations is back-to-front where

no incremental transparency or opacity updates are performed giving $2W - 1$ multiplications and $2W - 2$ additions. (See Appendix B for details.)

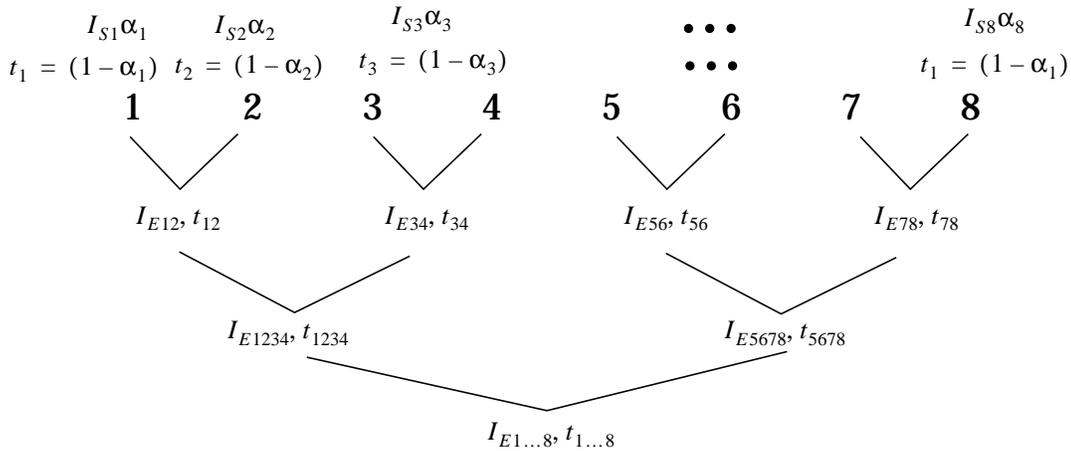


FIGURE 63 Fully Parallel Compositing

Spatial volume rendering, as FIGURE 57 shows, requires combining of all factors along the ray (CS). No ray surface intersections are found. Compositing is not usually computed in parallel [DREB88][LEVO90b][SCHR90]. If there are enough processors, all rays are computed in parallel, and the intensities and opacities can be calculated in parallel. Special purpose architectures have been proposed for parallel compositing [FOLE90][MALZ90]. Several general purpose machine's interconnection networks perform parallel product [THIN89][BLAN90].

Theorem 4.2: Parallel Volume Rendering is an optimal parallel algorithm by definition 6, 7, 8, and 9 (Chapter II) for $P = O(S/\log S)$ processors on CREW and EREW PRAMs.

Proof: The preprocessing stage is point operations, reads require only neighboring data which is accessed in directional phases. Time is $O(S/P)$, for S sample points.

By Theorem 3.1 warping is calculable with exclusive reads and is $O(S/P)$ for rigid body transforms. By partitioning object space subcubes are warped in data parallel or overlap fashion. If rigid body transforms are used the volume's extents remain constant and the overlap of any source subcube in image space subcubes is fixed to at most 12 other cubes. Either a constant number of subcubes are intersected (overlap approach), or the collision of messages is restricted to a small constant (data parallel approach).

Subcubes are then composited local to each processor for $O(S/P)$ work. The parallel product then operates on local frames. The local frames must be combined through a parallel product evaluation. The number of samples, starting with $S_{\text{frame}} = R/(P_x P_y)$ samples at each processor, is halved at each increment, FIGURE 64. The run time differs de-

pending on the number of processors, either $1 < P \leq R$ and $R < P \leq RW$. The total work is always $R(P_z - 1)$, where P_z is the ray sampling or initial number of subframes.

For $1 < P \leq R$ each processor remains busy for all compositing and time is $O(R(P_z - 1)/P) = O(S/P)$. For $R < P \leq RW$, processors are idled at some point during compositing. Two terms are one for all processors busy which equals $S_{\text{frame}} - 1$. The subframes are halved until there is one sample in the subframe. There are $\log P_z - \log S_{\text{frame}}$ remaining composites, enough single sample composites to combine all P_z samples. The time for compositing the subframes is $S_{\text{frame}} - 1 + \log P_z - \log S_{\text{frame}}$ for $S_{\text{frame}} < P_z$.

The total compositing time when $R < P \leq S$ is $O(S/P + \log P_z / S_{\text{frame}})$ which equals $O(S/P + \log P)$. Compositing achieves linear speedup for $P = O(S/\log S)$.

Each stage PPS, VWS, and CS is $O(S/P)$ for $P = O(S/\log S)$, therefore linear speedup is achieved over the fastest sequential algorithms which are $O(S)$ (Definition 6, Definition 9). $O(S)$ storage is used for optimal space complexity (Definition 8). The lower bound for computation on an EREW PRAM is $\log n$ for n inputs and the run time of the fastest algorithm $P \geq S$ is $\log W$ which is below the lower bound (Definition 7). ■

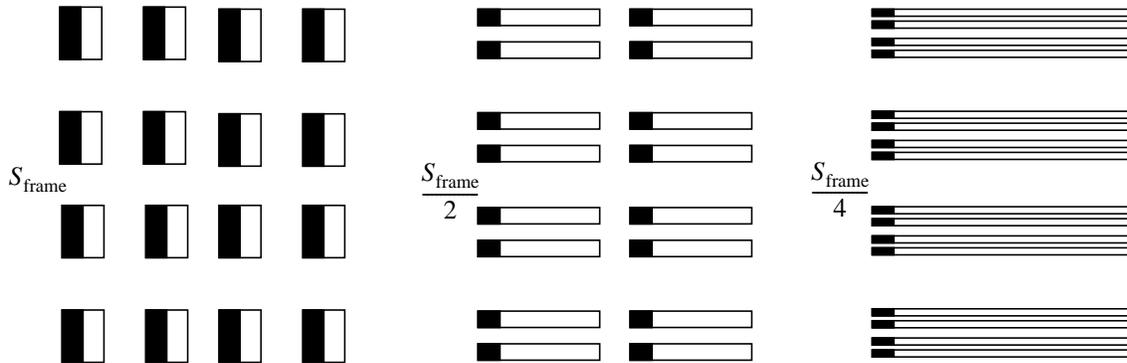


FIGURE 64 Halving of Frames During Parallel Product for Compositing

FIGURE 65 summarizes the overall complexity. Here I show with varying amounts of virtualization, the speedup. Because all three steps of the algorithm parallelize ideally from $P = 1$ to $O(S/\log S)$ processors there is linear speedup. Parallel product thresholds at $O(\log W)$ shown for $P = S \cong RW$. Using more and more processors does allow more accurate filters for reconstruction.

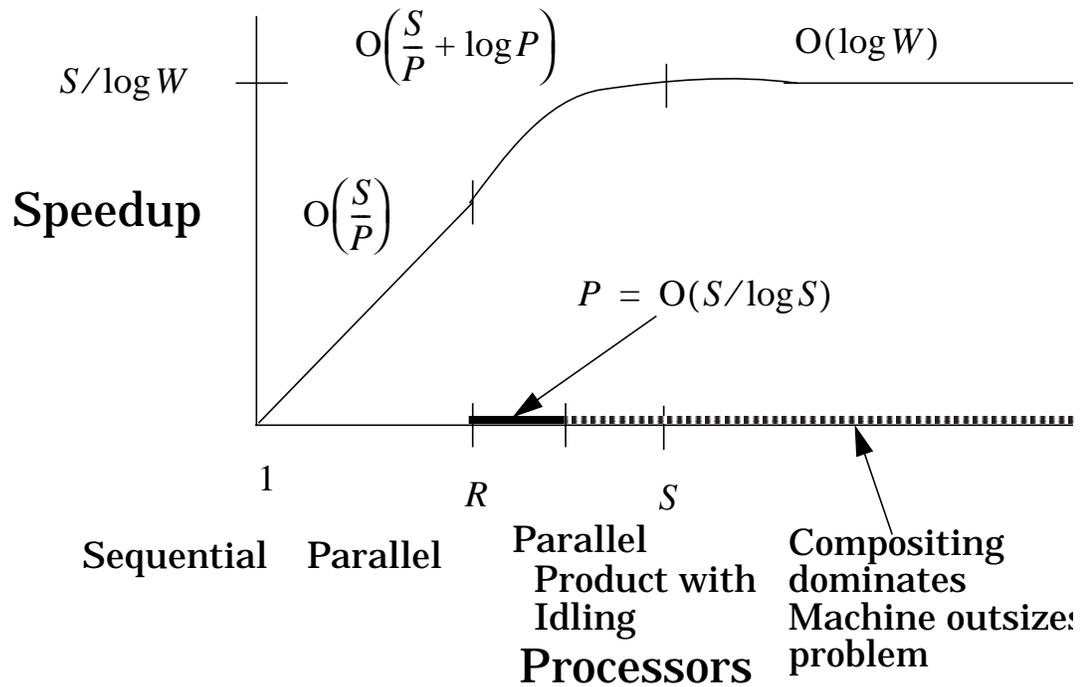


FIGURE 65

Overall Volume Rendering Complexity

The amount of parallelism for the volume rendering algorithm can be broken down into 4 regions. The first region is parallel $1 < P \leq R$. Processors are assigned subcubes that they preprocess, warp, and composite. Each processor stays busy through the parallel product calculations. Run time is $O(S/P)$.

The second region is work efficient parallel product $R < P < S$, $P = O(S/\log S)$. Now processors are idled during the final steps of compositing. When $P = S$ half of the working processors are idled upon each step. Run time is $O(S/P + \log P)$.

The third region is non work efficient because processors become idled during compositing and efficiency starts to drop off of linear speedup, $O(S/P + \log P)$.

The fourth region is fully parallel with $P \geq S$ providing the fastest algorithm possible, with multiple processors per voxel, but compositing dominates so run time is $O(\log W)$, determined by the number of sample points along a ray.

4.5 Permutation Warping For Parallel Volume Rendering

Permutation warping is essentially a processor assignment technique that provides a general approach for efficient parallel transform algorithms. Permutation warping is better than prior parallel algorithms because it is simultaneously memory efficient, processor efficient, general, and accurate. The algorithm (FIGURE 66), calculates the same image as FIGURE 57, but gives specific memory layout and communication requirements necessary for the EREW PRAM. Processors $\pi[r, s, t] \in \mathbb{N}$ (natural or whole numbers are assigned sample points $p[x, y, z] \in \mathfrak{R}$ requiring $P = S$ processors. Processor locations are on a whole number lattice, and samples are a discrete sampling $[x, y, z]$ of space (u, v, w) .

```

 $I_{\text{ray}}[ ] \leftarrow \text{Permutation\_Volume\_Render}(V, \Gamma, T, \text{classify}, \text{shading}) \{$ 
  1.0) PPS, calculate  $\alpha_p, I_{Sp}$ .
  2.0) VWS, Processor  $\pi$  does:
    2.1) Calculate processor assignments  $\pi' = M(\pi)$ 
    2.2) Calculate reconstruction point  $p_{\pi'} = T^{-1}(p'_{\pi'})$ 
    2.3) Perform resampling of  $\alpha_p$  and  $I_{Sp}$ 
    2.4) send resampled values to SS processors  $\pi'$ 
  3.0) CS, calculate ray intensities  $I_{\text{ray}}$  with parallel product.
 $\}$ 

```

FIGURE 66

Permutation Warping Parallel Volume Rendering Algorithm

Step 1.0 is the same as the PPS in FIGURE 57. Processors classify and shade by reading necessary neighboring data in directional phases, east, northeast, north, northwest, etc.

In Step 2, the view transform T points p to points p' by $p' = T(p)$. Resampling is required because the discrete rays do not line up with transformed voxels, $p'[x', y', z'] \notin N$. This results from allowing general viewpoints. Either viewpoints can be restricted, or any viewpoint can be supported by randomly accessing voxels $p[i, j, k]$ that surround the inversed SS point $p(x, y, z) = T^{-1}(p'[r, s, t])$. This backwards (ray tracing) solution is done by the warping from Chapter III. The assignment is guaranteed to be one-to-one, Theorem 3.1 Chapter III. FIGURE 67 illustrates the transformations calculated by a single processor. The OS and SS are separated, the OS on the left and the SS on the right. First the processor $\pi[r, s, t]$ shown as a circle in the OS lattice calculates who to resample for in Step 2.1 of the algorithm (FIGURE 66). The result is labelled processor $\pi'[r', s', t'] = M(\pi[r, s, t])$ and the logical connection is shown by the dotted line in FIGURE 67. Next, processor π calculates the inversed point position of π' , $p_{\pi}(x, y, z) = T^{-1}(\pi[r', s', t'])$ in Step 2.2. The inverse transformation is shown by a solid

line labelled T^{-1} . The point at which to perform reconstruction is $p_{\pi'}(x, y, z)$ and is shown as an asterisk * in FIGURE 67.

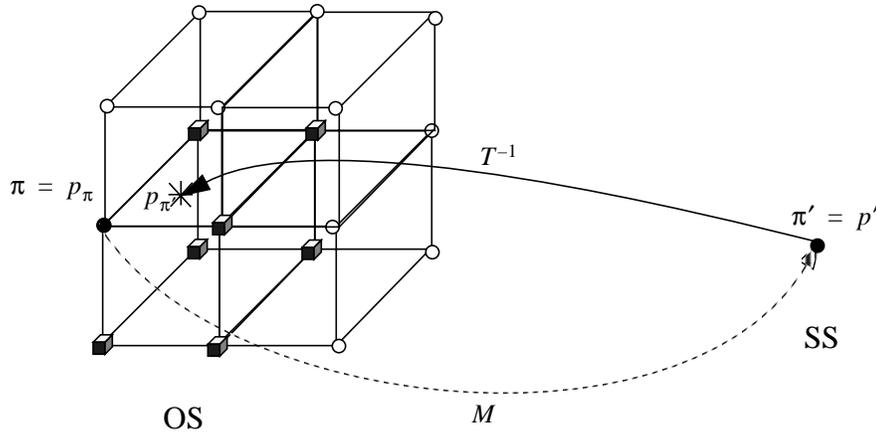


FIGURE 67 Transformations and Communications in Permutation Warping for a Single Voxel

Processor π reads the values of I_{S_p} and α_p of its neighboring processors. The number of neighbors used determines the filter order. A zero order hold reads only one neighboring voxel, the closest one to $p_{\pi'}$. A first order hold, or trilinear interpolation, reads 7 neighboring voxels surrounding $p_{\pi'}$, and FIGURE 67 shows processor π 's 7 neighbors as cubes. To avoid conflicts each processor reads in directional stages.

The final step in the VWS, Step 2.4, is sending the reconstructed values to π' , by an explicit send in the MCCM $\pi[r, s, t]$ to $\pi'[r', s', t']$ or by writing to a unique memory location in the EREW PRAM.

To understand the advantage of this extra work FIGURE 68 shows all communications taking place in parallel. There are no conflicts. The OS processor bounding box is green, and the forward T warped version is also given as green in the SS. The SS processor bounding box is red in both spaces. Of course processors are both OS processors π and SS processors π' with $\pi[r, s, t] = \pi'[r, s, t]$. This is shown by those processors who interpo-

late for themselves, the blue processors in the interior where communications arcs are not drawn.

color photograph inserted

FIGURE 68 Volume Transformations in Parallel

Also, see FIGURE 69 where the OS and SS have been properly merged, and the communication that is taking place can be seen to be nontrivial. The Viewing transforma-

tion is a rotation of the cube by 15 degrees about \hat{y} and 15/2 degrees about \hat{z} . Permutation warping calculates a one pass resampling.

color photograph inserted

FIGURE 69

Transformation with OS and SS Merged

The final step in the algorithm, step 3, combines resampled intensities and opacities using parallel product evaluation. Binary tree combining computes products for any associative operator (\otimes), $I_1 \otimes I_2 \otimes \dots \otimes I_w$ [KRUS85]. Compositing (I_i over I_j) is associative. Numerical integration is also associative.

The processor assignment calculated by $\pi' = M(\pi)$ works for any equiareal transform, $\det(T) = \pm 1$. The equiareal transform is,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & b_{12} & b_{13} \\ 0 & 1 & b_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ c_{21} & 1 & 0 \\ c_{31} & c_{32} & 1 \end{bmatrix} \begin{bmatrix} 1 & d_{12} & d_{13} \\ 0 & 1 & d_{23} \\ 0 & 0 & 1 \end{bmatrix}. \quad (\text{EQ 75})$$

(EQ 75) is solved symbolically for coefficients (b_{12} , b_{13} , b_{23} , c_{21} , c_{31} , c_{32} , d_{12} , d_{13} , and d_{23}) in Chapter III. Note that the decomposed matrices are not used for a multipass resampling, only to calculate the permutation, M . $M = \overline{M}_c \overline{M}_1 \overline{M}_2 \overline{M}_3 \overline{M}_r = (\overline{M}_1 + \text{Tran}) \overline{M}_2 (\overline{M}_3 + \text{Tran})$. where Tran is a composed pretranslation and postranslation so only three rounding steps occur for a total of seven rounds for any equiareal transform.

The inverse used in determining the reconstruction point is numerically stable, because equiareal transforms are by definition invertible. For arbitrary centered transforms use a product of translation matrices, $\text{Tran}(x, y, z)$, and the equiareal transform T . Transform about the point (r_x, r_y, r_z) and center the transform about (c_x, c_y, c_z) .

For arbitrary centered rotations the inverse T^{-1} is easily calculated because rotation $R(\psi, \phi, \theta)$ is orthogonal, meaning $T^{-1} = T^T$, and translations are inverted by negating their values,

$$\begin{aligned} T^{-1} &= (T(c_x, c_y, c_z)R(\psi, \phi, \theta)T(-r_x, -r_y, -r_z))^{-1} \\ &= (T(-r_x, -r_y, -r_z))^{-1}(R(\psi, \phi, \theta))^{-1}(T(c_x, c_y, c_z))^{-1} \quad \text{(EQ 76)} \\ &= T(r_x, r_y, r_z)(R(\psi, \phi, \theta))^T(T(-c_x, -c_y, -c_z)) \end{aligned}$$

The rotation matrix and a translation matrix are given in (EQ 77) and (EQ 78), and the transpose of (EQ 77) is composed with the translations for calculating the inverse with the minimum number of calculations.

$$R(\psi, \phi, \theta) = \begin{bmatrix} (\cos \phi \cos \psi) (-\cos \theta \sin \psi + \cos \psi \sin \phi \sin \theta) (\cos \psi \cos \theta \sin \phi + \sin \psi \sin \theta) & 0 \\ (\cos \phi \sin \psi) (\cos \psi \cos \theta + \sin \phi \sin \psi \sin \theta) \cos \theta \sin \phi \sin \psi - \cos \psi \sin \theta & 0 \\ (-\sin \phi) (\cos \phi \sin \theta) (\cos \phi \cos \theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(EQ 77)}$$

$$T(c_x, c_y, c_z) = \begin{bmatrix} 1 & 0 & 0 & c_x \\ 0 & 1 & 0 & c_y \\ 0 & 0 & 1 & c_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(EQ 78)}$$

4.5.1 Data Parallel Virtualization

To apply permutation warping without a processor for each sample point can be thought of as v virtual processors running on $P < v$ processors. With current technology an algorithm designer has $1 \leq P \leq RW$ processors, RW is the number of rays times samples on each ray. I have found that a data parallel approach uses permutation warping very efficiently.

I conserve as much storage as possible and calculate the correct image. The constant space requirements are $S + R/P^{1/3}$ for $1 \leq P \leq RW$ where P is the number of processors. When the number of processors equals the number of SS samples $S' = RW$ the virtual algorithm becomes the nonvirtual algorithm FIGURE 66.

To virtualize I make an assignment of P processors to the S voxels. If $P > S$ then samples are replicated to several processors. I evaluate the rule only once per processor so the cost of M is amortized over the virtual sub volumes. The assignment maintains the

communication efficiency. OS voxel points are assigned to processor id's by an address tiling. Address tiling in three dimensions is an extension of two dimensional tiling techniques [BLIN90] [WITT91]. FIGURE 70 shows how to calculate the tiled version of a slice, row major addressed volume, or $\langle t|k|r|i|s|j \rangle \rightarrow \langle t|r|s|k|i|j \rangle$

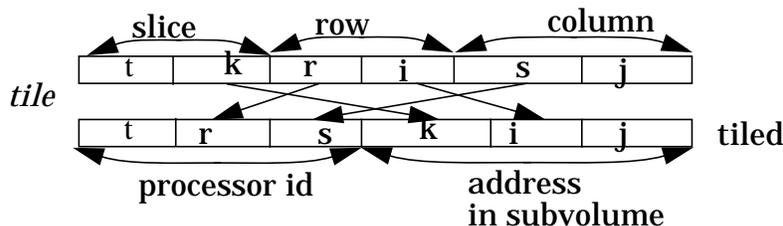


FIGURE 70

Three Dimensional Tiling To Calculate Processor Identification and Subvolume Addresses from (row, column, slice) Coordinates.

Such virtualization is amenable to a wide variety of architectures such as mesh [BLAN90], hypercube [SOMA91], and multistage interconnection networks. FIGURE 73 shows how machines with 16 processors are virtualized into a three dimensional volume. Each dimension gets approximately $P^{1/3}$ cuts.

The algorithm is the same as that in FIGURE 66, except now processors have more points to iterate over, S/P points each. In step 1, 2, and 3 a for loop is added for S/P points do, and the only challenging part is that the screen space assigned to each processor shrinks after each parallel product evaluation so they all remain assigned. FIGURE 72 shows the processors start with 1/4 of the screen, then get 1/8, and finally 1/16. The SS samples being calculated are unique, and in the EREW PRAM there will be no conflicts, but because of virtualization processors may receive more than one message. The density of messages across the network is the same if the slice and dice virtualization is used and communication remains efficient.

The data parallel algorithm requires storing an entire resampled volume. The source volume is not duplicated, but a 1 voxel overlap may be stored at each processor to avoid any local communication for the warping stage. In the worst case all resampled points are communicated, but in any communication period there is small congestion, and for any filter, only the final interpolated point is sent. All compositing takes place in the SS processors along preferred communication directions of the architecture. When there are enough processors for every data point the communication is one-to-one so this algorithm scales smoothly for machines which support fine message granularity. The algorithm uses exclusive reads and writes or a small amount of communication congestion.

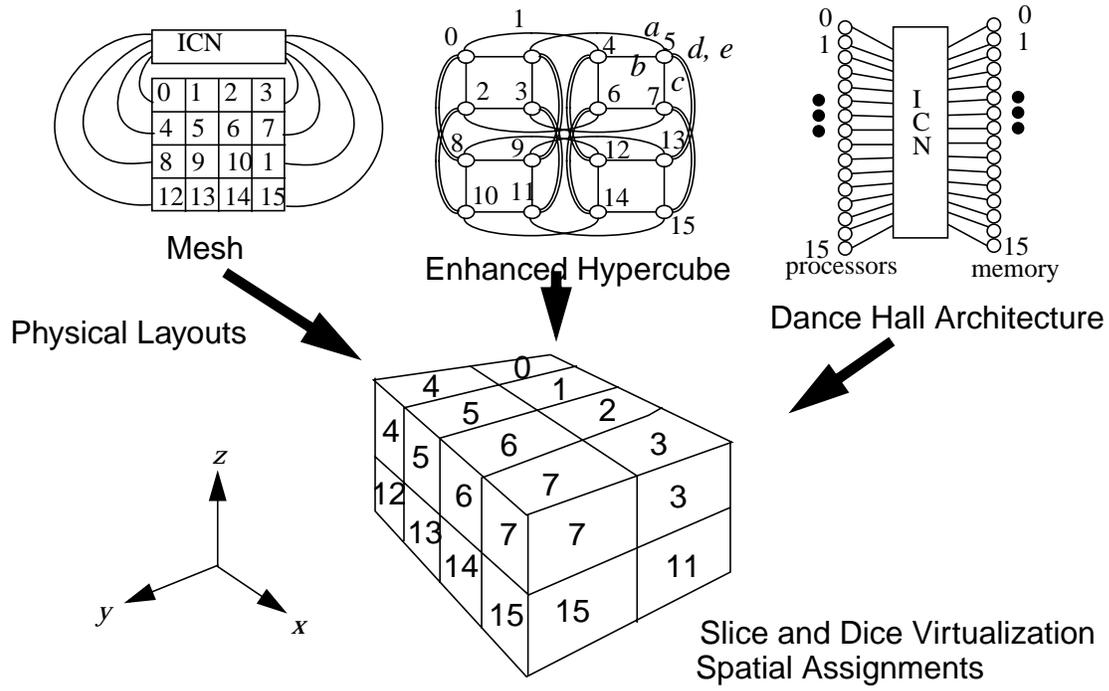


FIGURE 71

Spatial Volume Virtualization For a Variety of Architectures

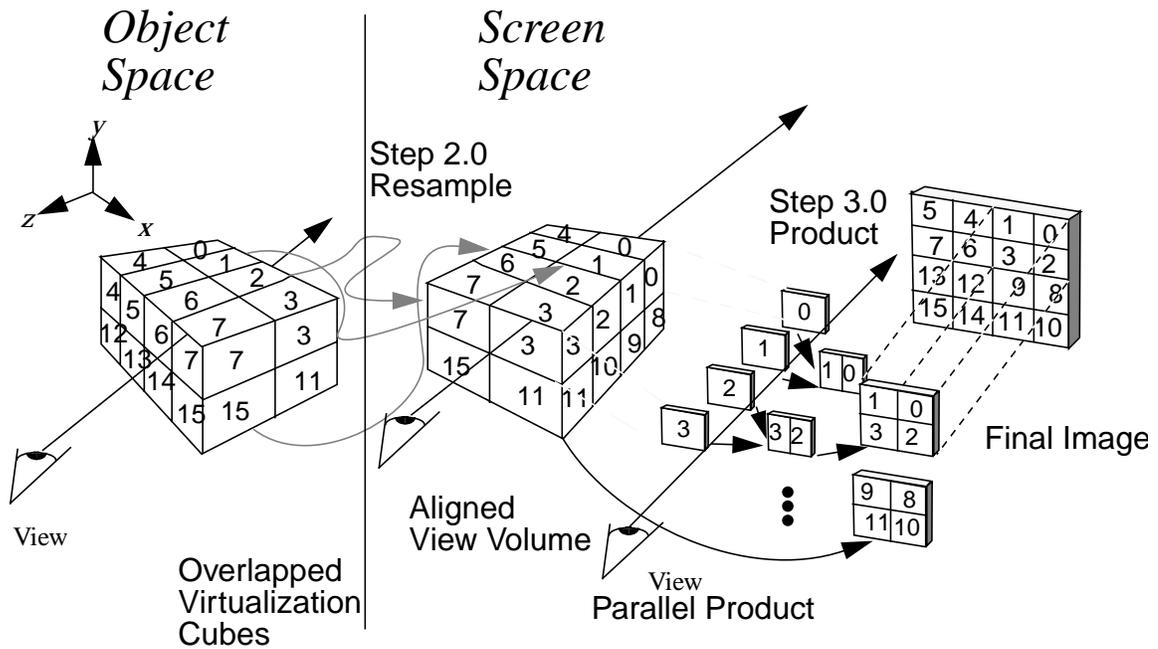


FIGURE 72

Steps of Virtual_Permutation_Volume_Render,
Virtualized SubVolumes to SubFrames to Final Image

4.5.2 High Granularity Virtualization

In high granularity machines such as Proteus [SOMA91] a message for each resampled point cannot be sent efficiently. I use instead an algorithm that takes advantage of the high virtualization ratio and sends large messages. There is no overlapped storage, and each processor completely render's it's subframe to the aligned SS, then subframes are sent to SS processors. Each processor can overlap at most 18 SS subvolumes. Then 18 messages can be sent. For Proteus, there are currently only 8 clusters of 4 Intel i860's. The volume is partitioned eight ways, and the screen is partitioned eight ways. There are only seven messages to be sent, and each cluster sends the messages without conflict because the SS partitioning is fixed, the OS partitioning is fixed, and the cluster may or may not have to send a value to the particular area of the screen.

In step 1) as in the Permutation_Volume_Render FIGURE 66, is preprocessing. Each processor calculates the opacities and shaded intensities for a subvolume assigned through slice and dice virtualization.

In step 2.1), the VWS, processors calculate the inversed coordinate frame. Step 2.2) warp each point by differencing.

Reconstructions are composited into temporary subframes using local data, alternating warping and compositing, steps 2.4 and 3.1. The number of subframes depends only upon the overlap of your data in the SS. Traversal is determined from the SS ray directions, and the subvolume data of the processor is randomly accessed as needed. Once the subframes are completed, they are sent to the aligned screen space (step 3.2) where further compositing takes place. This send requires sending parts of the calculated subframe to several processors.

In step 3.2 subframes are combined. The final frame is distributed across all of the processors, and every processor remains busy compositing data. FIGURE 74 shows Steps 2.4 and 3.1, 3.2, and 3.3 of the algorithm. Various physical processor layouts were shown in FIGURE 71 which correspond to the same OS spatial layout. The subvolumes are used as temporary storage while combining.

```

Iray[ ] ← HighGrain_Permutation_Volume_Render(V, Γ, T, classify, shading) {
  1.0) PPS  $\alpha_p, I_{Sp}$  calculated
  2.0) VWS
      2.1) For each subcube calculate overlap into SS to choose messages
      2.2) Calculate subcube coordinates  $p_{\pi'} = T^{-1}(p'_{\pi'})$ 
      2.3) Choosing and iteration start point at back corner of cube
      For each sample in subcube {
          2.4) reconstruct  $\text{Tmp}[\alpha_{p'}, I_{Sp'}] = \text{Reconstruction}[ ]$ 
      (CS) 3.1) Composite into subframe  $I_{Ew} = I_{Sp'}\alpha_{p'} + I_{Ew}(1 - \alpha_{p'})$ 
      3.2) Rounds of Permutation send of temporary subframes
      3.3) Parallel product compositing of subframes
      }
}

```

FIGURE 73

High Granularity Permutation Algorithm for $P \leq RW$,
Image order resampling storage $O(R)$.

The large granularity algorithm, FIGURE 73 and FIGURE 74, stores only temporary subframes, and is very memory efficient. The trade-off is that is more involved to calculate the messages to be sent, and the boundaries of those messages. Fewer values are communicated also, because compositing is partially done in the OS. The algorithm uses exclusive reads and writes which can be implemented on distributed or shared memory machines.

Both algorithms calculate compositing through a product evaluation. A parallel product evaluation [KRUS85], is work efficient up to $P = O(n/P + \log P)$ processors in the view depth dimension. The product evaluation is a speed limit for the time when parallel

machines get millions of processors. But for now input volume sizes and architectures are clearly in the linear speedup region shown in FIGURE 65.

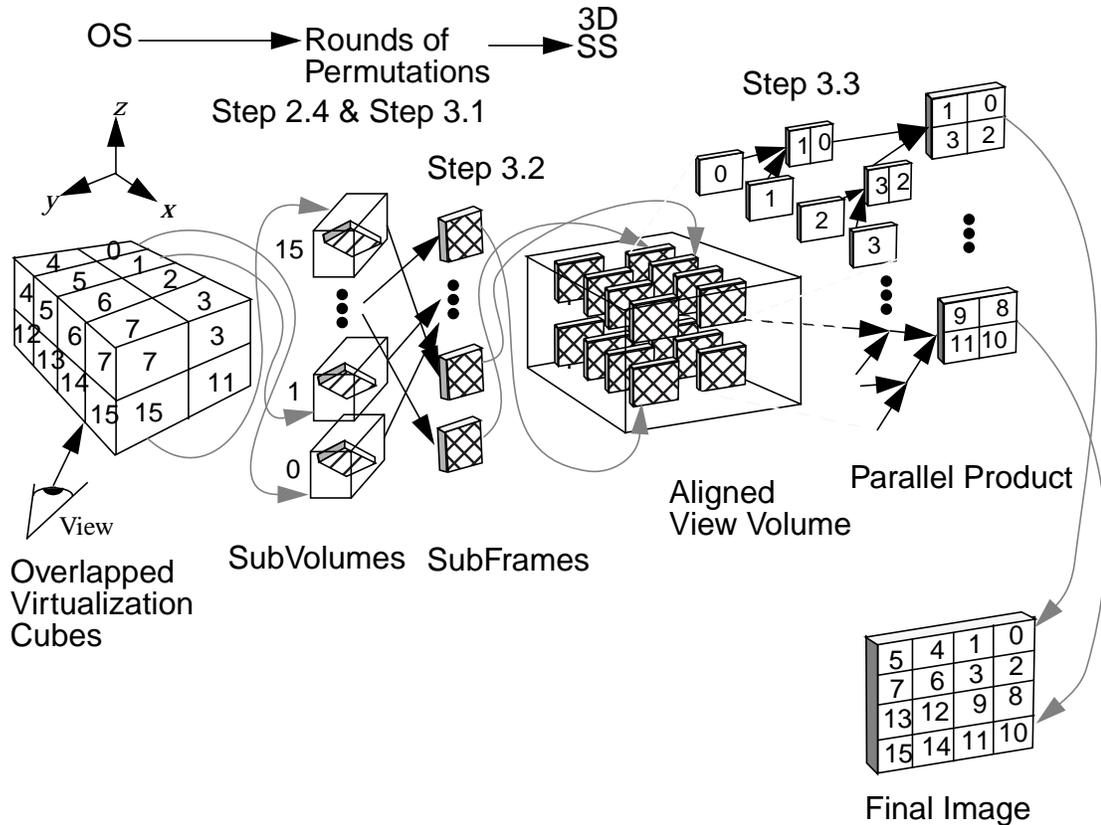


FIGURE 74

High Granularity Rounds of Permutation Sends

4.6 MasPar and Proteus Performance Results

I have implemented the data parallel algorithm on the MasPar MP-1 [BLAN90] and the high granularity algorithm on Proteus [SOMA91]. In this section I show filter quality and time and trade-offs, followed by performance measurements.

Multipass shears and direct warping are not equivalent. Because each resampling stage loses the original data, a shear filtering approach has more resolution error and interpolation error than a comparable direct filter (See [PRAT78] for one pass filters). After a shear all that is stored is the new samples, hence a multipass shear is not equivalent to trilinear interpolation. I demonstrate here the quantitative difference in the filters. FIGURE 77 shows the empirical error in my direct warp compared to the multipass shearing. I show direct warps using zero order hold, trilinear interpolation, and shearing with linear interpo-

lation. The multipass method has more error than the trilinear reconstruction in all cases. I used two test objects to calculate the reconstruction error: a cube of 65535 intensities and a sphere whose intensities are zero at the edge and 65535 at the center. 16 bit intensities were used. The volumes were 128x128x128 voxels with the sphere and cube centered and of diameter/width 64. A Sun Sparc 2 was used to calculate the comparison to ease implementation of the shearing approach. The errors were calculated by differencing each sample point for an altered viewpoint with the analytically defined cube or sphere. Absolute errors were summed on each ray. FIGURE 77 is the error in rotating 45 degrees about all three axes simultaneously.

TABLE 21 shows the mean of the summed error for all rays in an image for the volumes resampled. The mean error varies little with rotation angle, and the mean for all measured cases from 5 degrees to 45 degrees is given. The trilinear is clearly better than shearing, but the zero order hold is the same as the trilinear for the cube and worse than trilinear and shearing for the sphere. This results from the frequency content of the volumes. The cube is a step function and has infinite frequencies. The zero order hold maintains the resolution very well. Because of the high frequencies, the multipass approach has repeated aliasing steps which degrades the reconstruction considerably.

TABLE 21

Mean of the Measured Absolute Summed Error over all rays for 45 degree rotation about all axes.

	Cube	% worse than trilinear	Sphere	% worse than trilinear
zoh	48370	0%	4079	1468%
multipass	64068	32%	775	198%
trilinear	48385	0%	260	0%

The cube has sharp edges and high frequencies so errors were higher. FIGURE 75 shows the maximum error for the three approaches versus rotation angle about all angles simultaneously. The cube is not frequency limited, and the zero order hold does very well, because it can preserve resolution better than the other fil-

ters. The trilinear is more accurate though, as in the sphere case. FIGURE 76 shows the maximum error for any ray rendering the sphere at different view angles.

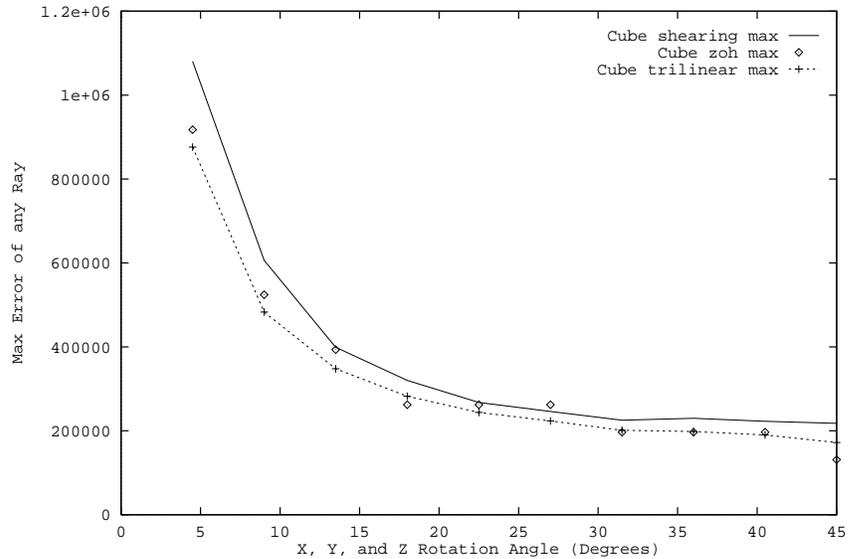


FIGURE 75 Maximum Error in Reconstruction of Cube

FIGURE 76 shows the maximum error for any ray rendering the sphere at different view angles. FIGURE 77 shows the error in a direct warp using zero order hold and trilinear interpolation compared to the multipass shearing. The multipass method has more error than the trilinear reconstruction in all cases. The mean error for all rays in the image remains the same across view angles. FIGURE 77 shows how error is placed in the figure, with error ramped from the maximum errors. The trilinear filter has low error across the whole image. The zero order hold has more error on the sphere than the multipass shear, but less error on the cube. The zero order hold is useful because it is inexpensive to calculate.

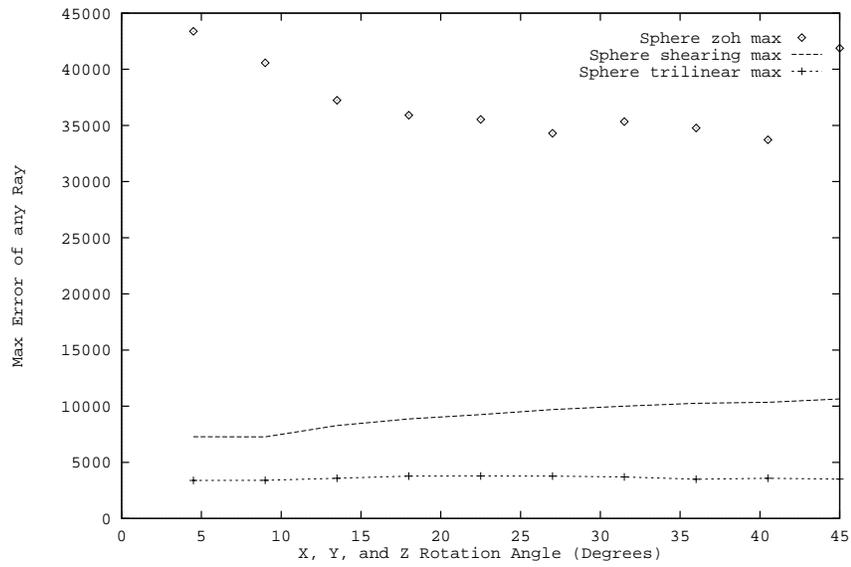


FIGURE 76

Maximum Error in Reconstruction of Sphere

FIGURE 77

OMAX
Error for 45x45x45 rotations,
Top: Zero Order Hold, Middle: Multipass, Bottom
Trilinear

The mean error for all rays in the image remains the same across view angles. FIGURE 77 shows how error is placed in the figure, with error ramped from the maximum errors in the pseudo colored images. The range of error is from 0 to 217719 (shear highest error) for the cube and zero to 41886 for the sphere (zoh highest error). TABLE 22 shows the mean and max errors for the 45, 45, 45 degree rotation.

TABLE 22

Absolute summed error on rays for 45, 45, 45 degree rotation (See FIGURE 77)

	Cube		Sphere	
	mean	max	mean	max
zoh	48372	131070	3977	41886
multipass	65028	217719	802	10648
trilinear	48309	172078	259	3501

The trilinear filter has low error across the whole image. The zero order hold has more error on the sphere than the multipass shear, but less error on the cube. The zero order hold is useful because it is inexpensive to calculate.

Visual differences on application data illustrate the qualitative differences. My multipass implementation could not process the medical data because of memory limitations. Max intensity is useful because it is simple, and works on volumes which contain a lot of noise. FIGURE 78 shows the noise inherent in the MR angiography data. A comparison of the filter quality using a trilinear and a zero order hold with max intensity are given for a magnification of the 256x256x28 data to a 512x512 image using 8X magnification. The filter difference on these medical image is readily apparent.

color photograph inserted

FIGURE 78

Data with Ramp to Show Noise

color photographs inserted

FIGURE 79

8X magnification, Zero Order Hold/ Trilinear

This empirical study shows how a direct warp has improved filter characteristics over multipass shear approaches of [SCHR91] and [VEZI92]. The direct warp calculates reconstruction filters identical to ray tracing of sequential algorithms. Next I present performance measurements.

4.6.1 MasPar Implementation

Performance measurements were taken on the MasPar MP-1 [BLAN90]. I briefly review the architecture, and then discuss implementation details. The MasPar used for the performance study was a 16384 SIMD processor MP-1 whose peak performance is 26,000 MIPS (32 bit integer) and 1,200 MFLOPS (32 bit floating point). The architecture supports frame buffers through VME frame grabbers, HIPPI connection, or through MasPar's frame buffer (not available yet). Image display in the current implementation is done on the X host. The processors are interconnected through both a toroidally connected mesh with 23,000 Mbytes/sec peak bandwidth, and through a general multistage crossbar router with 1,300 Mbytes/sec peak bandwidth. The array controller provides a software accessible hardware timer that accurately captures the elapsed run time.

My implementation in MPL, a C like parallel language, uses the slice and dice virtualization discussed, and virtualizes processors across all three dimensions. The neighboring processors do not need to be accessed in the resampling step by providing a 1 voxel overlap of volume storage on each processor. This allows the octant wherein the interpolation point lies to be accessed by a random access in each processor's local array avoiding the need for a costly case decision in the SIMD language. The storage overhead does not affect the size of volumes that can be processed, because there is a slight overhead for dynamic memory allocation. The data is loaded directly from disk into the slice and diced overlapped array.

The zero order hold is most efficiently calculated without using permutation warping, and I therefore use a backwards calculation and no overlapping for the zoh. The first order hold uses the rule calculation for a significant advantage over the backwards algorithm. My implementation takes advantage of the MasPar instruction ScanMax. Once each processor composites its subcube, ScanMax composite across z in segments to finish each parallel product of each ray with one instruction. The over operator can be done similarly using the Scan operator to create the proper transparency at each processor, and then doing a parallel addition.

A max intensity parallel product operator was used to generate like sized (32x32x32 to 32x32) byte images. Sizes of 32^3 to 256^3 were processed. Measurements given are the average of multiple runs at each angle. FIGURE 80 shows the run times to render a 128x128x128 byte volume to a 128x128 image. See TABLE 23. The rotation only times are given in FIGURE 80 also showing how the resampling for rotation takes the

majority of the time. The many lines for each filter show rotation about x, rotation about y, rotation about z, and rotation about x, y, and z.

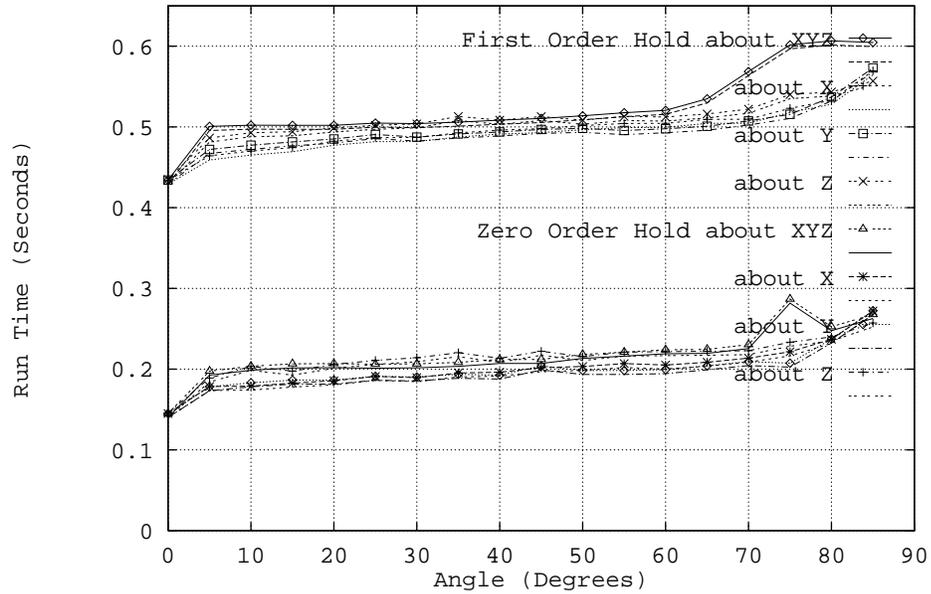


FIGURE 80 Nearly Constant Run Time Versus Angle

TABLE 23 16k Processor MP-1 128x128x128 Volume Rendering Times in Milliseconds

Filter	Rotation Axes	0	20	40	60	80
Foh	About x	434.235	482.359	496.150	504.567	533.154
	About y	434.235	485.119	493.886	497.846	537.060
	About z	434.811	498.804	508.437	512.326	543.372
	About x, y, and z	434.235	502.090	508.438	520.601	606.616
Zoh	About x	145.060	185.513	196.245	205.139	236.907
	About y	145.060	186.502	192.709	199.392	237.332
	About z	145.614	205.881	213.413	221.776	239.916
	About x, y, and z	145.060	206.853	212.068	224.002	252.644

By using a single decomposition of the rotation all of the rotation angles can be efficiently calculated with tunable (zoh, trilinear, or cubic) filters. There is no artificial barrier

er at 45 degrees as with the multiple pass approaches, and with a decomposition the uses reflected inputs to the matrix the cotangent can be used instead of the tangent, to maintain stability for angles 90 degrees to 180 degrees.

FIGURE 81 gives the mean run time across all angles, and using the min and max as error bars, for different volume sizes. See TABLE 23 and TABLE 24. Note that the performance is tightly bounded for each volume. The performance is predictable. The effectiveness of direct warps lies in the performance filter tunability. The zero order hold takes from 73% to 146% less time than the first order hold, and can be used for interactive performance in viewing the larger volumes. The trilinear interpolation, or first order hold, has comparable performance to the multipass warps but is more accurate. FIGURE 79 shows the filter quality tuning for the foh and zoh. Comparisons to [VEZI92] and [SCHR92] show that my resampling times are about a factor of 4 slower than [VEZI92] and 1.3 to 5 times faster than [SCHR91] for rotation only. See TABLE 25.

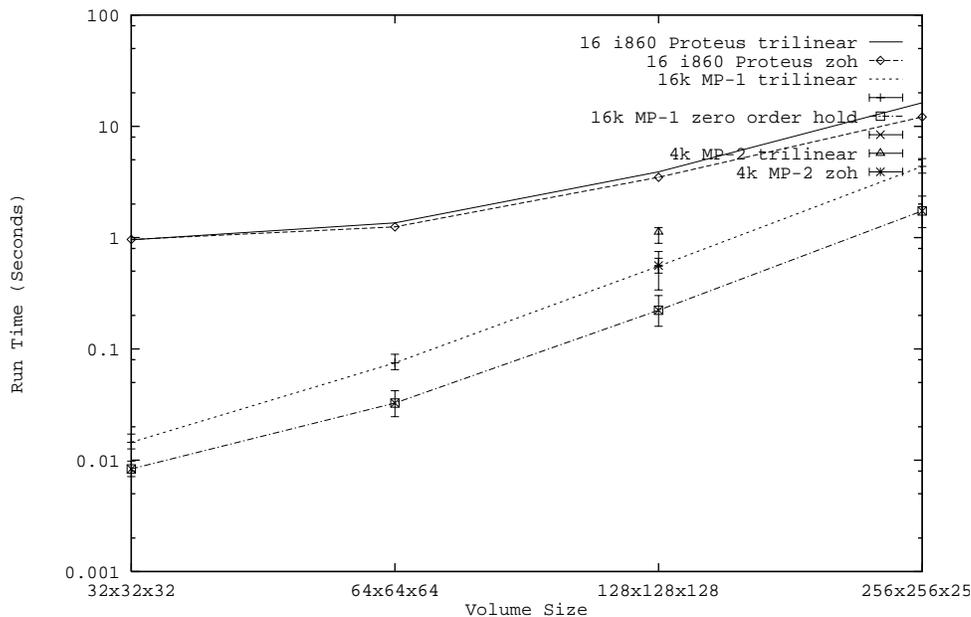


FIGURE 81

Run Times Versus Volume Size for the 16384 processor MP-1

The factor of 4 slowdown is clearly a result of the general router and mesh router mismatch, recall 1300 Mbytes/s versus 23,000 Mbytes/sec. But the communication congestion is low for the permutation warp. Using the rotation speed of 0, 0, 0 degrees in TABLE 23 the congestion is 19% to 29% of the run time for the rule algorithm, first order hold. The congestion is 40% to 43% for the backwards algorithm, or zero order hold, but for that simple filter the overhead of the rule calculation makes a backwards algorithm

more efficient. The router start-up penalty and/or the rule overhead account for the rest of the difference.

A comparison to other performance numbers in the literature reveals that there is intense competition for constant factor speedups TABLE 20. Because of the wide variation in view transforms, voxel formats, shading, preprocessing, and image sizes direct comparisons are difficult. The closest comparisons are to [VEZI92][SCHR91][SCHR92] who use similar voxel sizes. Comparison of resampling times shows that direct filters cost more (4 times more, [VEZI92]) but the direct filter is superior to the shear locally then send approach [SCHR91] with up to a 5 times speedup depending on the machine compared to. The forward wavefront approach [SCHR92] trades view angle freedom for high performance and a 1.35 speedup over my work TABLE 20.

The architecture strongly controls the algorithm features and performance. Shared memory of the Dash [NIEH92] or full data replication [YOO91] provide the highest performance arbitrary view solutions. But these approaches cannot be used on data parallel machine such as the MasPar MP-1/MP-2, CM-2, and CM-200. I have through permutation warping provided improved quality and view angle freedom for data parallel machines.

TABLE 24

16K Processor MP-1 Slice And Dice Timings For Warping Only, Milliseconds. Reconstruction to align and resample byte voxels with orthographic view.

	vol size	Mean	Min	Max
Forward foh	32x32x32	12.223	10.403	14.923
	64x64x64	66.698	56.962	81.431
	128x128x128	501.673	429.225	601.604
	256x256x256	3977.112	3407.390	4749.763
Backward zoh	32x32x32	6.591	5.384	8.076
	64x64x64	28.422	20.378	37.909
	128x128x128	203.063	140.105	281.743
	256x256x256	1602.002	1096.583	2223.762

implementation of the algorithm would not have communication congestion, but might be computation inefficient. Various timings on the MasPar MP-1 with 1K and 16K processors, the MP-2 with 4K processors, and Proteus with 16 processors follows.

TABLE 27 Volume Rendering Times For 1K MP-1, Seconds

Filters	vol size	Mean	Min.	Max
Trilinear	32x32x32	0.143146	0.116181	0.247137
	64x64x64	1.088516	0.885357	1.922671
	128x128x128	8.651044	7.027133	15.303813
	256x256x256	69.061064	56.129872	122.29240
Zero Order Hold	32x32x32	0.073746	0.051313	0.168320
	64x64x64	0.557138	0.379754	1.326254
	128x128x128	4.442529	3.014020	10.646267
	256x256x256	35.589276	24.132464	85.505345

TABLE 28 4K MP-2 Column Virtualization Timings for 128x128x128 Volume, Seconds

Filter	Mean	Min	Max
First Order Hold	1.342751	0.827207	4.233842
Zero Order Hold	0.852031	0.391235	3.564636

TABLE 29 Proteus Run Times, all output images are 256x256, Seconds

vol size	32 PE's Tril	32 PE's Zoh	1PE tril	1 PE Zoh
32x32x32	0.161	0.150	0.241	0.097
64x64x64	0.291	0.203	1.760	0.554
128x128x128	1.046	0.498	13.846	3.870
256x256x256	4.316	1.411	95.064	24.523

TABLE 30

4K MP-2 Slice and Dice Timings for 128x128x128 Volume,
Seconds

Filter	Mean	Min	Max
First Order Hold	1.123582	0.889419	1.230521
Zero Order Hold	0.560859	0.338280	0.751107

4.6.2 Proteus Implementation

Proteus is a scalable MIMD (multiple instruction multiple data) parallel computer originally intended for computer vision. The strong interconnection network provides fast I/O necessary for interactive visualization [SOMA91]. The machine, shown in FIGURE 71, has from 32 to 1024 processors, with 32 processors to a group. Each group has an aggregate I/O of 16*250M bits/second.

A prototype group with 32 Intel i860's has been implemented. The Intel i860 is a 40MHz/Mips processor with built in floating point capability 80 peak MFlops, 8k byte on chip data cache, and 4k byte on chip instruction cache. Each cluster has 4 i860's. Each i860 has 1 Mbyte of external cache accessible by 160Mbyte/sec bus. The cluster's shared memory is 8 Mbytes of DRAM upgradeable to 32 Mbytes, and has a 40 Mbyte/sec bus. Each cluster is controlled by a 33 MHz Intel i960 which sets up communication and handles interrupts freeing the i860's for computation. A group has 8 clusters of 4 i860's for 32 processors and 2.560 peak GigaFlops. FIGURE 71 shows the physical layout of 32 processors in eight clusters labelled C0 to C7. The interconnection network is a bit serial crossbar with single link transfer rates of 250 Mbits/second which achieve a throughput of roughly 20 Mbytes/second. A frame buffer is interfaced through the serial links so a 256x256 byte image can be refreshed at 1280 frames/second, a 1024x1024 byte image at 80 frames/second, and a 32 bit/pixel 1024x1024 image at 20 frames/second. The current implementation uses the communication interface board as the frame buffer. Display is done through a Sun server.

FIGURE 71 also shows the spatial layout of the 8 clusters assigned voxel data. The network communicates cluster to cluster so the partitioning is done by clusters. The data set is not replicated. Each communication uses a permutation which the crossbar interconnection network (ICN) provides. Within each cluster, the four processors render one fourth of the subimage being calculated. Because the cluster composites locally resampled data, only 7 messages are sent. Data sending is started before all of the local computation is complete, and compositing begins before all of the data has been transmitted.

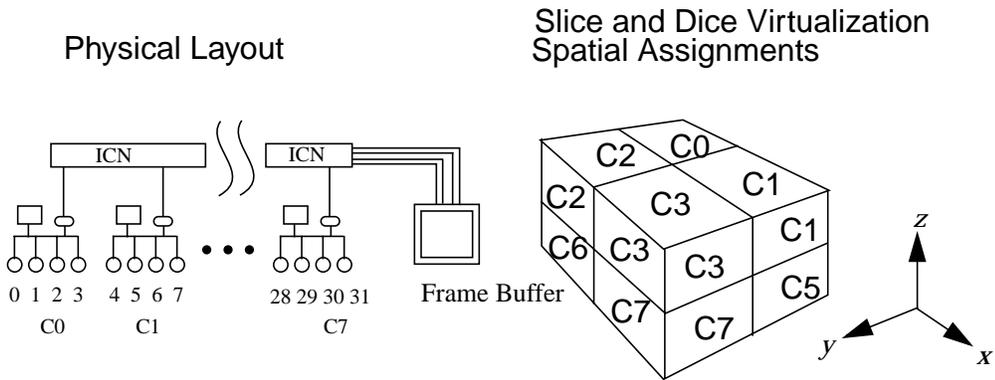


FIGURE 82 Spatial Volume Virtualization For Proteus

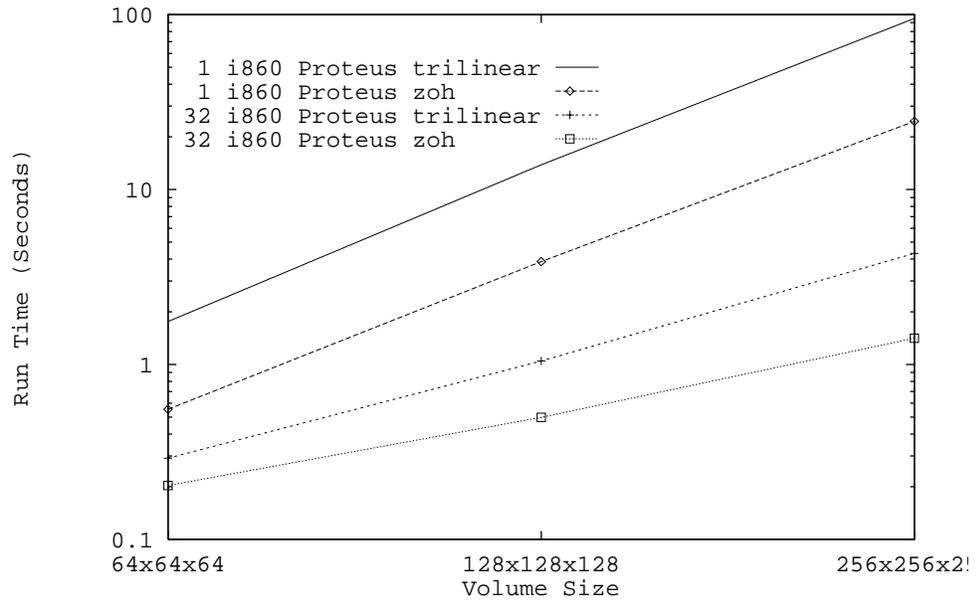


FIGURE 83 Run Time Versus Volume Size for Proteus and 16k processor MP-1

I implemented the high granularity algorithm on the Proteus Supercomputer [SOMA91]. Two different reconstruction filters are used, a first order hold and a zero or-

der hold. Direct warps support high order filters more effectively. Max intensity ray combining is used, and different shading is possible with the same filters. MR angiography images were used after being window and leveled to 8bits/voxel. Images created were 8bits/pixel. All measurements were taken using multiple runs of the code, and averaging.

FIGURE 83 shows the Proteus volume rendering algorithm's run time versus volume size (TABLE 29). The output image is 256x256 for all volume sizes. Speedup is given in TABLE 31. Proteus provides a speedup of 24.

TABLE 31 Speedup Versus for 32 Processors

Volume Size	Trilinear	Zoh
64x64x64	6.05	2.73
128x128x128	13.24	7.80
256x256x256	22.03	17.38

4.6.3 Comparison of Proteus With Existing Methods

My algorithm calculates backwards viewing, use tunable filters, and have limited congestion and memory overhead for efficiency. The efficiency is $O(S/P)$ run time, $O(S)$ storage for $S = RW$ samples to render with R rays, W samples per ray, on P processors. Measured performance is strongly controlled by the implementation. My numbers are near those of comparable powered machines, and I don't use data dependent optimization. The important result of this study is efficient arbitrary viewpoint rendering with distributed volumes. Different shading, preprocessing, and voxel sizes make results difficult to compare quantitatively (TABLE 19 and TABLE 20). Qualitatively I have shown that explicit distribution of source data is efficient and that parallel product can allow scaling processors beyond the number of rays.

Comparison numbers for [VEZI92][YOO91][CAME92][CHAL92][SCHR91][STRE92][NIEH92] and [SCHR92] are included for timing reference. My frame rates are 2 frames/second for 128x128x128 volumes and 0.7 frames/second for 256x256x256 volumes. I am able to visualize volumes of size 512x512x128 of byte voxels. This uses 32 Megabytes leaving 32 Megabytes for program code and other variables. The total memory capacity of Proteus is 64 Megabytes (8 Mbytes per cluster) and can be increased to 32 Mbytes/cluster with higher density DRAMs. The performance in millions of voxels per second ranges from 3 to 12. This compares to the 31.77 Mvoxels/second of [SCHR92], 23.30 Mvoxels/second of [NIEH92], and 21.18 Mvoxels/second of [STRE92].

4.7 Summary and Discussion

I presented optimal EREW PRAM algorithms for volume rendering, and demonstrated their efficiency on parallel machines. General reconstruction filters provide time/quality trade-offs not possible in previous parallel approaches making parallel implementations more useful for volume rendering. Volume rendering is ideally parallelizable with linear speedup. Theoretically, volume rendering can be a constant run time algorithm $O(1)$, provided that the network can composite all of the ray samples. As parallelism grows, parallel prefix and parallel product are more valuable for volume rendering. But, today's machines fall well into the linear speedup region $P = O(S/\log S)$, S is the number of samples taken to create the output image.

Volume rendering costs are linked to the data structures and representations, because of the high compute and storage costs. Volumes, while conceptually simple, do not provide the fastest visualization. Octrees, amalgams, and transparent surfaces can improve efficiency. In fact few applications require explicit voxelization, only the effects of light in semitransparent media, which boundary surfaces can represent.

Regardless of the data structure, object space partitioning gives easy parallel assignment shown by my parallel algorithms. I have shown SIMD and MIMD results and found more important differences lie in the supported message granularity.

The data parallel version can be ported to massively parallel general purpose computers and the high granularity version can be implemented on less parallel machines. This fact, and the ability to change combining rules, shading, or reconstruction filters, shows that permutation warping achieves high efficiency with great flexibility on general machines. Special purpose machines cannot offer this flexibility in shading, combining, and filter choices. My streamlined communication supports many filters for truly useful and general algorithms. My algorithms also support arbitrary viewpoints efficiently. Before these results, researchers thought general viewpoints were inefficient. Permutation warping proves this not to be the case.

My implementation on the MasPar allows rendering with changing viewpoints of 5 frames/second and 2 frames/second for higher quality trilinear reconstruction ($128 \times 128 \times 128$ volumes). This improves on previous results [DREBB88][SCHR91b][VEZI92][KABA92][WRIG92] because of the better filters used, and I illustrated the filter differences. Permutation warping is also memory efficient, and the data parallel algorithm requires $S + S'$ memory and the high granularity algorithm requires $S + R/(P_x P_y)$. The practical effect is larger data sets can be rendered, and on Proteus and the MasPar I rendered $512 \times 512 \times 128$ volumes of 32 Megabytes. On Pixel Planes 5 [YOO91], for example, network inefficiency required storing the volume on every processor, limiting data sizes to 128^3 . My algorithms are simultaneously tunable for filter quality, communication efficient, space efficient, and general. Providing sequential algorithm features in an efficient parallel algorithm is a most significant contribution.

Chapter V

Fourier Volume Rendering

In this chapter I review Fourier volume rendering and discuss possible algorithm improvements, and recent developments by other researchers.

5.1 Background

Fourier volume rendering [MALZ91][DUNN90][LEVO92] uses transitions to and from the frequency representation of a volume for rendering. Because the majority of volume data is created by the use of the projection slice theorem [KAK88], it was clear to several researchers that Fourier volume rendering held promise. This approach is fast because 2D frequency data creates 3D spatial information. There are similar techniques in the spatial domain [HARR78][JAFF82].

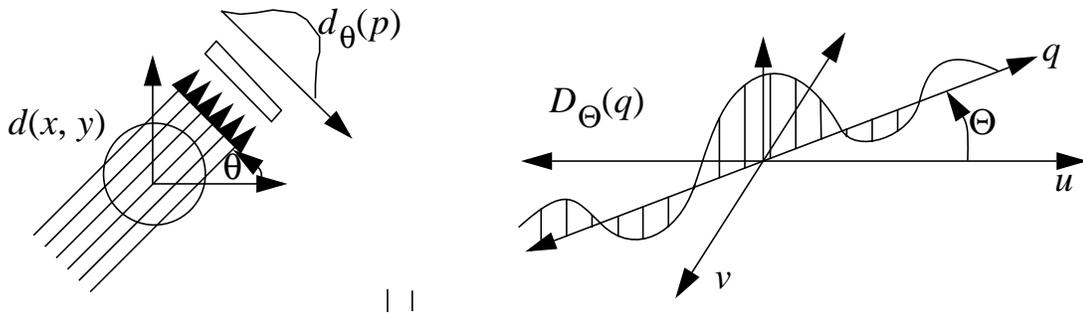


FIGURE 84 Fourier Slice Theorem, projection top, spectra bottom

The approach is as follows. Compute the three dimensional Fourier transform of the volume $V(r, s, t)$ saving calculation and storage by using the 3D real Hartley transform [BRAC86]. The Hartley transform is more efficient because the data is not complex but real. Given the Fourier transform $F(f)$, the Hartley transform $H(f)$ is

$$H(f) = F_{\text{real}}(f) - F_{\text{imag}}(f), \quad (\text{EQ 79})$$

and the three dimensional Hartley transform is,

$$H(u, v, w) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} V(r, s, t) \text{cas}(2\pi(ur + vs + wt)) dr ds dt, \quad (\text{EQ 80})$$

where $\text{cas}\theta = \cos\theta + \sin\theta$.

Then, by the Fourier slice theorem [KAK88], any planar slice through the origin of the spectra $H(u, v, w)$ is the Hartley transform of the projection of the volume $V(r, s, t)$ at that same angle where u', v' is a 2D plane oriented at angle θ, ϕ in the spectral volume.

FIGURE 84 illustrates the Fourier slice theorem, where $I(p, \theta)$ is the transmitted intensity, $d(p, \theta) = \log\left[\frac{I_o}{I(p, \theta)}\right]$ is the projected density that passes through the object. The transmitted density's spectra represents one line of spectral information for the entire object. I_o is the incident beam intensity, p distance along detector array, and the line $q = \sqrt{u^2 + v^2}$. Reversing the projection and going from the spectra, $D_\theta(q)$ to the density $d(x, y)$ calculates the line integral or shadowgram $d(p, \theta) = \int_{-\infty}^{\infty} d(x, y) dl$.

From a three dimensional spectra, $H(u, v, w)$ represents the Hartley transform of an angle of projection, and an inverse transform of a slice calculates the projection itself. The inverse Hartley transform is

$$I(x, y) = \int_{-\infty}^{\infty} H(u', v') \text{cas}(2\pi(u'x + v'y)) du' dv' \quad \text{(EQ 81)}$$

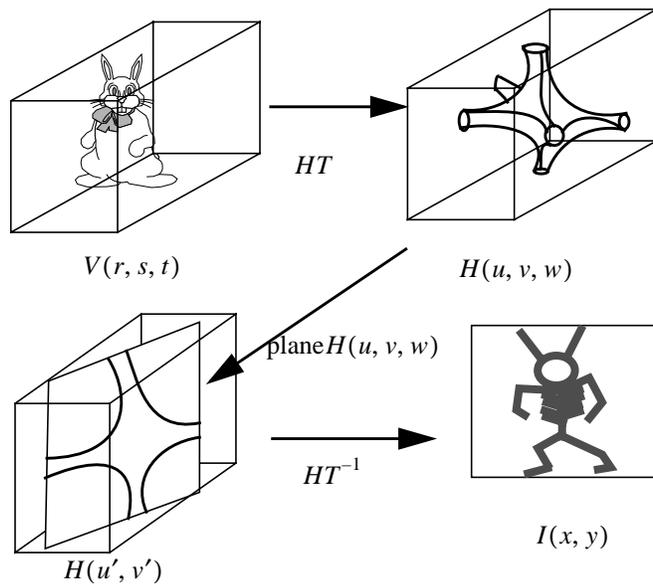


FIGURE 85 Fourier Volume Rendering

Malzbender [MALZ91] has implemented a Fourier approach, and shown higher efficiency than backward mapping algorithms. The difficulty that he runs into is in resampling and reconstruction in the Fourier domain, a problem also seen by [DUNN90]. Malzbender found the filtering to reconstruct the planar slice of $H(u, v, w)$ was more costly than the inverse fast Hartley transform HT^{-1} , a direct analog of the fast Fourier transform.

FIGURE 85 shows the transformation process. The rabbit in the upper left is the spatial model, voxels. The Hartley transform, HT , computes a 3D spectra. By sampling and reconstruction a 2D plane of spectra through the origin is obtained in the lower left. Different plane orientations create projections in different directions. The inverse Hartley transform, HT^{-1} , computes a 2D plane of intensities, $I(x, y)$, that is a shadowgram of the original data. The shadowgram represents attenuation and is like an artificial X-ray. The speed of the process is obvious because the 2D plane selection and inverse Hartley transform work with only a slice of data. This contrasts with both forward mapping, backward mapping, and surface fitting algorithms that work with a full volume of data.

Because the Fourier slice theorem dictates that a line integral through the volume is formed it may not be possible to achieve hidden surface and surface shaded renderings. Nevertheless reprojection is valuable for medical and speed critical applications.

5.2 Possible Fourier Volume Rendering Approaches

I had proposed to perform Fourier volume rendering using the Radon transform, and several architectures have surfaced to compute the Radon transform [CURR92]. Important issues for Fourier volume rendering have turned out to be how to get anisotropic shading, and how to avoid aliasing artifacts. Levoy has implemented an algorithm that uses gradient volumes in orthogonal directions to give directional shading results. The results are somewhat ambiguous because of the view independent shading model. The computational advantage is somewhat reduced because four precalculated volumes are used for rendering instead of one.

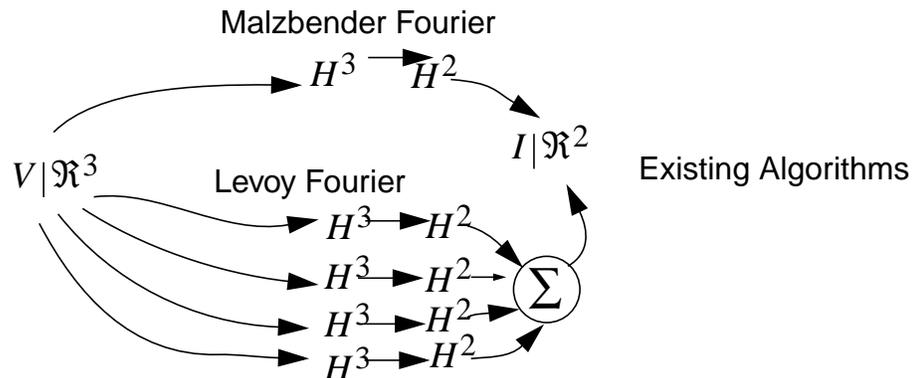


FIGURE 86

Volume Rendering Transform Graph

For those applications that do not require directional shading, such as MR angiography, Fourier volume rendering is ideal. In MR angiography because data is collected in the frequency domain, rendering is even more direct, because a spatial representation does not have to be created until a projection angle is chosen. A three dimensional fast Hartley

transform takes $O(S \log S)$ work [DUDG84]. The two dimensional inverse would take $O(R \log R)$ for each projection. This can be fully parallelized for $O(\log R)$ parallel run time. The fastest algorithms, using $P = R$ processors for Fourier volume rendering is $O(\log R)$, and $P = S$ for spatial volume rendering is $O(\log W)$. But the Fourier volume rendering algorithm works with less data, and therefore with a comparable number of processors is faster. Using $P \approx R$ processors for both, Fourier rendering is still $O(\log R)$ while spatial rendering is $O(S/R) = O(W)$ for a clear advantage.

Polar coordinate transforms may ease the resampling problem. Because the reconstruction, or resampling, is the costliest step, reducing the resampling cost is important. By sufficient prefiltering, a high quality polar coordinate representation of the volume can be created, which allows less expensive spatial reconstruction following projection. Survey of the literature reveals that polar coordinates are often used in derivation of circularly symmetric function transforms or the Hankel transform [JAIN89]. A fast polar coordinate transform is difficult to derive because the sampling geometry is not separable like rectangular coordinates.

$$g(r, \theta, \phi) = \int_0^\infty \int_0^\pi \int_0^{2\pi} G(s, \Theta, \Phi) e^{i2\pi s r [\cos \Theta \cos \theta + \sin \Theta \sin \theta \cos(\phi - \Phi)]} s^2 \sin \Theta ds d\Theta d\Phi \quad (\text{EQ 82})$$

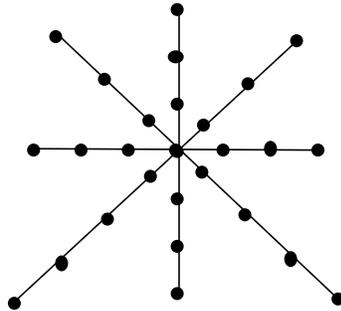


FIGURE 87

Polar coordinates

The Radon transform is generalized Hough transform, which can be adapted for rendering purposes. The Radon transform is the projection of spatial data [JAIN89], but how to create anisotropic, or directional lighting effects is unclear.

5.3 Summary and Discussion

There are intriguing possibilities for development of frequency representation both for filtering and rendering of sampled data sets. Researchers are skeptical about the development of anisotropic shading in Fourier methods [MALZ90b], but Fourier rendering provides the fastest asymptotic run time complexity method for both sequential and parallel volume rendering algorithms.

Chapter VI

Conclusions

In this dissertation I have presented an algorithm design framework, and several new algorithms for optimal parallel volume rendering. My framework is a collection of the knowledge used in developing algorithms. A directed graph representation allows working with algorithms at a high level using representations and algorithms as building blocks. Because of the multiplicity in transform calculations, including spatial warping and volume rendering, the most efficient algorithms were derived using my methodology. Such an approach creates more portable and flexible algorithms using abstractions where resource and efficiency trade-offs are easier to make. My volume rendering algorithms are optimal because they achieve linear speedup, are memory efficient, and achieve lower bounds on the EREW PRAM. My algorithms are also practically efficient and implementation on SIMD (MasPar) and MIMD (Proteus) confirm the complexity analysis.

6.1 Applying the Framework to Other Algorithms

Evaluating algorithms before coding is important and the bridging model that I introduced, the MCCM, allows designers to evaluate parallel algorithms. Intended to be used as an engineering tool, the MCCM is simple, and adds communication costs and a general network topology to the PRAM. I used the directed graph representation and the MCCM to develop parallel warping and volume rendering algorithms. The MCCM run time complexity accurately matched the MasPar and Proteus performance measurements.

For any application many algorithms will work, but by setting clear goals, using knowledge of resources and parameters, one can design efficient algorithms. An important contribution of this dissertation is development of parallel algorithms that give the same fidelity as sequential algorithms, and that are portable to existing parallel computers. Algorithms designers should not have to give up features to use parallel computers. My new parallel warping and volume rendering algorithms achieve linear speedup with unprecedented flexibility in view angles, reconstruction filters, and image fidelity. In the future, portable algorithms will be easier to develop, because slowdown compilers provide a technology for efficient portability. Efficiently parallelized algorithms are portable, machine scalable, problem size scalable, and generation scalable.

6.2 Designing Parallel Warping Algorithms

My parallel spatial warping algorithms are $O(1)$ with a processor per sample. The exclusive read exclusive write (EREW) algorithm is restricted to equiareal transforms, but joined with the CREW algorithm can down sample or up sample to achieve more general transforms. Because an EREW PRAM algorithm strictly limits interaction of processors,

the EREW algorithm turns out to be much more efficient than the concurrent read algorithm. The MCCM makes these differences explicit.

I also illustrated how to slow down the algorithm for machines with fewer processors than samples. Any parallel computer with a general interconnection network can efficiently warp an image with $O(S/P)$ run time, for P processors and S samples. On the MasPar, slice and dice virtualization, and a 1 sample boundary overlap on each processor kept the density of messages low for any rigid body two dimensional or three dimensional warp.

Other parallel warping algorithms relied primarily on multipass warp techniques [PAET86][TANA86][SCHR91][HANR90] which have poorer filter quality. Higher order transforms are possible with multipass warps, but my algorithms provide a two pass algorithm, one with scaling, and the other equiareal, for a good mix of generality, filter quality, and efficiency.

6.3 Designing Parallel Volume Rendering Algorithms

Warping algorithms are useful for volume rendering, because they generalize to any dimensional image and in fact there are greater advantages with higher dimensional images. For example two dimensional improvement is as high as 59% and three dimensional improvement is up to 100%. I compare existing parallel volume rendering algorithms grouped into four categories determined by their viewing transforms: backwards, multipass forwards, forwards wavefront, and forwards splatting.

Existing backwards parallel volume rendering algorithms have general reconstruction filter support, but restrict platforms or data set sizes. Existing multipass algorithms are very efficient, but restrict viewpoints and reconstruction filter quality. Forwards wave front algorithms have higher quality projection filters, but require post processing, and limit view points similar to the multipass approaches. Forwards splatting algorithms have filtering error from out of order compositing [WILH91]. My algorithms calculate backwards viewing, use tunable filters, and have limited congestion and memory overhead for optimal efficiency. The efficiency is $O(S/P)$ run time, $O(S)$ storage for $S = RW$ samples to render with R rays, W samples per ray, on P processors. My fastest EREW spatial volume rendering algorithm is $O(\log W)$ which can be improved upon by using stronger networks.

6.4 Future Research

There are several research areas touched upon in this dissertation: parallel algorithm design, volume rendering techniques, and parallel software methods. Each area has important open research problems.

I showed how transition graphs are a starting point for global optimization of algorithms. The algorithms developed in this dissertation were discovered by using the transition representation, investigating the alternatives, and working hard on those edges that represented the best features and performance. Automation of the transition choices and automated postulation of transitions is one topic. Optimization techniques and algorithm representations are the first issues to investigate.

There is much research left for constant factor speedups in volume rendering such as ray termination, bounding hulls, adaptive sampling, and adaptive quadrature. A distributed algorithm using my permutation warping, and optimizations such as adaptive sampling, ray termination, and bounding hulls would provide the best of both parallel speedup and data dependent optimizations. Another important research area is effective benchmarks and standard data sets for comparison of volume rendering hardware, algorithms, and packages.

Extension of the warping techniques to free form deformations (FFD's) [SEDE86] would provide parallelism for interactive solid modelling. Because of the unique subset of FFD's that are volume preserving, I believe that an optimal warping algorithm exists for them. Additionally FFD's would allow interactive viewing of sampled data, such as shearing away material instead of simply changing the transparency. This interactivity could provide a better understanding of the 3D nature of the data than possible before.

I have highlighted important research for parallel software. Slowdown compilers can provide unprecedented parallel code portability. Although the development costs are high, standards, and widespread use would provide longevity currently missing in parallel software. Missing technologies for slowdown include general efficient communication decomposition, control of slackness or multithreading bounds necessary for efficiency, and accepted parallel languages. Effective progress requires collaboration and standards.

My investigation of parallel volume rendering has been fruitful, and points to important problems in parallel algorithms research. I have developed optimal parallel volume rendering algorithms, and introduced a methodology to control the decision space when designing parallel algorithms. Further generalization of my approach can increase accessibility of parallel computing.

Bibliography

- [AHO83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Bell Telephone Laboratories, 1983.
- [AHO86] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.
- [ALVE90] G. A. Alverson, W. G. Griswold, D. Notkin, and L. Snyder, "A Flexible Communication Abstraction for Nonshared Memory Parallel Computing," in *Proceedings of Supercomputing '90*, 1990.
- [BAJU92] M. Bajura, H. Fuchs, and R. Ohbuchi, "Merging Virtual Objects with the Real World: Seeing Ultrasound Imagery within the Patient," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 203-210.
- [BARN88] M. Barnsley, *Fractals Everywhere*. San Diego, CA: Academic Press, Inc. 1988.
- [BARR81] A.H. Barr, "Superquadrics and Angle-Preserving Transformations," *IEEE Computer Graphics and Applications*, January 1981, pp. 11-23.
- [BATC80] Kenneth E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol. C-29, No. 9, September 1980, pp. 836-840.
- [BELL92] G. Bell, "Ultra Computers: A Tera Flop Before Its Time", *Communications of the ACM*, Vol 35, No. 8, Aug 1992, pp. 27-47.
- [BENN84] P.P. Bennet and S. A. Gabriel, "System for Spatially Transforming Images," U.S. Patent 4,472,732, Sep. 18, 1984.
- [BERS88] B. N. Bershad, et al. "An Open Environment for Building Parallel Programming Systems," tech. rep. Dept. of Computer Science, University of Washington, Seattle, WA, Jan. 1988, tech rep. 88-01-03.
- [BERS88b] B. N. Bershad, et al. "PRESTO: A System for Object-Oriented Parallel Programming," tech. rep. Dept. of Computer Science, University of Washington, Seattle, WA, Jan. 1988, tech rep. 87-09-01.
- [BLAN90] T. Blank, "The MasPar MP-1 Architecture," in *Proceedings of Comcon Spring 90 The Thirty-Fifth IEEE Computer Society International Conference*, San Francisco, CA Feb. 26-March 2, 1990, pp. 20-24.
- [BLIN82] J. F. Blinn, "Light Reflection Functions for Simulations of Clouds and Dusty Surfaces," *Computer Graphics*, Vol. 16, No. 3, pp. 21-29, July 1982.
- [BLIN90] J. F. Blinn, "Jim Blinn's Corner 'The Truth About Texture Mapping,'" *IEEE Computer Graphics and Applications*, Mar. 1990, pp. 78-83.
- [BRAC86] R. N. Bracewell, *The Hartley Transform*. New York, NY: Oxford University Press, 1986.
- [BRAD92] R. Brady and C. Potter, "A Real-Time 3D Volume Rendering Technique On a Massively Parallel Supercomputer," abstract in *SPIE/ IS&T's Symposium On Electronic Imaging Science and Technology*, San Jose, CA, Feb. 9-14, 1992, p. 104.
- [BRIGG87] W. L. Briggs, *A Multigrid Tutorial*. Philadelphia, PA: Society for Industrial and Applied Mathematics 1987.

- [BRUN90] P. Brunet and I. Navazo, "Solid Representation and Operation Using Extended Octrees," *ACM Transactions on Graphics*, Vol. 9, No. 2, pp. 170-197. Apr. 1990.
- [CAME92] G.G. Cameron and P.E. Undrill, "Rendering Volumetric Medical Image Data on a SIMD Architecture Computer," in Proceedings of the Third Eurographics Workshop on Rendering, Bristol England 17-20 May 1992
- [CANN92] D. Cann, "Retire Fortran? A Debate Rekindled," *Communications of the ACM*, Vol. 35, No. 8, Aug. 1992, pp. 81-89.
- [CARR90] N. Carriero and D. Gelernter, *How To Write Parallel Programs, A First Course*. Cambridge, MA: The MIT Press, 1990.
- [CAST79] K.R. Castleman, *Digital Image Processing*. Prentice Hall, Englewood Cliffs NJ 1979.
- [CATM80] E. Catmull and A.R. Smith "3D Transformations of Images in Scanline Order," *Computer Graphics*, Vol. 14, No. 3, July 1980 pp. 279-285.
- [CHAL92] J. Challenger, "Parallel Volume Rendering on a Shared-Memory Multiprocessor," Technical report UCSC-CRL-91-23, UC Santa Cruz, 1992.
- [CHAN60] S. Chandrasekhar, *Radiative Transfer*. Dover, NY: Oxford University Press, 1960.
- [CHAS89] J. S. Chase, et al., "The Amber System: Parallel Programming on a Network of Multiprocessors," tech. rep. Dept. of Computer Science, University of Washington, Seattle, WA, Sep. 1989, tech rep. 89-04-01.
- [CLIN88] H. E. Cline, W. E. Lorensen, S. Ludke, C.R. Crawford, and B. C. Teeter, "Two Algorithms For the Three-Dimensional Reconstruction of Tomograms," *Medical Physics*, Vol. 15, No. 3, May/Jun. 1988, pp. 320-327. Also in [KAUF91] pp. 64-71.
- [CORM90] T. H. Cormen, C. E. Lieserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 1990.
- [CROW88] F. C. Crow, "Parallelism in Rendering Algorithms," In Proceedings Graphics Interface '88, Edmonton, Alberta, June 6-10, pp. 87-96.
- [CROW89] F. C. Crow, G. Demos, J. Hardy, J. McLaughlin and K. Sims, "3D Image Synthesis on the Connection Machine," in *Proceedings of the Conference on Scientific Applications of the Connection Machine*, World Scientific Publishing Co., PTE Ltd., Ed H. D. Simon, 1989.
- [CURR90] W. Current, P. Hurst, E. Shieh, and I. Agi, "An Evaluation of Radon Transform Computations Using DSP Chips," *Machine Vision and Applications*, Vol. 3, 1990, pp. 63-74.
- [CYBE92] G. Cybenko and D.J. Kuck, "Supercomputers/Reinventing the Machine: Revolution or evolution?," *IEEE Spectrum*, Sept. 1992, pp. 39-41.
- [DANS92] J. Danskin and P. Hanrahan, "Fast Algorithms for Volume Ray Tracing," in Proceedings of 1992 Workshop on Volume Visualization, Boston, Oct. 19-20, 1992, pp. 91-97.
- [DREB88] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume Rendering," *Computer Graphics*, Vol. 22, No. 4, pp. 65-74, Aug. 1988.
- [DUDG84] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1984.

- [DUFF85] T. Duff, "Compositing 3-D Rendered Images," *Computer Graphics*, Vol 19, No. 3, July, 1985, pp. 41-43.
- [DUNN90] S. Dunne, S. Napel, and B. Rutt, "Fast Reprojection of Volume Data," in First Conference on Visualization in Biomedical Computing, May 22-25, 1990, Atlanta GA, pp. 11-18.
- [ELVI92] T. T. Elvins, "A Survey of Algorithms for Volume Visualization," *Computer Graphics*, Vol. 26, No. 3, Aug. 1992, pp. 194-201.
- [ELVI92b] T. T. Elvins, "Volume Rendering on a Distributed Memory Parallel Computer," in IEEE Visualization '92, Boston, MA, Oct. 19-23, 1992, pp. 93-98.
- [ESPO79] L. W. Esposito, "Extensions to the Classical Calculation of the Effect of Mutual Shadowing in Diffuse Reflection," *Icarus*, Vol. 39, pp. 69-80, 1979.
- [FARI88] G. Farin, *Curves and Surfaces for Computer Aided Geometric Design, A Practical Guide.* San Diego, CA: Academic Press, 1988.
- [FEIB80] E. Feibush, M. Levoy, R. Cook, "Synthetic Texturing Using Digital Filters," *Computer Graphics*, Vol. 14, No. 3, pp. 294-301, July 1980.
- [FELD92] Y. Feldman and E. Shapiro, "Spatial Machines: A More Realistic Approach To Parallel Computation," *Communications of the ACM*, Vol. 35, No. 10, Oct. 1992, pp. 60-73.
- [FIRM90] D. N. Firmin et al., "The Application of Phase Shifts in NMR for Flow Measurement," *Magnetic Resonance in Medicine*, Vol. 14, pp. 230-241, 1990.
- [FOLE90] J. Foley, A. vanDam, S.K. Feiner, and J.F. Hughes, *Computer Graphics Principles and Practice, Second Edition.* Reading, MA: Addison Wesley Inc., 1990.
- [FRASE85] D. Fraser, R. A. Schowengerdt, and I. Briggs, "Rectification of Multichannel Images in Mass Storage Using Image Transposition," *Computer Vision Graphics and Image Processing* Vol. 29, No. 1, Jan. 1985, pp. 23-36.
- [FRIE85] G. Frieder, D. Gordon, R. A. Reynolds, "Back-to-Front Display of Voxel-Based Objects," *IEEE Computer Graphics and Applications*, Vol. 5, No. 1, pp. 52-60, Jan. 1985.
- [FRIE88] G. Frieder, O. Frieder, and M. R. Stytz, "A High Performance Parallel Approach to Medical Imaging," in IEEE Second Symposium on The Frontiers of Massively Parallel Computations, 1988 Oct 10-12 Fairfax, VA, pp. 282-288.
- [FUCH89] H. Fuchs et al., "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," *Computer Graphics*, Vol. 23, No. 3, pp. 79-88, July 1989.
- [FUCH89b] H. Fuchs, M. Levoy, and S. Pizer, "Interactive Visualization of 3D Medical Data," *IEEE Computer*, Vol. 22, No. 8, pp. 46-51, Aug. 1989.
- [GALL89] R. Gallagher and J. Nagtegaal, "An Efficient 3-D Visualization Technique for Finite Element Models and Other Course Volumes," *Computer Graphics*, Vol. 23, No. 3, pp. 185-194, July 1989.
- [GEIS90] G. A. Geist, et al., *PICL A Portable Instrumented Communication Library C Reference Manual*, tech. rep., Oak Ridge National Laboratory, ORNL/TM-11130, Oak Ridge, TN, July 1990.

- [GEIS90] G. A. Geist, et al., A User's Guide To PICL A Portable Instrumented Communication Library, tech. rep., Oak Ridge National Laboratory, ORNL/TM-11616, Oak Ridge, TN, Sep. 1990.
- [GELE90] D. Gelernter, A. Nicolau, and D. Padua, Editors, Languages and Compilers for Parallel Computing. Cambridge, MA: MIT Press, 1990.
- [GEME90] GE Medical Systems, "3D Image Display and Manipulation," in Integrated Diagnostics Update, Works-in-progress presentation, GE Medical Systems, P.O. Box 414, Milwaukee, WI 53201, 1990.
- [GIBB88] A. Gibbons and W. Rytter, Efficient Parallel Algorithms. Cambridge University Press, Cambridge, England, 1988.
- [GLAS88] A. S. Glassner, "Space-time Ray Tracing For Animation," *IEEE Computer Graphics and Applications*, Vol. 8 No. 2, pp. 60-70, Mar. 1988.
- [GLASS89] A. S. Glassner, Editor, An Introduction To Ray Tracing. San Diego, CA Academic Press Limited, 1989.
- [GOLD85] S. M. Goldwasser, R. A. Reynolds, T. Bapty, D. Baraff, J. Summers, D. A. Talton, and E. Walsh, "Physician's Workstation with Real-Time Performance," *IEEE Computer Graphics and Applications*, Vol. 5, No. 12, Dec. 1985, pp. 44-56, and in [KAUF91] pp. 321-334.
- [GOLD88] S. M. Goldwasser, R. A. Reynolds, D.A. Talton, and E.S. Walsh, "Techniques for the Rapid Display and Manipulation of 3D Biomedical Data," *Computed Medical Imaging and Graphics*, Vol. 12 No. 1, pp. 1-25, 1988.
- [GOLU89] G. H. Golub and C. F. VanLoan, Matrix Computations, Second Edition. Baltimore, Maryland: The Johns Hopkins University Press, 1989.
- [GOOD84] J. W. Goodman, F. J. Leonberger, S. Y. Kung, and R. A. Athale, "Optical Interconnections for VLSI Systems," *Proceedings of the IEEE*, Vol. 72, No. 7, July 1984, pp. 850-866.
- [GOSH89] A. Goshtasby, "Correction of Image Deformation From Lens Distortion Using Bezier Patches," *Computer Vision, Graphics, and Image Processing*, Vol. 47, No. 3, Sep. 1989, pp. 385-394.
- [GREE89] N. Greene, "Voxel Space Automata: Modeling with Stochastic Growth Processes in Voxel Space," *Computer Graphics*, Vol. 23, No. 3, pp. 175-184, July 1989.
- [GREE92] G. Greenwood, "A Methodology For Mapping Pipelined Algorithms Onto Hypercube Arrays," Ph.D. Dissertation, University of Washington, Dept. of Electrical Engineering 1992.
- [GRIS90] W. G. Griswold, G. A. Harrison, D. Notkin, and L. Snyder, "Scalable Abstractions for Parallel Programming," in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, South Carolina, April 1990.
- [HAAC90] E. M. Haacke et al., "Optimizing Blood Vessel Contrast in Fast Three-Dimensional MRI," *Magnetic Resonance in Medicine*, Vol. 14, pp. 202-221, 1990.
- [HANR90] P. Hanrahan, "Three-Pass Affine Transforms for Volume Rendering," *Computer Graphics*, Vol. 24, No. 5, pp. 71-78, Nov. 1990.
- [HANS92] C. D. Hansen and P. Hinker, "Massively Parallel Isosurface Extraction," in *IEEE Visualization '92*, Boston, MA Oct. 19-23, 1992, pp. 77-83.

- [HARR78] L. D. Harris, R. A. Robb, T. S. Yuen, and E. L. Ritman, "Noninvasive Numerical Dissection and Display of Anatomic Structure Using Computerized X-Ray Tomography," in *SPIE Vol. 152, Recent and Future Developments in Medical Imaging*, 1978, pp. 10-18.
- [HARR90] W. L. Harrison III and Z. Ammarguellat, "A Comparison of Automatic Versus Manual Parallelization of the Boyer-Moore Theorem Prover," in D. Gelernter, A. Nicolau, and D. Padua, Editors, *Languages and Compilers for Parallel Computing*. Cambridge, MA: MIT Press, 1990, pp. 402-422.
- [HATC91] P. J. Hatcher et al. "Short Notes-Data-Parallel Programming on MIMD Computers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 2, July 1991, pp. 377-383.
- [HECK86] P.S. Heckbert "Survey of Texture Mapping," *IEEE Computer Graphics and Applications* Vol 6, No. 11, November 1986 56-67.
- [HENN90] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann 1990.
- [HERM92] G. T. Herman, J. Zheng, C. A. Bucholtz, "Shape Based Interpolation," *IEEE Computer Graphics and Applications*, Vol. 12, No. 3, May 1992, pp. 69-79.
- [HILL85] W. D. Hillis, *The Connection Machine*. Cambridge, MA: The MIT Press, 1985.
- [HIRA92] S. Hiranandani, K. Kennedy, C. Tseng, "Compiling Fortran D: for MIMD Distributed Memory Machines," *Communications of the ACM*, Vol. 35, No. 8, Aug. 1992, pp. 66-80.
- [INMO84] Inmos Limited, *Occam Programming Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [JACK88] D. Jackel and W. Strasser, "Reconstructing Solids from Tomographic Scans, The PARCUM II System," in *Advances in Computer Graphics Hardware II*. Springer International, 1988, pp. 101-109. Also in [KAUF91] pp. 358-371.
- [JAFF82] S. M. Jaffey and K. Dutta, "Digital Perspective Correction for Cylindrical Holographic Stereograms," in *SPIE Vol. 367, Processing and Display Of Three Dimensional Data*, 1982, pp. 130-140.
- [JAIN89] A. K. Jain, *Fundamentals of Digital Image Processing*. Englewood Cliffs, N J: Prentice Hall, 1989.
- [KABA92] J. Kaba, J. Matey, G. Stoll, H. Taylor, and P. Hanrahan, "Interactive Terrain Rendering and Volume Visualization on the Princeton Engine," in *IEEE Visualization '92*, Boston, MA Oct. 19-23, 1992, pp. 349-355.
- [KAHA89] D. Kahaner, C. Moler, S. Nash, *Numerical Methods and Software*. Prentice Hall, Englewood Cliffs, New Jersey 1989.
- [KAJI84] J. T. Kajiya, "Ray Tracing Volume Densities," *Computer Graphics*, Vol. 18, No. 3, pp. 165-174, July 1984.
- [KAJI86] J. T. Kajiya, "The Rendering Equation," *Computer Graphics*, Vol. 20, No. 4, pp.143-150, Aug. 1986.
- [KAK88] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*. New York, NY: IEEE, 1988.

- [KAUF87] A. Kaufman, "Efficient Algorithm for 3D Scan-Conversion of Parametric Curves, Surfaces, and Volumes," *Computer Graphics*, Vol. 21, No. 4, pp. 171-179, July 1987.
- [KAUF88] A. Kaufman and R. Bakalash, "Memory and Processing Architecture for 3D Voxel-Based Imagery," *IEEE Computer Graphics and Applications*, Vol. 8 No. 6, pp. 10-23, Nov. 1988.
- [KAUF90] A. Kaufman, R. Bakalash, D. Cohen, and R. Yagel, "A Survey of Architectures for Volume Rendering," *IEEE Engineering In Medicine And Biology*, pp. 18-23, Dec. 1990, also appears as [KAUF91b]
- [KAUF91] A. Kaufman, Editor, Volume Visualization. Washington, D.C.: IEEE Computer Society Press, 1991.
- [KAUF91b] A. Kaufman, R. Bakalash, D. Cohen, and R. Yagel, "Chapter6: Architectures for Volume Rendering," in Volume Visualization, A. Kaufman, Editor. Washington, D.C.: IEEE Computer Society Press, 1991, pp. 331-320, also appeared as [KAUF90].
- [KLAS87] R. Klassen, "Modeling the Effect of the Atmosphere on Light," *ACM Transactions on Graphics*, Vol. 6, No. 3, pp. 215-237, July 1987.
- [KOCH89] P. D. Kochevar, "Computer Graphics On Massively Parallel Machines," Ph.D. Dissertation, Dept. of Computer Science, Cornell University, 1989.
- [KRUE90] W. Krueger, "Volume Rendering and Data Feature Enhancement," *Computer Graphics*, Vol 24, No. 5, pp. 21-26, Nov. 1990.
- [KRUS85] Kruskal, L. Rudolph, and M. Snir, "The Power of Parallel Prefix," in Proceedings IEEE International Parallel Processing Symposium, 1985, pp. 180-185.
- [KUNG88] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice Hall 1988.
- [LADN80] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, Vol. 27, No. 4, Oct. 1980, pp. 831-838.
- [LAUB90] G. Laub, "Displays for MR Angiography," *Magnetic Resonance in Medicine*, Vol. 14, pp. 222-229, 1990.
- [LAUR91] D. Laur and P. Hanrahan, "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *Computer Graphics*, Vol. 25, No. 4, July 1991, pp. 285-288.
- [LEIG92] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, CA: Morgan Kaufmann, 1992.
- [LENZ86] R. Lenz, B. Gudmundsson, B. Lindskog, and P. Danielsson, "Display of Density Volumes," *IEEE Computer Graphics and Applications*, Vol. 6, No. 7, pp. 20-29, July 1986.
- [LEVI84] Levinthal, A. and Porter, T., "Chap - A SIMD Graphics Processor," *Computer Graphics*, Vol. 18, No. 3, July, 1984, pp. 77-82.
- [LEVO89] M. Levoy, "Display of Surfaces From Volume Data," Ph.D. Dissertation, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, May 1989.
- [LEVO90] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Transactions on Graphics*, Vol. 9, No. 3, pp. 245-261, July 1990.

- [LEVO90b] M. Levoy and R. Whitaker, "Gaze-Directed Volume Rendering," *Computer Graphics*, Vol.24, No. 2, pp. 217-223, March 1990.
- [LEVO90c] M. Levoy, "A Hybrid Ray Tracer for Rendering Polygon and Volume Data," *IEEE Computer Graphics and Applications*, Vol. 10, No. 2, Mar. 1990, pp. 33-40.
- [LEVO90d] M. Levoy, "Design for a Real-Time High Quality Volume Rendering Workstation," *Computer Graphics Tutorial*, 1990, pp. 224-232.
- [LEVO90e] M. Levoy, H. Fuchs, S. M Pizer, J. Rosenman, E. L. Chaney, G. W. Sherouse, V. Interrante, and J. Kiet, "Volume Rendering in Radiation Treatment Planning," in *Proceedings of The First Conference on Visualization in Biomedical Computing*, IEEE Computer Society Press, May 1990, pp. 4-10.
- [LEVO92] M. Levoy, "Volume Rendering Using The Fourier Projection-Slice Theorem," in *Proceedings Graphics Interface '92*, Vancouver, Canadian Information Processing Society, May 1992.
- [LI91a] J. Li and M. Chen, "Compiling Communication- Efficient Programs for Massively Parallel Machines," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, July 1991, pp. 361-376.
- [LI91b] J. Li and L. H. Jamieson, "A System for Algorithm-Architecture Mapping Based on Dependence Graph Matching and Hypergraphs," in *Fifth International Parallel Processing Symposium*, Anaheim CA, April 30 - May 2, 1991, pp. 513-518.
- [LOWM91] P. Lowman and J. Stokes, *Introduction To Linear Algebra*. New York, NY: Books For Professionals, 1991.
- [LORE87] W. Lorensen, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics*, Vol. 21, No. 4, pp. 163-169, July 1987.
- [MACH92] R. Machiraju, L. Schwiebert, and R. Yagel, "Parallel Algorithms for Volume Rendering," Dept. of Computer and Information Science, The Ohio State Univ. OSU-CISRC-10/92-TR29, Oct. 17, 1992.
- [MAIL92] P. Maillot, "A New, Fast Method for 2-D Polygon Clipping: Analysis and Software Implementation," *ACM Transactions on Graphics*, Vol. 11, No. 3, July 1992, pp. 276-290.
- [MALZ90] T. Malzbender, "Hierarchically Compositing Ray Cast Volume Rendering," HP Laboratories Technical Report HPL-90-28, Apr. 1990.
- [MALZ90b] T. Malzbender, personal communication.
- [MALZ93] T. Malzbender, "Fourier Volume Rendering," to appear in *ACM Transactions on Graphics*, 1993.
- [MANI89] E. S. Maniloff and K. M. Johnson, "Holographic Routing Network for Parallel Processing Machines," in *Proceedings Holographic Optics II: Principles and Applications*, Editor G. M. Morris, 25-28 April 1989, pp. 283-289.
- [MARS90] R. Marshall, J. Kempf, S. Dyer, "Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie," *Computer Graphics*, Vol. 24, No. 2, pp. 89-97, March 1990.
- [MASP91] *Data-Parallel Programming Guide*. Sunnyvale, CA: MasPar Corp. 1991.
- [MEAG82] D. Meagher, "Efficient Synthetic Image Generation of Arbitrary 3D Objects," in *Proceedings of IEEE Computer Society Conference on Pattern Recognition and*

- Image Processing*, IEEE Computer Society Press, Washington DC, 1982, pp. 473-478.
- [MEAG82] D. Meagher, "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing*, Vol. 19, No. 2, pp. 129-147, June 1982.
- [MEAG84] D. Meagher, "A New Mathematics for Solids Processing," *Computer Graphics World*, Oct. 1984.
- [MEAG85] D. Meagher, "Applying Solids Processing To Medical Planning," in *Proceedings of NCGS '85*, Dallas, TX, 1985, pp. 101-109. Also in [KAUF91] pp. 372-378.
- [MEAG91] D. Meagher, "Fourth-Generation Computer Graphics Hardware Using Octrees," NCGA '91, in press.
- [MESE83] B. E. Meserve. *Fundamental Concepts of Geometry*. Toronto, Ontario: Dover Publications, 1983.
- [MIDK90] S. P. Midkiff, D. A. Padua, and R. Cytron, "Compiling Programs with User Parallelism," in D. Gelernter, A. Nicolau, and D. Padua, Editors, *Languages and Compilers for Parallel Computing*. Cambridge, MA: MIT Press, 1990, pp. 402-422.
- [MOLN92] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 231-240.
- [MONT92] C. Montani, R. Perego, and R. Scopigno, "Parallel Volume Visualization on a Hypercube Architecture," in *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Oct. 19-20, 1992, pp. 9-16.
- [MITC88] D.P. Mitchell and A. N. Netravali, "Reconstruction Filters in Computer Graphics," *Computer Graphics*, Vol. 22, No. 4, pp 221-228, Aug. 1988.
- [NICK90] J. R. Nickolls, "The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer," in *Proceedings of Comcon Spring 1990*, San Francisco, CA, Feb. 26- Mar. 2, 1990, pp. 25-28.
- [NIEH92] J. Nieh and M. Levoy, "Volume Rendering on Scalable Shared-Memory MIMD Architectures," in *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Oct. 19-20, 1992, pp. 17-24.
- [NEYF90] Derek R. Ney, Elliot K. Fishman, and Donna Magid, "Volumetric Rendering of Computed Tomography Data: Principles and Techniques," *IEEE Computer Graphics and Applications*, Vol. 10 No. 2, March 1990, pp. 24-32.
- [NING92] P. Ning and L. Hesselink, "Vector Quantization for Volume Rendering," in *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Oct. 19-20, 1992, pp. 69-74.
- [NISH90] D. G. Nishimura, "Time-of-Flight MR Angiography," *Magnetic Resonance In Medicine*, Vol 14, pp. 194-201, 1990.
- [OHAS85] T. Ohashi, T. Uchicki, and M. Tokoro, "A Three-Dimensional Shaded Display Method for Voxel-Based Representations," in *Proceedings of Eurographics 1985*, C. E. Vandoni, Editor, Eurographics Association, Sep. 1985, pp. 221-232. and in [KAUF91] pp. 335-343.
- [OPPE89] A.V. Oppenheim and R. W. Schaffer, *Discrete-Time Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall 1989.

- [OWCZ89] J. Owczarczyk, W.J. Welsh, and S. Searby, "Performance Analysis of Image Registration Techniques," in IEE Third International Conference on Image Processing and Its Applications, Univ. of Warwick, U.K., 18-20 July 1989, pp. 10-13.
- [PAET86] A.W. Paeth, "A Fast Algorithm For General Raster Rotation," Proceedings Graphics Interface 1986 Vision Interface 1986 26-30 May 1986 Canadian Information Processing Society Vancouver, BC.
- [PAIN89] J. Painter and K. Sloan, "Antialiased Ray Tracing by Adaptive Progressive Refinement," *Computer Graphics*, Vol. 23, No. 3, pp. 281-288, July 1989.
- [PANG90] A. T. Pang, "Line-Drawing Algorithms for Parallel Machines," *IEEE Computer Graphics and Applications*, Vol. 10, No. 5, Sep. 1990, pp. 54-59.
- [PORT84] T. Porter and T. Duff, "Compositing Digital Images," *Computer Graphics*, Vol. 18, No. 3, pp. 253-259, July 1984.
- [POTM89] M. Potmesil and E. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, Vol. 23, No. 3, pp. 69-78, July 1989.
- [PRAT78] W. K. Pratt, *Digital Image Processing*. New York, NY: John Wiley & Sons, Inc. 1978.
- [PRESS88] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C, The Art of Scientific Computing*. New York, NY: Cambridge University Press 1988.
- [REEV83] W. Reeves, "Particle Systems-A Technique for Modeling a Class of Fuzzy Objects," *ACM Transactions on Graphics*, Vol. 2, No. 2, pp. 91-108, Apr. 1983.
- [RICE88] M. D. Rice, S. B. Seidman, and P. Y. Wang, "A Formal Model for SIMD Computation," in *IEEE Second Symposium on The Frontiers of Massively Parallel Computations*, 1988 Oct 10-12 Fairfax, VA, pp. 601-607.
- [SABE88] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *Computer Graphics*, Vol. 22, No. 4, pp. 51-58, Aug. 1988.
- [SCHR91] P. Schroeder and J. B. Salem, "Fast Rotation of Volume Data on Data Parallel Architectures," in *Proceedings IEEE Visualization '91*, San Diego, CA Oct. 22-25, 1991, pp. 50-57.
- [SCHR91b] P. Schroeder and J. B. Salem, "Fast Rotation of Volume Data on Data Parallel Architectures," Technical Report, TMC-195, Thinking Machines Corporation, 1991.
- [SCHR92] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, "Decimation of Triangle Meshes," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 65-70.
- [SCHR92b] P. Schroeder, G. Stoll, "Data Parallel Volume Rendering as Line Drawing," in *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Oct. 19-20, 1992, pp. 25-32.
- [SEDE86] T. W. Sederberg and S.R. Parry, "Free-Form Deformation of Solid Geometric Models," *Computer Graphics*, Vol. 20, No. 4, Aug. 1986, pp. 151-160.
- [SIEG87] H.J. Siegel, T. Schwederski, J. T. Kuehn, and N.J. Davis IV, "An Overview of the PASM Parallel Processing System," in *Tutorial: Computer Architecture*, edited by D.D. Gajski, V.M. Milutinovic, H.J. Siegel, and B.P. Furht, IEEE Computer Society Press, Washington, DC 1987 pp. 387-407.

- [SMIT87] A. R. Smith, "Planar 2-Pass Texture Mapping and Warping," *Computer Graphics* Vol 21, No. 4, July 1987 pp. 263-272.
- [SOCH90] D. G. Socha, "Compiling Single-Point Iterative Programs for Distributed Memory Computers," In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC., Apr. 1990.
- [SOCK91] D. G. Socha, Supporting Fine-Grain Computation on Distributed Memory Parallel Computers. Seattle, WA: University of Washington, Ph.D. thesis, 1991.
- [SOMA91] A. Somani et al., "Proteus System Architecture & Organization," in *Fifth International Parallel Processing Symposium*, Anaheim CA, April 30 - May 2, 1991, pp. 276-284.
- [STOC85] M. Stock, A Practical Guide to Graduate Research. New York, NY: McGraw Hill, 1985.
- [STON87] Harold S. Stone, *High Performance Computer Architecture*. Reading, MA: Addison Wesley, 1987.
- [STRE92] D. Stredney, R. Yagel, S. F. May, and M. Torello, "Supercomputer Assisted Brain Visualization with an Extended Ray Tracer," in *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Oct. 19-20, 1992, pp. 33-38.
- [TANA86] A. Tanaka, M. Kaneyama, S. Kazama, and O. Watanabe, "A Rotation Method For Raster Image Using Skew Transformation," *Proceedings IEEE Conference on Computer Vision and Pattern Recognition* (June 1986) pp. 272-277.
- [THIN89] Thinking Machines Corp., *Connection Machine Model CM-2 Technical Summary, Version 5.1*. Cambridge, MA: Thinking Machines Corp., 1989.
- [THIN88] Thinking Machines Corp., **Lisp Reference Manual, Version 5.0*. Cambridge, MA: Thinking Machines Corp., 1988.
- [THOM91] K.K. Thompson, Ray Tracing With Amalgams. Ph.D. Dissertation University of Texas, Austin, 1991.
- [TIED90] U. Tiede et al., "Investigation of Medical 3D-Rendering Algorithms," *IEEE Computer Graphics and Applications*, Vol. 10, No. 2, pp. 41-53, Mar. 1990.
- [TURK92] G. Turk, "Re-Tiling Polygonal Surfaces," *Computer Graphics*, Vol. 26, No. 2, July 1992, pp. 55-64.
- [UDUP90] J. K. Udupa and H. M. Hung, "Surface Versus Volume Rendering A Comparative Assessment," in *First Conference on Visualization in Biomedical Computing*, May 22-25, 1990, Atlanta GA, pp. 83-91.
- [UPSO88] C. Upson, and M. Keeler, "V-Buffer: Visible Volume Rendering," *Computer Graphics*, Vol. 22, No. 4, pp. 59-64, Aug. 1988.
- [VALI90] L. G. Valiant, "General Purpose Architectures," *The Handbook of Theoretical Computer Science*, Vol 1 Chap. 18, J. Van Leeuwen, Ed, 1990.
- [VALI90b] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, Vol. 33, No. 8, August 1990, pp. 103-111.
- [VEZI92] G. Vezina, P. A. Fletcher, and P. K. Robertson, "Volume Rendering on the MasPar MP-1," in *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Oct. 19-20, 1992, pp. 3-8.

- [VETT88] W. T. Vetterling, S. A. Teukolsky, W. H. Press, and B. P. Flannery, Numerical Recipes Example Book (C). New York, NY: Cambridge University Press 1988.
- [WEIM80] C. F. R. Weiman, "Continuous Anti-Aliased Rotation and Zoom of Raster Images," *Computer Graphics*, Vol. 14, No. 3, July 1980, pp. 286–293.
- [WEIN90] F. Weinhaus and M. Wallerman, "A Flexible Approach To Image Warping," *SPIE Vol 1244 Image Processing Algorithms and Techniques (1990)* pp. 108–122.
- [WEST89] L. Westover, "Interactive Volume Rendering," in C. Upson, Editor, *Proceedings of the Chapel Hill Workshop on Volume Visualization*, Univ. of North Carolina, ACM, May 1989, pp. 9-16.
- [WEST90] L. Westover, "Footprint Evaluation for Volume Rendering," *Computer Graphics*, Vol. 24, No. 4, pp. 367-376, Aug. 1990.
- [WEST92] L. Westover. Splatting: A parallel, feed-forward volume rendering algorithm. North Carolina Chapel Hill, University of North Carolina at Chapel Hill, Doctoral dissertation, 1991.
- [WILH91] J. Wilhems, "Decisions in Volume Rendering," *SIGGRAPH 91 Course Notes*, Vol. 8 State of The Art in Volume Visualization, pp. I.1-I.11.
- [WILH91b] J. Wilhelms, A.V. Gelder, "A Coherent Projection Approach for Direct Volume Rendering," *Computer Graphics*, Vol. 25, No. 4, 1991.
- [WILH92] J. Wilhelms and A.V. Gelder, "Octrees for Faster Isosurface Generation," *ACM Transactions on Graphics*, Vol. 11, No. 3, July 1992, pp. 201-227.
- [WITT91] C. M. Wittenbrink and A. K. Somani, "Cache Tiling for High Performance Morphological Image Processing," in *CAMP 91, Computer Architecture For Machine Perception*, Paris, France, Dec. 16-18, 1991, pp. 427-438. Prize paper. Full paper to appear in *Machine Vision and Applications*, 1993.
- [WITT92] C. M. Wittenbrink, "The Theory and Practice of Speedup Through Slowdown," technical report FTCL, July, 1992.
- [WITT93] C. M. Wittenbrink and A. K. Somani, "2D and 3D optimal parallel image warping," in *Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 13-16, 1993, pp. 331-337.
- [WITT93b] C. M. Wittenbrink and A. K. Somani, "Improved Filters and View Flexibility for Data Parallel Volume Rendering," to appear in the *Parallel Rendering Symposium, Visualization '93*, San Jose, CA, October 25-26, 1993.
- [WITT93c] C. M. Wittenbrink and A. K. Somani, "Permutation Warping for Volume Rendering," in *Proceedings of the Fifth Annual Western Computer Graphics Symposium*, Silver Star Mountain, British Columbia, March 28-30, 1993.
- [WOLB89] G. Wolberg and T.E. Boult, "Separable Image Warping With Spatial Lookup Tables," *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 369-378.
- [WOLB90] G. Wolberg "Digital Image Warping," IEEE Computer Society Press, Washington DC 1990.
- [WOLF88] S. Wolfram, *Mathematica, A System For Doing Mathematics By Computer*. Reading, MA: Addison-Wesley, 1988.
- [WOLF89] M. Wolfe, *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press, 1989.

- [WRIG90] W. E. Wright, "Parallelization of Bresenham's Line and Circle Algorithms," IEEE Computer Graphics and Applications, Vol. 10, No. 5, Sep. 1990, pp. 60-67.
- [WRIG92] J. R. Wright and J. C. L. Hsieh, "A Voxel-Based, Forward Projection Algorithm for Rendering Surface and Volumetric Data," in IEEE Visualization '92, Boston, MA Oct. 19-23, 1992, pp. 340-348.
- [YAGE92] R. Yagel and A. Kaufman, "Template-Based Volume Viewing," in Eurographics '92, A. Kilgour and L. Kjelldahl, Editors, Vol. 11, No. 3, 1992, pp. C-153 to C-167.
- [YAGE92b] R. Yagel, "High Quality Template-Based Volume Viewing," Technical report OSU-CISRC-10/92-TR28, The Ohio State University, Columbus, OH, 1992.
- [YOKO86] N. Yokobori, P.S. Yeh, and A. Rosenfeld, "Selective Geometrical Correction of Images," In IEEE Computer Society Conference on Computer Vision and Pattern Recognition, June 22-26, 1986, Miami Beach FL. pp. 530-533.
- [YOO91] 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," in Proceedings IEEE Visualization '91, San Diego, CA, Oct. 22-25, 1991, pp. 58-65.
- [ZORP92] G. Zorpette, "Supercomputers/Reinventing the Machine: The Power of Parallelism," IEEE Spectrum, Sept. 1992, pp. 28-33.

Appendix A

Glossary

affine transform	A map from one space to another that preserves ratios of distances and parallel lines, but does not preserve angles.
albedo	Reflectiveness or proportion of light reflected from a particle versus the light impinging.
alias	Multiple frequencies are seen as the same frequency because of inadequate sampling or reconstruction.
anisotropic medium	The phase function is dependent upon more than just the phase angle, such as gradient within a volume or the direction of a surface.
bilinear	A first order hold in two dimensions that has a cross term xy .
brightness	The perceived intensity of an object, not to be confused with intensity the measured intensity of an object. Brightness is different than intensity because of the psychological and physiological factors in perception.
CRCW	See PRAM.
CREW	See PRAM.
data dependent	For parallel algorithms, it is when the location of the data is given, but is not known apriori. Typically a linked list of values, and you are given only the head.
data independent	For parallel algorithms, when the data are strictly found by location and doesn't vary with the input.
EREW	See PRAM.
equiareal	A transform T whose determinant satisfies $\det(T) = \pm 1$ [MESE83].
frustum	Viewing pyramid formed by projecting the screen into object space.
globbing	Slang for grouping together, a way to describe virtualizing jobs.
initial prefix	Evaluation of all partial products of an associative operator. Examples are calculation of carries in addition, and compositing for volume rendering.
intensity	I , or radiant intensity, the amount of measured light energy.
isotropic medium	Phase function or reflectance function depends only upon the phase angle.

linear transform	A map from one vector space to another that preserves linear combinations.
moire	Beat patterns that arise if the image contains periodicity that are close to half the sampling frequency.
MCCM	Mixed cost communication machine. A theoretical machine similar to a PRAM, but takes into account the interconnectivity and communication costs.
opacity	α , the density of matter, or a measure of how opaque an object is. Values are from 0 to 1. See transparency.
optical depth	Describes amount of attenuation as light passes through a particular volume.
optimal efficiency	Work efficiency, or time for the parallel algorithm times the number of processors equals the time for the fastest sequential algorithm.
optimal run time	For the model of computation is typically a lower bound given the strength of the machine.
optimal space complexity	$O(n)$ on the order of the number of input elements.
optimal speedup	Linear speedup of the parallel program over the fastest known sequential program.
perf.	Performance, the speed of execution, such as frames/second.
parallel prefix	Evaluation of an initial prefix operation done in parallel.
phase angle	The angle between incident light and emitted/reflected light.
pixel	Picture element. The individual point light sources in a raster display.
PRAM	Parallel random access machine. A parallel theoretical machine model that has multiple processors that are strictly synchronized, and memory is readable by random access. The memory is typically restricted by disallowing concurrent reads or concurrent writes. The typical variants are: CRCW - concurrent read concurrent write CREW - concurrent read exclusive write EREW - exclusive read exclusive write
RAM	Random access machine. A theoretical machine used to develop sequential algorithms by comparing their asymptotic run times.
segment	Separate regions in an image as to their membership in desired sets.
shear	A transform that affects only one coordinate.

slackness	The amount of excessive parallelism in an application which allows for bundling of processing for better asymptotic bounds on theoretical machines. See [VALI90b].
speedup	The ratio of execution time without the improvement over the time with the improvement.
toroidally connected	Connected by modulus wrap around.
transform	The process of sending a point, image, or object into another space, commonly meaning a geometric transform of a coordinate. Can also mean calculating an alternative representation of an image such as the frequency representation calculated by the discrete Fourier transform.
transparency	t , the clearness of an object. It equals $1 - \alpha$, and varies from 0 to 1.
tril.	Trilinear, a first order hold in three dimensions called that has a cross term of xyz .
volume rendering	Creating a 2D image from 3D voxels using transparency/opacity effects.
voxel	Volume element, an abbreviation analogous to pixel. A volumetric data element within an image data cube.
warping	Spatial image transform such as rotation.

Appendix B

Derivation of Compositing Complexity

2.1 Background

There are many ways to compute volumetric compositing for a single ray of W sample positions. I derive the complexity of each alternative, and show the most efficient sequential and parallel methods. Compositing combines two image intensities taking into account their opacity, or opaqueness, ability to block light. The image in front will partially block out the image behind depending on its opacity value, α , which ranges from 0 to 1. If $\alpha = 1$ the image behind will be completely occluded and will not contribute anything.

A stack of images can be processed back-to-front, front-to-back, or any position within the stack. Trade-offs in partial updates, parallelism, and algorithmic optimizations create different complexities for each method. The asymptotic complexity is the same for the sequential methods $O(W)$, where, W is the number of images in the stack, and the parallel approaches are $O(\log W)$, but differences in constant notation allow for selecting the most efficient method. Sequentially the best is back-to-front, but if using adaptive ray termination then front-to-back must be used. In parallel, binary tree compositing is the most efficient, and a front-to-back progressive deepening may be used.

Compositing can use opacity or transparency based equations. A view ray passes through the stack of images and the final intensity of each ray is dependent upon all of the images. The ray's intersection with each image is a sample point in the volume considered a leaf of a tree. The internal nodes of the tree denote compositing calculations. Additionally, at each internal node not only the composited intensity is calculated, but also a combined transparency or opacity to be used in following computations. Five methods are presented described and the constant complexity is derived for each method. The five methods are front-to-back [LEVO90], back-to-front [LEVO90], binary-tree fully parallel, binary-tree front-to-back, and sum of attenuated emittances. I conclude by comparing the complexities of all of the methods. I show that calculating with transparencies is more efficient in all methods except back-to-front where it has the same complexity. It is therefore prudent to use transparency, especially in the parallel evaluation methods. I also claim my three parallel methods have optimal efficiency, and allow for different approaches varying communication and/or ray processing termination conditions. Adaptive ray termination may reduce the amount of computation and is described in section 1.3 and section 2.5.

2.2 Back To Front Compositing

I derive the compositing complexity step by step. The first step is to premultiply the shading intensity at each sample with the opacity to give an emitted intensity,

$$(I_{E_i} = I_{S_i}\alpha_i), i \in W, \quad (\text{EQ 83})$$

where I define the opacity (α_i), shading intensity (I_{S_i}), sample location (i), and the number of sample levels (W). If samples are combined from back to front intensities are calculated using,

$$I_{E_{ij}} = I_{E_i} + I_{E_j}(1 - \alpha_i) \quad (\text{EQ 84})$$

$$\alpha_{ij} = \alpha_i + \alpha_j(1 - \alpha_i), \quad (\text{EQ 85})$$

where i is the image in front and j is the image behind. Combine first the W^{th} and the $(W-1)^{\text{st}}$ sample point, then the $I_{E_{(W-1)W}}$ intensity is combined with the $(W-2)^{\text{th}}$ sample point and so on. FIGURE 88 shows how the compositing is performed. Each sample point is a leaf of the tree, and each internal node represents both an intensity, and the computation to calculate that intensity.

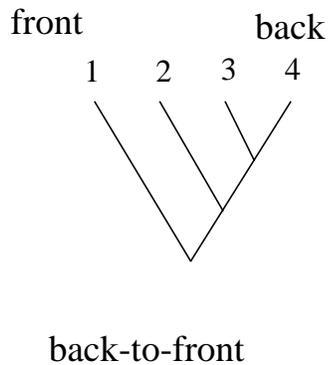


FIGURE 88

Back-To-Front Compositing Tree

FIGURE 89 shows an example compositing 4 image samples along one ray, $W = 4$. Assume the preprocessing stage of the algorithm provides the shading intensity, I_{S_i} , and opacities, α_i . Samples are labelled 1 to 4 with sample 1 at the front of the volume and sample 4 at the rear of the volume in FIGURE 89. Step 1 is the premultiplication to calculate the emitted intensities. Step 2 is the compositing of images (3) and (4). Step 3 is compositing image (2) with the previously combined (34) image. Step 4 is combining im-

age (1) with the (234) result. Notice that the updated opacities aren't used, and therefore did not need to be calculated.

$$\begin{array}{llllll}
 \text{Step 1} & I_{E_1} = \alpha_1 I_{S_1} & I_{E_2} = \alpha_2 I_{S_2} & I_{E_3} = \alpha_3 I_{S_3} & I_{E_4} = \alpha_4 I_{S_4} & W \text{ mult.} \\
 \text{Step 2} & & & & I_{E_{34}} = I_{E_3} + I_{E_4}(1 - \alpha_3) & \\
 & & & & \alpha_{34} = \alpha_3 + \alpha_4(1 - \alpha_3) & \\
 \text{Step 3} & & & I_{E_{234}} = I_{E_2} + I_{E_{34}}(1 - \alpha_2) & & \\
 & & & \alpha_{234} = \alpha_2 + \alpha_{34}(1 - \alpha_2) & & \\
 \text{Step 4} & I_{E_{1234}} = I_{E_1} + I_{E_{234}}(1 - \alpha_1) & & & & \\
 & \alpha_{1234} = \alpha_1 + \alpha_{234}(1 - \alpha_1) & & & & \\
 & & & & & W - 1 \text{ composited} \\
 & & & & & \text{intensities.}
 \end{array}$$

FIGURE 89

Back-To-Front Compositing Calculations

Because the opacities don't need to be updated the complexity is the same using either opacity or transparency. The number of calculations necessary is W multiplications for the emitted intensity calculation, Step 1, and $W - 1$ compositing operations, or the number of non-leaf nodes in the compositing tree, FIGURE 88. The cost of a compositing emitted intensities is 2 additions and 1 multiplication, (EQ 84). The total is $2W - 1$ multiplications and $2W - 2$ additions. This seems to be the most efficient way to composite, so why consider other methods? For sequential implementations a substantial savings may be achieved by using adaptive ray termination [LEVO90][DANS92]. This is a method for terminating processing on a ray once the ray becomes more opaque than a selected threshold. Adaptive ray termination can only be done when processing from front-to-back because the final intensity is a result from all objects in the volume and any ray may be occluded. Back-to-front cannot determine occlusion until the processing reaches the front of the volume. Also, progressive refinement [FOLE90] must be done by traversing the ray from front-to-back. This allows incrementally updating of an image that immediately represents an approximation of the image, and incremental improvements occur as each ray is further processed.

2.3 Front To Back Compositing

Front to back compositing allows adaptive ray termination and/or progressive refinement during the computation of an image. Calculate emitted intensities from samples exactly as shown in the previous section, but we change the order. This requires calculating a composited opacity which is used in following computations. FIGURE 90 shows front-to-back compositing with circled nodes representing those nodes at which we must also calculate a composited opacity value. FIGURE 91 shows the calculations taking place in FIGURE 90.

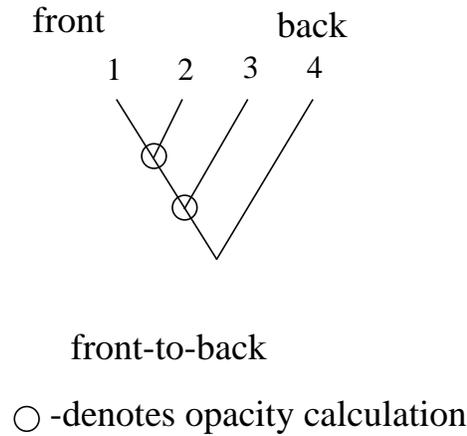


FIGURE 90 Front To Back Compositing Tree

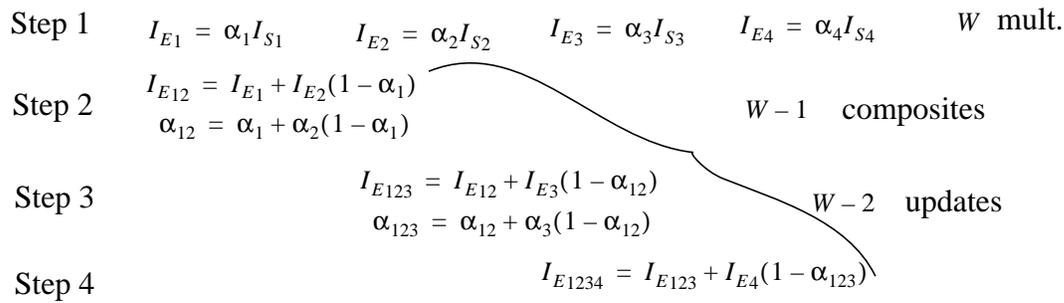


FIGURE 91 Front-To-Back Compositing Calculations

There are W initial premultiplies, $W - 1$ intensity composites, and $W - 2$ opacity composites. Again an intensity, or opacity, composite takes 2 additions and 1 multiply. The total for front-to-back compositing using opacity is $W + (2W - 3)$ multiplications and $2(2W - 3)$ additions, for a total of $3W - 3$ multiplications and $4W - 6$ additions. Saving transparency calculations $t = 1 - \alpha$ when computing intensities saves $W - 2$ additions for a total of $3W - 4$ additions. As described earlier compositing with transparencies is more efficient. Compositing versed in terms of transparency replacing (EQ 84) and (EQ 85) is,

$$I_{Eij} = I_{Ei} + I_{Ej}t_i \tag{EQ 86}$$

$$t_{ij} = t_i t_j. \tag{EQ 87}$$

Transparency calculations are included in the initialization,

$$t_i = (1 - \alpha_i) . \quad (\text{EQ 88})$$

FIGURE 92 shows computing the premultiplied emitted intensity, the transparencies, and then compositing from front-to-back calculating the tree in FIGURE 90.

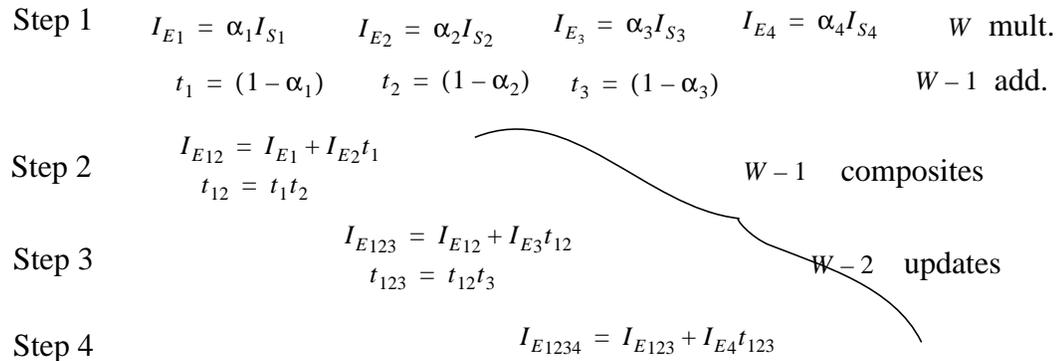


FIGURE 92 Front-To-Back Compositing with Transparency

The complexity is W multiplications and $W - 1$ additions for step 1 and $W - 1$ composited intensities for steps 2-4 and $W - 2$ updated transparencies for steps 2 and 3. The totals are $W + W - 1 + W - 2 = 3W - 3$ multiplications and $(W - 1) + (W - 1) = 2W - 2$ additions. Transparency compositing is more efficient than opacity compositing, but processing is not always sequential. I discuss parallel compositing in the next Section.

2.4 Parallel Binary Tree Compositing

Compositing is associative (See Chapter IV) and can be calculated optimally in a binary tree fashion. This is true because of arbitrary groupings shown below by the “over” operator [DREB88]. 4 image intensities can be grouped in any associative fashion.

1. back-to-front: $I = (I1 \text{ over } (I2 \text{ over } (I3 \text{ over } I4)))$
2. front-to-back: $I = (((I1 \text{ over } I2) \text{ over } I3) \text{ over } I4)$
3. binary-tree: $I = ((I1 \text{ over } I2) \text{ over } (I3 \text{ over } I4))$

Each method computes the same value. In parallel the associative groupings allow variety as in the sequential cases. I ignore communication costs and derive the constant notation complexity for each method.

Taking the associative groupings of three images to combine them pair wise may be done by either ((I1 over I2) over I3), or (I1 over (I2 over I3)). If W is a power of 2 there is no choice. From the derivation of the back-to-front and front-to-back schemes I showed that internal nodes that are near the front require updated transparencies (opacities) and internal nodes on the back edge of the tree do not. I represent the two groupings of three images in FIGURE 93.

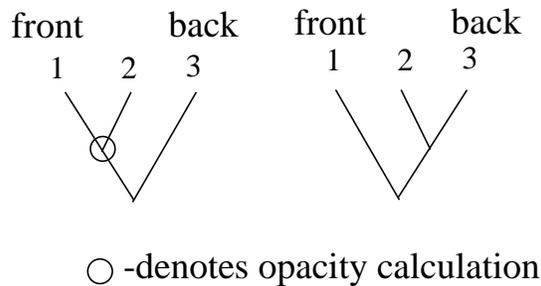


FIGURE 93

Binary Tree Compositing Associative Alternatives

Updating transparencies is avoided by placing as many internal nodes as possible along the back edge of the tree as shown by the right tree in FIGURE 93. Using right to left pair wise groupings does this. The complexity of binary tree compositing is therefore precomputed emitted intensities (leaves), composited intensities (internal nodes), and updated transparency updates (circled nodes). There are W leaves, and hence W pre-multiplies and for both opacity and transparency calculations. Add $W-1$ additions for transparency calculations when compositing by transparency. The number of internal nodes is always $W-1$. This is the number of intensity compositing calculations. The depth of the tree for any number of leaves W is $\lceil \log W \rceil$. Clustering the non updates to the full side of the tree there will always be $\lceil \log W \rceil$ internal nodes that do not require incremental transparency (or opacity) updates. This includes the root. All other internal nodes do updates. There are $W-1-\lceil \log W \rceil$ nodes performing updates. For transparency the update involves, $t_{ij} = t_i t_j$, 1 multiplication. For opacity the update involves $\alpha_{ij} = \alpha_i + \alpha_{ij}(1-\alpha_i)$, 1 addition and 1 multiplication, when $(1-\alpha_i)$ is saved during intensity calculations. The total cost for transparency is W emitted intensity initializations, $W-1$ transparency initializations, $W-1$ intensity composites, and $W-1-\lceil \log W \rceil$ transparency updates. This reduces to $W + (W-1) + (W-1-\lceil \log W \rceil) = 3W-2-\lceil \log W \rceil$ multiplications and $(W-1) + (W-1) = 2W-2$ additions.

The total cost for opacity is premultiplication + intensity composite + opacity updates, $W + (W-1) + (W-1-\lceil \log W \rceil) = 3W-2-\lceil \log W \rceil$ multiplications and $2(W-1) + (W-1-\lceil \log W \rceil) = 3W-3-\lceil \log W \rceil$ additions.

2.5 Front-To- Back Binary Tree Compositing

The binary tree method may be balanced in ways not intended to reduce the number of intermediate transparency (opacity) calculations. For example, an unbalanced tree gives a trade-off between adaptive ray termination and parallel computation. The updated intermediate transparency (opacities) in this case can be delayed until the terminate condition is evaluated saving operations upon termination. The front-to-back calculations in binary tree fashion are done as shown in FIGURE 94.

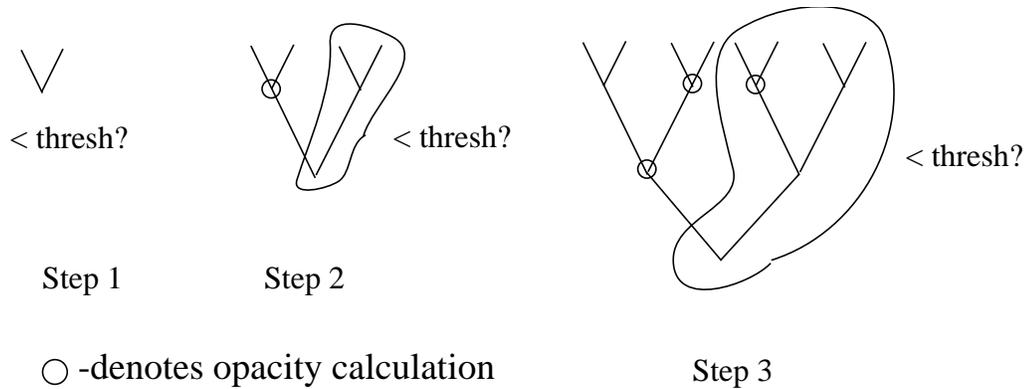


FIGURE 94

Front-To-Back Parallel Compositing

FIGURE 94 shows leaping into the samples by doubling the number of samples each time, and by updating the incremental transparencies (opacities) in each previous tree only after the threshold has been tested. Step 1 does not update the opacity, but step 2 does (See circle) because the termination condition failed. The costs for parallel compositing are still the intensity calculations, which remain unchanged: $2W - 1$ multiplications and $2W - 2$ additions, only the number of transparency (opacity) updates changes. The number of carry-forward opacity-calculations cannot be calculated by a closed form equation.

Given W sample points, a tree is formed by taking pairs starting from the left and building a binary tree. The depth of the tree is, as mentioned earlier, $\lceil \log W \rceil$ and the number of internal nodes is $W - 1$. Define an update node as the internal nodes of the tree excluding all nodes along the right most path to the root. Also exclude the root. For the right balanced tree the number of non update nodes is always $\lceil \log W \rceil$, but for left balanced trees

it is not as simple. Example trees for W equal to 2, 3, 4, and 5 are given below with updated nodes circled.

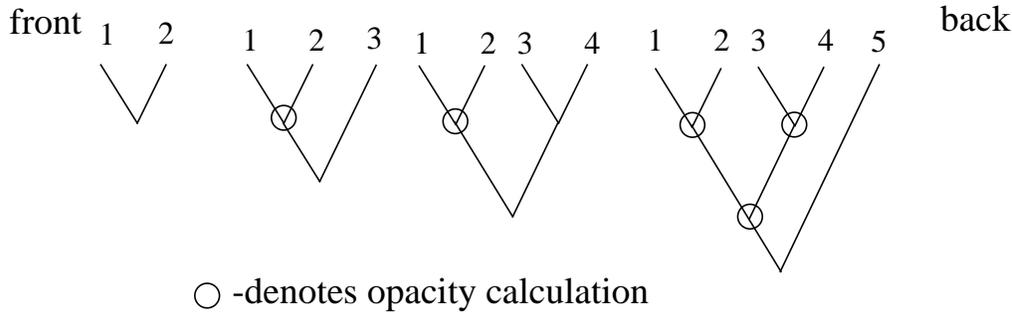


FIGURE 95 Update Node Problem

A closed form equation is not possible, but a dynamic programming approach can calculate the number for any W , and further the update nodes will always be bounded by $(\# \text{ update nodes} \leq 2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil)$ which is just the number of update nodes in the full binary tree with a number of nodes greater than or equal to our own. This is the number of internal nodes $2^{\lceil \log W \rceil} - 1$ minus the number of internal nodes along the right branch $\lceil \log W \rceil$. The number can be found by bit counting. The exact form for any W is the summation

$$\# \text{update nodes} = W - 1 - \sum_{i=0}^{\lceil \log W - 1 \rceil} (W - 1) / 2^i, \quad (\text{EQ 89})$$

using integer division. (EQ 89) adds the bits in the binary $W - 1$ word, which determines directly the number of full binary sub trees in the tree.

I use the upper bound of $2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil$ update nodes. The total number of calculations for opacity updates is $2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil$ multiplications and $2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil$ additions. The combined intensity and opacity calculations are $2W + 2^{\lceil \log W \rceil} - 2 - \lceil \log W \rceil$ multiplications and $2W + 2^{\lceil \log W \rceil} - 3 - \lceil \log W \rceil$ additions. To simplify discussion define the intensity computation minimum bound as $W - 1$ compositing operations, and front-to-back binary-tree compositing incurs an additional $2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil$ opacity updates.

Compositing with transparency, $t = 1 - \alpha$, gives slightly different results. The number of compositing operations remains the same, but the costs of initialization, compositing, and updates differs. The compositing cost changes to 1 addition and 1 multiplication using the transparency equations, (EQ 86), and the cost of updating a transparency

requires only a single multiplication, (EQ 87). So, by using all of the previous results derived for opacity compositing, the number of compositing operations is $W - 1$. The number of transparency updates is the same as the number of opacity updates giving $< 2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil$ multiplications for an upper bound, or (#update nodes) (EQ 89) for exact results. The totals for initialization, compositing, and updates are $(W) + (W - 1) + 2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil = 2W + 2^{\lceil \log W \rceil} - 2 - \lceil \log W \rceil$ multiplications and $(W - 1) + (W - 1) = 2W - 2$ additions.

2.6 Sum of Attenuated Emittances Approach

The line integral can be evaluated in other parallel fashion by calculating the intensity of each sample point as seen by the eye. From the line integral equation

$$I = \int_{\lambda_1}^{\lambda_2} t(l) I_S(l) V(l) dl \quad (\text{EQ 90})$$

it is possible to derive a direct evaluation formula

$$I = I_{E_1} + I_{E_2} t_1 + I_{E_3} t_1 t_2 + I_{E_4} t_1 t_2 t_3, \quad (\text{EQ 91})$$

This has the same complexity as the binary tree method, and in fact is done in binary tree fashion, but different communication is used. It requires communicating the opacities/transparencies to the levels preceding your local level. You can add the W samples attenuated emittances in a binary tree fashion. Which approach is more efficient is determined by the communication.

Create the sum of attenuated emittances by performing the transparency calculations and then combining. For example, FIGURE 96 shows 4 images being sequentially processed.

$$\begin{array}{rcll}
 I_{E_1} = \alpha_1 I_{S_1} & I_{E_2} = \alpha_2 I_{S_2} & I_{E_3} = \alpha_3 I_{S_3} & I_{E_4} = \alpha_4 I_{S_4} & W \text{ mult.} \\
 t_1 = (1 - \alpha_1) & t_2 = (1 - \alpha_2) & t_3 = (1 - \alpha_3) & & W - 1 \text{ add.} \\
 & & t_{12} = t_1 t_2 & & W - 2 \text{ mult.} \\
 & & & & t_{(12)3} = t_{12} t_3 \\
 & & & & \\
 I_{E_{\text{ray}}} = I_{E_1} + t_1 I_{E_2} + t_{12} I_{E_3} + t_{123} I_{E_4} & & & & W - 1 \text{ add.} \\
 & & & & W - 1 \text{ mult.}
 \end{array}$$

FIGURE 96 Sum of Attenuated Emittances Sequential Calculations

The intensity of the ray, $I_{E_{\text{ray}}}$, is the sum of the first image's intensity, unattenuated as there is nothing blocking it, and the sum of all of the following image intensities which are attenuated by specific amounts. Totals are $W + W - 2 + W - 1 = 3W - 3$ multiplications and $W - 1 + W - 1 = 2W - 2$ additions. The above technique is not fully parallel because the creation of the transparencies requires sequential communication.

If instead calculation is done as shown in FIGURE 97, the calculation is fully parallel and limited only by the speed of the parallel add, $\log W$.

$$\begin{array}{rcll}
 I_{E_1} = \alpha_1 I_{S_1} & I_{E_2} = \alpha_2 I_{S_2} & I_{E_3} = \alpha_3 I_{S_3} & I_{E_4} = \alpha_4 I_{S_4} & W \text{ mult.} \\
 t_1 = (1 - \alpha_1) & & t_{123} = (1 - \alpha_1)(1 - \alpha_2)(1 - \alpha_3) & & \text{add.} \\
 & & t_{12} = (1 - \alpha_1)(1 - \alpha_2) & & \\
 I_{E_{\text{ray}}} = I_{E_1} + t_1 I_{E_2} + t_{12} I_{E_3} + t_{123} I_{E_4} & & & & \text{mult.} \\
 & & & & W - 1 \text{ add.} \\
 & & & & W - 1 \text{ mult.}
 \end{array}$$

FIGURE 97 Sum of Attenuated Emittances Parallel Calculation

This has the same complexity as binary-tree compositing. The α_i values must be retrieved from all previous images, which may be slow. Also the product calculations for the transparencies will be not as efficient as binary tree approaches but perhaps the com-

munication network can calculate the products in a scan [THIN89][BLAN90] type operation. In fact, I use this approach in my MasPar implementation.

2.7 Summary and Discussion

For sequential methods, back-to-front is the most efficient but does not allow adaptive ray termination or progressive refinement. To do this a slightly higher cost method is front-to-back, which saves calculation if only part of the ray is processed.

For parallel methods I combine samples in binary fashion. Without a power of 2 number of samples update costs are minimized by associative groupings that place many internal nodes along the back of the tree. Balancing the tree forward allows mixing parallel progressive refinement and parallel evaluation.

An alternative evaluation directly calculates local transparencies and sums up all of the results. This sum of attenuated emittances approach may be more efficient than binary tree compositing depending on the communication overhead. The constant notation complexities are summarized below. FIGURE 98 shows all methods for five images. Notice again that internal nodes are intensity composite step, and circled nodes are opacity/transparency updates, and + nodes denote addition for the sum of attenuated emittances approach.

The number of updates is the biggest variance in the respective methods. It is also more efficient to calculate with transparencies. The update costs are shown in TABLE 34 and TABLE 35. The total costs are shown in tables TABLE 36 and TABLE 37.

TABLE 32

Initialization costs

emitted intensity premultiply	1 multiplication
transparency only transparency calculation	1 addition

TABLE 33

Number of Intensity Compositing Steps

All Methods $W - 1$

TABLE 34

Compute Cost, Update Cost

intensity	opacity	transparency
1M, 2A	1M, 2A	1M

TABLE 35

Number of Composites for Updates to transparency/opacity

method	cost
back-to-front	none
front-to-back	$W - 2$
binary-tree	$W - 1 - \lceil \log W \rceil$
binary-tree, front-to-back	$< 2^{\lceil \log W \rceil} - 1 - \lceil \log W \rceil$ or (EQ 89)
sum of attenuated emittances	$W - 2$

TABLE 36

Multiplications for All Methods

Method	Transparency/Opacity
back-to-front	$2W - 1$
front-to-back	$3W - 3$
binary-tree	$3W - 2 - \lceil \log W \rceil$
binary-tree, front-to-back	$< 2W + 2^{\lceil \log W \rceil} - 2 - \lceil \log W \rceil$
sum-of-atten.	$3W - 3$

TABLE 37

Additions for All Methods

Method	Transparency	Opacity
back-to-front	$2W - 2$	$2W - 2$
front-to-back	$2W - 2$	$4W - 6$
binary-tree	$2W - 2$	$3W - 3 - \lceil \log W \rceil$
binary-tree, front-to-back	$2W - 2$	$< 2W + 2^{\lceil \log W \rceil} - 3 - \lceil \log W \rceil$
sum-of-atten.	$2W - 2$	NA

The tables show clearly that the number of multiplications is the same for both the transparency and opacity calculation approaches. Opacity approaches require more additions for opacity updates, such as in the front-to-back, and binary-tree methods. I recom-

mend using transparency, both because of its simplicity in expression and reduced calculation for both sequential and parallel algorithms.

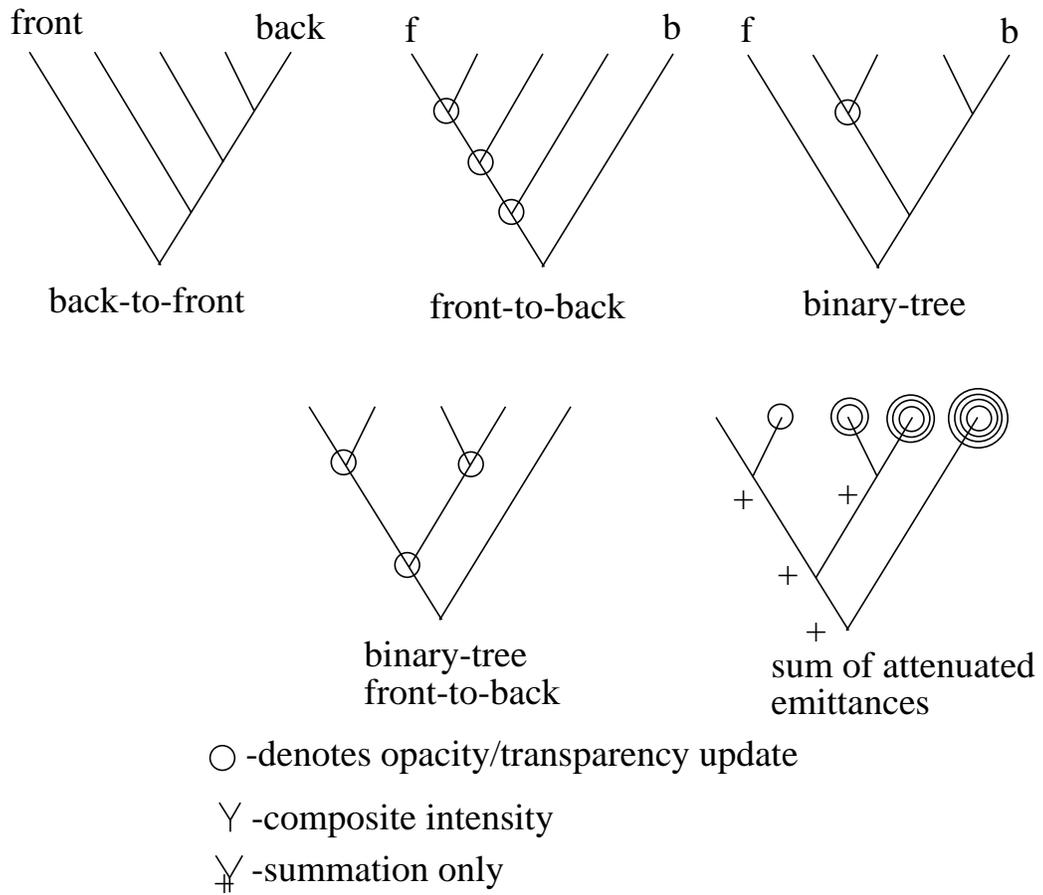


FIGURE 98

Vita

Craig Michael Wittenbrink was born in Denver, Colorado on June 29, 1965. He received his B.S. in electrical engineering and computer science from the University of Colorado in May 1987. After working at The Boeing Company designing computer image generators for flight simulators, he returned to get his M.S. in electrical engineering in December 1990 and Ph.D. in 1993 at the University of Washington. Craig has published in the journal *Machine Vision and Applications*, is a coinventor on a patent application for the Proteus Supercomputer, and has published and presented his work internationally. The work presented herein was done with support from the NASA Graduate Student Researcher's Program. The image warping chapter was published in part in the International Parallel Processing Symposium, 1993 as "2D and 3D Optimal Parallel Image Warping" [WITT93]. Material from the volume rendering chapter, Chapter IV, was partly published in the Parallel Rendering Symposium, Visualization '93 as "Improved Filters and View Flexibility for Data Parallel Volume Rendering," [WITT93b] and in SkiGraph as "Permutation Warping for Volume Rendering," [WITT93c]. With support from his NASA fellowship he also presented results from this dissertation in Washington, D.C. in 1992 and 1993 at the NASA GSRP Annual Symposium.