UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids

A dissertation submitted in partial satisfaction
of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in
COMPUTER AND INFORMATION SCIENCE
by
Judith Ann Challinger
December 1993

The dissertation of Judith Ann Challinger is approved:

Dr. Jane Wilhelms

Dr. Charles McDowell

Dr. Nelson Max

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Scalable Parallel Direct Volume Rendering for Nonrectilinear Computational Grids

Judith Ann Challinger

ABSTRACT

Various approaches to the problem of parallelizing direct volume rendering algorithms are discussed, and a scalable approach to parallel direct volume rendering of nonrectilinear computational grids is presented. The algorithm is general enough to handle non-convex grids and cells, grids with voids, grids constructed from multiple grids (multi-block grids), and embedded geometrical primitives. The algorithm is designed for a highly parallel MIMD architecture which features both local memory and shared memory with non-uniform memory access times. It has been implemented on a BBN TC2000 and benchmarked on several datasets. An analysis of the speedup and efficiency of the algorithm is given and any sources of inefficiency are identified. The trade-off between load balancing requirements and the loss of coherence due to the image-space task decomposition is examined. A variation of the algorithm which provides fast image updates for a changing transfer function is also presented. A distributed approach to controlling the execution of the volume render is used and the graphical user interface designed for this purpose is briefly described.

Keywords: volume rendering, parallel processing, scientific visualization

Preface

Acknowledgements

I would like to acknowledge the help and encouragement of my advisor, Jane Wilhelms, who has always encouraged independence, provided support and guidance when needed, and been an excellent role model. Nelson Max gave me the confidence and incentive to pursue this degree, and helped direct my interest towards computer graphics from the beginning. His committment to excellence has always been an inspiration to me. Charlie McDowell sparked my interest in highly parallel computing through an excellent graduate course, and has been an important resource for me. A special thanks goes to Jim Murphy of CSU Chico, who introduced me to UC Santa Cruz, and convinced me to come here. The efforts of all the faculty and staff of the CIS and CE Boards have made my experiences here pleasantly stimulating, as well as occasionally challenging! Lynne Sheehan and Melissa Queen have both provided support and encouragement beyond the call of duty in times of doubt.

Thanks also goes to the staff of the National Energy Research Supercomputing Center at Lawrence Livermore National Laboratory for use of the BBN TC2000 in this research. Several insightful conversations were had with Scott Whitman and Brent Gorda, and I especially appreciate Carol Hunter for her support and encouragement of my goals.

The most important support came to me from my family: my sweetheart, Peter, and our two beautiful kids, Michael and Jonathan. They all carried their share of the burden while I was off pursuing my own goals. I am very lucky to have them. A large amount of encouragement was also forthcoming from my extended family: Mom and Raymond, Dad, my sisters and brothers (especially my sister Lynn), Peter's Mum, and even my aunts (especially Nancy).

Finally, a very special thanks to my friend, Pam Ganzberger, who gave me the best advice of all: "Judy, just write it up and turn it in!"

Publication History

The early work described in section 4.3.1 is available as a technical report:

> Judy Challinger, "Parallel Volume Rendering on a Shared-Memory Multiprocessor", Technical report UCSC-CRL-91-23, University of California, Santa Cruz, 1991.

The early results presented in section 4.3.2 were published in:

> Judy Challinger, "Parallel Volume Rendering for Curvilinear Volumes", Proceedings of the Scalable High Performance Computing Conference, pp. 14-21, April 1992, IEEE Computer Society Press.

An earlier version of chapter 5 with preliminary results will appear in:

> Judy Challinger, "Scalable Parallel Volume Raycasting for Nonrectilinear Computational Grids", Proceedings of Visualization '93 Parallel Rendering Symposium, October 1993.

1. Introduction

There is a major shift in paradigm underway in the area of supercomputing. The powerful vector processors epitomized by the Cray series of supercomputers will gradually be replaced with massively parallel systems of hundreds or thousands of processors and extremely large, scalable memories. This trend towards a new architectural approach to achieving high performance is being driven from above by the performance requirements of the so-called *grand challenge* problems, and from below by engineering, physics, and economic factors which make the cost of computation less on mass-produced VLSI single chip processors.

With the advent of extremely powerful supercomputers and massively parallel systems, numerical simulations of physical systems are being done in three spatial dimensions, and at increasingly higher levels of resolution and complexity. Tools which provide for the *visual* analysis of the results of such simulations are extremely important.

A *volumetric dataset* is a collection of scalar data in which each datum has an associated location in three-dimensional space. Many numerical models of physical systems are based upon such scalar fields. *Direct volume rendering* is a powerful, but computationally intensive, computer graphics technique for rendering volumetric datasets that has been shown to be useful for the visual analysis of the results of scientific computations.

The extremely large size of the results of the numerical simulations can make it difficult and time consuming to extract useful visualizations of the data. Typically the scientist producing the result is located remotely from the massively parallel machine. A common approach has been to move the data set to a local graphics workstation and render it there. This can be problematic for very large data sets, and especially so if the simulation of interest is unsteady (time-varying). One motivation for this research is to provide a powerful *interactive* tool for the visual analysis of the results of simulations. In this context, it is desirable for the analysis tool (volume renderer) to be made available on the same machine that the simulation is running on. This will facilitate closer coupling of scientific simulations and visualization tools in the future, ultimately leading to the ability to interactively steer scientific computations based on visual feedback. As the trend towards putting large scientific simulation applications on massively parallel machines continues, it is essential that similar research is conducted on the parallelization of visualization techniques.

This thesis addresses the efficient parallel implementation of direct volume rendering algorithms on highly parallel MIMD architectures. Of primary interest are architectures which exhibit non-uniform memory access latency (i.e. those in which the access time for a memory location depends on its location relative to the requesting processor.) This class of architectures covers a wide range of MIMD systems, including strictly message passing architectures which implement some type of virtual shared memory. Several approaches to parallel direct volume rendering are discussed in chapter 4, and a new parallel algorithm for direct volume rendering is presented in chapter 5. The algorithm is general enough to handle all of the commonly used complex computational grids: curvilinear, multi-block, tetrahedral finite element, hexahedral finite element, etc. Each approach has been implemented and benchmarked on several test datasets.

For computational models which can execute a single (or a few) time steps at near interactive speeds, the close coupling of volume rendering for visualization with the executing computational model will provide a powerful exploratory tool for scientists. The design and implementation of the rendering algorithms is done in such a way as to allow such a close coupling. A distributed graphical user interface has been developed in order to demonstrate the remote interactive use of the parallel rendering software.

An overview of volume rendering is given next, followed by a survey of related work in chapter 2. Chapter 3 describes highly parallel architectures and section 3.2 discusses the specific parallel machine used for this work. General issues with regard to the parallelization of direct volume rendering algorithms are covered in chapter 4, along with some early research results and the rationale behind the specific focus of this thesis. The algorithms and data structures for parallel direct volume rendering that have been developed for this research are described in chapter 5. A description of the datasets and images used as test cases for benchmarking purposes is given in chapter 6, followed by benchmarking results and discussion. Conclusions are presented in chapter 7.

1.1 Introduction to Direct Volume Rendering

This section introduces the basic forms that current direct volume rendering algorithms take. There has been a lot of research into volume rendering algorithms; the many variations on these algorithms are covered in the next chapter.

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

How can such an entity be visualized so as to allow internal structure to be seen while preventing loss of information? This question has led researchers to a technique for rendering volumetric datasets called *direct* volume rendering [DCH88, UK88, Sab88, Lev88]. This approach is called "direct" to differentiate it from other methods, such as isosurface extraction or volume slicing, which utilize only a subset of the data for image creation. Using direct volume rendering, the entire volume of data can be rendered using various levels of transparency related to the scalar data values in order to see the interior structure. There are many algorithms for accomplishing this; figure 1.1 shows how lookup tables (commonly called transfer functions) can be used to obtain color and opacity values for a specific scalar data value in the array [UK88]. In general, the sample points desired within the data array will not lie directly on those sample points that are given. An interpolation method will be employed to generate the scalar data value at the desired sample point.

Rendering the volumetric dataset involves sampling the entire dataset in some fashion and mapping each sampled value to a color and opacity. The contribution from each sample must then be composited or accumulated into the image [PD84]. There are currently four basic algorithms which are widely used for sampling the entire volume: raycasting, cell projection, splatting, and shear transformations. The decision on whether to support

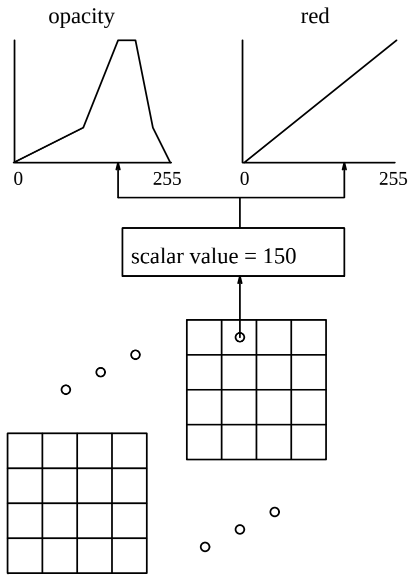

Figure 1.1: Volume rendering using transfer functions.

parallel or perspective viewing projections can have implications for sampling and aliasing. These algorithms can also be extended to render volumetric datasets on grids of increasing complexity.

1.1.1 Volumetric Grids

Volumetric datasets which are arranged on a regular rectilinear lattice or grid are of the simplest form. More general grids are commonly used by researchers to generate volumetric datasets. Nomenclature for volumetric grids is hardly standardized; a brief overview of grid types is given in [SK90] and more detailed information can be found in [Fle88, ZT89]. We will call a single data point that has been sampled or computed a *node*. Two neighboring nodes may be said to define an *edge*, and three or more define a *face*. In the case where a face is defined by more than three nodes, it is possible that the face will be non-planar. A *cell* is the space in $\Re^3$ defined by four or more nodes. A cell may be a tetrahedron, hexahedron, degenerate hexahedron, and so on. A face that is shared by two cells is an *internal face*, otherwise it is an *external face*.

An example of a simple, rectilinear, volumetric dataset is many magnetic resonance images stacked together in memory to create a three-dimensional array of data. In this case, each image sample is a node and has an associated implicit location in $\Re^3$. Many numerical simulations of physical systems also produce data in this format. The grid spacing may be irregular (in a given dimension the nodes will not be evenly spaced, but will vary,) as in some computational fluid dynamics (CFD) applications where portions of the space need to be sampled/computed at a higher resolution than others. Or the grid may be curved to match the simulation geometry, as in the *curvilinear* grids commonly used in CFD. In this case, a regular, rectilinear, computational grid has been *shaped*, i.e. mapped by a smooth function from $\Re^3$ to $\Re^3$ to give a curved shape, resulting in cells with non-planar faces in physical

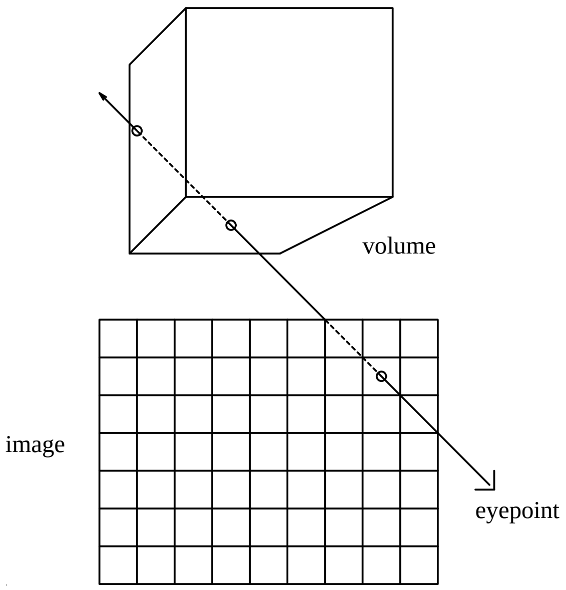

Figure 1.2: Volume rendering using the *raycasting* algorithm.

space [Fle88]. These array-organized grids are also called *structured* grids [SK90]. Very large simulation geometries are sometimes gridded using a *multi-block* approach [plo89]. In this situation several curvilinear grids are combined in one simulation in order to represent the solution space. These grids may overlap spatially, and cell faces belonging to two different grids may intersect. Two grids may occupy the same space, with one utilizing a finer gridding and more accurate result. Or the grids may be used to represent different parts of the simulation space, with only their boundaries overlapping. *Unstructured* computational grids in $\Re^3$ made up of a collection of tetrahedral or hexahedral cells that have been shaped are also common in computational fluid dynamics and finite element analysis applications [ZT89]. Typically the definition of these grids is given as a list of cells, defined by pointers into a list of nodes. There is no regular rectilinear array of nodes in computational space, as is the case with curvilinear datasets. Thus information on shared faces and neighboring cells is not inherent in the data structure. Finally, the volumetric dataset may not be defined on a grid at all, but may simply be given as a list of nodes. This type of dataset is commonly referred to as *scattered data.*

1.1.2 Image-space Rendering Algorithms

Raycasting is an image-space algorithm which involves casting a ray from the viewpoint through each image pixel and testing for intersection with the volume [UK88, Lev89b, Sab88, GO89, Cha90, Gar90, WCA$^+$90, Cha92, Koy92, NL92, MPS92, CM92, Gie92]. If a ray intersects the volume, the contents of the volume along the ray are sampled, transformed into color and opacity, and composited, and the resulting value is taken as the pixel contents. In essence, we are integrating along the length of the ray that is passing through the volume of data. This process is illustrated in figure 1.2.

There are two approaches to sampling along a given ray. The first involves finding the entry point of the ray into the volume and taking equally spaced steps along the ray, sampling and compositing, until the exit point from the volume is reached. Depending on the step size, it is possible to miss very small cells using this technique. The second approach involves finding the entry and exit points for each cell intersected by the ray. The contribution from each cell is then calculated, using either analytic integration or an approximation, and composited.

In this algorithm, the primary loop is through the image pixels. Each ray may be processed completely independently from any other ray, and the rays may be processed in any order. Compositing operations along a ray must proceed in either front-to-back or back-to-front order.

Raycasting of curvilinear or unstructured grids is especially time-consuming if no effort is made to reduce the ray/grid intersection testing requirements. One approach to doing this is to intersect external faces of the grid only, and then walk through the grid cell by cell [Gar90, Koy92]. Another approach is to sort the grid cells or faces in screen-space, and then test for ray intersection only with those cells or faces that are active at a given pixel [Cha92]. These approaches may still be classified as image-space algorithms since the primary loop is through the image pixels, and the algorithm will require access to many cells for one ray.

1.1.3 Object-space Rendering Algorithms

Projection is an object-space algorithm in which each volume cell or node is sampled to give its contribution to every pixel in the image onto which the cell or node projects. There are two approaches to the projection method for volume rendering: *cell projection* [UK88, MHC90, ST90, WV91, Wil92c, SH92, Luc92, VW93] and *splatting* [Wes89, Wes90, Neu92, Elv92, LH91]. Both of these methods begin by creating a visibility sort of the cells or nodes in order to ensure correct ordering of compositing operations. For rectilinear datasets a visibility sort is trivial, however, it is more complex for more general grids.

Cell Projection

For a regular, rectilinear volume, a cell is defined by eight neighboring scalar values in the volumetric dataset which form the corners of a hexahedron. For a given cell we are essentially integrating across the depth of the cell to give a contribution at each pixel the cell covers. Reconstruction of samples on the cell faces or inside the cell is accomplished by interpolating between the scalar values at the nodes of the cell. Scan conversion techniques and principles of spatial coherence may be utilized to speed cell sampling. An illustration of this process is given in figure 1.3. Given interpolation and integration methods, cell projection is equivalent to raycasting where the sample points used are the entry and exit of the ray through the cell.

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

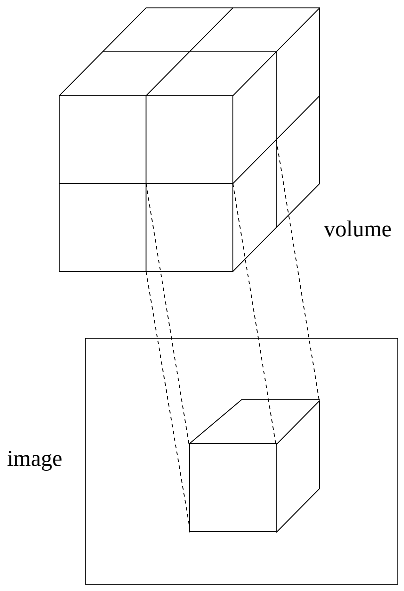

Figure 1.3: Volume rendering using the *cell projection* algorithm.

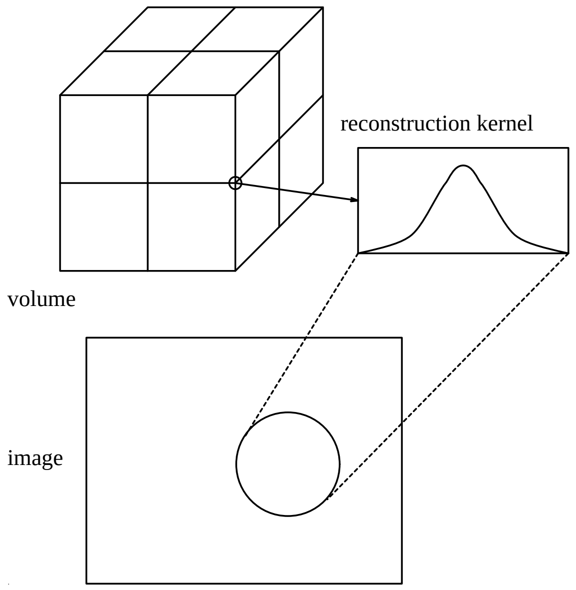

Figure 1.4: Volume rendering using the *splatting* algorithm.

Splatting

In splatting, the volume is processed one node at a time rather than one cell at a time. The scalar value at each node in the volume is processed with a reconstruction kernel that spreads the "energy" present in the node over several pixels in the image. Interpolation between node values is not done, but is approximated by the overlapping kernels. An illustration of this process is given in figure 1.4.

In this algorithm, the primary loop is through the nodes of the volumetric dataset. As with cell projection, each node may be splatted to the image independently, but the compositing operations between splats originating from nodes which vary in distance from the projection plane and which project to the same pixel must be ordered. Design of the reconstruction kernel may be complicated for more complex grids.

1.1.4 Shear Transformations

Direct volume rendering using shear transformations is a hybrid approach, rather than being either an image-space or object-space algorithm [DCH88, SS91, VFR92, KMS+92]. The algorithm implements the viewing transformation as a series of one-dimensional shear transformations and resampling of the dataset. The result gives a volume of data in memory that is view aligned. Compositing may then be accomplished simply by striding through the memory. This approach has only been applied to rectilinear datasets.

1.1.5 Complexity

Both the raycasting and projection algorithms are $O(n^3 + s^2 n)$ where n is the number of samples in each dimension of the volume (i.e. the volume consists of n^3 samples) and s is the number of pixels in each dimension of the image (i.e. an s^2 image is being created). Consider volume rendering an unrotated, n^3 rectilinear dataset to an image that is s by s, and let the volume exactly fill the image. The basic unit of work can be considered to be a single line integration through one cell of the volume, and the compositing of the resulting contribution into one pixel of the image.

If a viewing transformation is to be applied to each grid node before rendering begins, the raycasting algorithm will perform n^3 viewing transformations. A raycasting algorithm which does not require this step is $O(s^2 n)$. The raycasting algorithm will cast s^2 rays into the volume. For more complex grids, if a z-buffer algorithm is employed to determine the first intersection of the volume with a given ray, $(n-1)^2$ front-facing surface faces will be scan-converted into $(s/(n-1))^2$ pixels. An integration and compositing step will be performed $n-1$ times for each ray. In general, this integration and compositing step will require access to the node values of the given cell. If we are zoomed in on part of the volume not all of the cells will be accessed and processed, but those that are will likely be accessed many times. If the volume does not fill the image many of the rays will miss the volume entirely, leading to trivial termination of the ray processing. In addition, many of the cells may project between pixels and will not be intersected by any ray.

The projection approach will require the processing of $(n-1)^3$ cells. In general, the processing required for each cell will involve $(s/(n-1))^2$ integration and compositing steps, one for each pixel covered by a cell. It is possible to utilize coherence principles to approximate the contributions at each pixel, reducing the complexity of the integration step at some loss in accuracy [WV91]. The processing for each cell will require access to the node values of the cell, as well as access to the pixels it covers. If we are zoomed in on part of the volume all of the cells will still be processed, even if only to trivially clip them. If the volume does not fill the image, cells may project to very few pixels. Some may project between pixels and not make any contribution to the image.

For volumes on more complex grids, the analysis is also more complex. For example, volumes on curvilinear grids may contain a few very large cells projecting to portions of the image, and many very small cells projecting to other portions of the same image.

1.2 Motivation for Parallel Direct Volume Rendering

Direct volume rendering is a computationally intensive process. A rectilinear volumetric dataset may consist of 256^3 samples, or 16MB of data if the scalar data values are bytes. Multi-block curvilinear datasets with a million or more nodes are not uncommon. The time required to render an image from such a volume will vary greatly depending on which of the existing algorithms are used and the desired image resolution and quality. Achieving interactive or near-interactive rendering rates has been the topic of much research.

Researchers are actively moving their numerical simulations to highly parallel architectures. The increase in both computational power and memory capacity of these systems has made it possible for scientists to increase both the spatial resolution (grid size) of their numerical simulations, and to extend previously 2D computations to three spatial dimensions. As a result, these highly parallel architectures are allowing researchers to produce large volumetric datasets. In addition, many of the numerical simulations are unsteady (time-varying) meaning that many solutions at different time steps are produced.

In many cases, the most time-consuming and difficult part of a given numerical experiment will be the analysis, understanding, and verification of the contents of the voluminous results of the computation [WCH+87, SS89b]. The most useful visualization tools allow a researcher to interactively explore a dataset. Research into parallel algorithms for volume visualization on all commonly used computational meshes will provide researchers with a powerful means of investigating the results of their three-dimensional numerical simulations.

1.2.1 Scalability Is Important

The *scalability* of a parallel algorithm is generally a statement about how much faster that algorithm will run, given more processors and either input of the same size (for the standard definition of speedup) or input of increasing size (for scaled speedup). Definitions of these measures are presented in section 2.2.2. Another way of looking at it is, "how efficiently is the algorithm using the processors that are available?" Several factors may conspire to reduce the efficiency of a parallel algorithm. These include load imbalance, task generation overhead, synchronization requirements, remote memory latency, and network contention, among others. It is possible to design a parallel algorithm that does very well on a few (2 - 20) processors, and then rapidly loses efficiency with the addition of more processors. In fact, when executing on many processors, such an algorithm may run more slowly than the sequential version. These types of algorithms do not exhibit good scalability, and will not perform well on a highly parallel architecture. Sometimes the measured scalability of an algorithm can be improved by increasing the size of the input as the number of processors is increased. This approach assumes that larger input is desired, and is sometimes reasonable given that highly parallel machines are capable of handling larger problems.

Scalable algorithms are very important if they are to be of practical use in a highly parallel environment. For sufficiently large input, users should be able to interactively request any number of processors, and get performance commensurate with the resources they are using.

1.3 Context for Use

The computing paradigm in which this research has been conducted is one in which a user has large complex volumetric datasets residing on a massively parallel computer which is located remotely from the user's site. The user works on a fairly low-end X workstation with a 24-bit frame buffer, with access to the remote host via a network. This research deals specifically with volumes defined on complex (nonrectilinear) grids in which the scalar field may be time-varying, or in fact the grid itself may be time-varying.

Ideally, the parallel direct volume rendering algorithm is being run repeatedly with several possible variations. These variations could include:

- A change in the viewing transformation. Rotation, especially, is extremely important for spatial understanding.
- A change in the transfer functions. Finding a transfer function that brings out interesting aspects of the data is very difficult. The possibility of interactively exploring transfer functions (as is currently done with two-dimensional datasets) is very attractive.
- A change in the scalar data values of the grid. If the rendering algorithm is coupled to an executing simulation, it may be desirable to watch the scalar field develop over time.
- A change in the grid itself. This would be accompanied by changes in the scalar data values. Simulations which operate using Lagrangian methods are not uncommon.

Algorithms which could provide rapid feedback given one or more of these changes would be extremely useful to scientists.

1.3.1 Distributed Graphical User Interface

The user interface designed and used for this research was motivated by the desire to have a practical, portable system for interactively controlling volume rendering software running on a remote host. X Windows was chosen for its portability and widespread use; in particular, Motif widgets are used wherever possible. A distributed approach was chosen in order to make use of image compression in the transfer of rendered images from the host to the local workstation, and in order to get good performance in highly interactive operations such as transfer function and viewing specification modifications.

X Windows is not particularly suitable for the transmission of images over today's networks, thus a distributed approach allows the images to be intelligently compressed before being transmitted over the socket connection to be displayed. In addition, the operating systems of today's massively parallel hosts are not really suitable for highly interactive tasks such as rubberbanding a line in an X window. It is more efficient, and a better use of the host's resources, to perform such highly interactive tasks locally and send the necessary information to the host as it is needed. All highly interactive tasks are performed locally on the workstation, giving better performance and freeing up the massively parallel host for more intensive computations. The near-interactive, intensive task of volume rendering a scalar field is done on the host; the resulting image is compressed and sent back to the local workstation for display.

Figure 1.5: Distributed graphical user interface.

The concept of using a distributed approach as described here was inspired by a system using a similar approach for grid generation (also a computationally intensive process), presented at the 1992 Computational Aerosciences Conference at NASA Ames Research Center [SM92]. In this work a graphical user interface and plotting software runs on a local workstation, and a multi-block grid generation program which generates 3D grids by numerical solution of the Poisson equations runs on a remote supercomputer.

Figure 1.5 shows the main components of the graphical user interface. In the upper left corner is a main window which provides menus for saving and restoring rendering scripts, for establishing a host connection, and for specifying the rendering method. Buttons are available to pop up windows providing other functionality, and for requesting an image to be rendered. A text box allows the user to specify a new computational grid (these are stored in Plot3D files [plo89] on the host). Buttons along the right side are used to specify which scalar field from the solution file is desired and the histogram of the scalar field is displayed in a window on the left. In addition to the main window, separate windows are provided for transfer function editing, view specification, and image viewing. These are pop-up windows that can be iconified or closed independently. When the host reads in a grid, it sends the external nodes back to the workstation to be displayed in the view window. These are used to provide feedback to the user as the view is manipulated. The bounding box of these nodes is also used to compute an initial zoom and translation which will display the entire volume in the desired image size. The simple image compression scheme used is lossless and generally has been found to compress images by 30% to 60%. Images can be enlarged locally (on the workstation, rather than being rerendered on the host) using bilinear interpolation, and can be saved or restored from a local file.

2. Related Work

There are several areas of active research that are relevant to the work presented here. A large body of literature exists on sequential algorithms for volume rendering, including isosurface extraction techniques, methods for integrating volumetric and geometric primitives, and extensions to basic direct volume rendering algorithms in order to accommodate more complex grids. Several approaches have been presented which trade image quality for rendering speed. The use of parallel processing to speed the production of volume rendered images began in the medical imaging and solid modeling communities. Several specialized parallel architectures have been developed to accomplish specific rendering tasks. Also relevant are parallel architectures and algorithms that have been developed for computer graphics. Very recently, researchers have begun to utilize general-purpose parallel architectures to develop parallel direct volume rendering algorithms. This section briefly surveys these topics and outlines the various approaches that have been taken.

2.1 Sequential Algorithms for Volume Rendering

2.1.1 Isosurface Generation

Isosurface generation is the oldest and most well-known and studied approach to investigating the contents of volumetric datasets. As early as the mid-1970s researchers were developing algorithms to generate three-dimensional geometric representations by connecting contour lines of adjacent two-dimensional contour maps [FKU77, CS78, GD82]. In 1979 an algorithm was presented that would generate a three-dimensional contour map by operating directly on the three-dimensional data [WH79]. More recently, the *marching cubes* algorithm [LC87] generates an isosurface by examining the eight vertices of each voxel and determining any surface intersections. Intersections along voxel edges are approximated using linear interpolation and a triangle mesh for the isosurface is returned. The *dividing cubes* [CLL+88] algorithm works along these same lines, but approximates the isosurface with points by doing recursive subdivision. All of these algorithms generate a geometric representation of a subset of the volumetric dataset. Closely related to the techniques for isosurface generation are those used in the generation of implicit surfaces [Nor82, WMW86, Blo88], and in specialized methods for raytracing to a contour surface [Bli82b, NHK+85].

2.1.2 Direct Volume Rendering

One drawback to the use of isosurfaces as a means to visualize the contents of volumetric datasets is the fact that this approach inherently presents a subset of the data, throwing the rest of the information away. This can be minimized to some extent by the use of multiple semi-transparent isosurfaces, but too many isosurfaces can actually impair understanding. Researchers addressed this problem by developing algorithms for direct volume rendering.

Volume rendering algorithms can be classified as being based on raycasting, cell projection, splatting, or shear transformations as described in chapter 1. Raycasting is an image-space approach in which a ray from the eyepoint is cast through each pixel of the image, intersected with the volume, and sampled along its length [Lev88, Lev89b, UK88, Sab88, GO89]. Cell projection and splatting are both object-space algorithms in which cells or nodes are projected to the screen [UK88, Wes89, Wes90, LH91, WV91, ST90, MHC90]. These methods require that the cells or nodes be sorted into a visibility ordering and projected either front-to-back or back-to-front. Methods using shear transformations first rotate the volume in memory so that it is view aligned. Compositing can then be done by simply striding through the volume [DCH88]. Approaches to direct volume rendering have also differed in the techniques used to map the scalar values to color and opacity. This section gives an overview of several of the different approaches that have been taken.

In work presented by Upson and Keeler [UK88] transfer functions for both color and opacity are defined on the range of the volumetric scalar field. These are then used, along with gradients estimated using finite differences, in shading computations. Both image-space and object-space (raycasting and projection) methods are addressed.

The algorithm developed at Pixar [DCH88] was based on experiences and requirements in the medical imaging field. It requires that each input value in the volume be translated into a set of material percentages. Materials are then given properties such as color, opacity, and density. Surface locations are estimated by looking for areas of high gradient magnitude for density and the direction of the gradient is used in the shading calculations. These representations are combined to give a "shaded color volume" which may then be rotated and resampled using shear transformations, and then projected using a simple compositing scheme.

Sabella modifies Kajiya's [KH84] raytracing algorithm for computational efficiency by eliminating shadowing in order to render a scalar field as a varying density emitter [Sab88]. In one of his rendering schemes, four values are computed for each ray cast through the volume: the maximum value encountered along the ray, the distance to this maximum value, the attenuated intensity, and the center of gravity. The image is generated using the maximum value for the hue, attenuated intensity for the value, and either distance or center of gravity for saturation.

The approach taken by Marc Levoy [Lev88] is to use the dataset values to generate both a color volume and an opacity volume. A raycasting algorithm is then applied to acquire a vector of colors and opacities corresponding to the path of a ray from the view point, through a screen pixel, through the volume. These colors and opacities are composited in back-to-front order, resulting in the final pixel color.

Researchers have also been investigating ways of emphasizing surfaces within volumes of data using a direct rendering approach rather than conversion to geometry. Levoy creates an opacity for each volume element as a function of the gradient field of the dataset. This technique is used to produce both isovalue contour surfaces and region boundary surfaces [Lev88, Lev89b]. Goodsell and Olson have implemented a raycasting volume renderer which produces isosurfaces by comparing each sample along a ray with the previous sample, checking whether an isosurface has been crossed. If so, then the color and opacity of the surface are composited into the contribution collected thus far for the ray [GO89]. Upson and Keeler generate isosurfaces by using a step function in their opacity transfer function [UK88]. Drebin, Carpenter, and Hanrahan do surface extraction by computing a "surface normal volume" and a "surface strength volume" based on the percentages and densities of each material present in the volume [DCH88]. These derived volumes are then used in the shading calculations.

Related efforts in the modeling of natural phenomena and the synthesis of images containing such models are also applicable. In particular, the attempts at rendering such things as clouds, fog, dust, and particle systems can be considered specific applications of volumetric rendering. Blinn utilized a volume of density values to model clouds and presented a technique for rendering them [Bli82a]. Nelson Max extended his work in various ways [Max86] and Kajiya addressed methods for raytracing such models [KH84]. A primary consideration in these efforts has been the means of solving the scattering equation for accurate depiction of the volume based on a given lighting model.

2.1.3 Integration of Rendering Techniques

Recently researchers have begun to address the integration of the new technique of direct volume rendering with more traditional rendering algorithms. An important question is how to handle the direct volume rendering of a dataset that has geometrically defined polygons embedded in it. For example, it may be desirable to define an isosurface using geometrical primitives and render it embedded in the volumetric dataset. Many simulations are done with respect to some fixed geometry (such as fluid flow about an aircraft in CFD); it enhances image understanding to be able to render the geometrically defined object embedded in the flow field it generates.

Marc Levoy discusses two approaches in his Ph.D. dissertation [Lev89b]. The first is a hybrid raytracing approach in which rays are cast simultaneously into the volumetric and geometric objects. Samples are taken from the volumetric object at equal intervals along the ray. Each polygon making up the geometric object is tested for intersection with the ray. All samples are sorted by depth and composited. The second approach involves shading, filtering, and scan converting each polygon of the geometric object at the resolution of the volumetric object to produce a second volume containing the geometry. The two volumes are then combined using volume matting. Care must be taken with the order in which the polygons are scan converted. The resulting composite volume is then rendered using the direct rendering algorithm. The raycasting approach was found to produce sharper polygon edges than the scan-conversion method, but at a higher cost. The algorithms were tested using only a few large polygons embedded in a volumetric dataset; additional techniques would be required to reduce the ray-polygon intersection testing cost if many polygons were to be rendered.

Johnson and Mosher address this problem by adding new volumetric primitives to a traditional rendering system [JM89]. Volume point, vector, ray, and polygon primitives are incorporated into a typical 3D graphics environment. Each of these new primitives has an associated mapping function which maps the volumetric data to a geometric primitive. It is not clear exactly how their approach handles the integration of volumetric and geometric primitives which occupy the same space. Their technique does provide the very useful features of arbitrary cutting planes for volumes and texture mapping using volumetric data.

We have investigated how object-oriented design and implementation techniques can be used to accomplish an integrated rendering approach [Cha90]. The raycasting algorithm presented addresses the rendering of geometrically defined primitives embedded in a volumetric dataset. Algorithms for rendering both rectilinear and nonrectilinear volumetric datasets are given.

Giertsen [GT92] has presented an algorithm for rendering embedded geometrical primitives in unstructured volumetric datasets. The algorithm is an extension of an earlier approach [Gie92] to volume rendering which uses a scan-plane buffer to store contributions from each volume cell to the pixels of a given scanline. Geometrical primitives are handled similarly with the use of a multi-layer z-buffer. All rendering proceeds in scanline order. At each scanline contributions are made to the scan-plane buffer from all intersected volume cells and to the multi-layer z-buffer from all intersection polygons. When all active elements have been processed, the contributions from the two buffers are combined to form the final pixel values along the scanline.

2.1.4 Sequential Methods for Fast Volume Rendering

The primary drawback of direct volume rendering is the amount of computation needed to produce a result. To achieve its ultimate usefulness as an exploratory tool, volume rendering must run at interactive speeds. There have been numerous approaches to speeding up the volume rendering process using sequential algorithms [Wes89, Wes90, LH91, Lev90, WV91, DH92, VW93, Mal93, TL93]. Many of these algorithms trade image quality for speed, typically under user control.

Hibbard and Santek make the case that interactivity is the most important feature in a system for the analysis of volumetric datasets [HS89]. They have presented work towards a fast approximating algorithm for rendering volumetric scalar data on a rectilinear grid as a transparent fog. They select the grid planes most perpendicular to the view direction, and render each, in visibility order, as a set of transparent rectangular polygons.

As discussed in chapter 1, the splatting algorithm estimates the contribution to the image from a single grid node without requiring access to neighboring grid points or interpolation within cells. Compositing of overlapping splat regions is a fast approximation for the actual contribution at points in between grid nodes. Lee Westover has published work in which the goal of interactive rendering is pursued through the use of a splatting algorithm, along with successive refinement of images and table-driven mappings for shading and filtering [Wes89, Wes90]. This approach has been extended by Laur & Hanrahan [LH91] to use a pyramidal representation and hierarchical enumeration of the data with progressive refinement to achieve interactivity, but with some degradation in image quality.

Cell projection approaches allow more accuracy in the determination of contributions to the image. In this case whole cells are projected rather than regions approximated from single nodes as in splatting. Spatial coherence can be utilized to create efficient projection algorithms. Max, Hanrahan, & Crawfis [MHC90] give an algorithm for the projection of any collection of sortable convex polyhedra in which analytic integration is used. Their algorithm is applicable to irregular and scattered data sets.

In addition to taking advantage of spatial coherence to create efficient projection algorithms, some researchers have combined these techniques with the production of hardware renderable geometric primitives to further speed the rendering process by taking advantage of the fast rendering hardware available in computer graphics workstations. Shirley and Tuchman [ST90] present an algorithm for efficiently projecting tetrahedral cells in which each cell is represented with hardware renderable semi-transparent triangles. Laur & Hanrahan [LH91] combine a pyramidal representation with hierarchical enumeration of the data and the ability to rapidly generate hardware-renderable primitives to represent the projection of different-sized cells. The hardware-renderable primitives are scaled depending on the level of resolution requested by the user. Progressive refinement is utilized to obtain interactive rates. Williams [Wil92a, Wil92b, Wil92c] explores several approximations to the algorithm of of Shirley and Tuchman [ST90] for projecting tetrahedral cells which trade image quality for speed and compares the results. Wilhelms and Van Gelder [WV91] describe a projection algorithm for rectilinear volumes in which a template for projection is formed and used to speed the scan conversion of cells. They also compare different approaches to the approximation of the integrals required with respect to speed and image accuracy and quality. Van Gelder and Wilhelms [VW93] have also presented techniques which extended this work to handle curvilinear computational grids.

Raycasting approaches allow more accurate images to be generated, but tend to be much slower than the other algorithms due to their computational requirements. Various techniques can reduce these computational requirements, some of which will introduce artifacts and degradation in image quality. Levoy proposes techniques for increasing the computational efficiency in the raycasting approach to volume rendering. Two of these techniques reduce the cost of tracing each ray: using a hierarchical enumeration of the volumetric dataset, and adaptively terminating the ray processing based on the accumulated opacity of the pixel [Lev90]. Another technique reduces the number of rays cast by using an adaptive approach to determining where the greatest number of rays are required [Lev89b]. The image is subdivided into square sample regions and rays are cast at the four corners of each region. Color differences are used to determine whether each region needs to be recursively subdivided and additional rays cast. Progressive refinement and image interpolation are used to generate the complete image. Levoy and Whitaker [LW90] present a raycasting algorithm in which both the number of rays and the number of samples per ray are reduced in proportion to the distance of the ray from the focus of the user's gaze. Volumes are stored in volume pyramids in which the lowest level of the pyramid contains the original dataset and higher levels contain lower resolution versions of the dataset. The number of samples taken per ray is reduced by choosing a volume higher up the pyramid for sampling. Danskin and Hanrahan [DH92] present and compare several approaches to speeding the raycasting process. Several raycasting algorithms are presented using volume pyramids and allowing the user to trade image quality for speed. Sakas and Gerth give a method for creating volume pyramids based on perspective views which may then be traversed to avoid aliasing without oversampling [SG91]. Their approach to pyramidal volume traversal makes the time to render an image dependent on the image resolution and independent of volume resolution.

Very recently, researchers have presented methods for doing direct volume rendering in the frequency domain, rather than in the spatial domain [Mal93, TL93]. This approach is attractive because it reduces the complexity of the rendering process from $O(n^3)$ to $O(n^2 \log n)$ by making use of the equivalence of the integral of a 1D signal and its spectrum. The Fourier projection slice theorem applies this in higher dimensions and is the basis of Fourier volume rendering [Mal93]. The primary drawback of Fourier volume rendering is that occlusion is not supported, thus the resulting images are similar to x-rays. Totsuka and Levoy have presented techniques for frequency domain depth cueing and directional shading in an effort to alleviate this problem [TL93].

2.1.5 Volume Rendering More Complex Grids

Most of the approaches discussed so far addressed volumetric rendering of data on a rectilinear grid. Researchers have recently begun to address volume rendering algorithms for more complex computational grids. Williams and Max [WM92] give a rigorous analysis of optical models for volume rendering and present a continuous model of volume density which is particularly suitable for rendering scalar fields on irregular grids.

Many of the projection approaches for rectilinear grids can be extended to handle more general grids. The visibility ordering requirements remain the same. Max, Hanrahan, & Crawfis [MHC90] give an algorithm for the projection of any collection of sortable convex polyhedra. Shirley and Tuchman [ST90] present an algorithm for efficiently projecting tetrahedral cells in which each cell is represented with hardware renderable semi-transparent triangles. Williams [Wil92a, Wil92b, Wil92c] gives algorithms for visibility ordering and rendering of nonrectilinear volumes. The algorithm of Shirley and Tuchman [ST90] for projecting tetrahedral cells is modified using several different approximations which trade image quality for speed and the results are compared. Lucas [Luc92] renders irregular grids by visibility ordering the faces of each cell and then scan converting them in order, using a z-buffer to record information which allows the depth through each cell to be used in the compositing operation. Van Gelder and Wilhelms [VW93] extend their work on fast projection algorithms [WV91] using hardware renderable primitives to handle curvilinear grids. Several methods which trade image quality for speed are presented, along with an algorithm for visibility ordering and a technique for improving the results of using hardware compositing.

Raycasting is also fairly easily extended to handle more complex grids. The most important aspect is how to deal with the computational complexity of the ray/cell intersection testing requirements. Approaches taken include interpolating the grid to a rectilinear one, finding the first intersection and then stepping through the cells, and techniques related to scanline algorithms. Wilhelms, et al. [WCA$^+$90] investigate the tradeoffs between resampling a curvilinear grid to a rectilinear one before volume rendering, and direct volume rendering on the curvilinear grid using a raycasting algorithm. The direct volume rendering approach utilizes a y-bucket sort of the cells to reduce the number of ray/cell intersection tests required. Garrity [Gar90] presents an algorithm for volume rendering nonrectilinear datasets which requires that the data be cell-organized in memory so that shared faces are known. A ray intersection with an exterior face is found, the face through which the ray exits that cell is determined, and the process continues stepping through the volume cell to cell. Koyamada [Koy92] presents a raycasting approach for rendering of nonrectilinear volumes which also uses a fast cell traversal. His approach is based on tetrahedra and also involves finding ray intersections with front-facing exterior faces and stepping through the volume from cell to cell. In general, the approach of stepping cell to cell may require expensive testing to resolve the ambiguity that results from a ray passing through an edge or node of a cell [RW92]. An early version of the raycasting algorithm presented in this thesis performed bucket sorts in x and y on the cells of a curvilinear grid to reduce intersection testing requirements [Cha92].

Scan-line methods, which seem to fall somewhere between raycasting and projection approaches, have recently been applied to the rendering of more complex volumetric datasets.

Giertsen [Gie92] gives an algorithm for rendering sparse unstructured grids which utilizes a scan-plane buffer to store contributions to each pixel from different cells in the grid. Cell intersections with the scan-plane are incrementally computed, discretized, and stored in the buffer. Intersections are stored in the buffer at a location corresponding to their x and z coordinates. One drawback to this approach is the aliasing in the z direction caused by the discretization required by the use of the scan-plane buffer. When all contributions to the scanline have been stored in the buffer it is processed to produce the pixel values for that scanline.

2.2 Parallel Volume Rendering

An alternative to pursuing sequential algorithms that increase the speed of the direct volume rendering process, typically by accepting a decrease in the image quality and accuracy, is to investigate parallel algorithms for direct volume rendering. This approach is especially important in view of the fact that many of the numerical simulations that produce the volumetric datasets of interest are themselves being moved to parallel architectures. Parallel algorithms for volume visualization become even more important in the case of highly parallel architectures on which very large volumetric datasets may be produced. In these cases it may be infeasible, or at least very inconvenient, to move these datasets to a workstation for visualization postprocessing.

2.2.1 Nomenclature for Parallel Architectures

Parallel architectures can be categorized along several dimensions. One common distinction refers to the autonomy of the processors. A system in which the processors execute the same instruction stream in lockstep, but use different data, is referred to as single-instruction, multiple-data (SIMD). Alternatively, if each processor executes its own instruction stream the system is called multiple-instruction, multiple-data (MIMD). A system may also be classified by its memory architecture. The memory may be completely shared by all processors, or it may be completely distributed and private, or some combination of these. Systems which utilize a globally-shared memory are limited by the memory bandwidth to a small number of processors. Processors which have no shared memory must communicate by message passing over the interconnection network. Software can blur these hardware distinctions: message passing may be implemented on a shared-memory architecture, and local caching may be used to support virtual shared memory on a distributed-memory architecture. Another categorization involves the type of interconnection network used. The processors and memory in a system may be connected by a bus, ring, mesh, hypercube, or multistage switching network. An excellent description of many of these parallel architectures, along with a discussion of language and operating system issues, is given by Almasi & Gottlieb [AG89]. Several of the newest highly parallel MIMD architectures are described in [Hor93].

2.2.2 Measurement Techniques

In order to assess the efficiency and scalability of various parallel implementations, measurements of performance characteristics are made. Common measures used to analyze

n	Number of processors.
T_1	Serial execution time.
T_n	Execution time on n processors.
$S_n = T_1/T_n$	Speedup on n processors.
$E_n = S_n/n$	Efficiency on n processors.
$TOTAL_n = T_n \times n$	Total computational resources consumed.
r_i	Time spent on rendering tasks on processor i.
$R_n = \sum_{i=0}^{n-1} r_i$	Total time spent on rendering tasks.
b_i	Time spent waiting on processor i for other processors to finish.
$B_n = (\sum_{i=0}^{n-1} b_i)/TOTAL_n * 100$	Percentage of total computational resources consumed due to load imbalance.

Table 2.1: Measures of parallel algorithm performance.

the performance of parallel implementations include the execution time, speedup, and efficiency. Let the execution time of a serial version of a given algorithm be T_1, and the execution of a parallel version of the same algorithm running on n processors be T_n. Speedup is defined as

$$S_n = \frac{T_1}{T_n},$$

and efficiency is

$$E_n = \frac{S_n}{n}.$$

These measures are important in that they predict the effect on performance when additional processing power is brought to bear on an algorithm. Ideally, speedup will be close to n and linear, implying that adding processors will have a significant effect on execution time. In this case the algorithm is said to be scalable. Scaled speedup is an alternative to the traditional definition of speedup in which the problem size is increased as processors are added [Gus88]. It has been shown that it is possible to have speedup greater than n, although this is unusual in general [HM90]. The trend is typically shown by running a given algorithm several times for varying n and graphing the speedup as a function of n. The total execution time over all processors ($T_n \times n$) can be broken down into components which indicate the sources of overhead that contribute to declining efficiency (such as load imbalance) [Whi92]. Measures used in this thesis to analyze and present results of parallel algorithm performance are summarized in table 2.1.

2.2.3 Specialized Architectures for Volume Rendering

Early efforts to achieve interactive rates for rendering of voxel-based objects come primarily from the medical imaging and solid modeling communities. Specialized architectures have been developed to speed rendering in those applications. A survey of some of these systems has been done by Kaufman [KBCY90]. Many of these systems were initially designed for rendering solid models and were later applied to medical imaging. In these cases the system design was intended to handle voxels which are opaque if they are visible, unlike the semi-transparent volumetric models and rendering techniques described in the previous chapter. These architectures either do not handle semi-transparent voxels, or they do so at a performance penalty.

The specialized architectures that have been proposed or developed for solid modeling applications include: the 3DP^4 [OUT85], Insight [Mea85], the PARCUM II [Jac88], and the RayCasting Engine [EKM$^+$91]. Architectures designed for medical imaging applications include: the Voxel Processor [GRB$^+$85], and the Cube [KB88]. All of the approaches described in this section utilize MIMD architectures having either shared or distributed memory. Two use a pipelined approach [OUT85, EKM$^+$91] with different processors assigned to different tasks in the rendering pipeline. In these cases the connections between memory and processors are specific to the task being executed. Memory may also be specialized to speed the rendering process [Jac88, KB88] by allowing simultaneous reads and writes.

For the 3DP^4 [OUT85], the goal is a fast solid-model rendering system for interactive use. Solid models are represented by PEARYs, picture element arrays; these are simply rectilinear volumes of voxels which are either opaque or transparent. The rendering algorithm is called the *linear interpolating projection method.* The volume is considered as a $yd \times zd$ set of lines. The projections of the starting and ending points of each line are found by interpolating the projections of the 8 vertices which define the entire PEARY. The projection points of the voxels along each line are found by linear interpolation of the starting and ending points. A z-buffer is used for hidden surface removal. The gradient is estimated from the z-buffer and used for shading (*depth-only shading*). A parallel architecture is proposed to support these algorithms. It is a MIMD architecture in which the object memory is distributed, and the image memory is shared between two processors. The architecture is a four stage pipeline. There are 256 projection processors with one PEARY memory of 64^3 voxels each. Each of these processors also has a frame buffer and z-buffer. The merging processor is the next stage, it merges the 256 individual frame buffers into one image. The two stage shading processor is next, it first computes the gradient from the z-buffer and then shades the final image. A software implementation of the algorithm has been done, and theoretical timing analysis indicates a throughput of 10 512x512 images per second.

Meagher [Mea85] gives an overview of some of the features of Insight, a commercially available medical analysis and planning workstation. Two case studies are presented. Detailed information about the architecture is not given, but the algorithm is based on rendering an octree representation of a volumetric object to an image that is represented as a quadtree [Mea82]. The octree encoding of the volume provides a straightforward method for back-to-front or front-to-back traversal of the volume for hidden surface removal. Efficient techniques for determining the intersection of an octree node (representing part of the volume) with a quadtree node (representing part of the image) are given. This approach has been extended by Doctor and Torborg [DT81] to allow semitransparency of the volumetric object.

Jackèl [Jac88] proposes the PARCUM II, an architecture that seems to be shared-memory MIMD, however the emulation system that has been built utilizes only one processor. It comes from a solid modeling perspective and uses only opaque voxels. A special memory architecture allows simultaneous read of a 64^3 bit “macro volume element” or MVE. Hidden voxel removal is done by simultaneously processing the MVE in the x, y, and z directions, and by using a z-buffer. The system supports 512^3 bits or 256^3 7-bit values. Shading is done using gradients calculated from the z-buffer. The emulation system produces images in 40 to 110 seconds.

Ellis, et al. [EKM+91] give an overview of the RayCasting Engine, a special purpose parallel architecture for processing CSG models for rendering and other analysis. It takes a CSG model and clips it against a 2D array of rays, producing a ray-rep model where segments of each ray which intersects a solid are kept. A fairly large prototype has been built, but no empirical results were given. The system is a distributed-memory, MIMD architecture. The processors are connected in a 2D lattice. The bottom row of the array contains primitive classifiers (PC). Each of these is programmed with a single half-space and sequentially clips that half-space against all the rays. The remainder of the array contains classification combiners (CC) which accept two sets of line segments and combines them using boolean operations. The communication network is programmed to match the tree structure of the CSG model. CCs pass data to the top and left neighbor only. Large models can be processed by making more than one pass.

Goldwasser, et al. [GRB+85] describe the Voxel Processor, a MIMD, bus-based, distributed-memory architecture with the goal of real-time (30 frames per second) generation and display of volumetric medical image data. A scaled-down prototype has been built, but no empirical results are given. The full-sized design incorporates 64 processors and is designed to handle volumes of 256^3 16-bit values. The prototype has one processor and handles a 64^3 volume of 4-bit values. It isn't clear that the architecture is scalable and that the full-sized version will not suffer from contention problems. The working prototype can generate 16 frames per second. The generated image size is 512x512. The volume of data is distributed, a sub-volume to each processor. There is no resampling or integration, the algorithm is voxel-oriented and voxels are opaque. Table lookup is used to perform segmentation and windowing, and for graphics overlays. A painter's algorithm (back-to-front) is used at each processor to render its sub-cube to one of two (double-buffered) private frame buffers. Intensity and depth information is maintained at each pixel. When all processors have completed rendering, the individual images are merged (using depth information) into the output frame buffer.

Kaufman and Bakalash [KB88] describe the Cube, a shared-memory multiprocessor architecture with two unique features: a skewed memory organization that allows simultaneous access of a full beam (row) of voxels, and a multiple-write bus that allows the nearest opaque voxel to be selected in O(log n) time. The projection approach also supports rendering of scan converted geometry and translucency (there is a performance penalty for the latter). The target is a 512^3, 8-bit system. A 16^3, 8-bit prototype has been built. It is estimated that the full system will produce 16 frames per second. Memory organization allows a full beam of voxels along any of the orthogonal viewing directions to be accessed simultaneously. Arbitrary views required the data to be "rotated" in the memory. A dedicated frame buffer processor does this along with other voxblt functions. The viewing processor performs the real-time image projection. It consists of a processor for each voxel along a beam. These processors each access one of the voxels in a beam, clip, map, and shade it using table lookups, and write it to the multiple-write bus. For opaque voxels, this effectively performs the projection of an entire beam simultaneously. For translucency, this operation must be repeated and the resulting values combined.

2.2.4 Parallel Direct Volume Rendering

Comparisons between different parallel implementations for direct volume rendering have proven to be difficult because of the wide variety of factors that influence performance. The choice of algorithm, architecture, dataset, and image size all contribute to the efficiency of a particular implementation. Specification of the dataset to be rendered may have implications in the most effective choice of algorithm. This is true even when considering sequential algorithms. For example, if the dataset consists of a few large cells the best algorithm will effectively utilize spatial coherence. If, however, the dataset consists of many small cells in which most cells project to very few pixels, then the overhead involved in utilizing spatial coherence may reduce efficiency rather than enhancing it. Specification of an architecture will also have ramifications in the selection of an algorithm that will perform well. This is especially evident when one compares SIMD and MIMD implementations. The type of memory available (distributed, shared, hybrid) is an important factor as well.

Differences in methods of measuring performance reported by various researchers make direct comparisons of implementations difficult. However, certain trends can be clearly seen. MIMD architectures are more flexible in terms of the variety of algorithms they can support, and have received more attention than SIMD architectures. Efficiency on highly parallel systems is much more difficult to achieve than on architectures that utilize just a few processors. Raycasting has received the most attention from researchers investigating parallel algorithms for direct volume rendering. The basic raycasting algorithm is easily parallelized without introducing inter-processor synchronization requirements because each ray can be processed independently from any other ray as long as the values at the nodes of each cell intersected by the ray are rapidly accessible to the processor. Enhancements which speed sequential algorithms may also limit the scalability of a highly parallel implementation if they impact the synchronization requirements.

SIMD

The major drawback to SIMD architectures is that they are appropriate only for a special category of algorithms. SIMD architectures perform best on data parallel problems that can make use of nearest neighbor communications. For direct volume rendering, this means that SIMD architectures are most suitable for rendering rectilinear datasets that are view-aligned in memory. All of the SIMD approaches to direct volume rendering have addressed rectilinear grids only. These approaches are discussed in more detail below and the results are summarized in table 2.2.

The first parallel algorithms for direct volume rendering on SIMD architectures utilized shear transformations. Schröder [SS91] has presented an algorithm for parallel volume rendering on the Connection Machine (CM-2) in which an important aspect is the communication requirements for rotating the volume in memory to match the requested view. Using 64K processors this algorithm renders a 128^3 rectilinear dataset to a 256^2 image in 821ms. Vézina et al. [VFR92] give a similar algorithm using shear transformations and describes its implementation on the MasPar MP-1. He reports performance statistics for rendering a 128^3 rectilinear dataset to a 128^2 image at 456ms using 16K processors.

Machine	Data Size	Image Size	Time in Milliseconds	Number of Processors
CM-2 [SS91]	128^3	256^2	821	64K
MasPar MP-1 [VFR92]	128^3	128^2	456	16K
CM-2 [SS92]	128^3	512^2	923	16K
Princeton Engine [KMS$^+$92]	256^3	512^2	500	256

Table 2.2: Results reported on SIMD architectures for rectilinear datasets.

More recently Schröder has presented another algorithm which does not rely on shear transforms, but rather steps the rays through the volume [SS92]. In this approach, rays are cast at an angle through the volume and the projection information is passed from processor to processor. All rays are processed in parallel and move in lockstep along a major axis. This algorithm has been implemented on the CM-2 and the Princeton Engine. On the CM-2 a 128^3 dataset can be rendered to a 512^2 image in 923ms using 16K processors. Kaba et al. [KMS$^+$92] discuss techniques for volume rendering on the Princeton Engine Video Supercomputer. In their approach each yz slice is stored on a different processor. Rotation about the x axis can be simply performed using shear transformations on each processor (or direct application of the rotation transformation). For multiple axis rotation the line drawing projection algorithm given by Schröder [SS92] is used. They report rendering times for a 512^2 image and a 256^3 dataset of 125ms to 250ms for x axis rotation using 1024 processors and 500ms for multiple axis rotation using 256 processors.

MIMD

More work has been done on parallel direct volume rendering for MIMD architectures. A wide variety of architectures have been addressed. These can be categorized as to whether they are highly parallel and scalable, or limited to a small number of processors. Implementations for highly parallel systems have been described for the Pixel-Planes 5, Stanford DASH multiprocessor, nCUBE 2, Fujitsu AP1000, and the BBN TC2000. Work on smaller systems has been reported for multiprocessor Silicon Graphics systems. A variety of algorithms have been investigated, with the raycasting approach receiving the most attention. Two researchers have reported on parallel splatting algorithms and a few have addressed implementations of parallel projection algorithms on smaller multiprocessor computer graphics workstations. We have conducted a comparison of parallel image-space versus object-space rendering algorithms and the problems inherent in the two approaches [Cha91]. The results indicate that image-space decompositions may be easier to parallelize with high efficiency. Most of the research conducted so far has been directed towards direct volume rendering of rectilinear datasets. These approaches are examined in more detail below and the results are summarized in table 2.3.

Levoy has proposed a parallelization of the raycasting approach on the Pixel-Planes 5, a parallel architecture developed for computer graphics [FPE$^+$89, Lev89a]. Pixel-Planes 5 is a MIMD architecture utilizing a ring network. There are two types of processors: 32 MIMD graphics processors (GP), and 512x512 pixel processors (PP) grouped into 16 independently programmable renderers. The 512x512 frame buffer is connected to the ring. Each of the GPs and PPs has some local memory; there is no shared memory. Shading and classification volumes are to be created (and stored) on the PPs. Ray tracing is done by the GPs. Each GP is assigned a group of rays to process, it must request the required voxel information from the PPs. Local caching is suggested where possible to speed rendering time. Hierarchical spatial enumeration, adaptive sampling, and successive refinement are also suggested to speed rendering.

Nieh and Levoy [NL92] describe the implementation of Marc Levoy's volume rendering algorithms including optimizations of hierarchical opacity enumeration, early ray termination, and adaptive image sampling, on the Stanford DASH Multiprocessor. The DASH consists of clusters of processors connected by a scalable interconnection network. Processors can be added as required to the hierarchy. The system provides shared memory and several levels of caching. The reported work used a 48 processor system. The basic algorithm is a raycasting algorithm for rectilinear volumes. An approach similar to a task adaptive technique described by Whitman [Whi92] is used for task decomposition. The data is distributed among the processors in an interleaved fashion. Allocating portions of the dataset to the processors in round-robin order prevents the formation of hot spots that can occur when several processors attempt to read data that is located on one processor. Rendering times do not include I/O (image or data) or preprocessing steps such as octree creation. A typical example rendered a 256x256x226 dataset to a 416^2 image using 48 processors in 700ms with a speedup of 40 and an efficiency of 83% for nonadaptive, and 340ms with a speedup of 33 and an efficiency of 69% for adaptive image sampling.

Montani et al. [MPS92] present a parallel raytracing algorithm for rectilinear volumes on the nCUBE 2, a distributed-memory, message-passing, hypercube architecture. The approach groups the processing nodes into clusters. The dataset is replicated on each cluster. Scanlines of the image are assigned to each cluster in an interleaved fashion for processing. Within each cluster, slabs of the dataset are distributed among the processors. Each processor accumulates contributions along a ray as it passes through the part of the volume stored there. When a ray passes out of a local slab, it is passed as a message to the neighboring processor containing the slab the ray is entering. When a ray has been completely traced, the pixel value is computed and sent to the host workstation. Special care has to be taken with termination detection and deadlock prevention. An adaptive load-balancing scheme is presented in which one cluster traces a subset of pixels regularly distributed throughout the image. The resulting timing information is used to redistribute the slabs non-uniformly. Scalability of their approach depends highly on replication of the dataset. The best results on 128 processors require the dataset to be replicated 64 times. Results reported were obtained for rendering a 97x97x116 dataset to a 350x250 image. The best time was 4.75 seconds on 128 nodes with the adaptive data distribution scheme (not including the time to determine the best data distribution). This gives a speedup of 102 over the uniprocessor time of 485.24 seconds and an efficiency of 80%. However, this was obtained using a cluster size of 2 processors - meaning that the dataset was replicated 64 times. On the other hand, using a cluster size of 16 processors (data replicated 8 times), rendering took 7.94 seconds for a speedup of 61 and an efficiency of 48%.

Corrie and Mackerras [CM92] have implemented a parallel ray-casting algorithm for rectilinear volumes on a 128 processor Fujitsu AP1000. This machine is an experimental distributed-memory architecture with three independent communication networks. These include a broadcast network, a 2D torus network, and a synchronization network. The dataset is subdivided and distributed among the processors for storage. A distributed virtual memory system was designed and implemented in order to provide access to the distributed dataset to any processor. The task decomposition is done in image space, with each processor being assigned some portion of the image to render. Several task decomposition schemes are analyzed, the most efficient being dynamic allocation of square image blocks with an adaptive technique for redistributing a task in order to minimize load imbalance. A careful analysis of the virtual memory performance and caching behavior is presented. Raycasting a 256x256x109 dataset to a 512x512 image takes 54 seconds using 127 processors, with an efficiency of 80-85% attained.

Neumann [Neu92] describes an implementation of a parallel splatting algorithm for rectilinear data on the Pixel-Planes 5. The volume is splatted one slice at a time. The GPs compute coefficients for a quartic kernel. Renderers evaluate and produce the kernel which is then used for splatting into the current slice. The data distribution is static allocation, with slices interleaved among the GPs in round-robin fashion. The data set is stored three times with slicing in each of the 3 dimensions to accommodate different views. Renderers are assigned square portions of the image. They receive splat instructions from GPs only for voxels that project to their region. Splat instructions are ordered from front to back by GPs. Token passing (one per Renderer) is used to order instructions from GPs. The performance was analyzed on a 128x128x124 dataset. The best speed achieved was about 263ms for a 512^2 image using 40 GPs and 16 Renderers. Uniprocessor speed of the algorithm, with scalability and speedup statistics, are not given. Load balancing is difficult and efficiency depends on the correct number of GPs versus Renderers, which in turn depends on the image contents (especially the number of transparent voxels).

Elvins [Elv92] describes a parallel implementation of the splatting algorithm for rectilinear volumes on a 64 processor nCUBE 2. A master processor is responsible for reading in the dataset, dynamically allocating slices to processors for splatting, and collecting, ordering and compositing the contributions from each slice. Several optimizations intended to reduce the communications overhead were implemented and benchmarked. The results presented indicate that the speedup for this approach peaks at about 8 processors, severely limiting the scalability of the algorithm. The time to render a 256x256x90 dataset to a 200^2 image was 2 minutes 53 seconds.

Sakas [SH92] presents several scanline based methods for volume rendering rectilinear volumes and gives results of parallelization on an 8 processor Silicon Graphics 4D/380 VGX. The scanline methods range from use of a fast DDA to traverse the visible volume segment for each pixel, to the creation and use of a pyramidal volume. The hardware capabilities of the machine are used to speed geometric transformation and image display and manipulation. Performance on this machine is limited by access to the shared memory; it is not clear how the algorithm would perform on a highly parallel architecture. Using the pyramidal volume sampling a 256^3 dataset can be rendered to a 400^2 image in 47 seconds.

Machine	Data Size	Image Size	Time in Milliseconds	Number of Processors
Stanford DASH [NL92] Raycasting	256x256x226	416^2	340	48
nCUBE 2 [MPS92] Raycasting	97x97x116	350x250	4750	128
Fujitsu AP1000 [CM92] Raycasting	256x256x109	512^2	54000	127
SGI 4D/380 VGX [SH92] Scanline	256^3	400^2	47000	8
Pixel-Planes 5 [Neu92] Splatting	128x128x124	512^2	263	40 MIMD + 256K SIMD
nCUBE 2 [Elv92] Splatting	256x256x90	200^2	173000	64

Table 2.3: Results reported on MIMD architectures for rectilinear datasets.

All of the above research has been directed towards rendering of rectilinear datasets. Very recently researchers have been investigating parallel direct volume rendering algorithms for datasets that are nonrectilinear. We have published results of an investigation of a parallel raycasting algorithm which was implemented on the BBN TC2000 [Cha92]. The BBN TC2000 is a distributed-memory architecture which provides virtual shared-memory. An image-space task decomposition was used in which each scanline of the image constitutes one task. The scanlines are dynamically allocated to processors for rendering. The volumetric dataset is stored in virtual shared-memory and can be accessed by any processor. The dataset is interleaved over the processors to avoid hot spots. Performance was measured on a 37,479 hexahedral-cell dataset. A 512x512 image can be rendered in 53 seconds on 100 processors at an efficiency of 72%. A 256x256 image is rendered in 16 seconds on 110 processors at an efficiency of 57%. Remote memory latency, switch contention, and load imbalance were found to be the primary inefficiencies. This work is described in more detail in section 4.3.2.

Williams [Wil92a, Wil92c] has presented an algorithm for visibility ordering and projection of nonrectilinear volumes. A parallelization of the algorithm is given and its implementation on a 6 processor Silicon Graphics 4D/360 VGX is discussed. The sequential algorithm is very fast, about 1-3 orders of magnitude faster than raycasting. This speed is obtained primarily from the use of the SGI polygon rendering hardware to render the footprint approximations of each cell. The visibility ordering algorithm determines the order of projection. The 37,479 hexahedral-cell dataset is rendered in 7.8 seconds on 6 processors at an efficiency of 31%. Although the sequential algorithm is fast, the results presented indicate that its scalability when parallelized is low. In addition, much of the speed of the sequential algorithm is obtained through the use of hardware-renderable primitives. It is not clear how the algorithm would perform on a highly parallel system that does not have hardware support for rendering.

Lucas [Luc92] briefly describes a parallel direct volume rendering algorithm for irregular grids. The grid is partitioned on input into non-overlapping spatially connected regions. Nodes shared by more than one partition are replicated in each partition. Irregular grids are represented as a collection of polyhedra which are the faces of the cells of the grid. The rendering algorithm proceeds in two passes. In the first pass, task decomposition is by data partitions and each processor performs point and normal transformations and lighting calculations on the nodes in the partitions assigned to it. The image is subdivided into rectangular screen patches, and in the second pass the task decomposition is by screen patch. Each processor examines all the data partitions using a bounding box test to eliminate partitions that do not project to the current screen patch. For those partitions that do project to the current screen patch, all faces are examined and scan converted in back-to-front order. The visibility sort uses the centroid of faces which does not always give a correct ordering. An augmented z-buffer algorithm is utilized in order to take the depth of each cell into account when computing a cell's contribution and compositing it. Very few experimental results were presented, making it difficult to analyze the efficiency of the method. The machine which produced the results was not identified, but an efficiency of 77% was reported for 16 processors.

2.3 Parallel Computer Graphics

Related efforts in the parallelization of computer graphics algorithms are surveyed by Burke and Leler [BL90] and Crow [Cro90]. Several representative approaches are summarized here and will be described in more detail below. Fuchs proposed a technique for distributing the z-buffer hidden surface removal algorithm over a distributed-memory MIMD architecture [Fuc77]. Cleary, et al. give an algorithm for raytracing which utilizes a world-space decomposition on a distributed-memory MIMD system [CWBV83]. The rays are represented as messages passed between the processors. Dippé and Swensen extend this approach to achieve better load balancing by using an adaptive world-space decomposition [DS84]. Parallelization of global illumination algorithms in a distributed workstation environment has been addressed by Tampieri and Greenberg [TG88]. Parallelization of the raytracing algorithm on a distributed-memory MIMD architecture using an image-space decomposition is presented by Badouel [BP90]. This algorithm depends on an implementation of shared virtual memory with local caching. Whitman explores several image-space decompositions and scheduling strategies on a shared-memory MIMD machine [Whi92]. A detailed analysis of the overhead incurred in the parallelization is presented.

Fuchs [Fuc77] presents an algorithm for a MIMD architecture with distributed memory. Each processor is responsible for generating some portion of the screen image and part of its local memory will be used for the image and z-buffer. The image memories are dual-ported and a separate video scan converter handles the output of the image. A special processor which is connected to all others via a bus broadcasts each polygon in the scene to all the processors. Each processor performs a z-buffer scan conversion algorithm on the polygon and signals when it is done. When all processors have finished a "done" signal is present at the control processor which initiates the broadcast of the next polygon. An interlaced memory scheme is used to achieve load balancing. Several of today's powerful computer graphics workstations utilize a scheme very similar to this to speed the rendering process [Ake93, DN93].

Cleary, et al. [CWBV83] give an algorithm for raytracing on a distributed-memory MIMD architecture. A theoretical analysis of expected speedup was done, with software simulation to verify results. Performance is compared on two topologies; a 2D lattice and a 3D lattice with fast nearest neighbor communication and a slow connection to a controlling host. A world space decomposition is used, where traced rays are messages which are passed from processor to processor. The results indicate a less than linear speedup, with performance on a 2D lattice better than a 3D lattice.

Dippé and Swensen [DS84] describe an adaptive algorithm for generating raytraced images in which object space is divided into subregions. Using a theoretical argument, the algorithm is shown to be faster than the standard raytracing algorithm (in which every ray is tested against every object). They also describe a multiprocessing architecture onto which the algorithm is mapped. Three dimensional space is divided into subregions, with one or more subregions assigned to each processor. The processor containing the eye point initiates rendering by casting rays for each pixel. Each processor processes rays that enter its subregion by testing them against only the objects residing in that subregion. When a ray leaves a subregion and enters a neighboring subregion it is passed as a message. When a ray terminates and becomes a leaf of the raytracing tree, a pixel message is generated and propagated to the processor which has access to the frame buffer. Each subregion is a general hexahedron. The corner points are adaptively moved to accomplish load balancing. Neighboring regions must share load information. In the architecture described each processor is only connected to six neighbors, thus load and redistribution information must be routed through neighboring processors. Processors on the edge of the three-dimensional space have fewer duties, thus are assigned to handle external tasks such as the frame buffer, disk I/O, and the user interface. A simulator has been coded to test these ideas, but no empirical results were presented.

Tampieri and Greenberg [TG88] give an overview of global illumination algorithms (raytracing, radiosity, progressive refinement radiosity, and monte carlo), and give results of experiments with execution of these algorithms in a distributed workstation environment. Message passing over an ethernet connection is used for communication. For raytracing, an image-space subdivision is used and the entire scene database is stored at every workstation. The classical radiosity algorithm does not parallelize as it uses Gauss-Seidel; the progressive refinement radiosity algorithm is parallelized by having each workstation compute one column of the form-factors. The monte carlo method uses a coarse progressive refinement radiosity solution and then an image-space subdivision to trace rays. Demand scheduling is used for load balancing.

Badouel and Priol [BP90] describe parallel raytracing on a distributed-memory, MIMD architecture. The key feature is the use of shared virtual memory with local caching to make the data structures available to all processors. A task adaptive scheduling strategy is utilized. It is an image-space decomposition where each processor begins with a square subimage. Processors which finish early request more work from other processors. The speedup is very sensitive to the size of the local cache.

Franklin and Kankanhalli [FK90] present a parallel object-space hidden surface removal algorithm which uses the *uniform grid* technique. In this approach a GxG grid is cast on the eye-space scene. The polygons are sorted into any cells they cover, and are discarded if they are completely invisible. Within each cell all edges are subdivided into segments, where each segment will be completely visible or invisible. The visible regions in each cell are reconstructed from edge segments. A conflict detection and backoff strategy is given for the parallelization of the reconstruction step. The algorithm was implemented on a Sequent Balance 21000, which has 16 processors and shared memory, with almost linear speedup achieved.

Whitman [Whi92] explores several decomposition and scheduling schemes for the image-space decomposition of a scan conversion algorithm on the BBN GP1000. The decompositions include static data nonadaptive, data adaptive, and task adaptive. Techniques for determining the components of overhead were developed and empirical results are given for several images. The overhead components that were measured include scheduling, memory latency, block transfers, memory contention, algorithm adaptation, and synchronization. His conclusions were that use of local memory is important for good results, the task adaptive approach (utilizing local memory) has the best overall performance.

3. Highly Parallel Architectures

The research presented in this thesis is based on the premise that massively parallel architectures and operating systems will gradually become sophisticated enough that programmers will not need to deal directly with the low-level details of inter-processor communication as is currently required on many distributed memory systems. Highly parallel architectures are made feasible through the use of distributed memory and scalable interconnection networks. Attempts to allow the programmer to view these distributed memories as a single shared address space (whether through software techniques such as shared virtual memory or through architectural features) results in a paradigm in which accesses to memory are nonuniform. Latency for memory accesses will depend on many factors including physical location, cache coherency, and network contention. The advantage to designing efficient scalable algorithms that are based on a model of nonuniform memory access is that these algorithms may then be ported *(while maintaining their efficiency)* to any system which fits this model, regardless of the underlying physical hardware or interconnection network topology.

This chapter presents an overview of several highly parallel architectures from early attempts to provide a scalable shared-memory system to commercial and research machines still in development. The particular parallel machine used in this research, a BBN TC2000, is described in some detail.

3.1 Overview

Almasi and Gottlieb [AG89] describe several of the first highly parallel MIMD architectures which utilize shared memory including the Denelcor HEP, NYU Ultracomputer, BBN Butterfly, IBM RP3, and UI Ceder. A survey of more recent highly parallel MIMD architectures is given by Hord [Hor93]. Several of the more well-known or interesting architectures are briefly covered here. All of these architectures utilize physically distributed memory.

Four recent architectures which do not provide support for shared virtual memory are the Intel iPSC/2, nCUBE, Thinking Machines CM-5, and Fujitsu AP1000. The Intel iPSC/2 may contain up to 128 80386/80387 processors connected in a hypercube topology. The iPSC/2 system only provides support for message passing, however applications based on software implementations of shared virtual memory have been shown to efficiently utilize the machine [BP90]. The nCUBE may be configured with 8 to 8K processors. Each processor is a proprietary 64-bit VLSI chip. The interconnection network utilizes a hypercube topology with data and control messages routed via high-speed DMA communications channels with hardware routing. The nCUBE only supports message passing. Thinking Machines Corporation has recently introduced the CM-5, a scalable architecture which may contain up to 16K SPARC processors. A control network provides tightly coupled communications services. It is used for synchronization, broadcasting, and combining operations. A data network provides loosely coupled communications services for point-to-point data delivery. Only message passing is supported although the data network hardware provides a relative destination address that gives the illusion of a contiguous address space across processing nodes. This feature could be used to support shared virtual memory. Fujitsu has developed the AP1000, a distributed memory machine which utilizes from 64 to 1K SPARC processors interconnected using a 2D mesh topology. It also provides only message passing services, however recent work at the Australian National University has demonstrated an efficient application based upon a software implementation of shared virtual memory [CM92].

Four recently developed architectures which do support shared virtual memory are the Intel Paragon XP/S, Kendall Square Research KSR1, Cray T3D, and the Stanford Dash Multiprocessor. The Intel Paragon XP/S can be configured with up to 1K processing nodes, each containing two i860 processors. The nodes are connected in a 2D mesh topology. At each node, one of the processors is dedicated to application processing and the other to message processing, freeing the application processor from all details of communication services. Operating system support for shared virtual memory is provided. The KSR1 [Hen92] developed by Kendall Square Research supports 8 to 1088 RISC-style superscalar 64-bit processors. Their innovative ALLCACHE memory system distributes the memory physically as local caches associated with processors. The network consists of a two-level hierarchy of uni-directional rings. The ALLCACHE search engine moves data on reference to appropriate local caches and maintains cache coherence. Currently under development at Cray Research, Inc. is the T3D [Cra92], a highly parallel machine which can utilize "hundreds or thousands" of DEC Alpha chips. The memory is physically distributed with the processors and connected via a 3D torus topology. All memory is globally addressable and nonuniform (latency) access. The Stanford Dash Multiprocessor [LLG$^+$92] demonstrates the feasibility of building scalable parallel architectures with a single address space and coherent caches. A prototype based on 4 Silicon Graphics 4D/340 workstations has been developed. Each 4D/340 is considered a base cluster and contains 4 processors. Each processor has a cache, as well as shared memory available in each base cluster. The prototype connects the 4 base clusters in a 2D mesh, but any low-latency interconnection technology could be used. A single address space with coherent caches is maintained via directory structures. Cache levels include processor, local cluster, home cluster, and remote cluster.

3.2 The BBN TC2000

The machine used for this research is a BBN TC2000 located at the National Energy Research Supercomputing Center at Lawrence Livermore National Laboratory. This particular machine is configured with 128 processors and 2GB of main memory. The architectural features of this machine permit the extrapolation of results reported here to other MIMD architectures. The TC2000 provides memory that is physically distributed among the processors, but is globally addressable. Accesses to memory are nonuniform in that the latency for access to a remote shared-memory location is greater than for a local (on-board) memory reference. In addition, remote memory latency may increase with heavy use of globally-shared memory due to contention for the interconnection network. Many different MIMD architectures can be modeled as nonuniform memory access machines, from shared-memory architectures to distributed-memory architectures that support shared virtual memory.

	Local Read	Local Write	Remote Read	Remote Write
Uncached	0.550	0.600	1.913	1.889
Cache hit	0.150	0.600	0.150	1.889
Cache miss	0.850	1.200	2.529	4.168

Table 3.1: BBN TC2000 memory access times in microseconds. The cache hit and miss timings are for writethrough mode.

3.2.1 TC2000 Architecture

The BBN TC2000 is a multiprocessor architecture with a distributed shared memory [BBN89]. The TC2000 processors access the shared memory through an interconnection network called the Butterfly switch. The architecture is modular and scalable and can be configured to contain between 1 and 512 function boards. The main components of each function board include a Motorola 88100 RISC processor, a 16 kilobyte instruction cache and 88200 cache/memory management unit (CMMU), a 16 kilobyte data cache and 88200 CMMU, 4 to 16 megabytes of main memory, a switch interface, and a VMEbus interface. The 88100 is clocked at 20 MHZ, giving a manufacturer's rating of 17 MIPS and 20 MFLOPS for single precision floating point. These components are connected by an on-board bus called the T-bus. A system physical address in the TC2000 consists of 34 bits. The upper 9 bits specify one of 512 switch ports and the lower 25 address up to 32 megabytes on each function board.

When shared memory is allocated by an application, it can be specified as uncachable, cachable with copyback, or cachable with writethrough. Table 3.1 gives nominal memory access times for the system. The remote memory reference times do not include possible delays due to contention. The TC2000 also supports interleaving of shared memory to reduce switch contention. References made to a contiguous shared address space by a processor will be spread over several function boards by some mapping hardware. The basic clump size is 16 bytes which is also the maximum switch message data size. Each quad-page (32 kilobytes) may be either interleaved or non-interleaved. There may be multiple interleave pools or collections of function boards over which a given quad-page is interleaved.

The 88100 has one read-modify-write instruction, xmem, which exchanges the contents of a register with the contents of a memory location. The TC2000 is designed to honor the xmem instruction, meaning that atomic operations on memory locations can be executed over the interconnection network. When an xmem instruction references local memory, the CPU interface holds the T-bus preventing any other access until the transaction is complete. Likewise, an xmem instruction referencing remote memory holds the switch connection open, and holds the T-bus on the remote function board until the transaction is complete. Thus the xmem instruction is atomic on the TC2000.

Each function board contains a switch interface which is composed of a requester port and a server port. The requester port is used by the function board to request operations on remote function boards and to receive replies from those operations. The server port accepts requests for operations on this function board from remote function boards and sends replies to these requests. The reply to a given request is sent back over the same path as the request arrived on. Each Butterfly switch node is an 8 by 8 crossbar switch. These are arranged in two columns for a configuration with 64 function cards, or three columns for a configuration with more than 64 function cards.

When a requester port initiates a request, the header contains the route through the switch. The route consists of a list containing the output port required at each switch node and thus specifies a complete and exact path through the switch. At each switch node the output path for that node is removed from the header. If the output port is available, the connection is established and the remainder of the message is forwarded. If the output port is not available, the message is immediately rejected. As the rejects propagate backwards any partial connection is torn down and the switch resources freed. It is the responsibility of the requester port to reissue the request until a connection can be made. The requester ports support several pacing (backoff) strategies in order to reduce switch contention. It is possible to lock a switch connection and process several requests before releasing the connection, as long as the requests can be processed within a certain period of time. This is enforced by the use of a connection timeout. It is possible to bypass memory locking on a function board. For example, instruction fetch is always done in bypass mode. This allows instruction fetches from local memory to continue even though a remote function board may have the local memory locked.

3.2.2 The Uniform System

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

Memory Management

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

- `UsAlloc(Type, SizeInBytes)`

  Allocates a block of shared memory.
- `UsAllocLocal(Type, SizeInBytes)`

Allocates a block of shared memory on the local processor.

- `UsAllocOnUsProc(Type, Processor, SizeInBytes)`
  Allocates a block of shared memory on a given processor.
- `UsAllocScatterMatrix(Type, Rows, Cols, SizeInBytes)`
  Allocates a vector of `Rows` pointers in shared memory, and `Rows` separate vectors each containing `Cols` elements of size `SizeInBytes`. The `Rows` vectors are allocated in separate memories. Scattering data structures across several memories of the machine reduce memory and switch contention when many processors are accessing that data.

Another way to reduce switch and memory contention is to keep as much information as possible in process private memory. Constants and frequently referenced variables that are not changing rapidly can be replicated in each processor's local memory. Functions which propagate process private data to other processors include:

- `Share(&X)`
  Causes the (four byte) value of `X` to be copied into each processors private local memory prior to the execution of the first task worker function on that processor. `X` must be global or static.
- `ShareBlk(&X, SizeInBytes)`
  Causes the block of data beginning at `&X` to be propagated. The entire block must be global or static.
- `SharePtrAndBlk(&X, SizeInBytes)`
  Causes the pointer `X` and the block of data pointed to by `X` to be propagated. The pointer `X` must be global or static, the block of data has typically been allocated on the heap (via malloc).
- `ShareScatterMatrix(&X, Rows)`
  Propagates the vector of pointers created by `UsAllocScatterMatrix` to each processor, further reducing memory and switch contention.

Processor Management

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

- The generator mechanism is very efficient. It is implemented in one process per processor with each processor executing a tight loop of generate task - execute task with no context switches.
- The application is insensitive to the number of processors it is being run on.
- The load is balanced dynamically.

Many task generators, both synchronous and asynchronous, are provided. This research primarily uses an index generator of the form:

- `GenOnIFull(Initial, Worker, Final, Arg, R, MaxProcs, Abortable)`

The processor that calls `GenOnIFull()` will also participate in the execution of the parallel tasks. This task generator calls the function `Initial()` on each processor before beginning to execute tasks, and calls the function `Final()` on each processor before returning after all available tasks have been executed. Tasks are dynamically generated and are executed by calling the function: `Worker(0, i)` for $0 \leq$ `i` $<$ `R`. The `Initial()` and `Final()` functions are extremely useful in collecting performance statistics.

Index generation for 2D arrays is provided using the generator:

- `GenOnAFull(Initial, Worker, Final, Arg, R1, R2, MaxProcs, Abortable)`
  Generates tasks of the form `Worker(0, i1, i2)` for $0 \leq$ `i1` $<$ `R1` and $0 \leq$ `i2` $<$ `R2`.

Asynchronous index generation is possible using the generator:

- `GenID =`
  `AsyncGenOnIFull(Initial, Worker, Final, Arg, R, MaxProcs, Abortable)`
  In this case the function `AsyncGenOnIFull()` returns immediately to the caller with a generator handle enabling the caller to work on other things while the parallel tasks are executed by other processors. The generator handle may be used by the caller to wait for task completion using:
  `WaitForTasksToFinish(GenID)`,
  or to join in completing unfinished parallel tasks using:
  `WorkOn(GenID)`.

It is also possible to generate a single task on each processor using the generator:

- `GenTaskForEachProc(Worker, Arg)`.

Atomic operations are provided for 16-bit and 32-bit quantities by the Mach 1000 operating system (i.e. `atomadd32`, etc.). In addition, the Uniform System provides busy wait locks.

- `UsLock(lock, n)`
- `UsUnlock(lock)`

The variable `lock` is a short allocated in globally shared memory. The variable `n` specifies how long to wait between attempts to set the lock.

3.2.3 Parallel Job Control

The processors of the TC2000 at the National Energy Research Supercomputing Center is are divided into two conceptual groups, the public cluster and the parallel cluster. Currently 18 processors are reserved for the public cluster and these support user logins, editing, compiling, and other such activities. The remaining 110 processors are designated the parallel gang and run under the control of a UNIX daemon called the *Gang Scheduler* [GB91] which provides space and time sharing of the parallel gang. It is in the parallel gang that parallel user jobs run.

The Gang Scheduler provides several modes of operation for users in order to accommodate differing requirements. Time sharing and space sharing are offered for both batch and interactive execution. In time sharing, each user job (which will consist of several processes, one per processor) is given a time slice on the processors it is using. The entire job (processes on all processors) is started, run for its time slice, and then put to sleep to allow the next job in the queue to run. Space sharing is provided when a user job does not utilize all of the processors in the parallel gang. In this case a separate job may be running simultaneously on another set of processors in the parallel gang.

In order to provide for accurate performance measurements, a benchmarking mode is provided in which neither time nor space sharing is permitted in the parallel gang. Short benchmarking jobs (60 seconds or less) are allowed any time and long benchmarking jobs (2 hours or less) may be run between midnight and 6 AM. During benchmarking the user job is started and is not disturbed until the specified benchmarking time has elapsed. Although only the job running in benchmarking mode is executing in the parallel cluster, the public cluster will still be running user jobs and potentially generating network traffic. In addition, other parallel user jobs which may have been interrupted in order to run the benchmarking job will still be waiting to execute. These other jobs will have local memory allocated on the processors they are using. This may affect the amount of paging required in the execution of a new job. To summarize, the priority queues supported by the Gang Scheduler are:

- **Interactive:** Time and space sharing is provided with a small time slice.
- **Production:** Time and space sharing is provided with a large time slice (currently 10 minutes). Jobs in the production queue are run when they may share space with interactive jobs, or when there are no interactive jobs.
- **Standby:** Jobs in the standby queue are run when there are no jobs in either the interactive or production queues.
- **Benchmark:** Jobs in the benchmark queue run without space or time sharing until termination, or until the benchmark time is exhausted. Short jobs requiring 60 seconds or less will run immediately. Longer benchmarking jobs will be queued until midnight.

4. General Issues in Parallel Volume Rendering

This chapter presents a broad overview of the issues involved in designing parallel direct volume rendering algorithms for highly parallel MIMD architectures. The focus is on MIMD architectures in which some form of globally-shared memory is available, either through built-in architectural features, or through a software-based virtual shared-memory approach. The underlying model assumes a non-uniform memory access machine, in which various levels of memory exist with varying latency. Several possible approaches are examined and analyzed with references to existing literature. The rationale behind specific design choices for this research is made clear through examination of these issues.

4.1 Parallelization Issues

Parallelization of the direct volume rendering algorithms will involve decisions on two primary issues. The first is task generation, and the second is how to utilize the memory. Complicating these decisions is the fact that these two issues are related and a decision made in one area may affect the choices available in the other.

4.1.1 Task Generation

Task generation is the decomposition of a large job into smaller tasks which may then be performed in parallel. Task generation may be done in a wide variety of ways. There are three basic issues:

- the basis for decomposition,
- how task generation is done, and
- the size of tasks generated.

Decisions on these three issues have a significant impact on the scalability of a particular parallel algorithm.

The decomposition of tasks may be done in image space where each processor takes responsibility for generating some portion of the image. For example, each processor may generate one or more scanlines, or some rectangular portion of the image. Alternatively, the tasks may be based on a decomposition in object space where each processor is assigned some collection of cells or nodes in the volumetric dataset. Another approach involves subdividing world space. Each processor would handle those cells or nodes which fell into its world space (after the viewing transformation). This approach differs from an image-space decomposition by generating a 3-dimensional, rather than a 2-dimensional, decomposition.

Task generation may be done statically or dynamically. Static task generation involves deciding up front which portions of the total task are to be assigned to each processor. This decision should be based on some effective heuristic or this approach can lead to extreme load imbalance. Dynamic task generation means that the total work is broken up into tasks that are typically not uniform in size and are then allocated to processors in a dynamic fashion until all tasks are complete.

The size of the tasks generated is very important. For static task generation, the distribution of work needs to be as uniform as possible in order to minimize load imbalance. For dynamic task generation, a small task size and many tasks will improve load balancing. However, a very small task size also means the generation of many tasks will be necessary, generally introducing a serial section of code which quickly dominates and reduces efficiency. The optimal task size is probably a function of (at least) the size of the grid, the image size, the particular viewing transformation, and the number of processors. Coming up with an heuristic for the determination of a "good" task size at run time is an open problem.

There is a tradeoff between using a static task allocation strategy (which typically will incur inefficiencies due to load imbalance) and a distribution of data elements to each processor's local memory, or using a dynamic task allocation strategy and keeping the data elements in globally-shared memory (and paying the remote-reference penalty).

4.1.2 Memory Management

A decision must be made regarding how to store the large volumetric dataset and how to access portions of it for rendering. This decision has implications for task management. The dataset may be stored in globally-shared memory, providing access to any portion of it from any processor. This approach facilitates an image-space decomposition with dynamic task generation for rendering. Alternatively, the dataset may be partitioned and portions of it stored at individual processors. This approach suggests an object-space decomposition with static task generation and synchronization requirements for compositing the image.

There are advantages and disadvantages to storing the volumetric dataset in globally-shared memory. The simplicity of this approach minimizes the complexity introduced by parallelization and increases the understandability and maintainability of the code. It may also be desirable for the integration of the rendering code with an executing simulation. Regardless of how the simulation code has its data partitioned, if the data structures are available via globally-shared memory then it is probably desirable for the rendering algorithm to execute using those data structures without requiring a lot of data movement.

The primary disadvantage of the use of globally-shared memory is lower efficiency due to remote-memory latency and contention. Access time to globally-shared memory is greater than for local memory. In addition, an increase in the number of processors also increases contention for the interconnection network in order to gain access to globally-shared data structures, adversely affecting the scalability of the algorithm. Interleaving or scattering large data structures across the memories of the machine (as described in section 3.2) reduces memory and switch contention when accessing globally-shared data structures. Typically, globally-shared memory can also be specified as being cachable. An algorithm that takes advantage of coherence to efficiently utilize the caching capabilities of the system will exhibit improved scalability over one that does not.

In addition, local storage of required data structures can sometimes reduce the inefficiencies related to remote memory access. If the parallel algorithm moves globally-shared data into private local memory for further reference, block transfers will be more efficient, in general, than moving a word at a time. This will be particularly true on architectures such as those in which "globally-shared memory" is accessed via a message passing mechanism. It is conceivable that a parallel algorithm may perform a lot of preprocessing of the scalar dataset, the results of which may be stored locally (e.g. resampling for a given view, storing transformed points, gradients, etc.). Care must be taken with static local storage of partial results as performance will be greatly degraded if so much local memory is used that a given processor must resort to paging off disk.

4.1.3 Synchronization

Synchronization requirements between processors can add another level of complexity and inefficiency to parallel algorithms. For direct volume rendering in particular, the compositing operations to a pixel must be ordered. For a raycasting approach where a task is never smaller than one complete ray there are no inter-processor synchronization requirements. For a projection or splatting approach, the processing of cells or nodes will need to be ordered to ensure that the resulting compositing operations are also correctly ordered. This will lead to synchronization requirements [Cha91].

4.1.4 Other Issues

There are other issues that are important for developing parallel direct volume rendering as a useful, interactive, integrable, visualization tool for volumetric datasets:

- As the time to render an image approaches interactive rates the problem of getting the generated image out to the screen where it can be seen will present itself. This is especially critical in the context of users remotely located from the site of the massively parallel host. It would useful to output the image to an X-window on the user's workstation. Doing this in the most efficient manner may affect the decision about how to allocate the image memory.
- Also important is the ability to integrate traditional rendering algorithms based on geometrical primitives with the volume rendering algorithm. Rendering simulation geometry embedded in the scalar field, for example, greatly enhances visual understanding of the results. Is there an efficient approach to extending the parallel direct volume rendering algorithm to allow for embedded geometric primitives?

4.2 Parallelization of Specific Algorithms

It seems somewhat intuitive that an image-space algorithm such as raycasting would be most easily parallelized using an image-space decomposition for task generation. Likewise, an object-space algorithm such as cell projection seems to lend itself to an object-space decomposition for task generation. It is important to considered all the permutations however, even if to say that they are inefficient or not easily realized.

One important advantage of any image-space decomposition for volume rendering is the elimination of inter-processor synchronization requirements. In addition, an image-space decomposition with dynamic task generation results in good load balancing regardless of the desired viewing transformation or complexity of the grid. This is not the case for object-space decompositions where portions of the dataset are statically assigned to processors for rendering. Depending on the viewing transformation, some of these data partitions may not even project to the image (e.g. during a zoom, a common operation), while other partitions may cover much of the image.

A disadvantage to the use of an image-space decomposition is that it may preclude or make less efficient techniques that utilize spatial coherency to speed rendering.

4.2.1 Parallel Raycasting Algorithms

An image-space decomposition of the raycasting algorithm is very straightforward. The basic task can be considered to be the computation of a single pixel, and these can be grouped together as needed to generate tasks of optimal size. There are no inter-processor synchronization requirements. This approach lends itself to the use of a distributed image in which the image memory is distributed among the processors and is private. The pixel contents can be accumulated in the local memory of the processor executing the raycasting task. This implies that the image memory can easily be distributed; since the entire pixel contents are computed by the same processor, a pixel of the image will reside in the memory of the processor that computed it. Patterns of access to the volumetric dataset will be dependent upon the selected view, complicating the use of local memory for distribution of the dataset. Some sort of preprocessing or prefetching of required portions of the volumetric dataset could relieve contention, as could use of dynamic local storage mechanisms. The volumetric dataset is typically accessed repeatedly in a view dependent manner. Assuming that the volumetric dataset is located in globally-shared memory, several options that could be considered include:

- Keep the volumetric dataset in shared memory, making no attempt to do any local storage of portions of it. This approach will result in inefficiencies due to remote memory latency and contention. It is anticipated that this will adversely affect the scalability of the algorithm.
- Keep the volumetric dataset in shared memory and rely on the dynamic caching mechanisms provided by the architecture/operating system to locally cache required data values.
- The first time a portion of the dataset is intersected by a given ray, explicitly store the variables in local memory. If it required by that processor again, the local copy is used rather than the shared-memory copy. An algorithm for determining whether a cell is locally stored, and where, will be required.
- Sort the volumetric dataset for a given view, determining which cells may be intersected by a given ray or group of rays. This information can be compiled by examining the projected bounding boxes of the cells. These cells can then either be distributed to the processors (requiring a static task generation approach), or the information can be stored and used by each processor to prefetch the cells that will be required for rendering (allowing dynamic task generation). This approach is view dependent, requiring a new sort to be performed for a change in view. It will also require a significant amount of shared memory to store the results of the sort, and a significant amount of local memory to store the cells to be rendered. The size and shape of the tasks should minimize these additional storage requirements.

- Sample the volumetric dataset for a given view and store the samples in local memory. This approach is also view dependent, requiring that the volumetric dataset be resampled every time the viewing transformation is changed. A significant amount of local memory will be required to store the samples. The sampling tasks could be dynamically generated using explicit labeling of the resampled data, the compositing tasks would be statically generated (one per processor).

It is likely that some combination of these approaches will allow the desired levels of efficiency to be achieved when utilizing globally-shared memory.

An object-space decomposition of the raycasting algorithm is in some ways the equivalent of the cell projection algorithm, when considered at the level in which each task constitutes the rendering of one cell. The basic task might be the generation of all ray segments for a given cell. The compositing operations will need to be ordered between cells, whether they are processed by the same, or a different, processor. This will complicate the grouping of basic tasks in order to achieve optimal task size. This approach will easily support the distribution of the volumetric dataset among the local memories of the processors, and it could be view independent. Pixels in the image memory will be accessed by numerous processors, complicating the distribution of the image. A possible solution would be to give each processor some contiguous group of cells and have each processor generate an image (using either raycasting or cell projection) of its portion of the volumetric dataset in local memory. These images will then need to be composited together (in the correct order), requiring synchronization between processors. The grouping of cells for distribution would need to ensure that the required ordering on the compositing operations would not be violated.

4.2.2 Parallel Projection Algorithms

An image-space decomposition of the projection algorithm will be achieved by assigning each processor some part of the image, and having it scan convert (project) only those portions of cells which project onto that part of the image. Each processor will need to order its own operations on cells, but there will be no inter-processor synchronization requirements. There will be some duplication of work along boundaries of the image decomposition due to the fact that a given cell may project to more than one image tile. This approach is most easily implemented using globally-shared data structures for the volumetric dataset. All of the issues just discussed for the image-space decomposition of the raycasting algorithm will also apply here.

Object-space decomposition of the projection algorithm is conceptually straightforward, but is complicated by the synchronization requirements imposed by the ordering constraints of the compositing operations. The projection method for direct volume rendering lends itself to a distributed approach to volume storage, but with some tradeoffs for task generation and load balancing. The cell projection algorithm processes cells one at a time, accessing each cell once. This implies that the distribution of cells to processors for local storage is straightforward. It has implications for task generation and load balancing, however, because storing a cell at a processor implies that it will be rendered by that processor. This in turn implies static task generation. The alternative is to keep the volumetric dataset in globally-shared memory, and have the processors prefetch the cells which are dynamically assigned to them for rendering. Depending on the cell distribution, the contributions to a given pixel may come from several different processors. This implies the need for a shared image memory and synchronization of the tasks in order to prevent compositing conflicts. An alternative is to have each processor render its cells to a local image, followed by a post-processing phase where the local images are combined. Each processor will still need to order the rendering of its own cells to preserve the ordering of the compositing operations. In addition, the distribution of cells must make it possible to composite the distributed images without violating these ordering constraints.

4.2.3 Hybrid Algorithms

Both the cell-by-cell raycasting and cell projection methods have an equivalent inner function that consists of an integration and compositing step. The existence of this identical inner function facilitates direct comparisons between raycasting and projection algorithms. This basic unit of work is structured as follows. Given two points in a volume (the entry point and exit point of a ray through a cell) and the value of the scalar field at those points, estimate the integral across the distance between them, and use the result to generate a color and opacity sample. This color and opacity sample will be used in the compositing step. When structured in this way, the cell-by-cell raycasting and cell projection methods will give identical results. When the raycasting and projection approaches are broken down to this level and made as efficient as possible, they tend to merge and become a scanline algorithm.

Brute force raycasting would compare each ray to every cell in the volume, an extremely inefficient approach. One method of reducing the computational requirements of this approach is to examine only those cells whose image-space bounding box intersects a given ray. This information can be easily obtained using y-bucket and x-bucket sorts. To gain even more efficiency, intersections of the edges of the cell with the scanline can be computed and used to create the x-bucket sort. As the algorithm is modified to take advantage of spatial coherence it becomes akin to a scanline algorithm.

In the cell projection algorithm, all cells must be sorted into a partial order that is dependent on the viewing transformation. Each cell is then scan converted in order and rendered to the image. To reduce the inefficiency associated with scan converting shared faces and edges multiple times, the shared edges can be processed incrementally. This is easily accomplished using a scanline algorithm. This approach has the added advantage of eliminating the requirement of a visibility sort prior to rendering as will be shown in chapter 5.

Both the raycasting and projection algorithms for direct volume rendering can be improved by taking advantage of spatial coherence. The result of doing so is that the differences between the two approaches become blurred; raycasting and projection merge and become a scanline algorithm. Given an image-space decomposition, on any given processor the scanline approach is an intermediate algorithm which makes better utilization of coherence than raycasting and does not require the ordering of polyhedra as cell projection does.

4.3 Early Research Results

This section presents some of my early research results which, when combined with a review of the literature, has directed the focus of the research presented in this thesis. The rationale behind several key decisions is discussed, along with the statement of several key goals.

In order to experiment with various parallel volume rendering algorithms, an object-oriented volume renderer [Cha90] was ported to the BBN TC2000. This volume renderer initially used the raycasting algorithm as the method for rendering an image. The code was extended to provide the projection approach as another rendering method [MHC90]. The serial version of this volume renderer has provided a starting point for several experiments in parallel direct volume rendering.

4.3.1 Comparison of Decompositions for Rectilinear Grids

Some initial investigations into parallelization techniques for volume rendering are presented in [Cha91]. This work examines parallel implementations for both the raycasting and cell projection approaches to volume rendering for rectilinear grids. Only orthogonal viewing projections are considered in each case. The raycasting algorithm uses a sampling technique in which the entry and exit points of the ray with the volume are computed and the contents of the volume are sampled and composited at equally-spaced points along the ray segment that intersects the volume. The step size used for the sampling interval is such that every cell is sampled at least once. Code for the projection algorithm was provided by Nelson Max [MHC90] and performs scan conversion of the front and back faces of each cell, and provides an analytic method of integrating color and opacity which is equivalent to the limit of compositing more and more closely spaced samples. The sequential form of this algorithm processes the cells one at a time, in depth-sorted order. Both algorithms use globally-shared memory for storage of the volumetric dataset and the resulting image.

The parallel raycasting algorithm employs an image-space decomposition for task generation, and compares the results with two task sizes (pixel-per-task, and scanline-per-task). The tasks are allocated dynamically. The primary inefficiencies for the scanline-per-task approach were found to be remote memory latency and contention, and load imbalance. For the pixel-per-task approach, inefficiencies were remote memory latency and contention, and task generation. The image-space decomposition approach could benefit from additional techniques for utilizing local memory, and an heuristic for determining optimal task size.

A parallel projection algorithm utilizing an object-space decomposition was studied in which dynamic task generation of one cell per task was used. The volumetric data and the image memory were both located in globally-shared memory. A view dependent partial ordering of the cells was used during task generation to prevent conflicts in the compositing operations. It was found that the ordering required for the compositing operations prevented this approach from being effective. The task size was too small (one cell per task), and the serial sections of code were too large (required for synchronization of the tasks), resulting in poor speedup. Repeated access to globally-shared data structures further degraded performance.

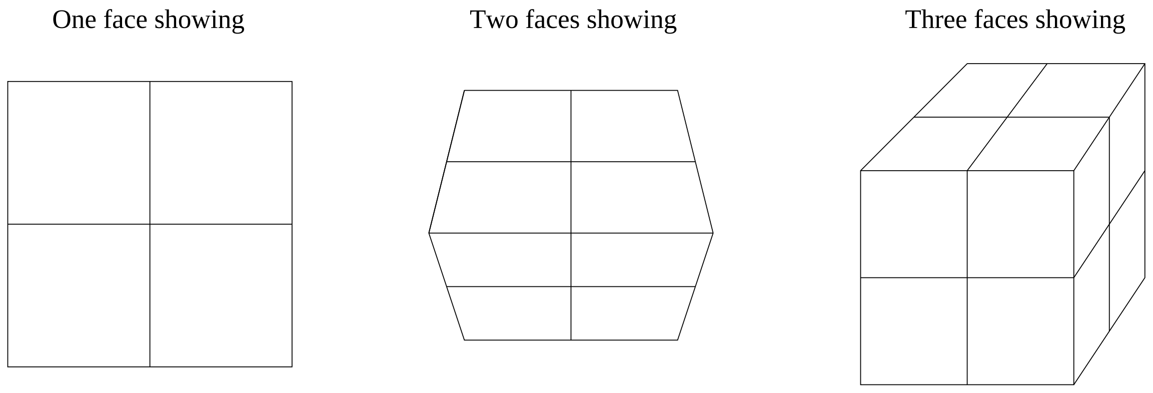

Figure 4.1: Three viewing cases affecting parallelism in the projection method.

Parallelization of the Raycasting Method

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

- Using `GenOnIFull()` a task is generated for each scanline of the image.
- Using `GenOnAFull()` a task is generated for each pixel of the image.

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

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

Data structures in process private memory:

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

Parallelization of the Projection Method

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

- One face visible: Initially an entire plane of cells will be available for rendering in parallel. Each cell rendered will affect the visibility of one other cell.

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

Figure 4.1 shows these three viewing cases. Although the figure presents the views in perspective for clarity, only orthogonal projections are being considered here. One possible approach to a decomposition for task management that would work for rectilinear grids would be to use a plane or slab of data as a task. Each processor would render the cells in its slab of data (in the correct order) to a local image. When all rendering is complete, the individual local images would be composited together (in the correct order) to form the final image. An approach of this type was not taken, primarily because it was desired that the algorithm be extensible to nonrectilinear grids. For this reason, two additional data structures are kept in order to drive the parallelism:

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

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

For rectilinear grids, the viewing case is easily determined by the number of back-facing faces. Only orthogonal projections are considered here; the visibility graph for a perspective view is slightly more complicated. One back-facing face corresponds to viewing case one in figure 4.1, two back-facing faces to viewing case two, and three back-facing faces to viewing case three. During an initialization phase each cell is initialized with a count of the number of cells it directly obscures. One or more cells which are farthest from the viewpoint will initially have count=0. After the counts are initialized, the ready list is initialized to include those cells with count=0.

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

In summary, for the parallel projection algorithm the task generator `GenOnIFull()` is used to generate one task for each cell in the volumetric dataset. Each task executes the following operations:

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

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

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

Results

The volumetric dataset used to produce these benchmarks was a $100 \times 120 \times 16$ electron density map for Staphylococcus Aureus Ribonuclease contributed by Dr. Chris Hill of the University of York. The dataset was rendered to a 512^2 image with the volume filling 500×512 pixels. This viewing configuration generated 512 tasks for the scanline-per-task approach; 262144 tasks for the pixel-per-task approach; and 176715 tasks for the cell-per-task projection approach. The measures used to analyze the performance of these approaches are discussed in section 2.2.2 and summarized in table 2.1. Figure 4.2 shows the speedup graphs for all three approaches. The speedup is basically linear for less than 10 processors, so measurements are given beginning at 10 processors. $T_1 = 548$ seconds for the raycasting algorithms and $T_{100} = 12$ seconds. $T_1 = 313$ seconds for the projection approach and $T_{100} = 26$ seconds. Let r_i be the total time processor i spends on rendering tasks. For the scanline-per-task approach, this is the sum of the times each processor spends rendering a scanline. For the pixel-per-task approach, this is the sum of the times each processor spends rendering pixels. For the projection method, this is the sum of the times each processor spends projecting cells to the image, including the explicit synchronization required for the use of the ready list and visibility graph. Then the total rendering task time, R_n, is defined to be $\sum_{i=0}^{n-1}(r_i)$. This total specifically excludes load imbalance and task generation overhead, but includes the overhead due to remote memory latency and contention. Figure 4.3 shows the percentage increase in the total rendering task time for increasing n. It can be seen that at 10 processors there is an increase of 82% for the raycasting approaches and 6% for the projection approach. These measurements give important insight into the amount and locality of remote memory accesses which in turn influence efficient use of the processor caching mechanism. For all methods, the total rendering time increases with the number of processors, reflecting the increasing memory and switch contention. This contention is exacerbated by a small task size and many tasks, probably due to numerous remote memory accesses for task generation.

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

Figures 4.4, 4.5, and 4.6, illustrate the execution profiles of the three approaches for various numbers of processors. All measurements are given as a percentage of $TOTAL_n$ (see table 2.1). States which account for under 2% of the total processing time are not shown. Thus, task generation does not appear in figure 4.4 because it is negligible for the scanline-per-task approach. The measurements collected include:

- **Startup time**: the sum of the times each processor spends getting started, that is from the time the code goes parallel to the first execution of a task. It is during this time that data structures which are to be propagated to each processor's local memory are moved. The startup time was found to be negligible for all three approaches, but may become an important factor as the rendering approaches interactive speeds.
- **Load imbalance**: the sum of the times each processor spends waiting after finishing its last task, until all processors have completed their last tasks. The load imbalance is seen to affect the speedup of the scanline-per-task approach for large n (see fig. 4.4).
- **Render time**: for the scanline-per-task approach, this is the sum of the times each processor spends rendering a scanline. For the pixel-per-task approach, this is the sum of the times each processor spends rendering pixels. For the projection method, this is the sum of the times each processor spends projecting cells to the image, excluding the explicit synchronization required for the use of the ready list and visibility graph.
- **Get cell synchronization**: for the projection method, this is the sum of the times each processor spends getting a cell to render off the ready list, including waiting if the list has been locked by another processor.
- **Update graph synchronization**: for the projection method, this is the sum of the times each processor spends updating the visibility graph after rendering a cell, including possibly waiting for access to the ready list and moving additional cells to it.
- **Other overhead**: for all methods, the sum of times for each processor in which that processor was not in one of the above states. This is primarily the time required for task generation, but may be affected by other unknown operating system inefficiencies. In particular, it can be seen in figure 4.5 that the `GenOnAFull()` task generator is extremely inefficient for large n.

Conclusions

Although the sequential projection algorithm is faster than the raycasting algorithm on the particular dataset and viewing configuration that was benchmarked, and the parallel projection approach generates less remote memory overhead than the parallel raycasting approach, the parallel algorithm for the projection method using a cell per task does not scale well with the number of processors. The synchronization requirements for the proper

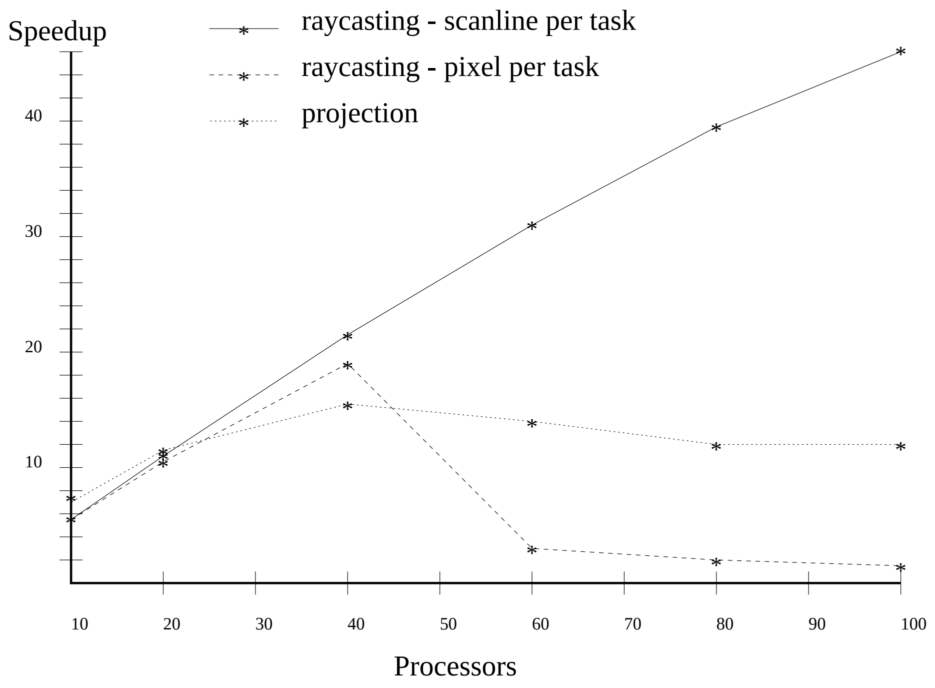

Figure 4.2: Speedup graphs for projection and raycasting on a rectilinear grid.

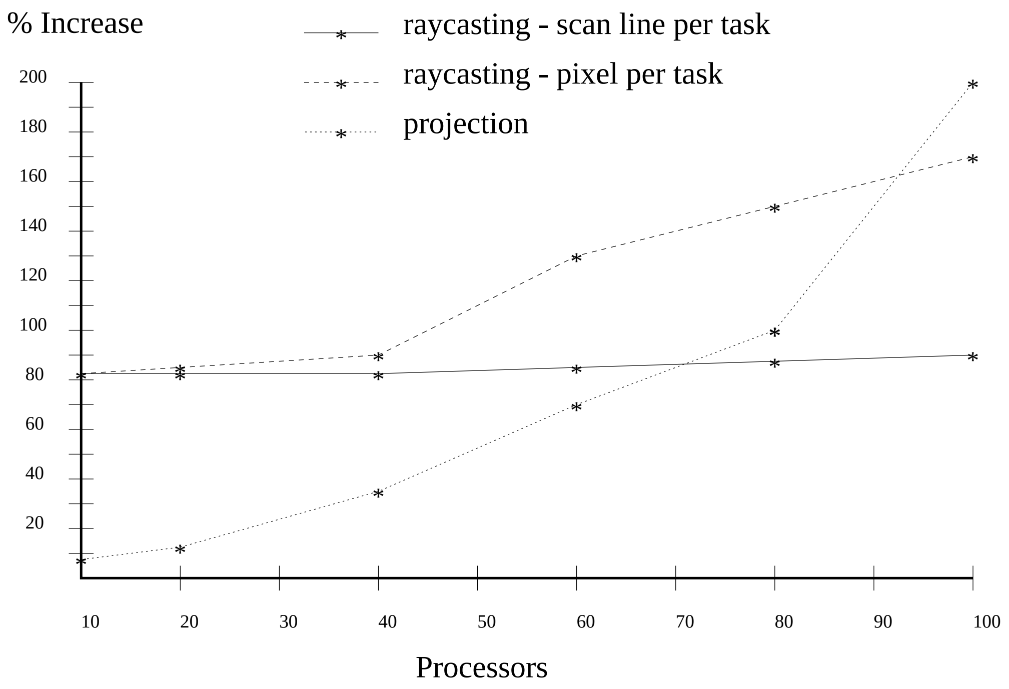

Figure 4.3: Increase in total rendering task time (R_n) on a rectilinear grid.

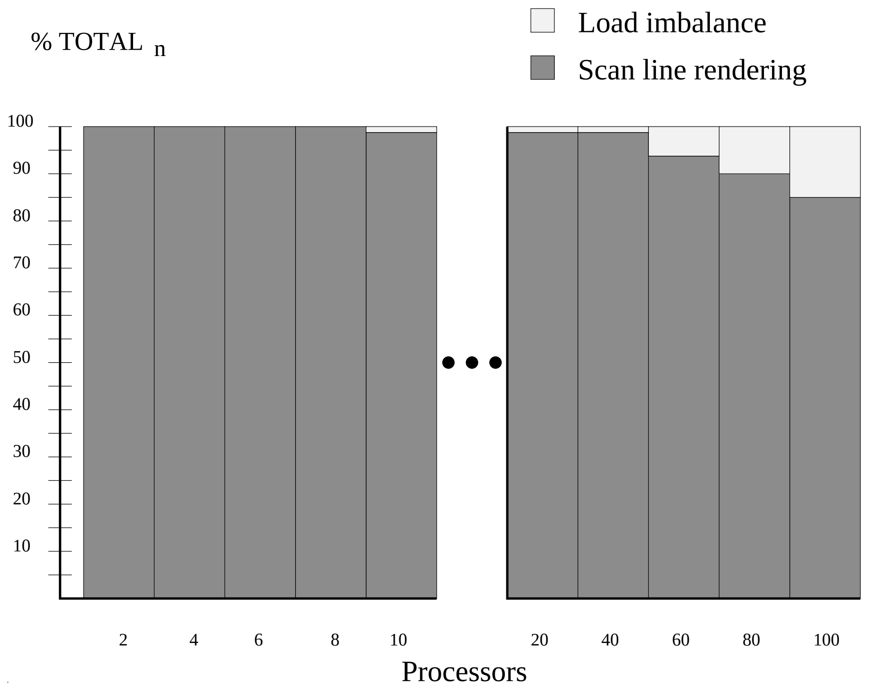

Figure 4.4: Execution profile for raycasting with one scanline per task.

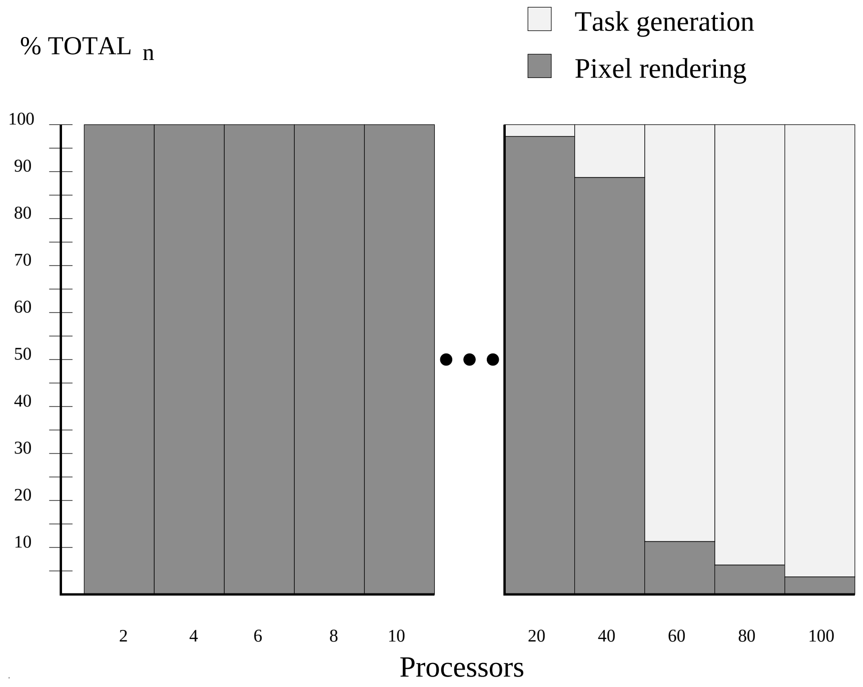

Figure 4.5: Execution profile for raycasting with one pixel per task.

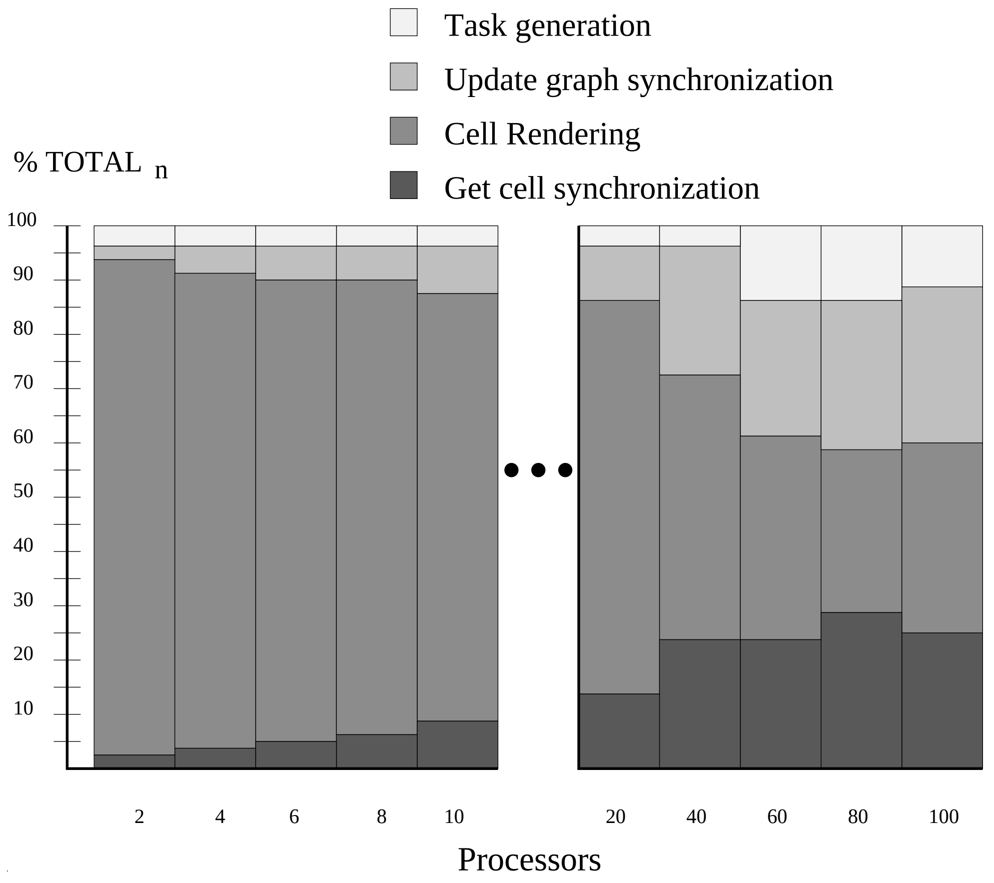

Figure 4.6: Execution profile for projection with one cell per task.

ordering of cell rendering generate a significant amount of overhead as n increases (as shown in figure 4.6). It is not clear why the synchronization cost decreased at 100 processors. The main inefficiencies for the scanline-per-task approach for the raycasting method are load imbalance for large n, and the penalty for using globally shared memory (see figure 4.4). The number of tasks, and therefore task size, has a significant effect on scalability as can be seen by the inefficiencies in task generation for raycasting with a pixel per task (figure 4.5).

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

The projection method, on the other hand, does have ordering constraints which complicate task generation. The techniques used for the parallelization of the projection approach require large amounts of memory in order to store the visibility graph, the use of locks around critical sections of code, and atomic operations. There are two requirements for making parallel projection viable. The task size must be increased, which implies grouping cells together to form tasks. In order to prevent compositing conflicts, these groups will need to be convex. To ease synchronization requirements, the cells in a group should be connected (adjacent to each other). It is anticipated that the determination of how many and which cells should be grouped together to form a task will be critical to the performance. For regular rectilinear volumes the decomposition may be fairly straightforward. It will be more complex for curvilinear and unstructured grids. The second requirement is to reduce the synchronization necessary between tasks. This problem will be eased somewhat by increasing the task size, but possibly not enough, and the larger task size may create load balancing problems. Two approaches that could be considered include:

- Decouple the rendering and compositing phases between processors. Have each processor render its cells to a local image. When all rendering is complete, composite the distributed images into one image.
- Use an image-space decomposition to determine the distribution of cells to processors, and to determine the distribution of image memory. Processors will then render the cells to a local image, the image space decomposition will specify which pixels reside at any given processor.

The determination of cell groupings to form tasks and the ordering of these tasks is an open problem.

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

4.3.2 Extension of Raycasting Approach to Curvilinear Grids

A technique for parallel volume rendering of curvilinear grids is described in [Cha92]. The cell-by-cell raycasting approach is enhanced with an algorithm for maintaining parallel active lists for each scanline in order to speed computation of ray/cell intersections. This approach also applies to the rendering of unstructured grids where neighboring cells are not known, without requiring the construction of an adjacency graph. Dynamic task generation is performed with a scanline per task. As with regular rectilinear volumes, remote memory latency and contention, and load imbalance, are found to be the primary inefficiencies. Reducing the task size and efficiently maintaining the active lists presents a problem. A two-phase algorithm in which the intersection lists for a given viewing transformation are explicitly stored in local memory is explored. This approach decouples the sampling and compositing phases of rendering, leading to the possibility of fast image update rates for changing transfer functions. A similar approach has previously been utilized to speed computation of successive ray-traced images with changing lighting conditions and surface properties [SS89a]. The performance of the two-phase algorithm is found to be comparable to that of the one-phase algorithm, with the advantage that the decoupled compositing phase takes less than a second with 100 processors.

Algorithms for Curvilinear Volumes

Two algorithms are presented for volume rendering of curvilinear volumes (in which a regular rectilinear grid in computational space has been warped or curved in $\Re^3$ to match the simulation geometry in physical space). Both algorithms use the raycasting approach to volume rendering, avoiding the issue of inter-processor ordering of the compositing operations for each pixel which arises with parallelization of object-space volume rendering algorithms [Cha91]. Only orthogonal viewing projections have been implemented, although the algorithms are equally applicable to perspective projections.

The algorithms presented here require that, for each ray, all intersections with cell faces be found. The algorithm proceeds cell by cell through the volume, estimating the line integral of the scalar field. An alternative to proceeding cell by cell is to take equidistant steps along the ray. This approach is commonly used for volume rendering regular rectilinear volumes. Although it avoids the problem of needing all face intersections for a ray, it instead requires that for each point along the ray, the cell containing that point be determined. This point location problem is trivial for regular rectilinear volumes, but much more complex for curvilinear volumes. Our approach is to reduce the number of intersection calculations required for each ray by utilizing the idea of a bucket sort and scanline algorithm from computer graphics [FD82, HB86]. For example, a y-bucket sort lists, for each scanline, the objects which begin on that scanline, and how many scanlines they are active. From this information a list of active objects (objects which may possibly be intersected) can be incrementally maintained from scanline to scanline. The algorithms presented here use a shared y-bucket sort to create private x-bucket sorts and active lists in order to reduce the number of intersection calculations required for each ray. A cell is considered active if its bounding box is intersected by a ray.

The volumetric dataset is stored in globally-shared memory as a cell-oriented data structure. Data shared by multiple cells (e.g. scalar value and location in physical space of a node) is stored once, with each cell maintaining a pointer to the information. Both algorithms begin by initializing the globally-shared data structures. Each cell is instantiated as an object and these cells are scattered across the globally-shared memory. A single cell stores three integers, (i, j, k), which identify one node of the cell. The spatial extent of the cell is defined by eight nodes: (i, j, k), $(i, j, k+1)$, $(i, j+1, k)$, $(i, j+1, k+1)$, $(i+1, j, k)$, $(i+1, j, k+1)$, $(i+1, j+1, k)$, $(i+1, j+1, k+1)$. The bounding box of each cell in screen coordinates is computed and stored with the cell, and a pointer to the cell is added to a shared y-bucket sort whose records are also scattered across the globally-shared memory. All other data structures (transfer functions, viewing specifications, interpolation methods, etc.) are in private local memory.

For the purposes of intersection calculation, each cell face is assumed to be comprised of two triangles. This is because the faces of a cell are not necessarily planar. For each ray, an ordered list of intersections is determined. Each intersection determines an entry/exit point of the ray with a cell. For each cell on the list, a sample of the scalar field is obtained at the midpoint of the ray between its entrance into and exit from the cell. The sample is obtained as a weighted sum of the eight scalar values at the nodes of the cell. If S_i is the known scalar value at grid node P_i, then $S(P)$ is the estimate of the scalar value at the desired sample point P

$$S(P) = \sum_{i=0}^{7} w_i S_i$$

In this work, inverse distance weighted interpolation was used in which the weights for each node are dependent on the distance of that node from the desired sample point

$$w_i = \frac{\prod_{k=0,k\neq i}^{7}[d_k(P)]^2}{\sum_{j=0}^{7}\prod_{l=0,l\neq j}^{7}[d_l(P)]^2}$$

where $d_k(P)$ is the Euclidean distance from the sample point P to the node P_k. Inverse distance weighted interpolation has the advantages of being fairly fast and rotationally invariant, but has the disadvantage that it does not have C^0 continuity along interior faces of the grid [WCA+90, Ram91]. The opacity value obtained by the transfer function mapping of the scalar value to opacity is weighted by the distance along the ray through that cell.

Single-phase Algorithm

The first algorithm proceeds to collect samples and composite them into the image in a single phase. Dynamic task generation is used to generate a task per scanline. The process private data is propagated before tasks begin. Each task executes the following initialization steps:

- Sweep through the y-bucket sort to create a list of cells active on this scanline.
- Create a local x-bucket sort from this list.
- Use the local x-bucket sort to create a local active cell list.

Then, for each pixel on the scanline:

- Generate an intersection list by testing active cells for intersection with the ray.
- Sample, map, and composite into the pixel.
- Incrementally update the active cell list.

The y-bucket sort, which is intended to be used incrementally, must be examined for entries from the first scanline to the current scanline by each processor. Other alternatives are possible, for instance, a list could be kept for each scanline of all the cells intersecting that scanline. This approach would require more memory than the y-bucket sort, but would be more efficient computationally. Profiling of the code showed that only 3% of the time to render an image using 100 processors was spent doing the y-bucket sweep. The sequential version of the algorithm utilizes the y-bucket sort in the traditional way, doing incremental updates to the list of active cells as each scanline is processed.

Two-phase Algorithm

In the second algorithm, the rendering proceeds in two phases. In the first phase, all the samples are taken and stored in local memory. In the second phase, the (locally) stored scalar samples are used to generate the color and opacity for that sample which are then composited into the image. In this way fast update rates may be attained for a given view of the volume, but with changing transfer functions. This would be useful for interactive exploration of transfer functions. Finding a good transfer function (one which highlights areas of interest) is currently one of the hardest aspects of volume rendering and is typically done by experimentation.

In this approach all of the intersection lists generated by the first phase of the algorithm are stored in local memory. As this is memory intensive, the algorithm is only intended to be used on many processors. Use of the algorithm on too few processors will result in paging as processors utilize more local memory than is physically present. To create an $s \times s$ image, the raycasting approach will cast s^2 rays into the volume generating s^2 intersection lists. For an n^3 volume where $s \approx n$, the total size of the intersection lists will be comparable to the size of the volume itself. If $s > n$ and the volume fills the image, the total size of the intersection lists will be larger than that of the volume. If the volume does not fill the image many of the rays will contain empty intersection lists, possibly resulting in a collection of intersection lists that require less space than the volume.

Using a static decomposition, allocate one task per processor. Given n processors, processor i (where $0 \leq i < n$) takes scanlines i, $n+i$, $2n+i$, etc. The rendering is split into a sampling phase and a compositing phase. During the sampling phase, each task performs the sampling for each pixel on each scanline assigned to it:

- Sweep the y-bucket sort.
- Create an x-bucket sort.
- Create an active cell list.
- For each pixel on the scanline:
  - Generate an intersection list by testing the ray against active cells.
  - Sample, and save in local memory.
  - Incrementally update the active cell list.

During the compositing phase, each task performs the mapping and compositing for each pixel on each scanline assigned to it:

- For each stored sample at each pixel:
  - Map sample to color and opacity.
  - Composite into the pixel.

Elimination of Small Cells

In either of the above algorithms it is possible to have cells which are small enough that their screen-space bounding boxes fall between scanlines or between pixels. This is a form of aliasing that can be reduced by stochastic sampling methods [Gla89], or by volume pyramid approaches such as those described by [SG91]. In the absence of anti-aliasing, these cells can be trivially eliminated from the bucket sorts (and thus not considered for possible intersection with a ray). Doing this improves both the absolute time required to render the image, and the load balancing of the parallel decomposition.

Task Order

The complexity of the rendering requirements in terms of the number of cells projecting to any given part of the image can vary widely across the image. The shaping of the curvilinear grid usually results in some areas of many small cells and other areas with just a few large cells. This complicates attempts at achieving good load balancing. One approach is to try to ensure that the potentially larger tasks are dynamically generated first. During the creation of the shared y-bucket sort, it is easy to produce a sorted order for the processing of scanlines based on the number of cells that become active on any given scanline. It was found that doing this somewhat improves the load balancing as the number of processors approaches the number of scanlines being generated (i.e. generating an image with 256 scanlines using 100 processors). A better heuristic might be to produce a sorted task order using the number of cells active on a scanline, rather than the number of cells that become active.

Storage Method

The raycasting algorithm repeatedly accesses the volumetric dataset in a view dependent manner. Thus the determination of how the volume is stored can have a significant effect on the performance of the algorithm. Of course the most efficient approach would be to store the entire volume at each processor. This is generally not feasible for large datasets. A possible compromise that increases efficiency on some architectures is to distribute multiple copies of the dataset to groups of processors [MPS92]. For systems that support virtual shared memory with local caching, another form of partial dataset replication is implicitly achieved. In this particular experiment the volume is stored in globally-shared memory. To reduce contention and to ensure that large datasets can be handled, the volume is scattered across the memories associated with each processor being used. Two approaches to doing this scattering have been examined and the results are given in the next section.

Results

The parallel algorithms described above have been implemented on the BBN TC2000 and their performance has been measured and analyzed (see table 2.1 for a summary of measures). The algorithms have been benchmarked on the blunt fin data set from NASA Ames Research Center [HB85]. This dataset represents a CFD simulation of air flow past a blunt fin on a grid resolution of $40 \times 32 \times 32$, or 37479 cells. Images have been created at resolutions of both 256^2, and 512^2. Even at these resolutions, many cells are so small they fall between pixels as described above.

All of the timing results presented here represent only the rendering portion of the algorithms described above. The time required to read the volume in and perform the view dependent preprocessing step (creation of the shared y-bucket sort) was not measured.

Single-phase Algorithm Performance: Figure 4.7 shows the time required to render a 256^2 image using the single-phase algorithm. The time went from 990 seconds on one processor using all local memory, to 16 seconds on 100 processors. This represents a speedup of about 62 and an efficiency of about 62%. The time to render a 512^2 image went from 3804 seconds on one processor to 53 seconds on 100 processors for a speedup of 72 and an efficiency of 72%. Speedup graphs are shown in figure 4.8. The speedup is basically linear for less than 10 processors, so measurements are given beginning at 10 processors.

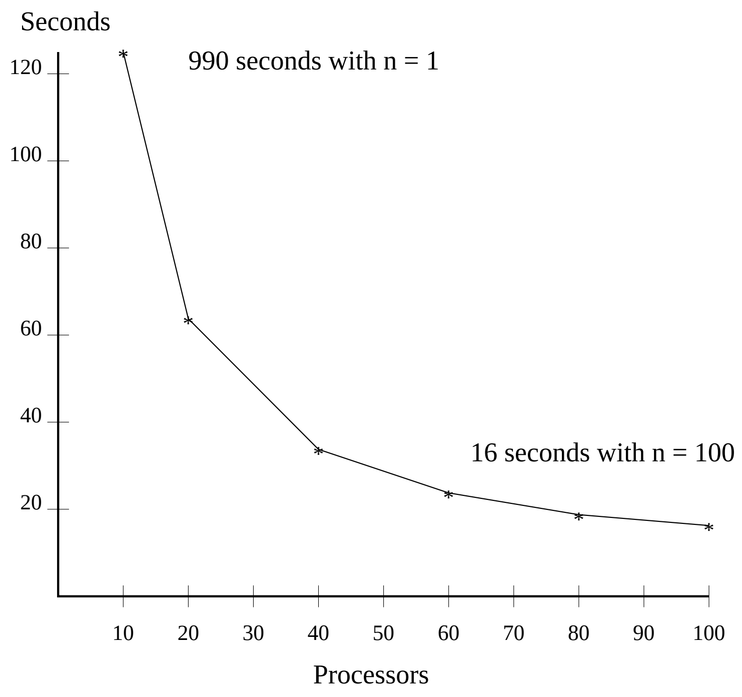

Figure 4.7: Rendering time (T_n) for the single-phase algorithm on a nonrectilinear grid.

The primary sources of inefficiency are load imbalance and remote memory latency and contention. Figure 4.9 shows the percentage of the total computational resources ($TOTAL_n$) actually spent on rendering, as opposed to waiting for a task (negligible in this case) or waiting for other tasks to complete (load imbalance). It can be seen that load imbalance is a significant contributor to overhead.

Let r_i be the total time spent on rendering tasks by processor i. Figure 4.10 shows the percentage increase in the total rendering task time (R_n) for increasing n. This graph demonstrates the other significant inefficiencies; remote memory latency and contention for access to shared data structures. It was not explicitly measured in this particular study, but it is anticipated that the increase in total rendering task time will be characterized by a jump on going from one processor using local memory to two processors using globally-shared memory, followed by a gradual increase as the number of processors increases. The initial jump is due to remote memory latency, and the gradual increase is due to contention.

Since the volumetric dataset is repeatedly accessed in a view-dependent manner during rendering, the choice was made to store the volume in globally-shared memory. In order to reduce contention, the contents of the volume are scattered across the memories associated with each processor in use. Two approaches to doing the scattering have been investigated. The first is to scatter planes of data across the processors. Let the dimensions of the computational grid be described by i, j, and k. For example, the blunt fin dataset has $k = 32$ planes of $i \times j = 1,280$ nodes. In the results just presented, each of these planes was stored at a different processor. However, this means that when $n > 32$, some processors will

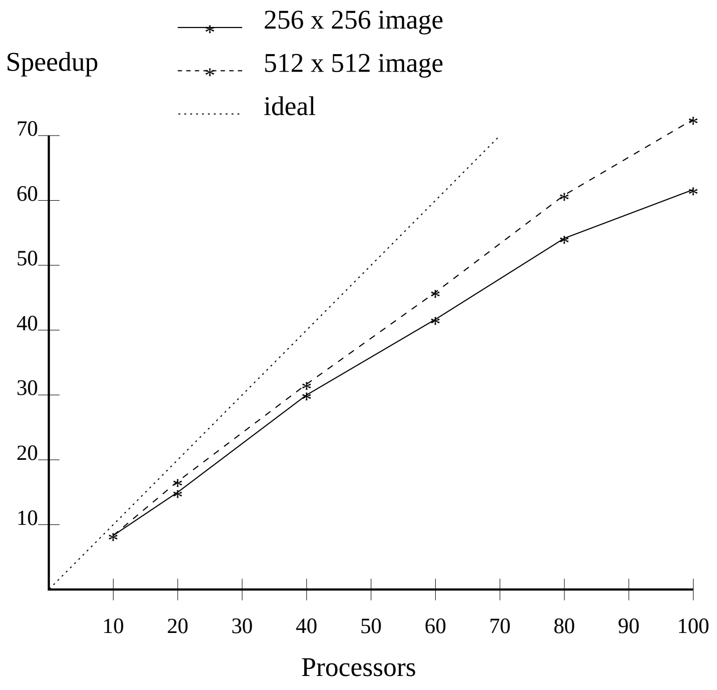

Figure 4.8: Speedup for the single-phase algorithm on a nonrectilinear grid.

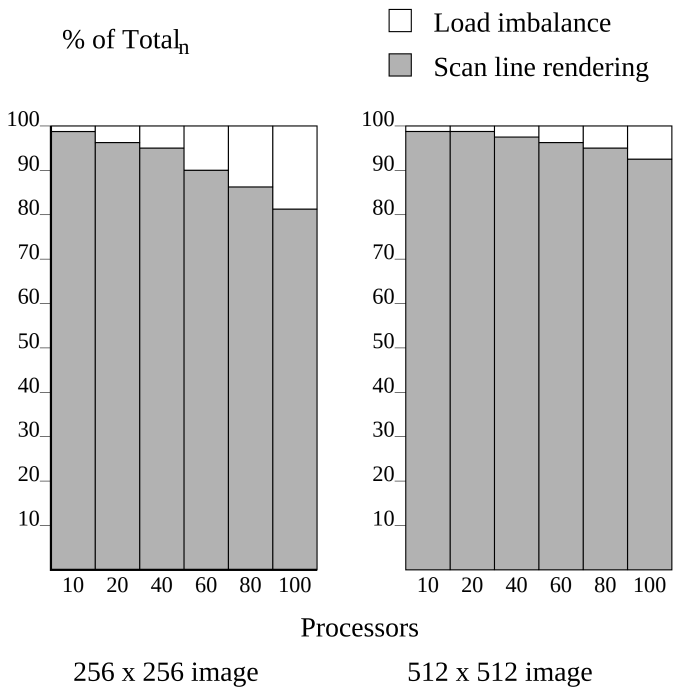

Figure 4.9: Execution profile for the single-phase algorithm.

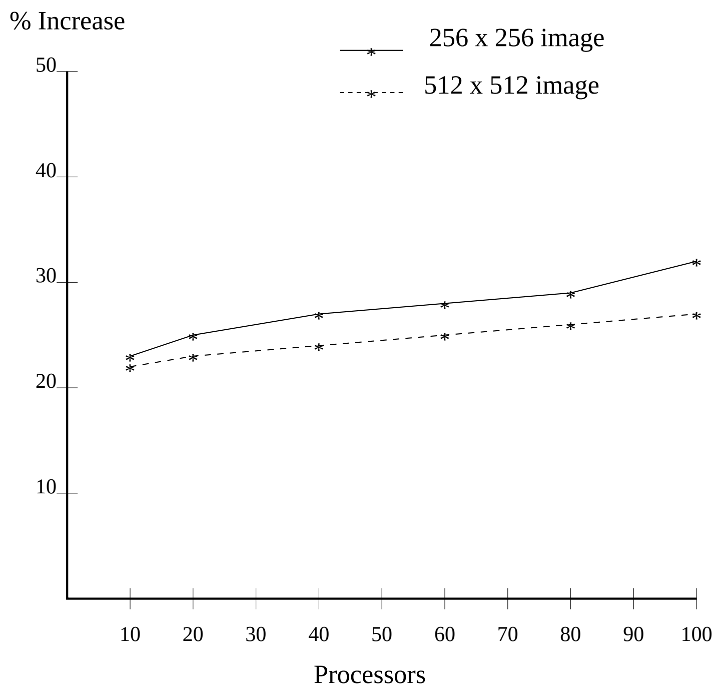

Figure 4.10: Increase in total rendering task time (R_n) on a nonrectilinear grid.

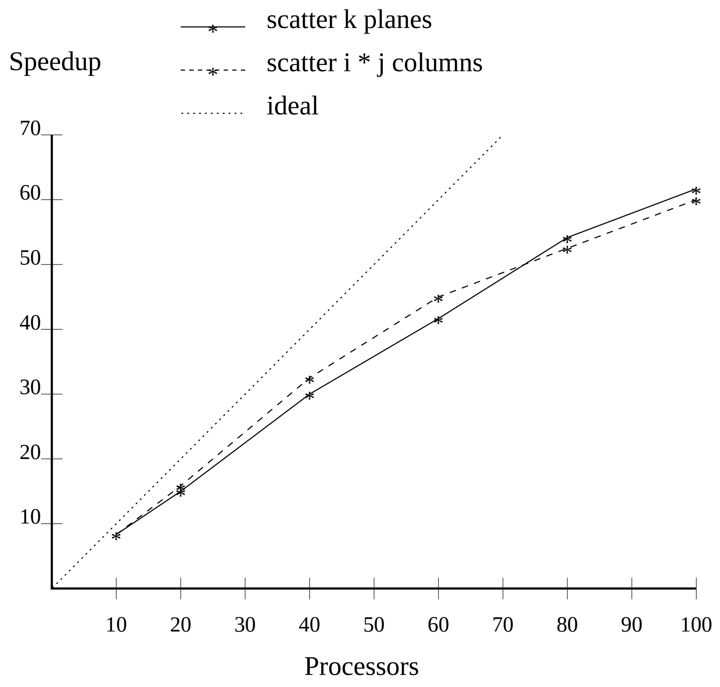

Figure 4.11: Speedup by scatter method.

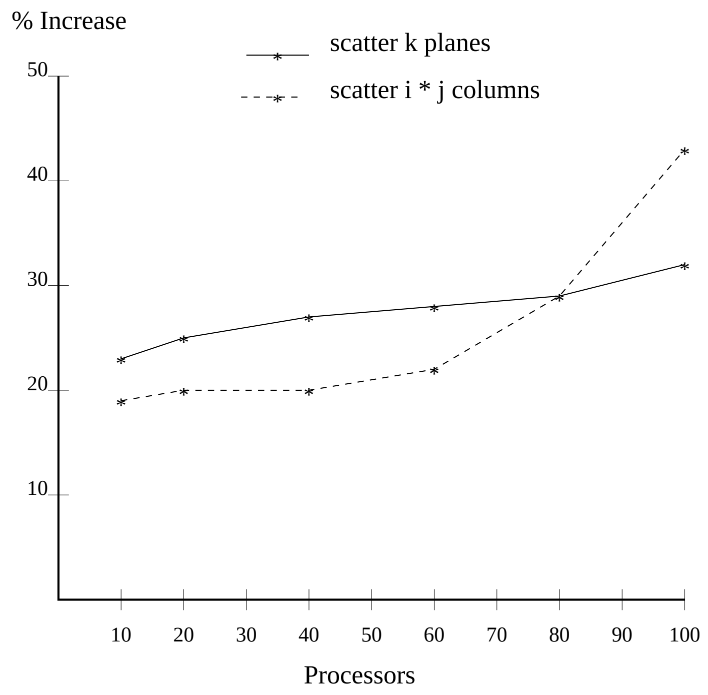

Figure 4.12: Increase in total rendering task time (R_n) by scatter method.

not have any data stored in their associated memory. An alternative approach would be to store $i \times j = 1,280$ columns of $k = 32$ nodes at different processors. This would spread the dataset more evenly across the available memories. It was expected that this would reduce contention, even for $n < 32$, by reducing the probability that multiple processors would require data from the same memory. Figure 4.11 shows that the approach was effective for less than about 80 processors. Figure 4.12 shows that the rate of increase in contention increased at about 60 processors for the column-scattering approach, resulting in more contention for the column-scattering approach than for the plane-scattering approach for $n > 80$. It is not at all clear what factors influenced this.

Two-phase Algorithm Performance: Using the two-phase approach, 20 seconds were required to complete both phases of the rendering algorithm on 110 processors, but the compositing phase takes about 400 milliseconds.

This algorithm has to perform more work than the single-phase algorithm because it must sample every cell along every ray. In the single-phase algorithm, sampling stops when a pixel becomes opaque and in many cases the entire ray need not be processed. In addition, the two-phase algorithm has a much higher use of local memory than the single-phase algorithm, thus for small numbers of processors the performance may be much worse due to paging of virtual memory. It was found that the implementation is sensitive to machine load (even in benchmarking mode), probably due to extensive use of local memory. It does achieve the desired goal, however, in that the time required for the compositing phase is very small. For rerendering a given view after changes in the transfer function, the algorithm runs at interactive rates.

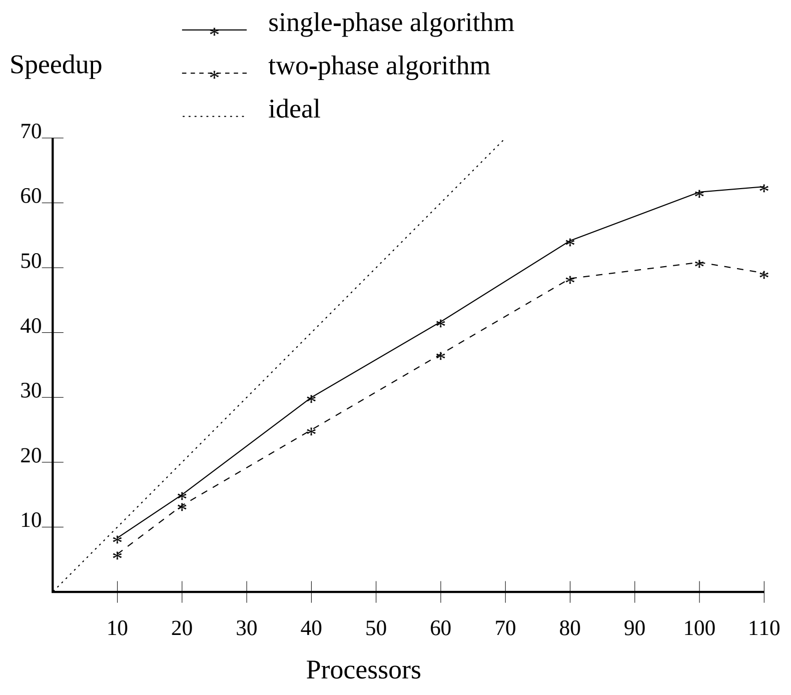

Figure 4.13: Speedup for the two-phase algorithm.

The speedup for this algorithm was computed using the serial time for the single-phase algorithm since the serial time for the two-phase approach was very large due to paging. The graph is shown in figure 4.13. The two-phase algorithm uses a static task decomposition which negatively affected the load balancing (see figure 4.14). One alternative may be to use dynamic task generation to allocate sampling tasks, followed by static task generation for the compositing phase.

Conclusions

The main inefficiencies were found to be remote memory latency and switch contention for globally-shared memory, and load imbalance. Load balancing is more difficult with curvilinear volumes than with rectilinear volumes. There are some direct tradeoffs between taking advantage of image coherence, through the use of the scanline algorithm for instance, and having the ability to decompose the image arbitrarily to achieve good load balancing.

The algorithms presented here will work equally well on unstructured grids such as those used in finite element analysis, without requiring an adjacency graph to be constructed. The algorithms also support the inclusion of embedded geometrical primitives (polygons), leading to the ability to render the simulation geometry with the scalar field to obtain a more easily understood image [Cha90]. Finally, the two-phase approach gives the user the capability to quickly explore different transfer functions for rendering. This will also lead to increased understanding of the scalar field.

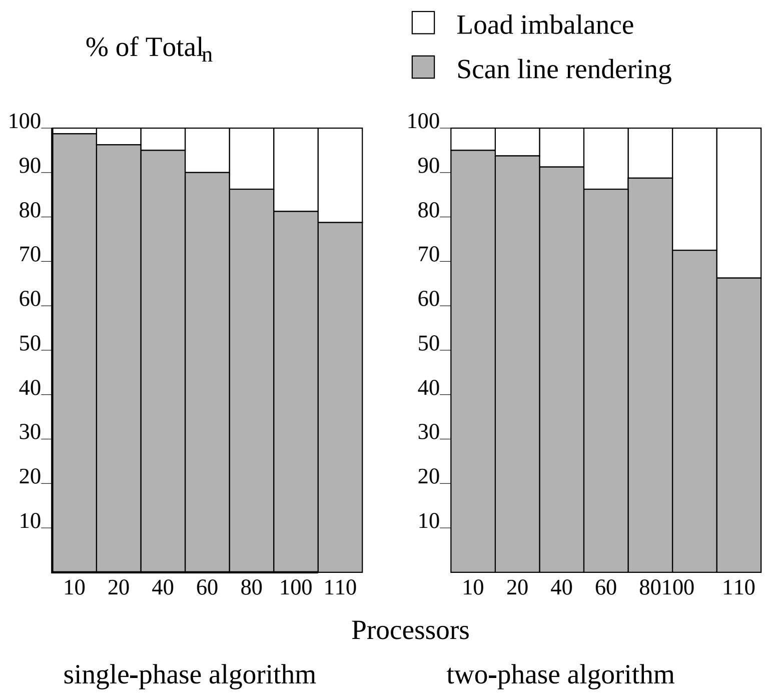

Figure 4.14: Execution profile for the two-phase algorithm.

4.4 Approach Chosen for Detailed Investigation

A very basic tradeoff between decomposition approaches has been identified. Image-space decompositions will be view dependent, but will not have any inter-processor synchronization requirements. Object-space decompositions can be view independent, but will introduce inter-processor synchronization requirements.

An important goal of this research is to design an algorithm that can be efficiently coupled with an executing simulation. For this reason it is important to consider how the application may be managing its memory (i.e. where does the application have the volume stored). It may be important to be able to render from the data structures maintained by the executing simulation, without requiring the movement of a lot of data. This will be especially important for unsteady (time-varying) calculations, or for Lagrangian approaches where the grid itself may be changing. For this reason, the assumption is made that the volumetric dataset is available to all processors via some form of globally-shared memory. No restrictions are placed upon the distribution of the volumetric dataset, it may be interleaved, scattered, or partitioned in any way that is beneficial to the executing simulation.

A review of the literature and the initial studies discussed above have suggested the direction taken in this research. Key decisions include the approach to memory management, task generation, and the basic algorithm. A review of the literature shows that image-space decompositions for raycasting [Cha92, NL92, CM92, Luc92] have achieved better scalability and levels of performance than object-space decompositions [MPS92]. Implementations of object-space decompositions for projection or splatting algorithms have been shown to be limited in terms of their scalability by synchronization requirements [Cha91, Wil92c, Elv92]. The parallel direct volume rendering algorithm described in the next chapter utilizes interleaved globally-shared memory to store the volumetric dataset. This approach facilitates the use of an image-space task decomposition with dynamic task generation, as well as providing the ability to render from an existing application's data structures. Image-space tasks utilizing square image tiles have been shown to perform better than scanline decompositions due to increased cache coherence [CM92]. Rectangular image tiles form the rendering tasks, improving both cache coherence and the possibilities for taking advantage of spatial coherence to speed the rendering process. Task size is easily varied by changing the size of the image tiles. The basic algorithm executed on each processor makes use of a scanline algorithm for generating each portion of the image.

The next chapter presents, in detail, the parallel scanline algorithm that has been developed and analyzed in the course of this research. Chapter 6 presents benchmark results and analysis of the performance of the algorithm on several test datasets. In particular, the analysis answers the following questions with regard to the parallel algorithm proposed here:

- Can we reach interactive or near-interactive speeds?
- What kind of speedup is attained?
- How scalable is the algorithm?
- What are the inefficiencies?
- Does the algorithm make effective use of local memory to reduce inefficiencies?

5. A Scanline Algorithm for Parallel Volume Rendering

Most parallel direct volume rendering algorithms previously presented for MIMD architectures have been for rectilinear datasets, and none has produced the level of scalability that is desired. The algorithm presented in this chapter focuses on the efficient (scalable) parallel implementation of a direct volume rendering algorithm for nonrectilinear datasets on highly parallel, multiple-instruction, multiple-data (MIMD) architectures with virtual shared memory. The design of the algorithm does not preclude its use on rectilinear datasets, however it has not been optimized to take advantage of the simpler geometrical structure inherent in rectilinear datasets. The algorithm as implemented utilizes orthographic parallel projections for viewing, although it could just as well be used to render perspective projections.

As the architectures and operating systems of massively parallel systems mature, virtual shared memory with non-uniform memory access times will almost certainly become a supported feature. It is likely that many applications will take advantage of this feature due to the increased programmer productivity it provides. Scalable algorithms designed for this type of execution environment will be portable while maintaining a high degree of efficiency.

5.1 System Overview

The earlier approach described in section 4.3.2 operated on the cells of the volume, rather than the faces, and utilized a task decomposition based on scanlines [Cha92]. The approach to be described here is faster and exhibits better scalability. The algorithm proceeds in three distinct phases. These include processing changes to the grid, processing changes to the view, and rendering the image. Not all phases need to be executed for each new image. For example, if the view has changed but the grid has not, then only the viewing sort and rendering phases are executed. If the transfer function is the only thing that has changed, then only the rendering phase is required. The basic algorithm represents the image as a set of image tiles. The current implementation uses square image tiles, however there is nothing inherent in the algorithm that requires square tiles. The image tiles form the basis of the task decomposition for the rendering phase in which processors dynamically acquire one image tile at a time for rendering. Figure 5.1 presents an overview of the main system functions which are summarized here and discussed in detail later in this chapter:

- **Grid Initialization:** Whenever a grid changes, the data structures that represent it must be updated. The grid data structures are presented in section 5.2, and grid initialization is discussed in section 5.3.
- **Parallel View Sort:** A parallel view sort creates for each tile a list of pointers to cell faces that project into that tile. This is done prior to rendering whenever the view has changed. The creation of the view sort reduces the complexity of determining ray-face intersections within a tile, and enhances the scalability of the rendering algorithm by limiting the number of shared data structures that need to be accessed in order to render any given tile. The parallel view sort is discussed in section 5.4.

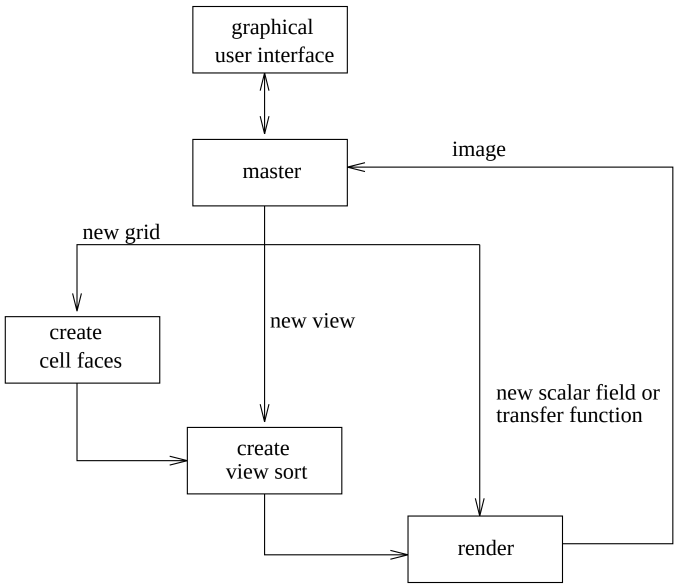

Figure 5.1: Overview of main functions.

- **Parallel Tile Rendering:** A scanline algorithm that takes advantage of spatial coherence is applied to the faces that project to a given tile. Within each tile-rendering task a local active list is incrementally maintained, resulting in the availability of a list of cell faces that are intersected by the ray through each pixel. The cell faces on this active list are processed, with spans across each cell face being incrementally maintained from pixel to pixel. Edges defining active faces are stored in local memory and are incrementally updated. These techniques are described in detail in section 5.5.

As depicted in figure 5.1, a master processor communicates with the user interface (described in section 1.3.1) via a socket connection to receive updates to grid, scalar, view, and transfer function specifications. The system has been designed in such a way (although this has not been implemented) that the master could also be communicating with an executing simulation. In the current implementation the grid only changes when the user specifies a new grid and solution file. If the system were coupled with an executing simulation which utilized adaptive grid methods, the grid could change during the course of the simulation. Currently, the user may select one of the 5 values (density, one of the three components of velocity, or energy) provided in the Plot3D solution file. If the system were coupled with an executing simulation of an unsteady flow, it would be possible for the scalar field to change at every time step. The user specifies the view and transfer functions. The master processor is also responsible for compressing and sending images back to the user interface. In addition to these special user interface tasks, the master participates in the parallel tasks for grid modifications, viewing changes, and rendering.

5.1.1 Object-Oriented Design and Implementation

The design is object-oriented and the code is written in C++ which supports the required object-oriented features fairly directly. The most important features used in this

```
class object_base {
private:
public:
    object_base(){}
    virtual void new_view(matrix, view_cmds){}
    virtual void new_tf(float*, color*){}
    virtual int get_xform_groups(){}
    virtual void view_xform(int){}
    virtual int get_sort_groups(){}
    virtual void view_sort(int){}
    virtual void view_init(){}
    virtual void make_buckets(){}
    virtual int allocate_buckets(int*){}
    virtual void free_buckets(){}
    virtual void zero_buckets(){}
    virtual void tile_init(int, int, int){}
    virtual void scanline_update(int, int, int){}
    virtual void pixel_update(int, int){}
    virtual void tile_free(){}
    virtual boolean intersect(ray*, intersect_list*){}
    virtual boolean shade(world*, intersection*, pix_contribution*){}
};
```

Figure 5.2: Base class definition for renderable objects.

implementation are the encapsulation of data and functions via class definitions, derived classes, and virtual functions. The ability to specify small functions "inline" was also considered important. Classes may be defined for all types of objects that may be rendered. These include curvilinear grids, unstructured grids, rectilinear grids, geometrical primitives, etc. Each of these classes provides virtual functions which maintain active lists if required (as the algorithm to be described here does), perform intersection testing and/or updating, and do the shading computations for resulting intersections. Instances of these classes may be generated and rendered together. Although this chapter focuses primarily on the class design and implementation for curvilinear grids, the design of the system is such that many types of geometric and volumetric objects can be defined and rendered by one or more rendering methods [Cha90].

Renderable Objects

The renderable objects are the different types of objects that can be specified, oriented, and positioned in the world description, and then rendered to an image. All renderable objects are derived from a base class, `object_base` (see figure 5.2). This mechanism provides several key capabilities. In particular, it allows the different renderable objects to be treated in a homogeneous fashion.

For example, intersection with a ray is performed on instances of the class `object_base`. The virtual functions for intersection provided with each renderable object derived from the class `object_base` ensure that appropriate intersection calculations are performed for the current instance of `object_base`. Once an intersection has been detected, the renderable objects can be kept on the intersection list without regard to the specific derived class being dealt with. A pointer is simply maintained to the base class. Then, when this intersection list is processed to determine the contribution to the current pixel of each intersection, virtual shading functions ensure that the appropriate shading calculations are carried out for each derived renderable object type.

Another desirable function for renderable objects is to be able to update them on a scanline and/or pixel basis. This functionality, which may speed the rendering of certain types of primitives considerably, is also provided via virtual functions. For example, at each new scanline the rendering function can ask each renderable object to update itself and the virtual function `scanline_update()` ensures that this is done appropriately for each derived object type.

Many different renderable objects, both geometric and volumetric, can be derived from this base class. For example, a single triangle could consist of the three points or vertices in $\Re^3$ which define it in clockwise order when viewed from the outside, along with other relevant data. Virtual methods would be provided to test for and generate an intersection with a given ray, or compute a shading contribution. A small triangle mesh could be represented as a collection of instances of the class for a single triangle. Data structures and methods supplied to support a scanline algorithm could reduce the number of intersection calculations that must be performed for each pixel. Triangle strips in which the shared vertices of individual triangles are represented once could also be supported. Private variables used to represent a rectilinear volume include the dimensions of the volume, the scalar and gradient data, color and opacity transfer functions, and other relevant information. Virtual functions provide methods to intersect the volume or compute the shading contribution along a ray.

Rendering Method

It is in the rendering method that the homogeneous treatment of renderable objects is best seen. Pseudocode describing the process is given in figure 5.3. Before rendering begins, each renderable object is instructed to initialize itself. As each new scanline in the image is begun, each renderable object updates any private representation. A similar call is made as each new pixel is begun. For each pixel of the image, a ray is generated through the pixel into the collection of objects. Each renderable object on the list tests itself for intersection with the ray and updates an intersection list. The intersection list is itself an object which maintains itself in sorted order. After all intersections have been tested for, each renderable object on the ordered intersection list computes its shading contribution to the pixel for the given intersection. Pixels are objects as well and know how to composite themselves. Less than a page of C++ code is required to implement this functionality for any collection of heterogeneous renderable primitives.

5.2 Grid Data Structures and Distribution

The algorithm presented here is designed and optimized to handle computational grids that are nonrectilinear. We will call a single data point that has been sampled or computed a *node*. Two neighboring nodes may be said to define an *edge*, and three or more define a *face*. In the case where a face is defined by more than three nodes, it is possible that the

```
/*
    Pseudocode for the parallel function to render one image tile.
*/

void render_tile(int tile)
{
    Compute pixel extents for this tile.

    For each renderable object o
        o->tile_init();

    For each scanline in this tile {
        For each renderable object o
            o->scanline_update();

        For each pixel in this scanline {
            Create an empty pixel.
            int n_active = 0;
            For each renderable object o
                n_active += o->pixel_update();

            if (n_active) {
                Allocate intersection list l for n_active intersections.
                For each defined renderable object o
                    o->intersect(l);

                For each intersection i on list, and while not opaque
                    opaque = i->shade();
            }

            If not opaque, composite over background.

            Store pixel in the image.
        }
    }
    For each defined renderable object o
        o->tile_free();
}
```

Figure 5.3: Homogeneous treatment of renderable objects.

face will be non-planar. A *cell* is the space in $\Re^3$ defined by four or more nodes, and four or more faces. A face that is shared by two cells is an *internal face*, otherwise it is an *external face*.

Grids may be curved to match the simulation geometry, as in the *curvilinear* grids commonly used in CFD. A curvilinear grid is defined by a rectilinear computational grid that has been shaped, resulting in cells with non-planar faces in physical space [Fle88]. These array-organized grids are also called *structured* grids [SK90]. Computational grids in $\Re^3$ that are not array-organized (sometimes called *unstructured* grids [SK90]) and are made up of tetrahedral or hexahedral cells that have been shaped are also common in computational fluid dynamics and finite element analysis applications [ZT89]. Typically the definition of these grids is given as a list of cells, defined by pointers into a list of nodes. Thus information on shared faces and neighboring cells is not inherent in the data structure. Information on which nodes define the faces of a cell is usually implicit for structured and unstructured grids. In the case of scattered data, a Delaunay triangulation could be used to give such a structure to the data.

The algorithm presented here has been designed to be general enough to handle structured or unstructured grids. The grids and cells may be non-convex, and grids may contain voids or holes. Grids may have been constructed from multiple grid definitions (multi-block grids). The multiple grid definitions may be spatially overlapping with intersecting faces. Information on which cells share faces is not required, which is useful in applying the algorithm to unstructured datasets. Embedded geometrical primitives can also be rendered by the algorithm. The remainder of this chapter describes a class definition for curvilinear volumes in detail.

5.2.1 Curvilinear Grid Data Structure

A class definition for curvilinear grids is derived from the base class for renderable objects (see figure 5.4). Multi-block grids are defined as several instances of `curv_volume`, one per block. The data structure records the grid index (for multi-block grids), grid dimensions, addresses of shared data structures, color and opacity transfer functions, transformation matrix, and data structures for the implementation of a scanline algorithm. The grid index, grid dimensions, and addresses of shared data structures are set up once when the grid is read in from a file. This information is propagated to all processors and stored locally. The spatial locations of the grid nodes and the solution components (data values) available at these nodes are read in from a file and stored in virtual shared memory. All available Plot3D solution components (density, velocity, and energy) are stored, and an additional pointer is maintained to the solution component the user desires to render. This allows the user to rapidly generate a new image using a different scalar field. Descriptions of cell faces defined by the grid are computed in parallel and stored in virtual shared memory. The addresses of these data structures are propagated to all processors. An array of buckets (one per image tile) are set up and initialized to be empty. The address of these buckets is also propagated to all processors. The buckets will be initialized by the parallel view sort prior to rendering whenever the view has been changed. Color and opacity transfer functions and the transformation matrix may change interactively according to user specifications. When this happens the affected data structures are updated and the changes propagated to all processors. The data structures associated with the implementation of the scanline algorithm are used locally and independently on each processor.

5.2.2 Node and Face Storage

Linear arrays of cachable interleaved shared memory are used to store the grid nodes and scalar values. Each grid node is identified by a unique index which may be used to obtain the spatial location and/or scalar values of the node. In addition, each face in the grid is represented by an instance of class `cell_face`. The `cell_face` class is derived from

```
class curv_volume: public object_base {
    friend class cell_face;
    friend class local_face;
private:
    int gn;             /* grid index */
    int xd, yd, zd;     /* grid dimensions */
    point* grid;        /* pointer to transformed grid points */
    point* orig_grid;   /* pointer to original grid points */
    int* iblnk;         /* pointer to iblank data */
    float* scalar_base; /* pointer to scalar data */
    float* scalar;      /* pointer to desired solution component */
    cell_face** faces;  /* pointer to groups of cell faces */
    color* clr_func;    /* pointer to color transfer function */
    float* opac_func;   /* pointer to opacity transfer function */
    matrix xform;       /* composite transformation matrix */
    bucket* bkts;       /* pointer to buckets for tiles */
    active_rec* y_sort; /* y-bucket sort of face records */
    int* y_index;       /* indices into y-bucket sort */
    int y_n_active;     /* number of active faces on scanline */
    int* y_active;      /* list of active faces on scanline */
    active_rec* x_sort; /* x-bucket sort of face records */
    int* x_index;       /* indices into x-bucket sort */
    int x_n_active;     /* number of active faces for this pixel */
    int* x_active;      /* list of active faces for this pixel */
    .
    .
    .
public:
    curv_volume();
    virtual void new_view(matrix, view_cmds);
    virtual void new_tf(float*, color*);
    virtual int get_xform_groups(){ return xd; }
    virtual void view_xform(int);
    virtual int get_sort_groups(){ return xd+yd+zd; }
    virtual void view_sort(int);
    virtual void view_init();
    virtual void make_buckets();
    virtual int allocate_buckets(int*);
    virtual void free_buckets();
    virtual void zero_buckets();
    virtual void tile_init(int, int, int);
    virtual void scanline_update(int, int, int);
    virtual void pixel_update(int, int);
    virtual void tile_free();
    virtual boolean intersect(ray*, intersect_list*);
    .
    .
    .
};
```

Figure 5.4: Class definition for a curvilinear volume.

```
class cell_face: public object_base {
    friend class local_face;
private:
    unsigned ef_cv_yminmax;  /* face flag, volume index, ymin, ymax */
    int v[4];                /* node indices */
public:
    cell_face(){}
    .
    .
    .
    virtual boolean shade(world*, intersection*, pix_contribution*);
};
```

Figure 5.5: Class definition for a cell face.

the base class for renderable objects. This allows intersections with cell faces to be placed on the intersection list for a pixel. For hexahedral cells (as in a curvilinear grid) four vertices or grid nodes define each face. The `cell_face` data structure contains an index into the grid node and scalar value arrays for each vertex. It also contains a word used to store the minimum and maximum scanline extents of the face, an index indicating which grid it belongs to (required for multi-block grid solutions), and a flag indicating whether the face is internal or external (required for handling non-convexity and voids in the grid). These components are packed into a single integer with the face flag in the upper 4 bits, the index in the next 4 bits, ymin taking the next 12 bits, and ymax in the lower 12 bits. The class definition for curvilinear cells is shown in figure 5.5.

The cell faces are divided up into sections called *face groups* by the dimensions of the grid. This division is motivated by the need to identify parallel tasks based on groups of faces. In this implementation for curvilinear grids, an $xd \times yd \times zd$ grid will have $xd + yd + zd$ face groups. These face groups may be different sizes, for example, there will be xd face groups containing $(yd-1) \times (zd-1)$ cell faces each, yd face groups containing $(xd-1) \times (zd-1)$ cell faces, and zd face groups containing $(xd-1) \times (yd-1)$ cell faces. An array of $xd + yd + zd$ pointers to cell faces is allocated. Each of the face groups is stored in cachable interleaved shared memory, the array containing the pointers to the face groups is propagated to each processor to be stored locally.

Curvilinear grids have a natural decomposition into this form since the grid is rectilinear in computational space. Unstructured grids such as those used in finite element analysis would need to be divided into groups of faces. This could be accomplished using a spatial decomposition of the original grid. The size of each group should be constructed so as to enhance load balancing of parallel functions which operate on one group at a time. Shared faces need only be represented once, and no information is required concerning which cells share a given face.

5.2.3 Bucket Storage

An array of instances of class `bucket` are allocated, one for each tile in the image (see figure 5.6). The array of buckets is allocated when the curvilinear volume is read in, or when the number of tiles forming the image is changed. Each bucket specifies the number

```
class bucket {
private:
    int n;            /* number of pointers currently in bucket */
    int size;         /* maximum size of bucket */
    cell_face** o;    /* pointer to array of cell_face pointers */
public:
    bucket(){}
    .
    .
    .
};
```

Figure 5.6: Class definition for a bucket.

of pointers in the bucket, the maximum size of the current bucket allocation, and a pointer to an array of cell face pointers. The buckets are allocated in interleaved shared memory, but are declared to be uncachable. This is because atomic increments to the counts will be made as different processors add pointers to the shared lists during the parallel view sort. The address of the array of buckets is propagated to all the processors.

5.2.4 Summary of Shared Data Structures

Table 5.1 summarizes the shared data structures. The grid nodes and scalar values are stored as floats. If there are N nodes in the grid $(N = xd \times yd \times zd)$, then there are slightly less than $3N$ cell faces. The transformed grid nodes and the cell face descriptors require approximately 18N words. The original grid specification and the solution require 8N words. Thus the extra memory needed to describe the volume to the rendering algorithm is 2.25 times the amount of space required to store the volume itself. Determining the amount of space that will be required for the buckets representing the view sort is complicated and will vary based on the characteristics of the grid and the viewing specification. There is one bucket per image tile and it is filled with a variable number of pointers to cell faces. In the simplistic case that the image is made up of T tiles with each cell face projecting to only one tile, then the combined buckets will contain approximately $3N$ pointers. Chapter 6 gives measured bucket sizes for several views of several different sized datasets.

5.3 Grid Initialization

When a new grid is specified by the user, the grid and solution files are read into shared memory. In the current implementation the file format supported is the Plot3D format generated by CFD applications at NASA Ames Research Center [plo89]. Initialization of the cell faces is done in parallel. The task decomposition is by face group and these tasks are dynamically generated. This phase generates $xd + yd + zd$ tasks during which indices into the grid and scalar arrays are computed and stored for each vertex of each cell face in the group.

Data Structure	Size of One Element	Number of Elements
Grid Nodes	3 floats	$xd \times yd \times zd$
Transformed Nodes	3 floats	$xd \times yd \times zd$
Scalars	5 floats	$xd \times yd \times zd$
Cell Faces	5 ints	$xd \times (yd-1) \times (zd-1) +$ $yd \times (xd-1) \times (zd-1) +$ $zd \times (xd-1) \times (yd-1)$
Empty Buckets	2 ints + 1 pointer	T

Table 5.1: Summary of shared data structures given a grid with dimensions xd by yd by zd and an image broken into T tiles.

5.4 Parallel View Sort

The objective of the view sort is to create a list of pointers to pertinent cell faces for each image tile. A second important function of the view sort is to eliminate from further consideration any cell face whose view-space bounding box falls entirely between two pixels, or entirely out of the image. The parallel view sort consists of four phases. These are the viewing transformation, determination of required bucket sizes and local compilation of cell face pointers, allocation of shared memory for buckets, and bucket initialization. All of these phases are parallel with the exception of the allocation of shared memory which is sequential for reasons discussed below.

5.4.1 Viewing Transformation

In the viewing transformation phase, the grid nodes are multiplied by a matrix representing the viewing transformation. Each instance of `curv_volume` maintains two separate arrays in shared memory for the grid nodes. The first contains the original, untransformed grid nodes. In an environment in which the rendering code was running simultaneously with an executing simulation, this data structure would actually belong to the simulation. The grid nodes from this array are multiplied by the transformation matrix and placed in the second array for use by the rendering functions. This is done in parallel with each of xd face groups constituting a task. For multi-block grids with n grid definitions, the number of tasks will be $\sum_{i=1}^{n} xd_i$. These tasks will be dynamically generated and the appropriate class instance will execute the viewing transformation.

5.4.2 Bucket Sorting

Due to the inefficiency of memory management functions for shared memory, sorting of the cell faces into buckets has been split into sorting and initialization phases with sequential allocation of the shared memory for the buckets in between. In particular, allocate and free commands are very slow for shared memory, and there is no reallocate function. In order to minimize use of these time-consuming functions, new memory for a bucket is allocated only if the new size required is greater than the existing size of the bucket. The first time an image is rendered every bucket will need to be allocated.

The sorting phase involves computing the view-space bounding box for each cell face, counting the number of cell faces in each bucket, and compiling local lists of cell face pointers for each bucket. The task decomposition is by face group generating $xd + yd + zd$ tasks using dynamic task generation. For multi-block grids with n grid definitions, the number of tasks will be $\sum_{i=1}^{n} (xd + yd + zd)_i$. In the first task executed by any given processor, a local array of integers called `counts` and a local array of `cell_face` pointers called `lists` are allocated, one per image tile. For each face group processed by that processor, bucket counts and cell face pointers are compiled locally. Each `cell_face` in the group computes its bounding box, increments the count for each image tile that the bounding box projects to, and stores a pointer to itself in the lists for those tiles. In addition, the minimum and maximum y extents of the face are stored in the `cell_face` for later use during rendering. The case in which a cell face bounding box projects to a tile, but the cell face itself does not, is handled during tile rendering as described in section 5.5.3. Once every `cell_face` in a given group has been processed, any non-zero local counts that were generated during the current task are atomically added to the shared `bucket` counts. When all the parallel tasks of the sorting phase are finished, the shared buckets each contain a total count of the number of pointers that will need to be stored in any given bucket, and each processor has stored in its local memory some portion of the pointers to cell faces for each tile's bucket.

5.4.3 Bucket Memory Allocation

A sequential portion of the code allocates the necessary amount of cachable interleaved shared memory for each bucket that is not currently large enough, and stores the pointer to it in the bucket. The shared bucket counts are then set to zero so they may be atomically incremented by processors doing bucket initialization in the next phase.

5.4.4 Bucket Initialization and Task Ordering

The bucket initialization phase of the view sort initializes the list of pointers in each shared bucket. A single task is statically generated on each processor. For each tile with a non-zero local count, the shared bucket count is atomically incremented and the list of pointers accumulated locally for that bucket is copied to the shared list. When all of the shared buckets have been updated, the local counts and lists are deleted.

A simple expedient for better load balancing using dynamic task generation is to order the tasks such that the largest tasks are handed out first. In this case, task size is defined to be the number of cell faces contained in a bucket. During the (sequential) memory allocation phase for buckets, each renderable object is instructed to allocate the necessary bucket space and the required space is summed for each bucket. This results in an array with an entry for each image tile in which each entry represents the total size of the task for all renderable objects. Associating a tile number with each entry and sorting the array based on the task size gives an ordering for task generation which will be shown in chapter 6 to improve load balancing. This ordering is stored in shared memory where it will be accessed during the rendering phase by processors to determine the correct tile to be rendered.

The sorting of the tasks is done in parallel with the initialization of the buckets. During the bucket initialization phase, the master processor sorts the task array using a heapsort ($O(n \log_2 n)$) before performing the bucket initialization.

5.5 Parallel Tile Rendering

In this section the rendering phase of the algorithm is presented. This is the most time-consuming phase and stands to benefit the most from an efficient parallelization. The design objectives have been an efficient sequential algorithm that is scalable when parallelized. Past experience has shown that load imbalance, remote memory latency, and switch contention have been the primary inhibitors of scalability. Load imbalance is addressed via an adjustable tile size and task ordering. Remote memory latency and switch contention are addressed by reducing the required number of shared-memory accesses through the use of the parallel view sort, local storage of face and edge information, and incremental updates. Pseudocode for the rendering phase is given in figure 5.3.

5.5.1 Task Decomposition

In the rendering phase, task decomposition is by image tiles. Since each tile contains a (possibly greatly) varying number of cell faces to be rendered, dynamic task generation is essential for good load balancing. As described in section 5.4.4, the tasks are sorted by size and the largest tasks are allocated first. When a processor obtains task i to be executed it uses this task number as an index into the shared order array to find the ith largest task. Thus the tile that will be rendered will be `tile = order(i)`. From this, and the knowledge of the tile and image size, the minimum and maximum pixel extents of the tile are computed.

5.5.2 Tile Rendering Algorithm

Within each tile, the approach taken to reduce the number of intersection calculations required at each pixel utilizes the idea of a bucket sort and scanline algorithm from computer graphics [FD82, HB86]. The algorithm presented here uses a y-bucket sort followed by an x-bucket sort to create a list of active cell faces for each ray. This approach differs from a traditional computer graphics approach in that the cell faces, rather than the edges, are used to create and maintain the active list. The high degree of edge sharing among faces makes it difficult to know which face is being interpolated when an edge becomes active. An approach based on the cell faces eliminates this ambiguity. Knowing which face is being interpolated allows us to determine which two edges should be used for the interpolation, as well as whether the face is external, internal, or part of a solid wall (simulation geometry).

Tile Initialization

Given an image tile to be rendered and the bucket from the view sort, the first step is to initialize the records that will be used to create and incrementally maintain the y-bucket sort of the cell faces. This functionality is provided in the virtual function `tile_init()`.

A record is created for every `cell_face` specified in the bucket for this tile from the view sort and stored in `y_list` (see figure 5.4). These records contain the minimum scanline of the cell face, number of active scanlines, and a pointer to the instance of `cell_face` that is being represented. The records are grouped by the minimum scanline of the cell face to form the buckets of the y-bucket sort. Once these records have been initialized, a loop is entered over the scanlines of the tile.

Processing at each Scanline

At each new scanline, the `y_active` list is updated using the y-bucket sort in `y_list` to incrementally maintain a list of cell faces active on the current scanline. The first step is to decrement the scanline counts of any cell faces that are currently active. If the count becomes zero, the cell face is deactivated and any local storage associated with the cell face is released. Next, any cell faces that become active on this scanline are added to the `y_active` list. As new cell faces become active, local storage for edge and span records is allocated and initialized as described in section 5.5.3. The intersections of the scanline with edges of each cell face active on the scanline are incrementally computed and stored. Each cell face is represented as two triangles for the purposes of interpolation because the faces of the curved hexahedron are not necessarily planar. It would also be possible to estimate the closest planar polygon to represent the cell face, however, representation of each cell face as two triangles has the advantage that interpolation is then rotationally invariant. Each cell face is thus represented by the four edges that define it plus one arbitrarily, but consistently, chosen diagonal. Each edge that is currently stored and active is incrementally updated to generate the required values at the intersection of the edge and the current scanline. The x, z, and scalar value at the edge intersection point are obtained this way and stored in the edge record. The final step before processing at each pixel can begin is to set up the x-bucket sort based on cell faces in the `y_active` list (those that are active somewhere on this scanline). This process is identical to that described for the y-bucket sort. The virtual function `scanline_update()` provides for all of these requirements.

Processing at each Pixel

Similarly to the processing of the `y_active` list prior to each new scanline, at each new pixel the `x_active` list must be updated using the x-bucket sort in `x_list` to incrementally maintain a list of cell faces active at the current pixel. This is accomplished in the virtual function `pixel_update()`. The first step is to decrement the pixel counts of any cell faces that are currently active. Cell faces that are no longer active are removed from consideration for the remainder of the scanline. Next, any cell faces that become active on this pixel are added to the `x_active` list. The x-bucket for the current pixel is sorted into descending order by the minimum z value of the face on this scanline using Shell's method ($O(n^{3/2})$) [PFTV86]. The x-bucket is then merged with the current active list. This process maintains the active list in nearly sorted order and greatly reduces the time required to insert intersections on the depth-sorted intersection list when it is generated. The intersection list is a linked list that is maintained in order by ascending z values so that compositing may be done front-to-back. Having the active list nearly ordered by descending

```
class intersection {
private:
    float depth;        /* intersection depth along ray */
    float scalar;       /* interpolated scalar value */
    unsigned char ef;   /* external face flag */
    object_base* obj;   /* pointer to object generating intersection */
    intersection* next; /* next intersection on list */
public:
    intersection();
    .
    .
    .
};


class intersect_list {
private:
    intersection* head;    /* first intersection on list */
    intersection* list;    /* preallocated space for list */
    int size;              /* size of preallocated list */
    int n;                 /* current number of intersections on list */
    struct more {          /* pointer to extra space if needed */
        intersection* i;
        more* m;
    } *m;
public:
    intersect_list();
    .
    .
    .
};
```

Figure 5.7: Class definitions for intersections and intersection lists.

z values means that most of the time an intersection is added to the intersection list, it will be added to the head of the list.

The `x_active` list now contains pointers to all of the cell faces that are intersected by the ray through this pixel. For each `cell_face` on the `x_active` list the virtual function `intersection()` is called to compute the intersection by incremental update of the spans for that cell face, and add it to a depth-sorted intersection list. Since each cell face is represented by two triangles, there are two spans possible (and thus two intersections possible) for each active cell face. The entry on the intersection list records the distance along the ray of the intersection, the scalar value of the field at that point, a flag indicating whether the face containing the intersection is internal, external, or a solid wall, and a pointer to the `cell_face` containing the intersection. Table 5.7 gives the class definitions for intersections and intersection lists. Each renderable object provides an estimate of how many intersections it may generate on any given pixel. For the algorithm described here, this estimate is the number of cell faces on the `x_active` list. Space for the entire list is preallocated using this estimate, greatly reducing the overhead associated with memory management. A method is provided to allocate extra space if the estimate falls short of the actual requirement.

Creation of an intersection list, rather than performing the shading and compositing on the fly as the `x_active` list is processed serves several purposes. It ensures that the intersections generated by entries on the `x_active` list are in depth-sorted order (they may not be exactly in order on the `x_active` list). It allows for the combination of several different types of renderable objects, such as embedded geometrical primitives, and provides a straightforward way of supporting multi-block grids. Finally, the intersection lists may be stored in local memory and used to rapidly update an image for a changing transfer function as will be described in section 5.5.4.

After all cell faces (and all other renderable objects) have been processed, the intersection list is traversed to compute the color and opacity for the pixel. For each intersection on the depth-sorted intersection list (or until the pixel is opaque), the virtual function `shade()` is called to calculate the shading contribution and composite the computed color and opacity into the current pixel. The shading function detects voids in the volume or executes a different shading technique for solid walls by using the flag stored in each intersection in the intersection list to detect these cases. For shading internal portions of the volume, the shading calculation uses the scalar value for the current intersection, the scalar value for the next intersection on the list, and the distance between them. Let t_0 be the distance along the ray to the point of intersection with the current cell face on the intersection list, and $\hat{\rho}_0$ be the normalized optical density at that point (as determined from the transfer function for opacity which has the range $(0, 1)$, and is indexed by the scalar value at the point t_0). Similarly, let t_1 be the distance along the ray to the point of intersection with the next cell face on the intersection list, and $\hat{\rho}_1$ be the associated optical density. Then the optical density in the range $(0, \infty)$ is given by

$$\rho_0 = \log \frac{1}{1 - \hat{\rho}_0}$$

and similarly for ρ_1. The optical density ρ for the cell is determined by averaging the two as

$$\rho = \frac{\rho_0 + \rho_1}{2}$$

The chromaticity components are averaged in the same way, and the opacity α is given by

$$\alpha = 1 - \exp^{-\rho(t_1 - t_0)}$$

The optical density is assumed to be independent of wavelength, thus one exponential is evaluated for each cell intersected by the ray. The use of this approximation to the integral can result in image artifacts in the case where discontinuities in the transfer functions occur inside a cell. These can be minimized by the avoidance of step functions in the transfer function. The volume density optical model proposed by Williams and Max [WM92] solves this problem by providing an exact solution to the integral inside a cell for piecewise linear transfer functions. Finally, the opacity is used to composite the contribution to the pixel using the standard method described by Porter and Duff [PD84]. After all intersections have been processed, or the pixel has become opaque, the pixel is stored in the image.

```
class local_face: public object_base {
private:
    cell_face* cfp;       /* pointer to remote cell_face */
    struct edge {         /* edge records */
        short miny, maxy;
        short count;
        float x, delta_x;
        float z, delta_z;
        float s, delta_s;
    } edges[5];
    struct span {         /* span records */
        short minx, maxx;
        short count;
        float z, delta_z;
        float s, delta_s;
    } spans[2];
    float minz;          /* minimum z value of face */
    short xvmin, xvmax;  /* min and max x of face for this scanline */
public:
    local_face(){}
    .
    .
    .
    virtual boolean intersect(ray*, intersect_list*);
};
```

Figure 5.8: Class definition for a locally stored face.

5.5.3 Coherence and Local Face Storage for Enhanced Scalability

Scalability can be enhanced by storing repeatedly accessed shared variables in local memory. Experience has shown that care must be taken in doing this, however. A brute-force approach would be to store all of the cell faces for a given tile in local memory before beginning to render. Although this reduces the remote memory references, it can require too much local memory resulting in paging at selected processing nodes. This eliminates any benefit gained from the local storage of cell faces by increasing the processing time at those nodes and upsetting the load balancing. The approach taken here involves locally storing only those cell faces that are active on any given scanline. A pointer to the shared `cell_face` is stored with the local variables providing access to the indices used to access node and scalar values, the grid index, and the external face flag. By locally storing edge records and making use of spatial coherence to do incremental updates of the edges defining a face, it becomes possible to effectively cache the node and scalar values of the cell face.

During the execution of `scanline_update()`, when a `cell_face` becomes active, a local data structure called a `local_face` is allocated. This data structure stores a pointer to the shared `cell_face`, as well as edge records for the edges defining the face (see figure 5.8). The edges are initialized at the first scanline that the face becomes active. Spatial coherence is used to update the edge intersections with the current scanline incrementally. This utilization of spatial coherence not only improves the efficiency of the algorithm, but it has the added advantage of effectively causing the grid node and scalar values that define the edges to be locally stored. For each `cell_face` projecting to a given tile, the node indices stored in the shared `cell_face` and the grid node and scalar values are accessed once in order to initialize the edge records stored in the `local_face`. For each edge several variables are computed and stored. These include the minimum and maximum y values for which the edge is active, a counter, and the x, z, and scalar value at the intersection with the scanline and their increments. As the edge intersections are updated at each scanline, span records are initialized for the scanline, and the minimum and maximum x values for the face are stored and used when creating the x-bucket sort. The minimum z value of the face on the current scanline is stored and used when inserting the cell face into the `x_active` list. The span records are updated at each pixel and used to generate intersections. In the case where the bounding box of the cell face projects to the current tile, but the cell face itself does not, none of the edges will ever become active, the spans will not be initialized, and the cell face will not generate any intersections. The `local_face` data structure will be deleted in `scanline_update()` when the scanline is reached at which the face being represented is no longer active, or after all scanlines of the tile have been rendered.

Each face is defined by five edges in order to represent it as two triangles. However, all but one of these edges are shared with one or more other faces. In the current implementation, all five edges are stored with each cell face. However, local storage requirements could be further reduced by storing shared edge data structures once. Two approaches to doing this have been explored, in both cases the time to utilize a hashing function for shared edge storage was longer than that required to simply update multiple copies of the edges. If each edge may be stored more than once (when it is shared by two or more faces), care must be taken to process multiply defined edges identically in order to avoid image artifacts. In particular, when doing incremental updates of intersection points along edges, the increments should begin from the same node and proceed to the opposite node identically for all replicated copies.

In one approach to shared edge storage, each `curv_volume` maintains an `edge_list` which is a two-dimensional array of pointers to lists of edge records. There is one list for each pixel in the tile. When a face becomes active, it computes the minimum x and y values in the tile at which each edge becomes active and submits the edge to the `edge_list` for storage. The `edge_list` uses the x and y values to determine which list may contain the edge. Since each grid node is uniquely identified by an index into a linear array, each edge is uniquely identified by a pair of these indices. If the edge is already stored, a use count is incremented and a pointer to the edge is returned to the face. If the edge is not yet stored it is inserted into the list with a use count of 1. When a face is deactivated, it decrements the use counts for all of its edges. Each scanline when the edges are updated, edges with use counts of zero are removed and the memory released. In this approach the hash table size is dependent on the tile size (because there is an entry for each pixel in the tile) and is constant for all tiles, resulting in too many collisions to make it effective. In another approach to edge storage, each tile dynamically creates a hash table whose size is dependent on the number of cell faces in the bucket for that tile. The hash code is taken to be the sum of the vertex indices for the edge, modulo the hash table size. This approach was much faster than the first approach tried, but was still less effective than simply maintaining multiple copies of edges where required.

```
class sample_container {
private:
    int which_tile;                /* tile these samples belong to */
    int n_samples;                 /* number of pixels in tile */
    intersect_list** samples;      /* pointer to samples */
    sample_container* next;        /* pointer to next sample container */
public:
    sample_container();
    .
    .
    .
};
```

Figure 5.9: Class definition for a sample container.

5.5.4 Local Storage of Intersection Lists

A second variation of the algorithm has been implemented in which the intersection lists are explicitly stored in local memory and retained between rendering phases. A similar approach has previously been utilized to speed computation of successive ray-traced images with changing lighting conditions and surface properties [SS89a]. The basic rendering algorithm proceeds in the same way, but sampling (intersection generation) and compositing have been split into two phases. The motivation for doing this is to attain fast image updates for a changing transfer function. Finding a transfer function which brings out features of interest in the data can be a time-consuming process, and this approach helps to alleviate that.

The algorithm begins by dynamically allocating tasks for the sampling phase, if required. The sampling phase is only necessary if the grid or viewing transformation has changed. Task decomposition is by image tile, and these tasks are dynamically allocated and ordered by size as described above. Each processor begins each task by allocating a `sample_container` (see figure 5.9) for the current tile and placing it on a linked list of sample containers stored on this processor. The processor then executes just the sampling portion of the algorithm for the tile, starting by setting up a y-bucket sort based on the cell faces in the bucket for the tile. For every scanline in the tile, the `y_active` list is updated, local storage for newly active cell faces is allocated, edge intersections of cell faces active on this scanline are computed, and the x-bucket sort is set up. For each pixel in a scanline, the `x_active` list is updated, the intersection list for the pixel is generated and placed in the sample container, and for each cell face on the `x_active` list, intersections are computed by incremental update of the span records and added to the depth-sorted intersection list for this pixel.

The compositing phase will be executed following a sampling phase, or alone if just the transfer function has changed. Static task generation is used (one task per processor), with each processor computing pixels from the intersection lists in the stored sample containers for the tiles assigned to it during the sampling phase. For each pixel of each tile resident on a given processor the intersections on the sorted intersection list for that pixel are processed and the shading contribution is calculated and composited. When all intersections have been processed, or the pixel has become opaque, the pixel is stored in the image.

6. Benchmark Results

In this chapter the performance of the parallel direct volume rendering algorithm presented in chapter 5 is analyzed. The various measures used in the analysis are discussed in section 2.2.2 and summarized in table 2.1. The particular datasets and viewing configurations are presented in the next section, followed by a summary of the best rendering times achieved, characterizations of each image, and a summary of data structure memory requirements. Many variations of dataset size, image size, number of image tiles, viewing specifications, and number of processors have been tested. Each configuration has been run three times to verify the consistency of the measurements. The statistics presented here are compiled from over 1000 runs. Results for the rendering phase are discussed in section 6.2, with attention given to the effects of image and tile size and any sources of inefficiencies. The effectiveness of storing intersection lists in local memory for fast image updates with a changing transfer function is examined in section 6.2.6. Finally, results for the parallel view sort are discussed in section 6.3.

6.1 Description of Datasets and Images

The parallel direct volume rendering algorithm described in chapter 5 has been benchmarked on several volumes and for multiple views of these volumes. Four curvilinear data sets obtained from NASA Ames Research Center were used in this study. The first is the *blunt fin* dataset [HB85]. This dataset represents a CFD simulation of air flow past a blunt fin on a grid resolution of $40 \times 32 \times 32$, or 40960 nodes defining 37479 cells. Images of this dataset are presented in figures 6.1 and 6.2. The second is the *post* dataset [RKK86], which was obtained from a numerical study of three-dimensional incompressible flow around multiple posts and has a grid resolution of $38 \times 76 \times 38$ giving 109744 nodes defining 102675 cells. Three views of this dataset have been benchmarked (figures 6.3, 6.4, and 6.5). The third view of the post dataset demonstrates rendering with a nonconvex grid containing a void (figure 6.5). The third dataset is the *delta wing* dataset, taken from a study of vortical flow over a delta wing [FGH87]. It contains $91 \times 51 \times 51$ grid nodes, or 236691 nodes defining 225000 cells (figures 6.6 and 6.7). The fourth dataset is the *shuttle* dataset [MS90], a multi-block grid consisting of 9 grids with a total of 941159 nodes defining 885898 cells. This dataset represents flow computations of the space shuttle ascent aerodynamics. Four views of the shuttle dataset have been benchmarked (figures 6.8, 6.9, 6.10, and 6.11). The solid walls in the grid which represent the shuttle geometry have been rendered in opaque white. The 9 grids represent a hemisphere surrounding half of the shuttle orbiter, external tank, and solid rocket booster, plus attachment hardware. Three of the images rendered view the shuttle and its flow field from the side of the hemisphere at which the flow field does not obstruct most of the shuttle geometry (figures 6.8, 6.9, and 6.11). In figure 6.10, the view is such that the shuttle geometry is embedded in the flow field. The transfer functions used to create the images are shown in figure 6.12. An overview of the rendering times for all images and several datset and image characteristics are discussed in the next section, followed by an in-depth analysis for each image.

Figure 6.1: View 1 of the blunt fin dataset.

Figure 6.2: View 2 of the blunt fin dataset.

Figure 6.3: View 1 of the post dataset.

Figure 6.4: View 2 of the post dataset.

Figure 6.5: View 3 of the post dataset.

Figure 6.6: View 1 of the delta wing dataset.

Figure 6.7: View 2 of the delta wing dataset.

Figure 6.8: View 1 of the shuttle dataset.

Figure 6.9: View 2 of the shuttle dataset.

Figure 6.10: View 3 of the shuttle dataset.

Figure 6.11: View 4 of the shuttle dataset.

Figure 6.12: Transfer functions used to create images: upper left for the blunt fin, upper right for the post, lower left for the delta wing, and lower right for the shuttle.

6.1.1 Image Characteristics and Dataset Sizes

Dataset & View	View Sort	256^2 Image	512^2 Image
Blunt fin - view 1	0.46	2.5	7.9
Blunt fin - view 2	0.49	2.5	8.8
Post - view 1	0.91	3.3	9.7
Post - view 2	0.93	4.0	11.9
Post - view 3	0.86	4.9	14.7
Delta wing - view 1	1.19	3.7	10.3
Delta wing - view 2	1.37	4.4	14.6
Shuttle - view 1	2.44	7.1	20.4
Shuttle - view 2	2.95	11.4	29.1
Shuttle - view 3	2.71	10.5	32.0
Shuttle - view 4	2.70	9.9	26.4

Table 6.1: Best rendering times in seconds (T_{110}) using 110 processors.

Table 6.1 summarizes the best rendering times (T_{110}) for the images using 110 processors. Execution times are given for the parallel view sort and for the rendering phases for both 256^2 and 512^2 images. The execution times for the parallel view sort are the sum of the best execution times for the node transformation, bucket sort, and bucket initialization phases using 110 processors. Faster execution times for the parallel view sort were obtained using fewer processors for the smaller datasets. In addition, obtaining consistent results for the parallel view sort proved difficult. These points are discussed further in section 6.3.

Table 6.2 presents some measures which characterize the complexity of each image. The total number of faces in each dataset is given. The number of faces culled by the parallel view sort and the number of faces remaining (those that appear in the image) are given for each image. The percentage of empty pixels and average depth complexity are also given. The average depth complexity is obtained by summing all of the cell face intersections in the image and dividing by the number of non-empty pixels. These measures show that the parallel view sort, as well as providing each task with a list of pertinent cell faces, significantly reduces the number of cell faces that must be dealt with by culling the ones that do not project to the image. For all the images tested, approximately 40% to 90% of the cell faces are culled. The larger and more varied the dataset, the higher the percentage of culled cell faces for any given image.

Table 6.3 presents measures which characterize the task decomposition. For each image the largest task size is given (measured as the largest number of cell faces to project to any one tile), as well as the number of pointers to cell faces stored in the buckets produced by the parallel view sort. The ratio of cell face pointers stored in buckets to the number of cell faces appearing in the image (those not culled) gives an estimate of coherence lost due to the division of the image into tiles. A value of 1.0 for this ratio would indicate that each cell face projects to only one tile and no coherence would be lost. However, a value of

Blunt Fin Dataset:	faces	faces	% empty	average
115816 faces	culled	in image	pixels	depth
View 1	41640	74176	22.8	35.6
View 2	83464	32352	2.7	33.8

Post Dataset:	faces	faces	% empty	average
314944 faces	culled	in image	pixels	depth
View 1	194792	120152	21.7	38.0
View 2	140920	174024	2.8	38.0
View 3	148404	166540	1.1	49.1

Delta Wing Dataset:	faces	faces	% empty	average
686500 faces	culled	in image	pixels	depth
View 1	411223	275277	17.3	37.0
View 2	538216	148284	9.0	52.8

Shuttle Dataset:	faces	faces	% empty	average
2711792 faces	culled	in image	pixels	depth
View 1	2388412	323380	0	39.2
View 2	1517138	1194654	0	66.2
View 3	2021838	689954	0	89.1
View 4	2023540	688252	0	96.2

Table 6.2: Summary of image complexity statistics showing the total number of faces in each dataset, the number of faces culled during the view sort, the number of faces appearing in each image, the percentage of empty pixels in each image, and the average depth complexity defined as the sum of the cell face intersections divided by the number of non-empty pixels.

2.0 would not necessarily mean that all of the cell faces project to two tiles, but only that some of the faces project to more than one tile. Thus the ratio is only an estimate of lost coherence. A more precise measurement of lost coherence is given in the detailed analyses which follow this section.

Table 6.4 summarizes the data structure sizes for each dataset. Storing the indices for each vertex of each cell face is the most memory-intensive requirement. However, the ready availability of these cell face definitions provides the basis for an efficient algorithm. In addition, most highly parallel systems have a lot of memory (the BBN TC2000 has 2 gigabytes), making memory issues slightly less significant than when designing codes for a workstation environment where memory may be extremely limited. The most important issue for these types of systems is how effectively the local memory on each processing node is utilized.

Blunt Fin Dataset:	256 tiles			1024 tiles		
	largest task	faces in buckets	bucket ratio	largest task	faces in buckets	bucket ratio
View 1	9253	125683	1.70	4310	198212	2.67
View 2	2695	66766	2.10	1368	116812	3.60
Post Dataset:	**256 tiles**			**1024 tiles**		
	largest task	faces in buckets	bucket ratio	largest task	faces in buckets	bucket ratio
View 1	12104	199198	1.66	5234	310783	2.59
View 2	11210	262275	1.51	5847	395548	2.27
View 3	4492	278610	1.67	1852	435143	2.61
Delta Wing Dataset:	**256 tiles**			**1024 tiles**		
	largest task	faces in buckets	bucket ratio	largest task	faces in buckets	bucket ratio
View 1	14903	359930	1.31	6187	467204	1.70
View 2	9111	222409	1.50	4318	320754	2.16
Shuttle Dataset:	**256 tiles**			**1024 tiles**		
	largest task	faces in buckets	bucket ratio	largest task	faces in buckets	bucket ratio
View 1	40468	417661	1.29	13472	543956	1.68
View 2	62687	1418622	1.19	25014	1711213	1.43
View 3	79360	858428	1.24	23678	1100295	1.59
View 4	39553	883899	1.28	15084	1125977	1.64

Table 6.3: Summary of the largest task (defined as the largest number of cell faces to project into a tile), total number of cell face pointers stored in buckets, and the ratio of the number of cell face pointers stored in buckets to the total number of cell faces (those not culled by the view sort) in the image.

Data Set	Grid Nodes	Scalars	Cell Faces
Blunt Fin	0.47	0.78	2.65
Post	1.26	2.1	7.2
Delta Wing	2.7	4.5	15.7
Shuttle	10.77	17.95	62.1

Table 6.4: Summary of shared data structure sizes in megabytes.

6.2 Timing Results for the Tile Rendering Phase

In this section timing results for the rendering phase are discussed. For each view of each dataset, measurements were collected for image sizes of 256^2 and 512^2, tile sizes of 16^2 and 8^2 for the 256^2 images (generating 256 and 1024 tasks respectively) and tile sizes of 32^2 and 16^2 for the 512^2 images (again generating 256 and 1024 tasks respectively). These rendering configurations were run on 10, 20, 40, 60, 80, 100, and 110 processors in order to study the scalability of the approach. Measurements of sequential rendering times were obtained for the blunt fin, post, and delta wing datasets; the shuttle dataset is too large to render using one processing node without excessive paging invalidating the measurement.

6.2.1 Blunt Fin Dataset

Configuration for 256^2 image	View 1	View 2
Local memory, 1 tile	206.8	223.4
Shared memory, 1 tile	208.4	223.6
Shared memory, 256 tiles	228.2	236.6
Shared memory, 1024 tiles	249.2	250.1

Table 6.5: Sequential execution times (T_1) for the blunt fin dataset in seconds using various configurations in order to show the effect of the use of shared memory and multiple image tiles. The times given are for a 256^2 image.

Breaking the image into tiles for rendering by different processors introduces an inefficiency due to the loss of coherence. Table 6.5 gives sequential execution times (T_1) for both views of the blunt fin dataset, for various tile sizes. The first row lists the time to execute on one processor using one tile and only local memory. The second row lists the time to execute on one processor using one tile and shared memory for the data structures that are normally shared during parallel execution. The third and fourth rows give execution times on one processor using shared memory with the image broken up into 256 and 1024 tiles respectively. It can be seen that the move from local to shared memory does not significantly affect the time required to render an image. This is due to the efficient local storage of active cell faces as described in chapter 5. However, breaking the image up into tiles does have a significant impact on the sequential rendering time. For view 1 of the blunt fin dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 10.3% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 20.5% increase in the sequential rendering time. For view 2 of the blunt fin dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 5.9% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 12% increase in the sequential rendering time. View 1 shows most of the dataset, and view 2 is a closeup of the edge of the blunt fin. It would seem intuitive that the image having more coherence (view 2) would display a larger increase in rendering time due to loss of coherence. The fact that view 1 contains roughly twice as many faces as view 2 (see table 6.2), coupled with the fairly small amount of coherence in either image (see table 6.3), explains the counter-intuitive results.

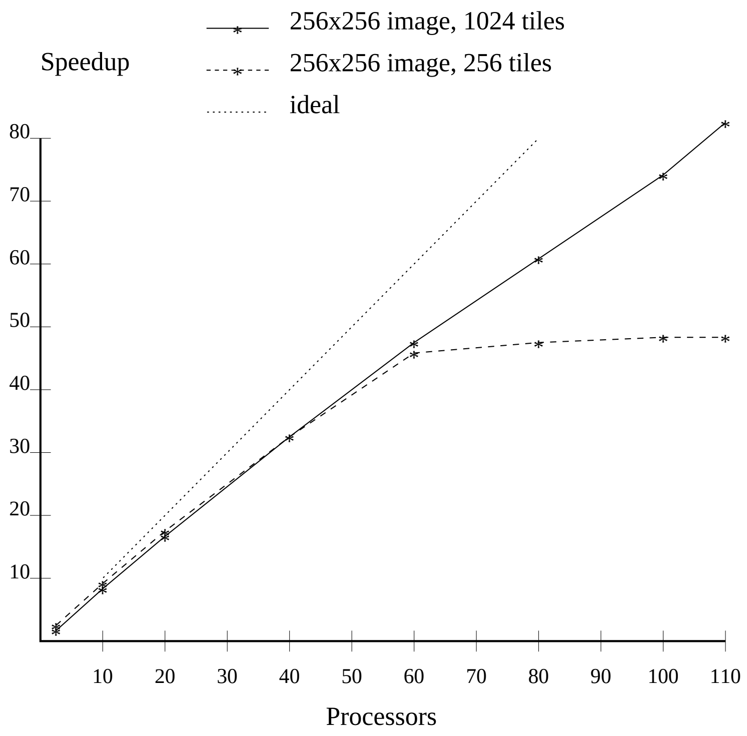

Figure 6.13: Speedup graph for a 256^2 image of view 1 of the blunt fin dataset.

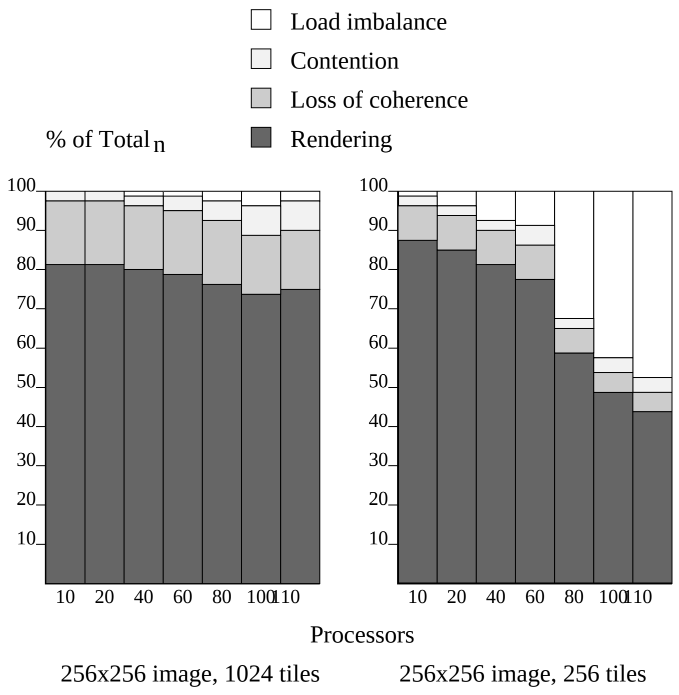

Figure 6.14: Execution profile for a 256^2 image of view 1 of the blunt fin dataset.

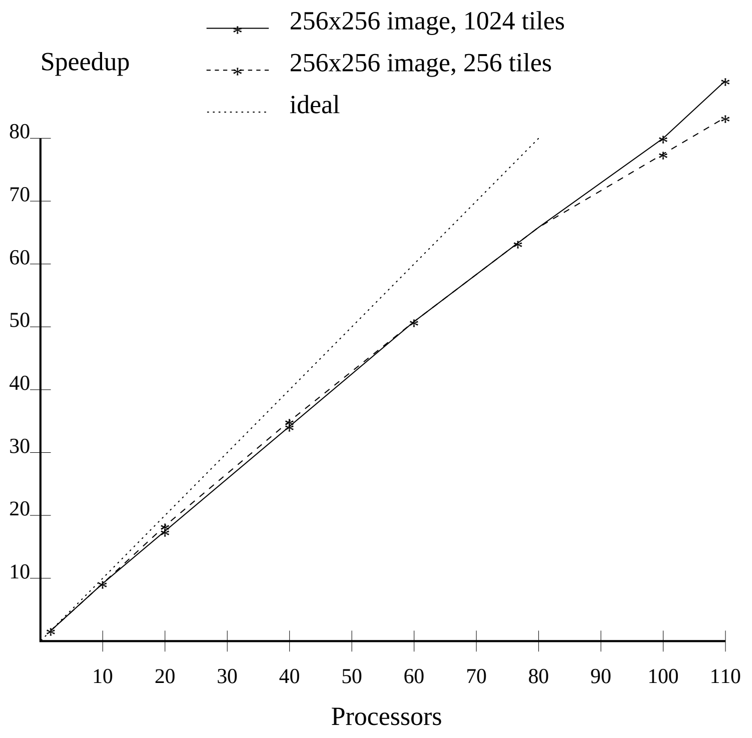

Figure 6.15: Speedup graph for a 256^2 image of view 2 of the blunt fin dataset.

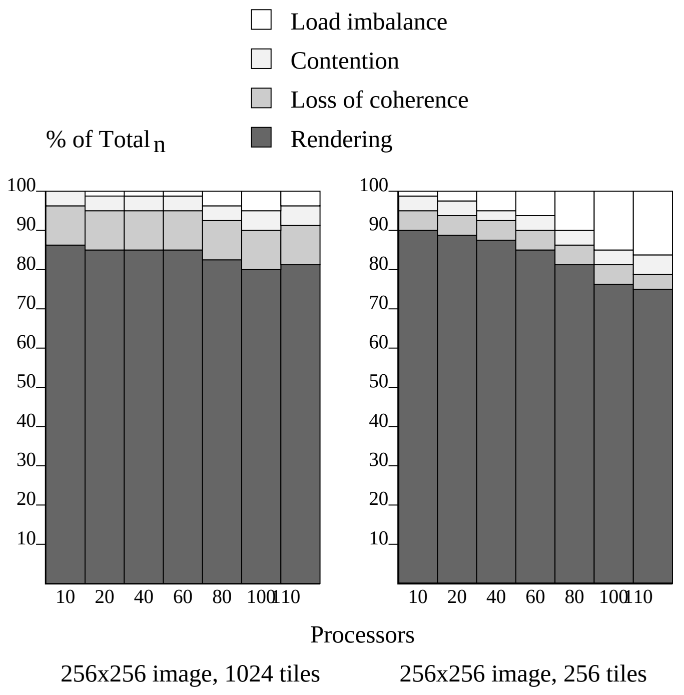

Figure 6.16: Execution profile for a 256^2 image of view 2 of the blunt fin dataset.

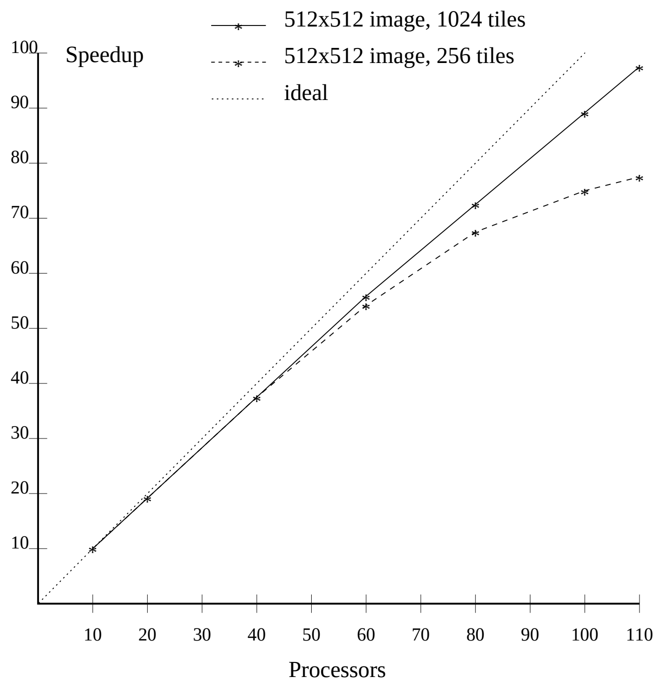

Figure 6.17: Speedup graph (relative to $T_{10} = 765.9$ using 256 tiles) for a 512^2 image of view 1 of the blunt fin dataset.

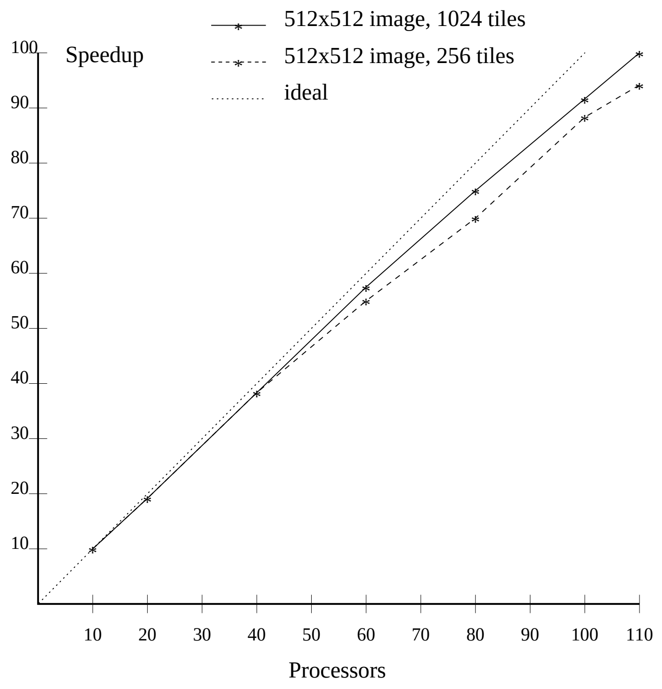

Figure 6.18: Speedup graph (relative to $T_{10}884.5$ using 256 tiles) for a 512^2 image of view 2 of the blunt fin dataset.

View 1 - 256^2 image - see figure 6.1										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
2	117.1	234.2	0	1.8	88.3	127.6	255.1	0	1.6	81.0
10	23.7	234.4	1.2	8.7	87.3	25.5	254.9	0	8.1	81.0
20	12.2	234.9	3.9	17.0	84.8	12.8	255.1	0	16.2	80.8
40	6.4	236.3	6.9	32.3	80.8	6.5	256.8	0.7	31.8	79.5
60	4.5	240.4	8.8	46.0	76.6	4.4	259.5	1.4	47.0	78.3
80	4.4	239.0	31.5	47.0	58.8	3.4	263.2	1.8	60.8	76.0
100	4.3	243.6	42.4	48.1	48.1	2.8	266.2	2.9	73.9	73.9
110	4.3	247.8	46.5	48.1	43.7	2.5	268.5	3.3	82.7	75.2

View 2 - 256^2 image - see figure 6.2										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
2	122.0	243.8	0	1.8	91.6	128.9	257.7	0	1.7	86.6
10	24.7	244.7	1.1	9.0	90.4	25.9	257.8	0.4	8.6	86.3
20	12.5	245.4	1.8	17.9	89.4	13.1	259.3	0.8	17.1	85.3
40	6.4	244.0	5.3	34.9	87.3	6.6	258.8	1.4	33.8	84.6
60	4.4	244.1	6.4	50.8	84.6	4.4	260.1	1.8	50.8	84.6
80	3.4	245.3	10.1	65.7	82.1	3.4	261.7	3.4	65.7	82.1
100	2.9	247.2	15.2	77.0	77.0	2.8	263.1	4.1	79.8	79.8
110	2.7	247.9	14.0	82.7	75.2	2.5	264.0	3.6	89.4	81.2

Table 6.6: Rendering phase performance measurements for the blunt fin dataset for two different tile sizes. Generated image size is 256^2, n is the number of processors, T_n is the execution time in seconds, R_n is the sum of the rendering times on all processors in seconds, B_n is the percentage of load imbalance, S_n is the speedup, and E_n is the efficiency.

Table 6.6 gives performance measurements for both views of the blunt fin dataset for 256^2 images. Speedup graphs for view 1 are given in figure 6.13, and for view 2 in figure 6.15. Speedup and efficiency have been calculated using the best sequential times (obtained using one image tile and all local memory) of $T_1 = 206.8$ seconds for view 1, and $T_1 = 223.4$ seconds for view 2. The results show a direct trade-off between maximizing the utilization of spatial coherence through the use of larger tile sizes, and promoting good scalability by minimizing load imbalance through the use of smaller tile sizes and dynamic task generation. At 110 processors the approach of using smaller tiles to reduce load imbalance is more effective, despite the increases due to loss of coherence. This is because the more effective load balancing provides near-linear speedup as n increases. As the execution profiles in figures 6.14 and 6.16 show, this is especially true for views that contain tiles that vary greatly in the number of cell faces projecting to them, and/or in their depth complexity. In view 1, the grid has been rotated about the x and y axes, and there is some empty space at the top and bottom of the image. This produces a large variation in the number of cell faces that project to any given tile. The largest task contains 4310 cell faces when the image is

View 1 - 512^2 image - see figure 6.1						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	77.4	765.9	1.1	80.3	803.1	0
20	39.6	764.1	3.5	40.3	803.4	0
40	20.7	767.4	7.3	20.4	807.0	0.3
60	14.5	774.4	10.9	13.8	812.8	1.7
80	11.5	777.2	15.5	10.5	819.7	2.2
100	10.2	785.5	22.7	8.6	826.2	3.7
110	10.0	786.2	28.6	7.9	828.6	3.8

View 2 - 512^2 image - see figure 6.2						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	89.5	884.5	1.2	91.4	911.2	0.3
20	45.4	882.3	2.9	46.0	914.6	0.6
40	24.1	884.5	8.0	23.1	913.8	0.9
60	16.0	886.7	7.3	15.6	918.0	2.1
80	12.6	892.0	11.6	11.8	924.4	2.3
100	10.0	894.9	10.6	9.6	928.8	3.5
110	9.4	899.0	12.7	8.8	930.0	3.9

Table 6.7: Rendering phase performance measurements for the blunt fin dataset for two different tile sizes. Generated image size is 512^2, n is the number of processors, T_n is the execution time in seconds, R_n is the sum of the rendering times on all processors in seconds, and B_n is the percentage of load imbalance.

broken into 1024 tiles, and 9253 cell faces for 256 tiles. In view 2, the grid has been rotated about the x axis and has been zoomed up significantly. Some cell faces will project outside the image and be clipped during the parallel view sort, and the remaining ones will be larger with respect to the tile size, and more evenly distributed over the image. The largest task in view 2 contains 1368 cell faces for an image with 1024 tiles, and 2695 cell faces for 256 tiles. Although view 1 contains roughly twice as many cell faces as view 2, it also has 22.8% of its pixels empty, as opposed to 2.7% for view 2 (see table 6.2). This explains why the sequential rendering time for view 1 is larger than the sequential rendering time for view 2, even though it seems intuitive that view 2 would take less time due to fewer cell faces and more coherence.

Examination of the total time spent rendering over all processors, (R_n) in table 6.6, shows that the storage of active cell faces in local memory minimizes the increase in contention as the number of processors increases. As n goes from 2 to 110 processors, the total time spent rendering over all processors increases 5.8% and 5.3% for view 1 using 256 and 1024 image tiles respectively, and 1.7% and 2.4% for view 2. Again, the particular viewing transformation has an effect in that the increase in contention is higher in those images having larger numbers of small cell faces per tile.

Table 6.7 gives performance measurements for rendering 512^2 images. As sequential execution times (T_1) could not be reliably obtained, speedup graphs are computed relative to the execution time using 10 processors and 256 image tiles. The speedup graphs are given in figures 6.17 and 6.18. Rendering statistics for the larger image size show the same characteristics, with the speedup for increasing n being more nearly linear when using more tasks and smaller image tiles.

6.2.2 Post Dataset

Configuration for 256^2 image	View 1	View 2	View 3
Shared memory, 1 tile	255.2	321.7	397.5
Shared memory, 256 tiles	288.5	357.4	438.0
Shared memory, 1024 tiles	324.5	402.6	488.2

Table 6.8: Sequential execution times (T_1) for the post dataset in seconds using various configurations in order to show the effect of multiple image tiles. The times given are for a 256^2 image.

Table 6.8 gives sequential execution times (T_1) for all three views of the post dataset, for various tile sizes. This dataset is too large to be rendered using all local memory, so only shared memory configurations were measured. The first row lists the time to execute on one processor using one tile and shared memory for the data structures that are normally shared during parallel execution. The second and third rows give execution times on one processor using shared memory with the image broken up into 256 and 1024 tiles respectively. Again, breaking the image up into tiles has a significant impact on the sequential rendering time. For view 1 of the post dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 13.0% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 27.2% increase in the sequential rendering time. For view 2 of the post dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 11.1% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 25.1% increase in the sequential rendering time. For view 3 of the post dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 10.2% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 22.8% increase in the sequential rendering time.

Table 6.9 gives the performance measurements for the post dataset. Speedup and efficiency have been calculated using the best sequential times obtainable (one image tile using shared memory) of $T_1 = 255.2$ seconds for view 1, $T_1 = 321.7$ seconds for view 2, and $T_1 = 397.5$ seconds for view 3. Speedup graphs and execution profiles are given in figures 6.19 and 6.20 for view 1, figures 6.21 and 6.22 for view 2, and figures 6.23 and 6.24 for view 3. These figures again show the trade-off between the inefficiency due to loss of coherence when breaking the image into smaller tiles, and the inefficiency due to load imbalance as n increases when using larger tiles. Using 1024 tiles, all three views produce nearly linear speedup. For smaller values of n (< 80), better execution times (T_n) result from using larger image tiles. View 1 contains the entire grid, meaning that there are many small cell faces and some areas of empty space in the image due to the shape of the grid. This view produces the most difficult load balancing requirements for larger image tiles. In

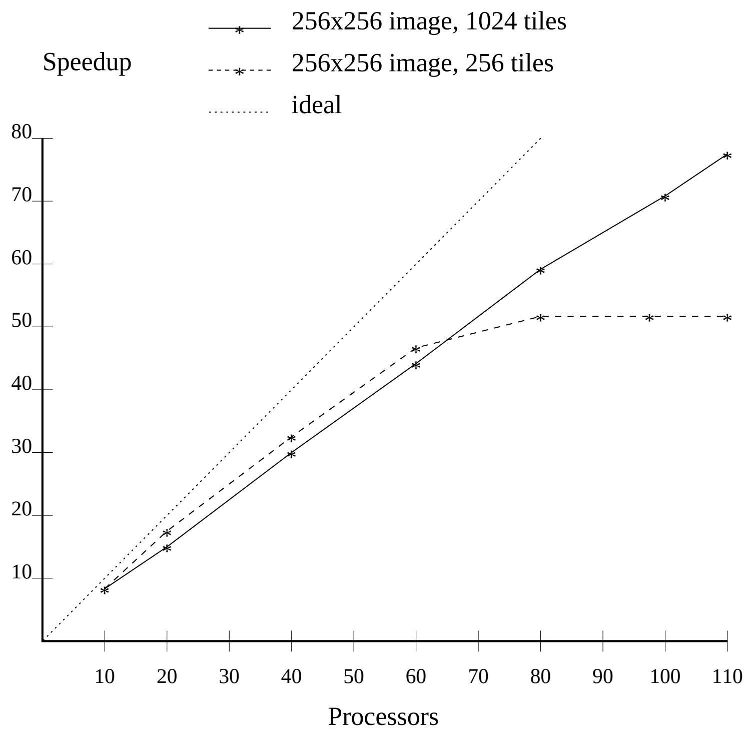

Figure 6.19: Speedup graph for a 256^2 image of view 1 of the post dataset.

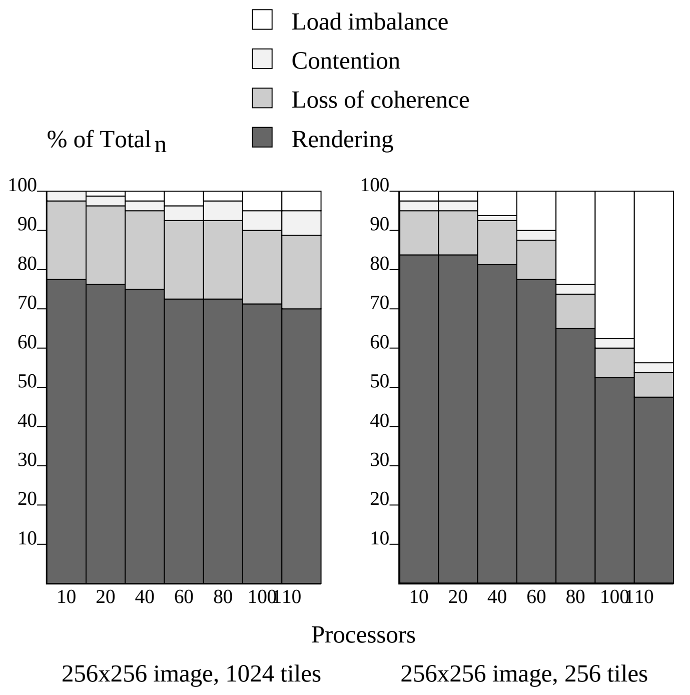

Figure 6.20: Execution profile for a 256^2 image of view 1 of the post dataset.

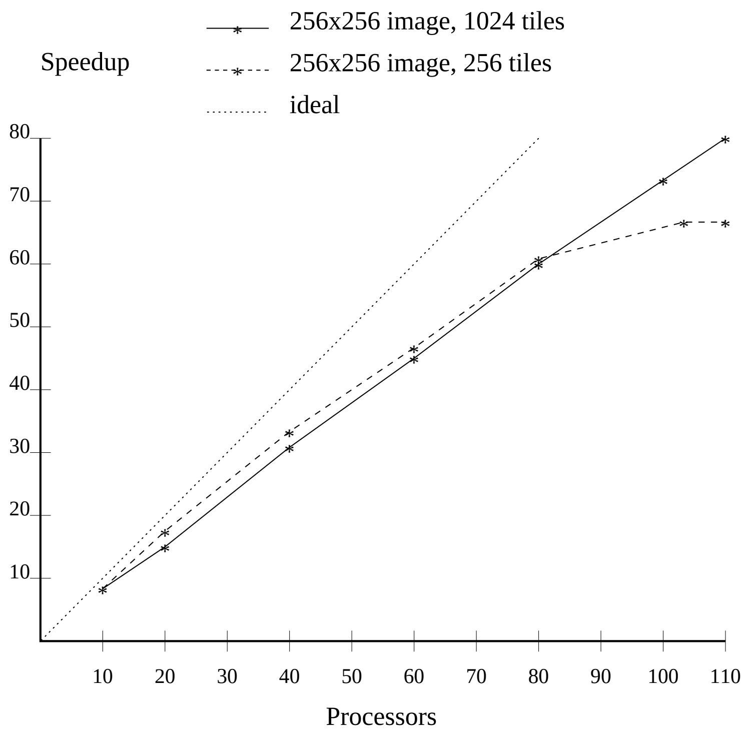

Figure 6.21: Speedup graph for a 256^2 image of view 2 of the post dataset.

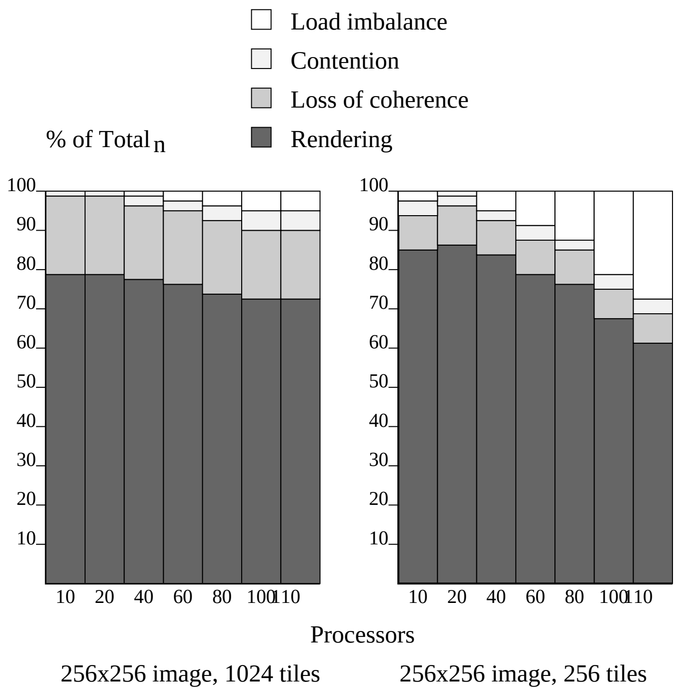

Figure 6.22: Execution profile for a 256^2 image of view 2 of the post dataset.

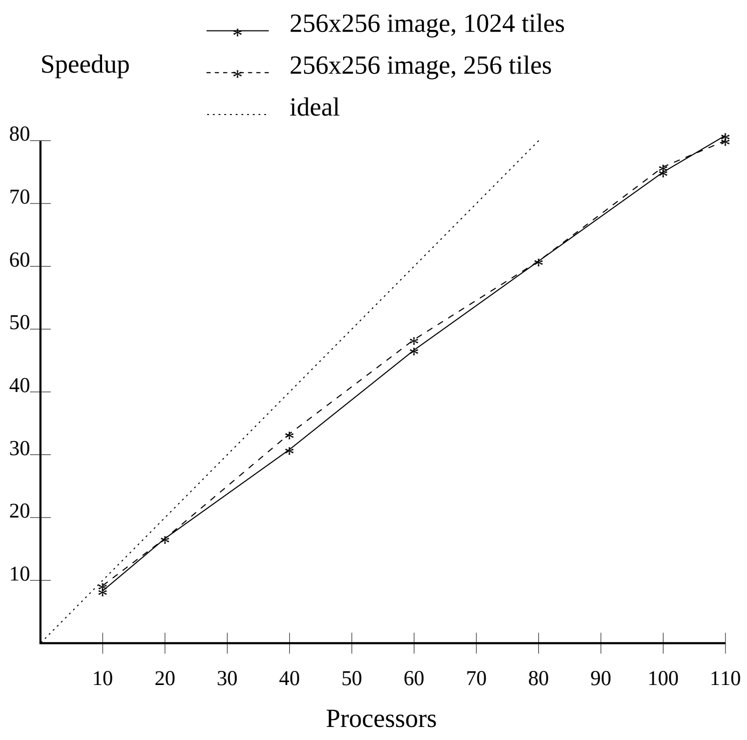

Figure 6.23: Speedup graph for a 256^2 image of view 3 of the post dataset.

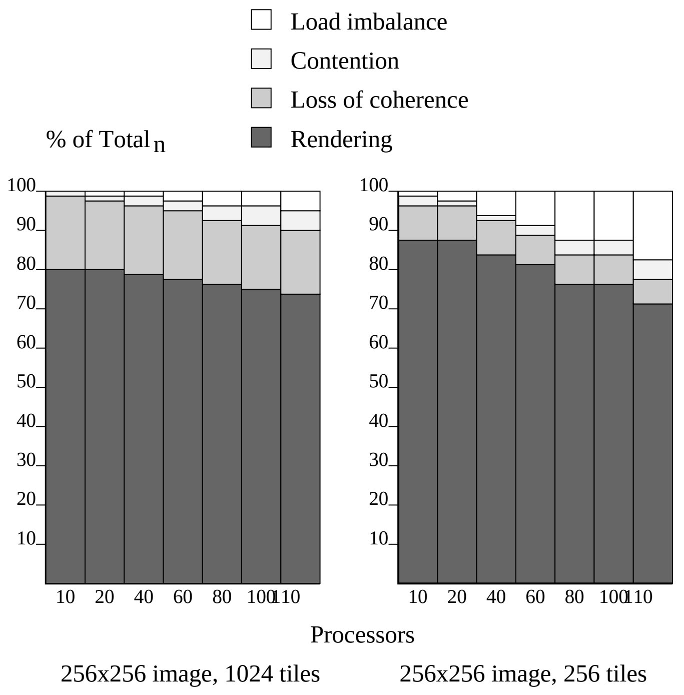

Figure 6.24: Execution profile for a 256^2 image of view 3 of the post dataset.

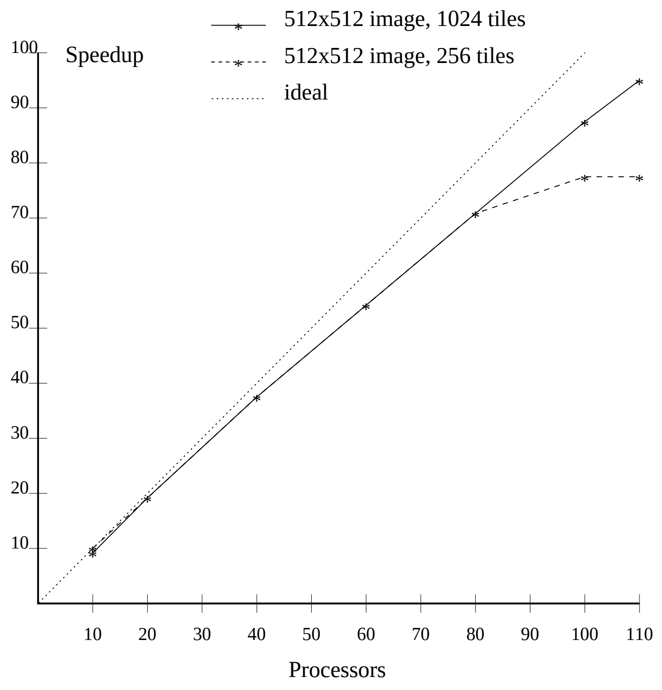

Figure 6.25: Speedup graph (relative to $T_{10} = 925.1$) using 256 tiles) for a 512^2 image of view 1 of the post dataset.

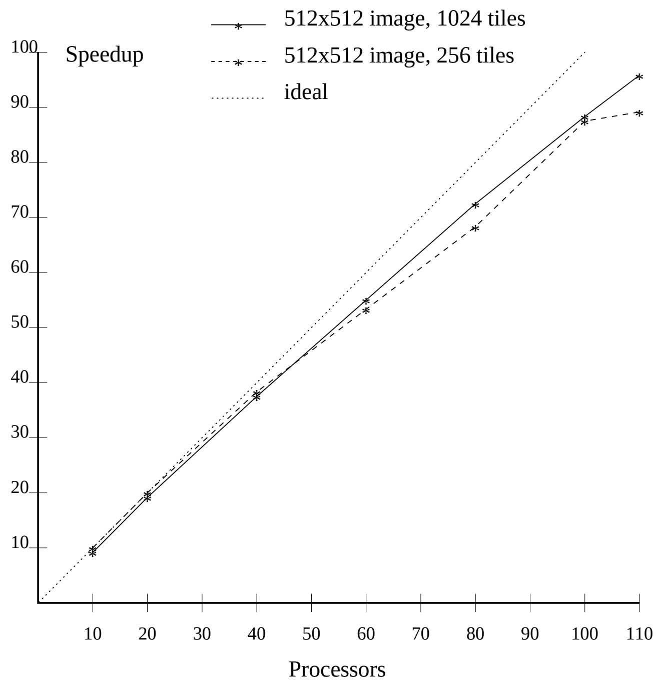

Figure 6.26: Speedup graph (relative to $T_{10} = 1141.9$ using 256 tiles) for a 512^2 image of view 2 of the post dataset.

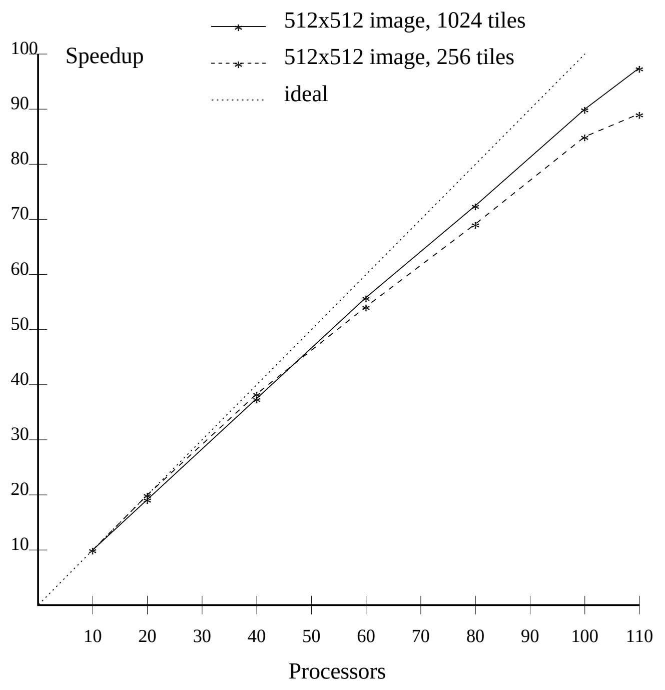

Figure 6.27: Speedup graph (relative to $T_{10} = 1429.7$ using 256 tiles) for a 512^2 image of view 3 of the post dataset.

view 2 we have zoomed in on the center of the grid. Many cell faces will be outside the image and will be clipped during the parallel viewing transformation. In addition, there is a region of empty space in the middle, and the individual cell faces are larger relative to the tile size. View 3 is a rotated version of view 2. The grid has been rotated about the y axis to show the effect of rendering through a nonconvex grid with a void. The result of the rotation is to give a fairly consistent depth complexity over the entire image. This fact is evidenced by the reduction in load imbalance over the other two views when using larger image tiles. For an image broken into 1024 tiles, the largest task sizes are 5234 cell faces in view 1, 5847 cell faces in view 2, and 1852 cell faces in view 3. For an image broken into 256 tiles, the largest tasks are 12104, 11210, and 4492, respectively.

Examination of the total rendering time R_n in table 6.9 shows a minimal increase in contention as the number of processors increases. As n goes from 10 to 110 processors, the total time spent rendering over all processors increases 1.6% and 4.1% for view 1 with 256 and 1024 image tiles respectively, 1.1% and 3.2%for view 2, and 2.7% and 3.2% for view 3.

The performance measurements for 512^2 images given in table 6.10 show characteristics similar to those for 256^2 images. The speedup is more nearly linear using 1024 tiles, and the point at which the improved load balancing offsets the loss of coherence to produce better execution times (T_n) happens earlier ($n > 60$). Since sequential execution times (T_1) could not be reliably obtained, speedup graphs for 512^2 images are computed relative to the execution time using 10 processors (figures 6.25, 6.26, and 6.27).

View 1 - 256^2 image - see figure 6.3										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
10	30.5	299.1	1.7	8.4	83.7	33.3	331.9	0	7.7	76.7
20	15.2	294.0	3.4	16.8	83.9	16.7	331.3	1.0	15.3	76.4
40	7.9	295.6	6.4	32.3	80.8	8.5	334.5	2.0	30.0	75.1
60	5.5	298.0	8.9	46.4	77.3	5.8	338.6	3.0	44.0	73.3
80	4.9	300.0	22.7	52.1	65.1	4.4	340.9	3.5	58.0	72.5
100	4.9	302.5	38.3	52.1	52.1	3.6	343.3	3.9	70.9	70.9
110	4.9	303.8	43.7	52.1	47.3	3.3	345.6	4.1	77.3	70.3
View 2 - 256^2 image - see figure 6.4										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
10	37.7	371.8	1.5	8.5	85.3	41.3	412.0	0.3	7.8	77.9
20	18.7	365.3	2.3	17.2	86.0	20.7	411.6	0.6	15.5	77.7
40	9.6	366.2	4.5	33.5	83.8	10.5	413.3	1.2	30.6	76.6
60	6.8	370.4	8.9	47.3	78.8	7.1	416.1	2.0	45.3	75.5
80	5.3	372.0	12.6	60.7	75.9	5.4	419.7	2.1	59.6	74.5
100	4.8	374.2	22.1	67.0	67.0	4.4	423.3	2.9	73.1	73.1
110	4.8	375.9	28.6	67.0	60.9	4.0	425.2	3.1	80.4	73.1
View 3 - 256^2 image - see figure 6.5										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
10	45.6	451.2	1.2	8.7	87.2	49.8	495.5	0.4	8.0	79.8
20	22.9	445.4	2.7	17.4	86.8	25.0	495.6	0.9	15.9	79.5
40	11.9	445.9	6.5	33.4	83.5	12.6	496.7	1.4	31.5	78.9
60	8.2	449.2	8.4	48.5	80.8	8.5	499.4	2.2	46.8	77.9
80	6.5	454.7	12.1	61.2	76.4	6.5	503.4	2.7	61.2	76.4
100	5.2	460.4	11.6	76.4	76.4	5.3	507.8	3.1	75.0	75.0
110	5.0	463.2	15.6	79.5	72.3	4.9	511.2	4.5	81.1	73.7

Table 6.9: Rendering phase performance measurements for the post dataset for two different tile sizes. Generated image size is 256^2, n is the number of processors, T_n is the execution time in seconds, R_n is the sum of the rendering times on all processors in seconds, B_n is the percentage of load imbalance, S_n is the speedup, and E_n is the efficiency.

View 1 - 512^2 image - see figure 6.3						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	94.7	925.1	2.3	98.6	980.9	0.5
20	48.0	911.7	5.0	49.6	979.0	1.2
40	25.2	915.0	9.0	25.2	983.9	2.3
60	17.3	919.5	11.1	17.1	991.1	3.2
80	13.1	925.1	11.4	13.0	996.9	4.2
100	12.0	930.5	22.4	10.6	1003.4	5.5
110	12.1	934.1	29.4	9.7	1006.3	5.4

View 2 - 512^2 image - see figure 6.4						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	115.6	1141.9	1.3	121.7	1212.8	0.3
20	57.3	1128.1	1.6	60.9	1211.9	0.5
40	30.3	1131.8	6.6	30.8	1214.3	1.4
60	21.5	1138.8	11.7	20.8	1223.9	1.7
80	17.0	1143.5	16.0	15.7	1229.4	2.3
100	13.1	1149.4	12.2	13.0	1236.3	4.9
110	12.9	1153.8	18.3	11.9	1241.2	4.8

View 3 - 512^2 image - see figure 6.5						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	145.9	1429.7	2.0	151.0	1505.2	0.3
20	73.4	1415.9	3.5	75.8	1506.6	0.6
40	38.0	1418.5	6.6	38.4	1507.6	1.8
60	26.7	1425.5	10.9	25.8	1518.4	2.0
80	20.6	1433.6	13.0	19.8	1527.9	3.2
100	16.8	1442.9	14.0	15.9	1533.8	3.7
110	16.0	1449.2	17.8	14.7	1537.2	4.7

Table 6.10: Rendering phase performance measurements for the post dataset for two different tile sizes. Generated image size is 512^2, n is the number of processors, T_n is the execution time in seconds, R_n is the sum of the rendering times on all processors in seconds, and B_n is the percentage of load imbalance.

6.2.3 Delta Wing Dataset

Configuration for 256^2 image	View 1	View 2
Shared memory, 1 tile	317.1	384.0
Shared memory, 256 tiles	336.2	405.2
Shared memory, 1024 tiles	360.2	430.7

Table 6.11: Sequential execution times (T_1) for the delta wing dataset in seconds using various configurations in order to show the effect of multiple image tiles. The times given are for a 256^2 image.

Table 6.11 gives sequential execution times (T_1) for both views of the delta wing dataset, for various tile sizes. This dataset is too large to be rendered using all local memory, so only shared memory configurations were measured. The first row lists the time to execute on one processor using one tile and shared memory for the data structures that are normally shared during parallel execution. The second and third rows give execution times on one processor using shared memory with the image broken up into 256 and 1024 tiles respectively. Again, breaking the image up into tiles has a significant impact on the sequential rendering time. For view 1 of the delta wing dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 6% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 13.6% increase in the sequential rendering time. For view 2 of the delta wing dataset, breaking the 256^2 image into 256 tiles of 16^2 pixels causes a 5.5% increase in the sequential rendering time, and breaking the image into 1024 tiles of 8^2 pixels causes a 12.2% increase in the sequential rendering time.

Table 6.12 gives the performance measurements for both views of the delta wing dataset for 256^2 images. Speedup and efficiency have been calculated using the sequential times (obtained using one image tile and shared memory) of $T_1 = 317.1$ seconds for view 1, and $T_1 = 384.0$ seconds for view 2. Speedup graphs and execution profiles are given in figures 6.28 and 6.29 for view 1, and figures 6.30 and 6.31 for view 2. Both views produce nearly linear speedup using 1024 image tiles. For smaller values of n better execution times (T_n) result from using larger image tiles due to the higher utilization of coherence and reduced load balancing inefficiencies. View 1 contains the entire grid, meaning that there are many small cell faces and some areas of empty space in the image due to the wing shape of the grid. This large variation in task size produces load balancing difficulties when using 256 tiles as seen in figure 6.29. In view 2, the grid has been rotated about the x axis and zoomed up to fill the image.

For an image formed of 1024 tiles, the largest task in view 1 contains 6187 cell faces, and the largest task in view 2 contains 4318 cell faces. For an image with 256 tiles the largest tasks are 14903 and 9111 respectively. As n goes from 10 to 110 processors, the total time spent rendering over all processors (R_n) increases 8.4% and 7.2% for view 1 using 256 and 1024 tiles respectively, and 2.9% and 2.2% for view 2. These figures again reflect the fact that the image in which the cell faces are fewer, and larger with respect to the tile size, require fewer remote memory accesses, thus reducing the rate at which contention increases when processors are added.

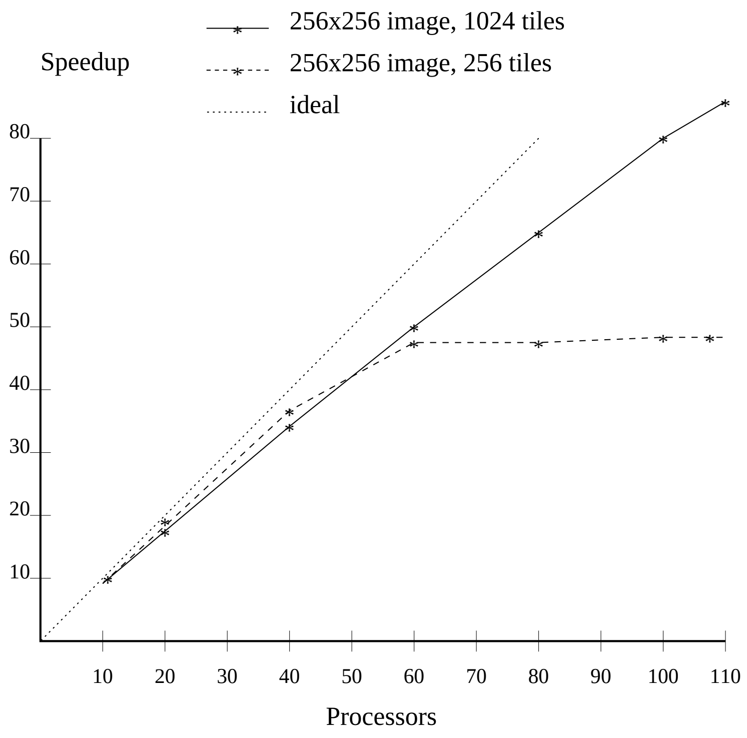

Figure 6.28: Speedup graph for a 256^2 image of view 1 of the delta wing dataset.

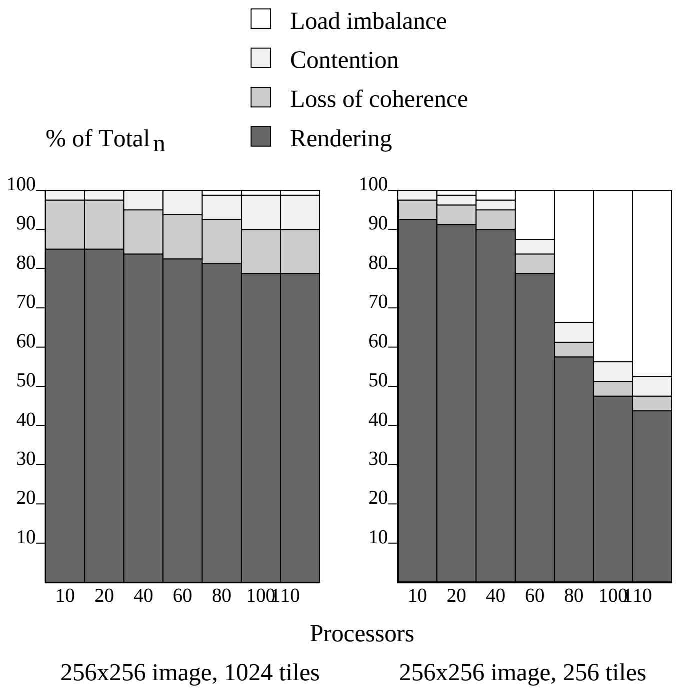

Figure 6.29: Execution profile for a 256^2 image of view 1 of the delta wing dataset.

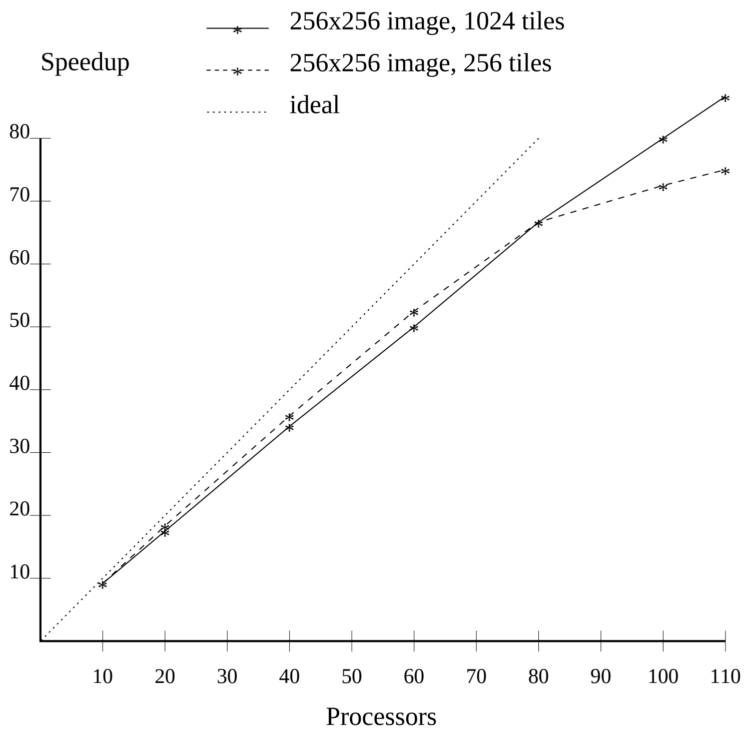

Figure 6.30: Speedup graph for a 256^2 image of view 2 of the delta wing dataset.

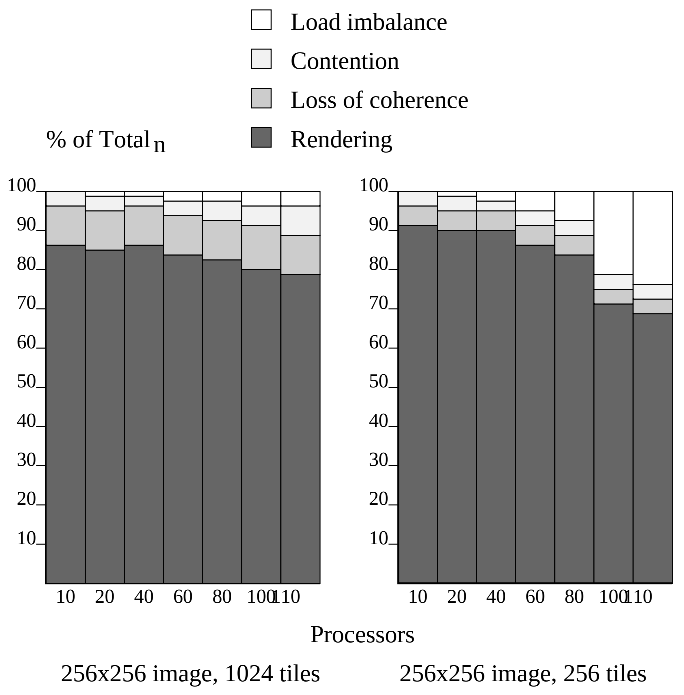

Figure 6.31: Execution profile for a 256^2 image of view 2 of the delta wing dataset.

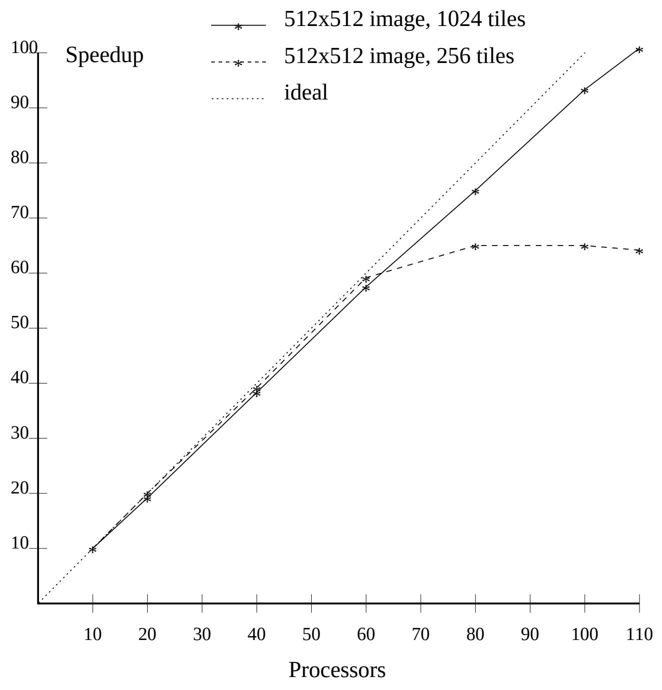

Figure 6.32: Speedup graph (relative to $T_{10} = 1036.7$ using 256 image tiles) for a 512^2 image of view 1 of the delta wing dataset.

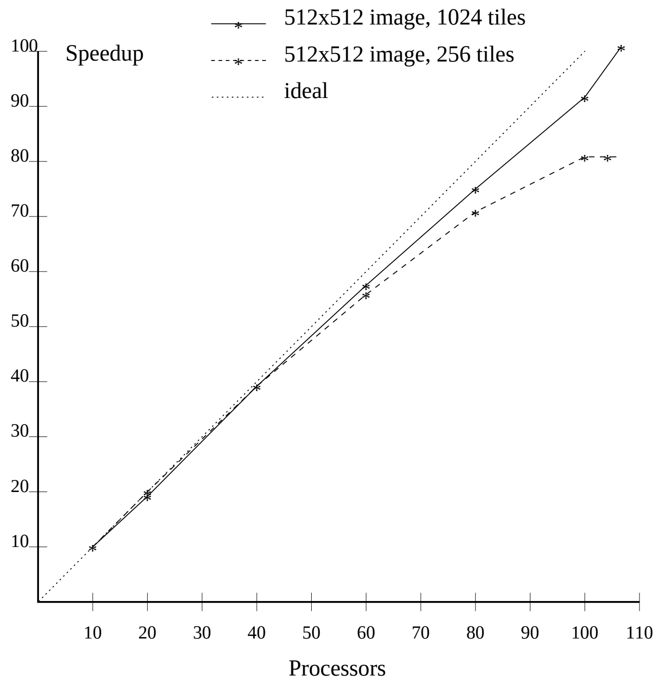

Figure 6.33: Speedup graph (relative to $T_{10} = 1479.9$ using 256 image tiles) for a 512^2 image of view 2 of the delta wing dataset.

View 1 - 256^2 image - see figure 6.6										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
10	34.5	345.1	0.1	9.2	92.0	37.2	371.9	0	8.5	85.2
20	17.4	346.4	0.5	18.2	91.1	18.7	373.5	0	17.0	84.8
40	8.8	346.6	1.7	36.0	90.0	9.4	373.9	0.2	33.7	84.3
60	6.8	351.2	13.8	46.6	77.7	6.4	381.6	0.4	49.5	82.6
80	6.9	361.1	33.8	46.0	57.4	4.9	389.6	0.9	64.7	80.1
100	6.7	369.8	44.5	47.3	47.3	4.0	396.2	1.1	79.3	79.3
110	6.6	374.1	48.1	48.0	43.7	3.7	398.6	1.4	85.7	77.9

View 2 - 256^2 image - see figure 6.7										
	256 tiles					1024 tiles				
n	T_n	R_n	B_n	S_n	E_n	T_n	R_n	B_n	S_n	E_n
10	41.9	416.8	0.4	9.2	91.6	44.8	447.2	0.2	8.6	85.7
20	21.3	417.5	1.9	18.0	90.1	22.5	448.5	0.5	17.1	85.3
40	10.7	415.1	3.3	35.9	89.7	11.2	445.7	0.9	34.3	85.7
60	7.4	417.9	5.4	51.9	86.5	7.6	448.0	1.8	50.5	84.2
80	5.7	422.9	6.6	67.4	84.2	5.8	453.9	2.4	66.2	82.8
100	5.3	426.9	19.4	72.5	72.5	4.8	458.6	3.1	80.0	80.0
110	5.1	429.0	21.3	75.3	68.5	4.4	457.1	3.2	87.3	79.3

Table 6.12: Rendering phase performance measurements for the delta wing dataset for two different tile sizes. Generated image size is 256^2, n is the number of processors, T_n is the execution time in seconds, R_n is the sum of the rendering times on all processors in seconds, B_n is the percentage of load imbalance, S_n is the speedup, and E_n is the efficiency.

Table 6.13 gives performance measurements for rendering 512^2 images. The speedup graphs for view 1 are given in figures 6.32, and in 6.33 for view 2. Since sequential execution times (T_1) could not be reliably obtained, speedup graphs for 512^2 images are computed relative to the execution time using 10 processors.

View 1 - 512^2 image - see figure 6.6						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	103.7	1036.7	0	108.7	1087.2	0
20	52.1	1039.0	0.2	54.4	1088.4	0
40	26.3	1035.3	1.4	27.1	1083.2	0.1
60	17.7	1043.2	1.7	18.2	1092.5	0.1
80	15.9	1054.1	17.0	13.8	1103.1	0.3
100	16.0	1068.9	32.8	11.2	1114.7	0.4
110	16.1	1074.5	39.2	10.3	1121.0	0.5

View 2 - 512^2 image - see figure 6.7						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	148.4	1479.9	0.3	152.8	1525.7	0.1
20	75.2	1482.7	1.4	76.6	1525.9	0.4
40	38.3	1473.1	3.8	38.2	1514.9	1.0
60	26.3	1481.2	6.0	25.9	1522.7	1.9
80	20.9	1491.5	10.5	19.7	1531.1	2.8
100	18.3	1500.7	17.9	16.1	1540.4	3.5
110	18.3	1506.1	25.2	14.6	1544.9	3.9

Table 6.13: Rendering phase performance measurements for the delta wing dataset for two different tile sizes. Generated image size is 512^2, n is the number of processors, T_n is the execution time in seconds, R_n is the sum of the rendering times on all processors in seconds, and B_n is the percentage of load imbalance.

6.2.4 Shuttle Dataset

The shuttle dataset is the largest and most complex of the four datasets tested. It is also probably most representative of the type of dataset that might be generated on a massively parallel system. Table 6.14 gives the performance measurements for all views of the shuttle dataset for 256^2 images, and table 6.15 gives performance measurements for 512^2 images. Speedup graphs are given in figures 6.34 to 6.41. Speedup and efficiency have been calculated relative to T_{10} using 256 image tiles due to the difficulty of obtaining a valid sequential execution time for such a large dataset. This difficulty also prevents the exact measurement of the inefficiency associated with the loss of coherence when breaking the image into tiles for sequential execution. However, examination of the statistics collected show results strikingly similar to those for the smaller datasets. In fact, the same features are present and intensified. Load balancing is much more critical (and more difficult) on such a large, complex, and widely varying dataset. The effects of the loss of coherence are apparent in that R_n (the sum of the rendering times over all processors) is smaller when using larger tiles.

As for previous datasets, the results show a direct trade-off between maximizing the utilization of spatial coherence through the use of larger tile sizes, and promoting good scalability by minimizing load imbalance through the use of smaller tile sizes and dynamic task generation. For this dataset, even more than the others, the approach of using smaller tiles to reduce load imbalance is clearly more effective, despite the increases due to loss of coherence. The time to render an image using 110 processors is two to three times faster using 1024 tiles than when using 256 tiles. Again, this is because the more effective load balancing provides near-linear speedup as n increases, as examination of the execution times (T_n) for both 256^2 and 512^2 images reveals.

The image in view 1 contains the entire grid. The majority of the complexity is in the center section immediately surrounding the orbiter, external tank and solid rocket booster. The view is from the flat side of the hemisphere, thus the unobstructed shuttle geometry can be seen in the center. The largest task size for this view is 13472 cell faces when using 1024 tiles and 40468 cell faces when using 256 tiles. Inspection of the execution timings in tables 6.14 and 6.15 show that load balancing is difficult for this view, even when using 1024 tiles. Using 10 processors to generate a 256^2 image, the time to render using 1024 tiles is 8.1% slower than the time to render using 256 tiles. This gives an estimate of the inefficiency due to the loss of coherence when using 1024 rather than 256 image tiles. As n goes from 10 to 110, the sum of the rendering times on all processors, R_n, increases 8.9% using 256 tiles and 18.4% using 1024 tiles.

View 2 is a zoomed up and rotated version of view 1. The view is still from the flat side of the hemisphere, thus the unobstructed shuttle geometry can be seen. The largest task size for this view is 25014 cell faces when using 1024 tiles and 62687 cell faces when using 256 tiles. Using 10 processors to generate a 256^2 image, the time to render using 1024 tiles is 6.8% slower than the time to render using 256 tiles, giving an estimate of the inefficiency due to the loss of coherence. At 110 processors however, the time to render using 1024 tiles is almost half the time required using 256 tiles. As n goes from 10 to 110, the sum of the rendering times on all processors, R_n, increases 6.5% using 256 tiles and 7.4% using 1024 tiles.

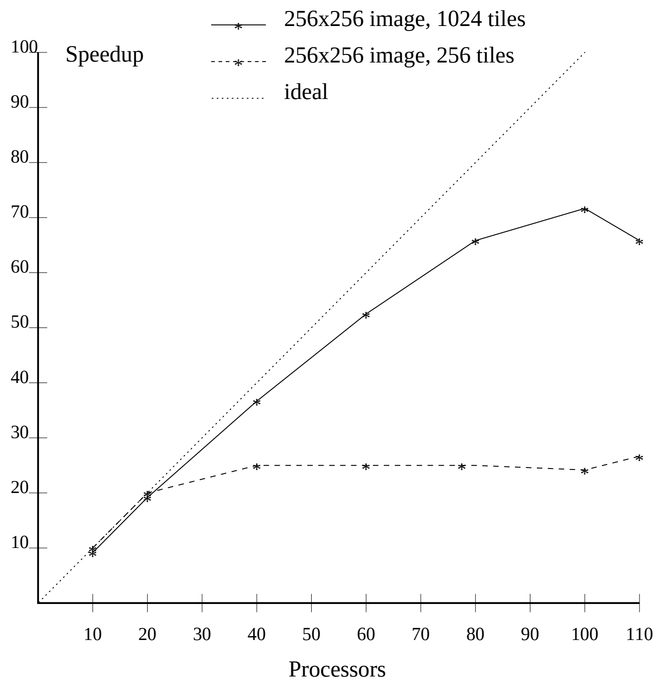

Figure 6.34: Speedup graph (relative to $T_{10} = 468.1$ using 256 image tiles) for a 256^2 image of view 1 of the shuttle dataset.

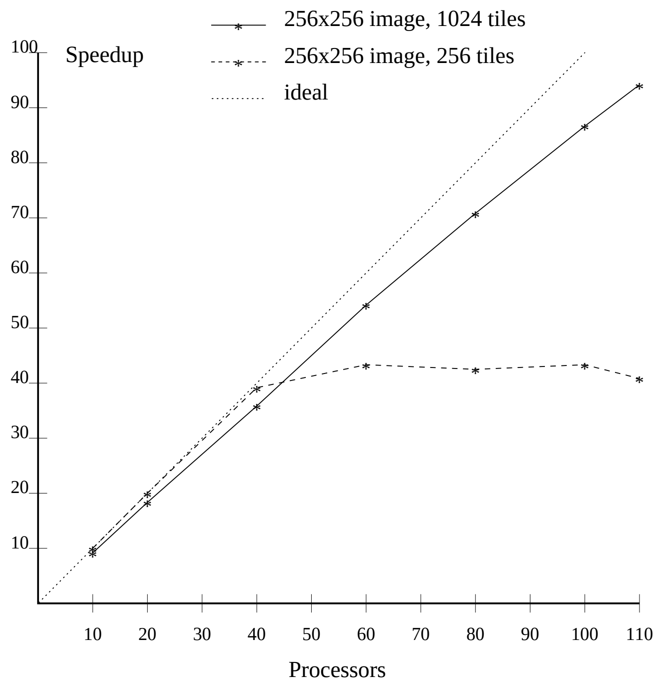

Figure 6.35: Speedup graph (relative to $T_{10} = 1076.6$ using 256 image tiles) for a 256^2 image of view 2 of the shuttle dataset.

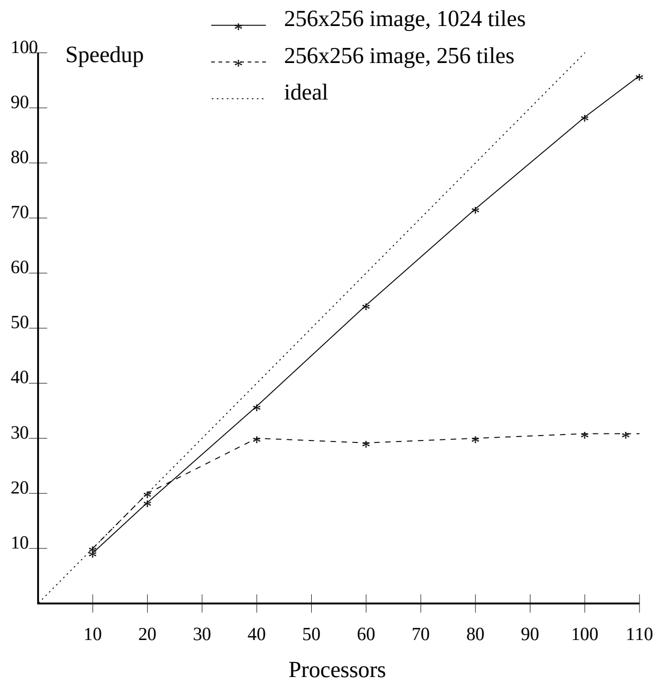

Figure 6.36: Speedup graph (relative to $T_{10} = 1006.7$ using 256 image tiles) for a 256^2 image of view 3 of the shuttle dataset.

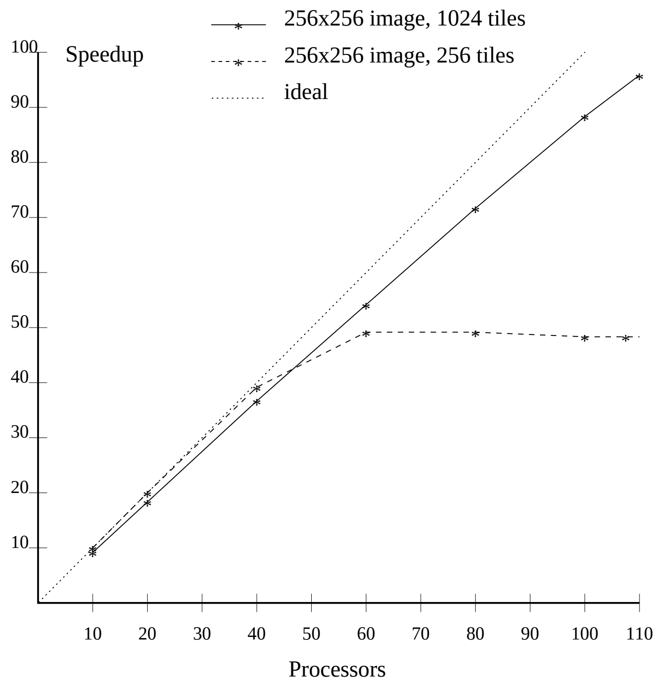

Figure 6.37: Speedup graph (relative to $T_{10} = 944.1$ using 256 image tiles) for a 256^2 image of view 4 of the shuttle dataset.

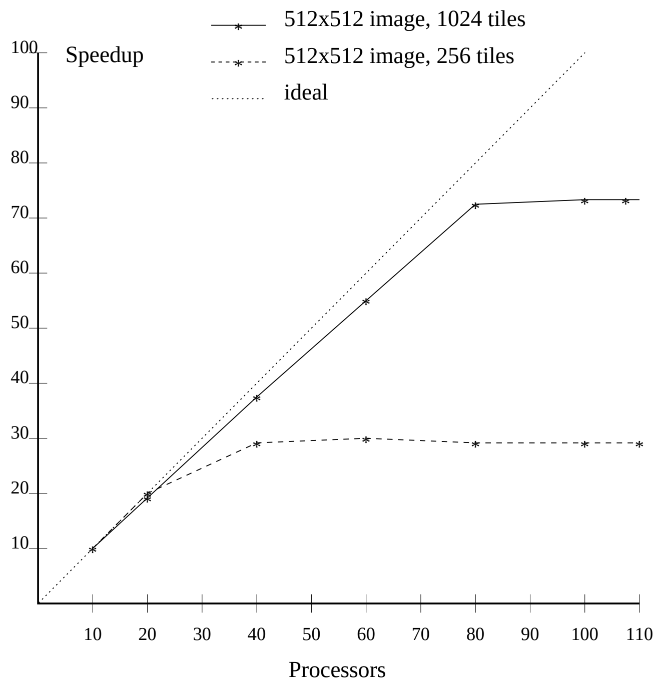

Figure 6.38: Speedup graph (relative to $T_{10} = 1488.5$ using 256 image tiles) for a 512^2 image of view 1 of the shuttle dataset.

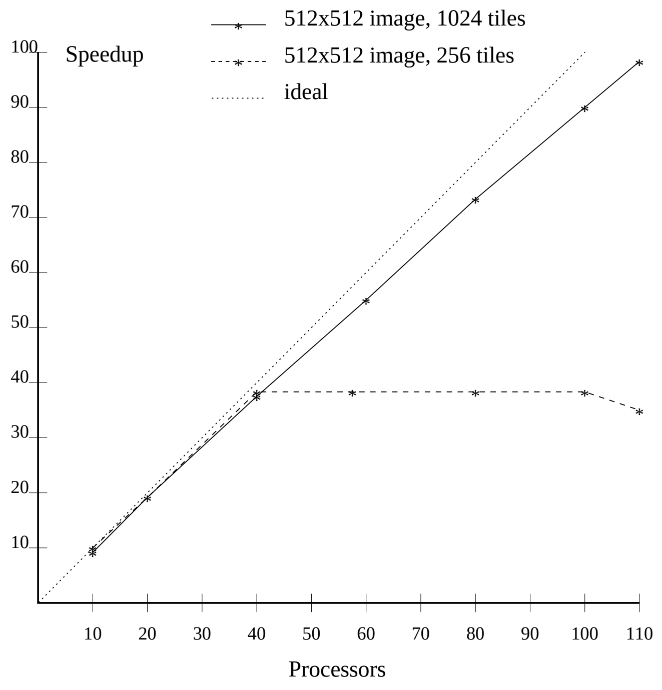

Figure 6.39: Speedup graph (relative to $T_{10} = 2839.3$ using 256 image tiles) for a 512^2 image of view 2 of the shuttle dataset.

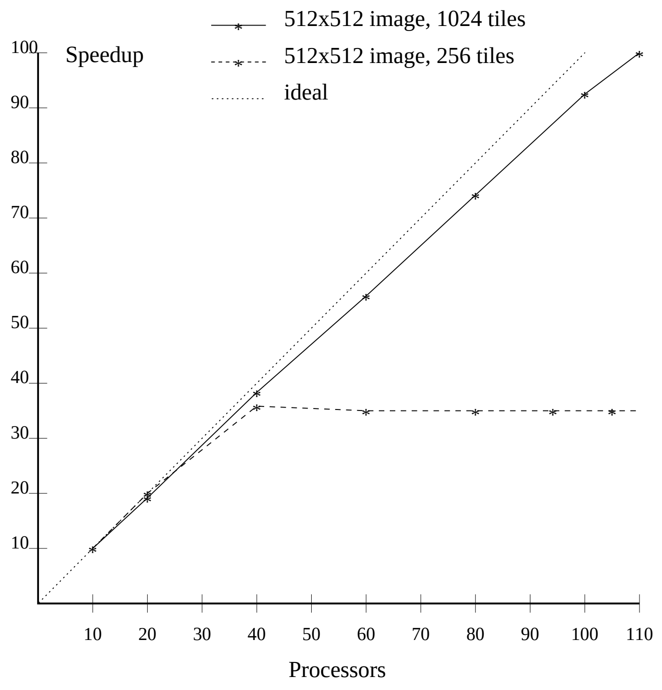

Figure 6.40: Speedup graph (relative to $T_{10} = 3211.7$ using 256 image tiles) for a 512^2 image of view 3 of the shuttle dataset.

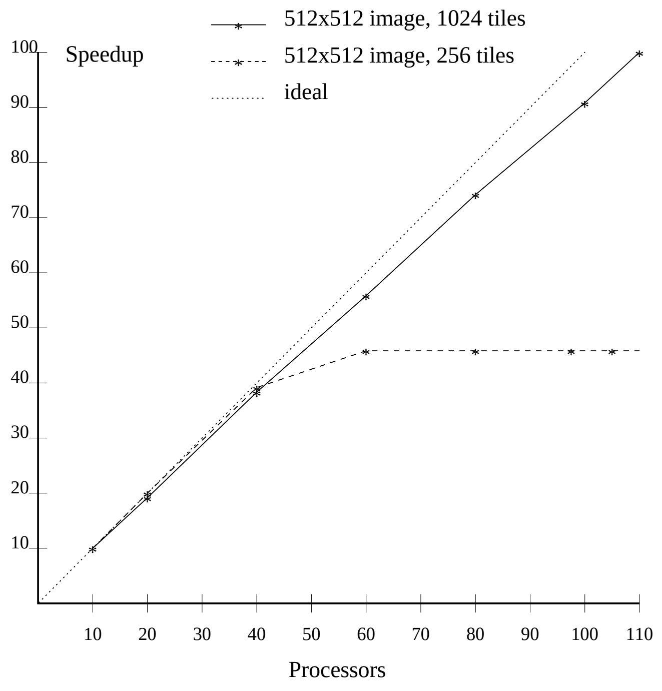

Figure 6.41: Speedup graph (relative to $T_{10} = 2636.4$ using 256 image tiles) for a 512^2 image of view 4 of the shuttle dataset.

View 1 - 256^2 image - see figure 6.8						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	47.0	468.1	0.4	50.7	506.4	0.1
20	23.7	468.6	1.0	25.4	506.4	0.2
40	18.9	482.0	36.1	12.9	515.5	0.4
60	18.8	492.4	56.2	8.7	519.3	0.8
80	18.9	498.8	67.0	7.1	559.0	1.3
100	19.5	515.2	73.5	6.5	556.3	14.8
110	17.0	509.7	72.6	7.1	599.5	23.1

View 2 - 256^2 image - see figure 6.9						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	108.0	1076.6	0.3	115.1	1150.1	0.1
20	54.8	1090.5	0.6	58.5	1167.4	0.3
40	27.9	1097.9	1.5	29.6	1177.0	0.5
60	24.9	1094.3	26.6	19.8	1179.5	0.8
80	25.9	1130.8	45.4	15.1	1194.6	1.1
100	24.9	1124.2	54.8	12.4	1222.5	1.3
110	26.6	1146.1	60.7	11.4	1235.2	1.7

View 3 - 256^2 image - see figure 6.10						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	101.2	1006.7	0.5	108.6	1085.3	0.1
20	51.5	1019.1	1.1	55.3	1103.4	0.2
40	34.1	1040.4	23.7	27.9	1109.2	0.5
60	34.7	1021.4	50.9	18.5	1101.6	0.7
80	33.9	1025.3	62.2	14.1	1115.1	0.9
100	32.9	1029.8	68.6	11.5	1133.1	1.4
110	32.9	1037.1	71.2	10.5	1138.7	1.4

View 4 - 256^2 image - see figure 6.11						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	94.9	944.1	0.5	102.1	1020.6	0
20	48.1	955.8	0.7	51.4	1026.6	0.2
40	24.5	965.8	1.6	26.1	1041.4	0.4
60	19.3	959.8	16.9	17.4	1033.8	0.6
80	19.3	968.2	37.4	13.2	1045.4	0.8
100	19.7	989.2	49.8	10.7	1061.1	1.2
110	19.7	996.4	53.8	9.9	1068.2	1.3

Table 6.14: Rendering phase performance measurements for the shuttle dataset.

View 1 - 512^2 image - see figure 6.8						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	149.7	1488.5	0.6	156.0	1557.4	0.2
20	76.5	1515.5	1.0	79.4	1583.6	0.3
40	51.1	1517.6	25.7	39.7	1579.6	0.5
60	50.5	1510.9	50.1	27.0	1602.3	0.9
80	50.7	1534.3	62.1	20.6	1628.6	1.3
100	50.9	1543.4	69.7	20.3	1666.0	17.8
110	51.9	1554.9	72.7	20.4	1673.7	25.3

View 2 - 512^2 image - see figure 6.9						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	285.5	2839.3	0.5	306.5	3061.5	0.1
20	146.2	2890.6	1.1	152.1	3037.0	0.2
40	75.0	2890.6	3.8	76.2	3034.9	0.4
60	74.6	2877.7	35.7	51.4	3061.4	0.7
80	75.0	2900.6	51.7	38.7	3062.4	1.0
100	75.3	2932.3	61.0	31.6	3105.8	1.5
110	80.5	2947.8	66.6	29.1	3123.8	2.4

View 3 - 512^2 image - see figure 6.10						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	322.8	3211.7	0.5	336.2	3360.0	0.1
20	163.1	3222.4	1.2	169.1	3372.0	0.3
40	90.3	3262.9	9.6	85.5	3400.5	0.5
60	92.0	3245.6	41.2	57.6	3424.9	0.8
80	90.7	3255.5	55.1	43.2	3416.1	1.1
100	90.7	3271.5	63.9	35.0	3445.9	1.4
110	90.9	3287.1	67.1	32.0	3457.2	1.6

View 4 - 512^2 image - see figure 6.11						
	256 tiles			1024 tiles		
n	T_n	R_n	B_n	T_n	R_n	B_n
10	265.4	2636.4	0.7	276.2	2760.6	0.1
20	133.9	2647.7	1.1	138.8	2770.4	0.2
40	68.0	2651.0	2.5	70.3	2798.6	0.5
60	57.0	2667.1	21.9	47.4	2818.1	1.0
80	57.0	2690.7	41.0	35.6	2812.8	1.2
100	57.0	2712.8	52.3	28.9	2837.2	1.8
110	57.1	2719.9	56.6	26.4	2846.5	1.9

Table 6.15: Rendering phase performance measurements for the shuttle dataset.

In view 3 the grid as positioned in view 1 has been rotated about the y axis. The flat side of the hemisphere is now on the right, and the shuttle geometry is embedded in the flow field. The majority of the complexity is on the right side of the image, immediately surrounding the orbiter, external tank and solid rocket booster. The largest task size for this view is 23678 cell faces when using 1024 tiles and 79360 cell faces when using 256 tiles. Using 10 processors to generate a 256^2 image, the time to render using 1024 tiles is 7.8% slower than the time to render using 256 tiles. Using 110 processors, the time to render using 256 tiles is more than three times slower than the time required when using 1024 tiles. As n goes from 10 to 110, the sum of the rendering times on all processors, R_n, increases 3.0% using 256 tiles and 4.9% using 1024 tiles.

Finally, in view 4, the grid is an even further zoomed up and rotated version of view 1. The largest task size for this view is 15084 cell faces when using 1024 tiles and 39533 cell faces when using 256 tiles. Using 10 processors to generate a 256^2 image, the time to render using 1024 tiles is 8.1% slower than the time to render using 256 tiles, giving an estimate of the inefficiency due to the loss of coherence. At 110 processors however, the time to render using 1024 tiles is again half the time required using 256 tiles. As n goes from 10 to 110, the sum of the rendering times on all processors, R_n, increases 5.5% using 256 tiles and 4.7% using 1024 tiles.

6.2.5 Summary of Rendering Phase Results

The algorithm described in chapter 5 has been tested on many variations of dataset size, image size, number of image tiles, viewing specifications, and number of processors. The statistics compiled from these test cases allow several conclusions to be drawn:

- The results consistently show a direct tradeoff between maximizing the utilization of spatial coherence through the use of larger tile sizes, and promoting good scalability by minimizing load imbalance through the use of smaller tile sizes and dynamic task generation. A larger tile size improves utilization of coherency, but also makes load balancing more difficult as n increases. At 110 processors the approach of using smaller tiles to reduce load imbalance is clearly more effective, despite the increases due to loss of coherence. This is because the more effective load balancing provides near-linear speedup as n increases.
- The results show that the storage of active cell faces in local memory minimizes the increase in contention as the number of processors increases. The design of the algorithm is such that each remotely stored cell face projecting to an image tile will be accessed once in order to store locally the edges defining the cell. This approach reduces the number of remote memory references required during a rendering task, thus enhancing the scalability of the algorithm.
- As expected, the specific viewing transformation can have a significant effect on performance results. Some views of the same dataset were shown to have very different load balancing requirements for instance. Zooming in on a dataset causes many cell faces to be clipped during the parallel view sort, reducing the number of cell faces that must be dealt with during the rendering phase. Also, smaller increases in contention can be seen for images making better use of coherence by containing cell faces that are larger with respect to tile size.

- Also as expected, the specific grid can have a significant effect of the performance results. Many structured and unstructured grids are nonconvex and it is quite likely that a viewing specification desired by the user will contain empty space in the image. For example, the first view desired may be one in which the entire grid exactly fills the image. Due to the nonrectilinear shape of the grid, this makes it necessary that there be portions of the image to which nothing projects. In addition, the largest grids currently in use, and the ones most likely to be in use on massively parallel systems, vary widely in their complexity. This was seen to have a significant impact on load balancing requirements.

In analyzing the performance of any parallel rendering algorithm, it is clearly very important to measure performance on a wide variety of datasets, viewing specifications, and any other relevant configuration parameters.

6.2.6 Local Storage of Intersection Lists

Table 6.16 compares execution times in seconds for the two-phase approach (described in section 5.5.4) in which the intersection lists are locally stored, and the single-phase approach (described in section 5.5) in which they are not. Since this algorithm is memory intensive and is only intended to be used on many processors, testing was performed using 100 processors. Since the goal is the fastest possible image updates, only 256^2 images were generated using 1024 image tiles. The results indicate that the primary goal of fast image updates for a changing transfer function are achieved, especially for the more complex datasets where image generation using only the compositing phase is two to three times faster than for the single-phase algorithm.

6.2.7 Reduction of Load Imbalance by Task Ordering

In all of the results given in this chapter, the tasks associated with each tile to be rendered have been ordered by descending size. In this section the effect of ordering the task generation by descending size is examined. Task size is defined to be the number of cell faces which project to a given tile. This information is readily available after the sorting phase of the parallel view sort, and can be used to produce an ordering for the tiles in decreasing order by size. Using this ordering, the tile with the most cell faces projecting to it can be generated first, and so on. Table 6.17 shows the difference in execution time (T_n) and load imbalance (B_n) for execution of 100 processors for all of the images discussed in this chapter. It can be seen that ordering the tasks by size improves load balancing and results in lower execution time in all cases. In the cases where load balancing is most critical (as for the highly complex shuttle dataset), ordering the tasks by size improves performance considerably. It cannot make up for too large a tile size, however, as can be seen by examining the results using 256 tiles. Overall, task ordering provides a very simple expedient for improving load balancing.

Blunt Fin Dataset				
		generate intersections	composite intersections	single-phase approach
View 1 (figure 6.1)	T_{100}	1.7	1.4	2.8
	B_{100}	2.7	24.0	2.9
View 2 (figure 6.2)	T_{100}	1.5	1.4	2.8
	B_{100}	3.9	8.8	4.1

Post Dataset				
		generate intersections	composite intersections	single-phase approach
View 1 (figure 6.3)	T_{100}	2.4	1.5	3.6
	B_{100}	2.5	19.1	3.9
View 2 (figure 6.4)	T_{100}	3.0	1.8	4.4
	B_{100}	3.0	17.5	2.9
View 3 (figure 6.5)	T_{100}	3.6	2.2	5.3
	B_{100}	8.1	8.9	3.1

Delta Wing Dataset				
		generate intersections	composite intersections	single-phase approach
View 1 (figure 6.6)	T_{100}	3.0	1.5	4.0
	B_{100}	1.1	17.6	1.1
View 2 (figure 6.7)	T_{100}	3.1	2.4	4.8
	B_{100}	7.5	18.0	3.1

Shuttle Dataset				
		generate intersections	composite intersections	single-phase approach
View 1 (figure 6.8)	T_{100}	7.6	2.2	6.5
	B_{100}	26.8	32.6	14.8
View 2 (figure 6.9)	T_{100}	11.1	2.5	12.4
	B_{100}	5.8	24.6	1.3
View 3 (figure 6.10)	T_{100}	10.3	4.4	11.5
	B_{100}	18.3	27.1	1.4
View 4 (figure 6.11)	T_{100}	8.7	3.4	10.7
	B_{100}	0.8	30.6	1.2

Table 6.16: Summary of the effects of storing intersection lists locally in order to provide fast image updates for changing transfer functions using 100 processors. For the two-phase approach execution time (T_n) and the percentage of load imbalance (B_n) are given for both the intersection generation phase and the intersection compositing phase. Execution times and load imbalance for the combined single-phase algorithm are also given for comparison. Execution times are in seconds.

Blunt Fin Dataset		256 tiles		1024 tiles	
		tasks ordered	not ordered	tasks ordered	not ordered
View 1 (figure 6.1)	T_{100}	4.3	4.5	2.8	3.0
	B_{100}	48.1	42.4	2.9	4.0
View 2 (figure 6.2)	T_{100}	2.9	3.9	2.8	3.2
	B_{100}	15.2	34.4	4.1	13.4

Post Dataset		256 tiles		1024 tiles	
		tasks ordered	not ordered	tasks ordered	not ordered
View 1 (figure 6.3)	T_{100}	4.9	5.7	3.6	3.6
	B_{100}	38.3	46.4	3.9	5.0
View 2 (figure 6.4)	T_{100}	4.8	6.0	4.4	4.4
	B_{100}	22.1	37.3	2.9	4.6
View 3 (figure 6.5)	T_{100}	5.2	5.8	5.3	5.3
	B_{100}	11.6	20.6	3.1	3.7

Delta Wing Dataset		256 tiles		1024 tiles	
		tasks ordered	not ordered	tasks ordered	not ordered
View 1 (figure 6.6)	T_{100}	6.7	6.8	4.0	4.5
	B_{100}	44.5	45.8	1.1	13.2
View 2 (figure 6.7)	T_{100}	5.3	8.2	4.8	6.0
	B_{100}	19.4	47.6	3.1	24.0

Shuttle Dataset		256 tiles		1024 tiles	
		tasks ordered	not ordered	tasks ordered	not ordered
View 1 (figure 6.8)	T_{100}	19.5	19.9	6.5	7.7
	B_{100}	73.5	75.2	14.8	30.9
View 2 (figure 6.9)	T_{100}	24.9	27.8	12.4	16.6
	B_{100}	54.8	60.2	1.3	27.9
View 3 (figure 6.10)	T_{100}	32.9	35.9	11.5	15.7
	B_{100}	68.6	71.2	1.4	28.9
View 4 (figure 6.11)	T_{100}	19.7	21.8	10.7	11.7
	B_{100}	49.8	54.9	1.2	9.5

Table 6.17: Summary of the effects of ordering task generation by task size (defined as the number of cell faces which project into a given tile) on load imbalance (B_{100}) and execution times in seconds (T_{100}) for $n = 100$.

Blunt Fin Dataset						
	line update	shade	intersect	pix update	tile init	other
View 1	35	25	16	9	6	9
View 2	27	31	18	10	5	9

Post Dataset						
	line update	shade	intersect	pix update	tile init	other
View 1	43	21	14	9	5	8
View 2	42	21	14	10	6	7
View 3	39	23	15	9	7	7

Delta Wing Dataset						
	line update	shade	intersect	pix update	tile init	other
View 1	44	18	12	9	10	6
View 2	33	26	17	10	7	7

Shuttle Dataset						
	line update	shade	intersect	pix update	tile init	other
View 1	42	16	16	9	12	6
View 2	50	9	15	13	9	4
View 3	39	17	20	11	8	5
View 4	36	12	29	9	9	5

Table 6.18: Profile summary for all images. The percentage of rendering time spent on each of the most time-consuming functions is given.

6.2.8 Execution Profile

The execution of the algorithm (described in chapter 5) on each test image has been profiled using `gprof`. The typical order of importance was:

- The `scanline_update` function which maintains the `y_active` list and creates the x-bucket sort. The most time-consuming parts of this function are the updating of active edges, and setting up new edge records when faces become active.
- The `shade` function which performs the integral approximation and compositing steps.
- The `intersect` function which maintains the span records and generates intersections.
- The `pixel_update` function which maintains the `x_active` list. This includes the sort of the records in each bucket of the x-bucket sort.
- The `tile_init` function sets up the y-bucket sort of the cell faces that project to the tile.
- Other requirements, the largest portion of which is made up of various memory management tasks.

In several cases the order of importance varied slightly. For example, view 2 of the blunt fin dataset is the only test case in which shading took more time than scanline updating. In three out of four of the shuttle images, intersection generation was more time consuming than shading. The percentages of time spent for each of these functions are summarized in table 6.18.

View 1 - 256^2 image - see figure 6.1						
	256 tiles			1024 tiles		
n	xform	sort	init	xform	sort	init
10	0.16	0.75	0.15	0.15	0.87	0.24
20	0.08	0.41	0.09	0.08	0.48	0.14
40	0.06	0.26	0.06	0.09	0.26	0.1
60	0.18	0.25	0.08	0.12	0.2	0.12
80	0.15	0.16	0.08	0.14	0.19	0.15
100	0.2	0.15	0.11	0.17	0.17	0.17
110	0.21	0.13	0.11	0.18	0.14	0.19

View 2 - 256^2 image - see figure 6.2						
	256 tiles			1024 tiles		
n	xform	sort	init	xform	sort	init
10	0.15	0.62	0.08	0.15	0.71	0.16
20	0.08	0.33	0.05	0.08	0.36	0.09
40	0.09	0.18	0.05	0.05	0.21	0.08
60	0.26	0.16	0.07	0.1	0.16	0.07
80	0.1	0.11	0.07	0.11	0.12	0.1
100	0.16	0.1	0.08	0.13	0.13	0.11
110	0.13	0.1	0.16	0.2	0.2	0.16

Table 6.19: Parallel view sort execution times (T_n) in seconds for the blunt fin dataset for two different tile sizes. Generated image size is 256^2.

6.3 Timing Results for the Parallel View Sort

It was found to be much more difficult to obtain consistent measurements for the parallel view sort than for the rendering phase. This is probably due to three things: the small task size, the overhead of going parallel three times, and the atomic updates required for initializing the buckets. For the smaller datasets there is not enough work in the applying the viewing transformation to each grid node to provide any speedup for this phase as n gets large. It could be combined with the sorting phase at the expense of transforming each grid node three times. Alternatively, a method of task generation in which the overhead of going parallel is reduced (i.e. a "split-join" as opposed to a "fork-join" approach [PWD93, BGWW91]) would reduce the inefficiency of having three small task phases. The sorting phase did see a speedup for all datasets, although it certainly tails off by 110 processors and is not close to being linear. This is most likely due to the atomic updates required to the shared bucket counts, as well as the small task size. The same can be said for the bucket initialization phase. Table 6.19 gives execution times (T_n) for each phase of the parallel view sort for both views of the blunt fin dataset, and using both 256 and 1024 image tiles. The pattern shown in these results is representative of that seen for all three of the smaller datasets. Table 6.20 gives representative execution times (T_n) for the parallel view sort for all of the datasets tested.

Blunt Fin Dataset		256 tiles			1024 tiles		
		xform	sort	init	xform	sort	init
View 1 (figure 6.1)	$n = 10$	0.16	0.75	0.15	0.15	0.87	0.24
	$n = 100$	0.2	0.15	0.11	0.17	0.17	0.17
View 2 (figure 6.2)	$n = 10$	0.15	0.62	0.8	0.15	0.71	0.16
	$n = 100$	0.16	0.1	0.08	0.13	0.13	0.11
Post Dataset		**256 tiles**			**1024 tiles**		
		xform	sort	init	xform	sort	init
View 1 (figure 6.3)	$n = 10$	0.44	1.79	0.28	0.39	1.85	0.36
	$n = 100$	0.41	0.28	0.15	0.46	0.33	0.29
View 2 (figure 6.4)	$n = 10$	0.44	2.05	0.34	0.39	2.03	0.48
	$n = 100$	0.28	0.28	0.43	0.28	0.33	0.28
View 3 (figure 6.5)	$n = 10$	0.44	1.95	0.39	0.39	2.06	0.54
	$n = 100$	0.29	0.28	0.54	0.26	0.33	0.25
Delta Wing Dataset		**256 tiles**			**1024 tiles**		
		xform	sort	init	xform	sort	init
View 1 (figure 6.6)	$n = 10$	0.89	3.62	0.42	0.89	3.79	0.53
	$n = 100$	0.61	0.5	0.25	0.57	0.54	0.39
View 2 (figure 6.7)	$n = 10$	0.89	3.31	0.27	0.88	3.5	0.39
	$n = 100$	0.23	0.61	0.53	0.23	0.66	0.78
Shuttle Dataset		**256 tiles**			**1024 tiles**		
		xform	sort	init	xform	sort	init
View 1 (figure 6.8)	$n = 10$	3.54	12.23	0.48	3.3	12.25	0.68
	$n = 100$	2.24	1.46	0.43	2.1	1.52	0.39
View 2 (figure 6.9)	$n = 10$	3.35	14.84	1.64	3.39	14.74	2.0
	$n = 100$	0.53	1.55	0.41	0.68	1.6	0.69
View 3 (figure 6.10)	$n = 10$	3.2	12.4	0.98	3.18	12.92	1.43
	$n = 100$	0.43	1.4	0.35	0.46	1.47	0.64
View 4 (figure 6.11)	$n = 10$	3.2	12.42	1.03	3.22	12.97	1.33
	$n = 100$	0.47	1.38	0.4	0.46	1.48	0.73

Table 6.20: Summary of execution times (T_n) in seconds for the parallel view sort.

7. Conclusions

A scalable approach to parallel direct volume rendering of structured and unstructured computational grids has been presented. The algorithm is general enough to handle non-convex grids and cells, grids with voids, grids constructed from multiple grids (multi-block grids), and embedded geometrical primitives. The algorithm is designed for a highly parallel MIMD architecture which features both local memory, and shared memory with non-uniform access times. A variation of the algorithm which provides fast image updates for a changing transfer function has also been presented. The scalability of the algorithm is based on the use of an image-space algorithm which does not require inter-processor synchronization for the compositing step in the rendering process, and the efficient local storage of portions of the volume that are active for any given rendering task. The image is broken into tiles and dynamic task generation is used to generate rendering tasks. The tasks are ordered from largest to smallest, as defined by the number of cell faces projecting to each one, for further improvement in the load balancing. Each task makes use of an efficient algorithm utilizing coherence and performing efficient local storage of active cell faces in order to reduce remote memory accesses and improve scalability.

The algorithm has been tested on many variations of dataset size, image size, number of image tiles, viewing specifications, and number of processors. The statistics compiled from these test cases allow several conclusions to be drawn:

- The results consistently show a direct tradeoff between maximizing the utilization of spatial coherence through the use of larger tile sizes, and promoting good scalability by minimizing load imbalance through the use of smaller tile sizes and dynamic task generation. A larger tile size improves utilization of coherency, but also makes load balancing more difficult as n increases. At 110 processors the approach of using smaller tiles to reduce load imbalance is clearly more effective, despite the increases due to loss of coherence. This is because the more effective load balancing provides near-linear speedup as n increases. It may be possible to obtain the efficiency gained through the use of large tile sizes while still maintaining good load balancing as n grows large, using a task-adaptive approach to task generation [Whi92, NL92]. An efficient method of sharing a portion of a remaining task (tile to be rendered), without incurring any loss of coherency, would need to be developed.
- The results show that the storage of active cell faces in local memory minimizes the increase in contention as the number of processors increases. The design of the algorithm is such that each remotely stored cell face projecting to an image tile will be accessed once in order to store locally the edges defining the cell. This approach reduces the number of remote memory references required during a rendering task, thus enhancing the scalability of the algorithm.
- As expected, the specific viewing transformation can have a significant effect on performance results. Some views of the same dataset were shown to have very different load balancing requirements for instance. Zooming in on a dataset causes many cell faces to be clipped during the parallel view sort, reducing the number of cell faces that must be dealt with during the rendering phase. Also, smaller increases in contention

can be seen for images making better use of coherence by containing cell faces that are larger with respect to tile size.

- Also as expected, the specific grid can have a significant effect of the performance results. Many structured and unstructured grids are nonconvex and it is quite likely that a viewing specification desired by the user will contain empty space in the image. For example, the first view desired may be one in which the entire grid exactly fills the image. Due to the nonrectilinear shape of the grid, this makes it necessary that there be portions of the image to which nothing projects. In addition, the largest grids currently in use, and the ones most likely to be in use on massively parallel systems, vary widely in their complexity. This was seen to have a significant impact on load balancing requirements.

In analyzing the performance of any parallel rendering algorithm, it is clearly very important to measure performance on a wide variety of datasets, viewing specifications, and any other relevant configuration parameters.

A distributed graphical user interface which is used to control the remotely executing volume renderer has also been presented. The entire system has been demonstrated to allow the user to interactively explore very large datasets from a remote location. The design of the entire system is such that the volume renderer could communicate with a simultaneously executing simulation on the massively parallel host. This approach could lead to the ability to steer a simulation based on visual feedback of its progress.

There are several areas that warrant further investigation. Techniques for task generation that take advantage of coherence, but can achieve good load balancing would increase the overall efficiency of the algorithm. General methods for doing so remain to be developed. It would be interesting to implement the algorithm presented here on a machine with higher remote memory latency and examine the performance. The ability to automatically generate transfer functions based on the contents of the dataset being examined would greatly benefit all users of volume rendering. Finding a good transfer function is currently one of the most difficult tasks in utilizing a direct volume renderer to visualize the contents of a dataset. By far the most exciting work left to be done is to examine the issues involved in closely coupling a highly parallel executing simulation with a visualization algorithm such as the one presented here. As highly parallel architectures and software evolve, I believe such a paradigm will be the research environment of choice for computational scientists in many fields.

References

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[Ake93] Kurt Akeley. RealityEngine Graphics. *Computer Graphics*, pages 109–116, 1993. Proceedings of SIGGRAPH '93.

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

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

[BGWW91] Eugene D. Brooks III, Brent C. Gorda, Karen H. Warren, and Tammy S. Welcome. Split-Join and Message Passing Programming Models on the BBN TC2000. In *The 1991 MPCI Yearly Report: The Attack of the Killer Micros*, pages 6–11. Lawrence Livermore National Laboratory, March 1991. UCRL-ID-107022.

[BL90] Andrew Burke and Wm Leler. Parallelism and Graphics: an Introduction and Annotated Bibliography. In *SIGGRAPH Course Notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[Bli82a] J. F. Blinn. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. In *Proceedings of SIGGRAPH '82*, pages 21–29, 1982.

[Bli82b] James Blinn. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.

[Blo88] Jules Bloomenthal. Polygonization of Implicit Surfaces. *Computer-Aided Geometric Design*, 5:341–355, 1988.

[BP90] Didier Badouel and Thierry Priol. An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures. In *Proceedings of the Fifth Eurographics Workshop on Graphics Hardware*, September 1990.

[Cha90] Judith Ann Challinger. Object-Oriented Rendering of Volumetric and Geometric Primitives. Master's thesis, University of California, Santa Cruz, 1990.

[Cha91] Judy Challinger. Parallel Volume Rendering on a Shared-Memory Multiprocessor. Technical Report UCSC-CRL-91-23, University of California, Santa Cruz, 1991.

[Cha92] Judy Challinger. Parallel Volume Rendering for Curvilinear Volumes. In *Proceedings of the Scalable High Performance Computing Conference*, pages 14–21. IEEE Computer Society Press, April 1992.

[CLL$^+$88] Harvey E. Cline, William E. Lorensen, Sigwalt Ludke, Carl R. Crawford, and Bruce C. Teeter. Two Algorithms for the Reconstruction of Surfaces from Tomograms. *Medical Physics*, 15(3):320–327, June 1988.

[CM92] Brian Corrie and Paul Mackerras. Parallel Volume Rendering and Data Coherence on the Fujitsu AP1000. Technical Report TR-CS-92-11, Department of Computer Science, The Australian National University, 1992.

[Cra92] Cray Research, Inc. *CRAY T3D Software Overview Technical Note*, sn-2505 1.0 edition, 1992.

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

[CS78] H.N. Christianson and T. W. Sederburg. Conversion of Complex Contour Line Definitions into Polygonal Element Mosaics. In *Proceedings of SIGGRAPH '78*, pages 187–192, 1978.

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

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume Rendering. *Computer Graphics*, 22(4):65–74, 1988. Proceedings of SIGGRAPH '88.

[DH92] John Danskin and Pat Hanrahan. Fast Algorithms for Volume Ray Tracing. In *1992 Workshop on Volume Visualization*, pages 91–98. ACM, 1992.

[DN93] Michael F. Deering and Scott R. Nelson. Leo: A System for Cost Effective 3D Shaded Graphics. *Computer Graphics*, pages 101–108, 1993. Proceedings of SIGGRAPH '93.

[DS84] Mark Dippé and John Swensen. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *Computer Graphics*, 18(3):149–158, 1984. Proceedings of SIGGRAPH '84.

[DT81] Louis J. Doctor and John G. Torborg. Display Techniques for Octree-Encoded Objects. *IEEE Computer Graphics and Applications*, pages 29–38, July 1981.

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

[Elv92] T. Todd Elvins. Volume Rendering on a Distributed Memory Parallel Computer. In *Visualization '92*, pages 93–98. IEEE, October 1992.

[FD82] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, 1982.

[FGH87] K. Fujii, S. Gavali, and T.L. Holst. Vortical Flow over a Delta Wing Computation. In *5th International Conf. on Numerical Methods in Laminar and Turbulent Flow*, July 1987. Montreal, Quebec.

[FK90] Wm. Randolph Franklin and Mohan S. Kankanhalli. Parallel Object-Space Hidden Surface Removal. *Computer Graphics*, 24(4):87–94, 1990. Proceedings of SIGGRAHP '90.

[FKU77] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal Surface Reconstruction for Planar Contours. *Communications of the ACM*, 20, 1977.

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

[FPE+89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics*, 23(3):79–88, 1989. Proceedings of SIGGRAPH '89.

[Fuc77] Henry Fuchs. Distributing a Visible Surface Algorithm over Multiple Processors. In *Proceedings of ACM*, pages 449–451, October 1977.

[Gar90] Michael P. Garrity. Raytracing Irregular Volume Data. *Computer Graphics*, 24(5):35–40, November 1990. Proceedings of the San Diego Workshop on Volume Visualization.

[GB91] Brent C. Gorda and Eugene D. Brooks III. The MPCI Gang Scheduler. In *The 1991 MPCI Yearly Report: The Attack of the Killer Micros*, pages 183–187. Lawrence Livermore National Laboratory, March 1991. UCRL-ID-107022.

[GD82] S. Ganapathy and T. G. Dennehy. A New General Triangulation Method for Planar Contours. In *Proceedings of SIGGRAPH '82*, pages 69–75, 1982.

[Gie92] Christopher Giertsen. Volume Visualization of Sparse Irregular Meshes. *IEEE Computer Graphics and Applications*, 12(2):40–48, March 1992.

[Gla89] Andrew S. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press Limited, 1989.

[GO89] David S. Goodsell and Arthur J. Olson. Molecular Applications of Volume Rendering and 3-D Texture Maps. In *Proceedings of the Chapel Hill Workshop on Volume Visualization.* Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[GRB$^+$85] Samuel M. Goldwasser, R. Anthony Reynolds, Ted Bapty, David Baraff, John Summers, David A. Talton, and Ed Walsh. Physician's Workstation with Real-Time Performance. *IEEE Computer Graphics and Applications*, 5(12):44–56, December 1985.

[GT92] Christopher Giertsen and Allan Tuchman. Fast Volume Rendering with Embedded Geometric Primitives. In T. L. Kunii, editor, *Visual Computing - Integrating Computer Graphics and Computer Vision*, pages 253–271. Springer Verlag, 1992.

[Gus88] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[HB85] C. H. Hung and P. G. Buning. Simulation of Blunt-Fin Induced Shock Wave and Turbulent Boundary Layer Interaction. *Journal of Fluid Mechanics*, 154:163–185, 1985.

[HB86] Donald Hearn and M. Pauline Baker. *Computer Graphics.* Prentice-Hall, Inc., 1986.

[Hen92] Henry Burkhardt III. Announcing the KSR1 Supercomputer. Article 2946 of comp.parallel, Febuary 1992.

[HM90] Dave Helmbold and Charlie McDowell. Modeling Speedup(n) greater than n. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):250–256, April 1990.

[Hor93] R. Michael Hord. *Parallel Supercomputing in MIMD Architectures.* CRC Press, Inc., 1993.

[HS89] William Hibbard and David Santek. Interactivity is the Key. In *Proceedings of the Chapel Hill Workshop on Volume Visualization.* Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[Jac88] D. Jackél. Reconstructing Solids from Tomographic Scans - The PARCUM II System. In *Advances in Computer Graphics Hardware II*, pages 101–109. Springer International, 1988.

[JM89] E. Ruth Johnson and Charles E. Mosher. Integration of Volume Rendering and Geometric Graphics. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*. Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[KB88] Arie Kaufman and Reuven Bakalash. Memory and Processor Architecture for 3D Voxel-Based Imagery. *IEEE Computer Graphics and Applications*, 8(11):10–23, November 1988.

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

[KH84] James T. Kajiya and B.P. Von Herzen. Ray Tracing Volume Densities. *Computer Graphics*, 18(3):165–174, 1984. Proceeding of SIGGRAPH '84.

[KMS+92] Jim Kaba, Jim Matey, Gordon Stoll, Herb Taylor, and Pat Hanrahan. Interactive Terrain Rendering and Volume Visualization on the Princeton Engine. In *Visualization '92*, pages 349–355. IEEE, October 1992.

[Koy92] Koji Koyamada. Fast Traversal of Irregular Volumes. In T. L. Kunii, editor, *Visual Computing - Integrating Computer Graphics and Computer Vision*, pages 295–312. Springer Verlag, 1992.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, 1987. Proceedings of SIGGRAPH '87.

[Lev88] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[Lev89a] Marc Levoy. Design for a Real-Time High-Quality Volume Rendering Workstation. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 85–92. Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

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

[Lev90] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[LH91] David Laur and Pat Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Computer Graphics*, 25(4):285–288, 1991. Proceedings of SIGGRAPH '91.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford Dash Multiprocessor. *Computer*, pages 63–79, March 1992.

[Luc92] Bruce Lucas. A Scientific Visualization Renderer. In *Visualization '92*, pages 227–233. IEEE, October 1992.

[LW90] Marc Levoy and Ross Whitaker. Gaze-Directed Volume Rendering. *Computer Graphics*, 24(2):217–223, 1990. Proceedings of 1990 Symposium on Interactive 3D Graphics.

[Mal93] Tom Malzbender. Fourier Volume Rendering. *ACM Transactions on Graphics*, 12(3):233–250, 1993.

[Max86] Nelson Max. Light Diffusion Through Clouds and Haze. *Computer Vision Graphics, and Image Processing*, 33:280–292, 1986.

[Mea82] D. Meagher. Efficient Synthetic Image Generation of Arbitrary 3-D Objects. In *Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing*, June 1982.

[Mea85] Dr. Donald J. Meagher. Applying Solids Processing Methods to Medical Planning. In *Proceedings of NCGA*, pages 101–109. National Computer Graphics Association, April 1985.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Computer Graphics*, 24(5), 1990. Proceedings of the San Diego Workshop on Volume Visualization.

[MPS92] C. Montani, R. Perego, and R. Scopigno. Parallel Volume Visualization on a Hypercube Architecture. In *1992 Workshop on Volume Visualization*, pages 9–16. ACM, 1992.

[MS90] F. W. Martin, Jr. and J. P. Slotnick. Flow Computations for the Space Shuttle in Ascent Mode Using Thin-Layer Navier-Stokes Equations. In P.A. Henne, editor, *Progress in Astronautics and Aeronautics, Vol. 125*, pages 863–886. American Institute of Aeronautics and Astronautics, Washington, D.C., 1990.

[Neu92] Ulrich Neumann. Interactive Volume Rendering on a Multicomputer. In *1992 Symposium on Interactive 3D Graphics*, pages 87–99. ACM, March 1992.

[NHK$^+$85] Hitoshi Nishimura, Makato Hirai, Toshiyuki Kawai, Toru Kawata, Isao Shirakawa, and Koichi Omura. Object Modelling by Distribution Function and a Method of Image Generation. *Trans. IEICE*, J-68D(4):718–725, 1985. in Japanese.

[NL92] Jason Nieh and Marc Levoy. Volume Rendering on Scalable Shared-Memory MIMD Architectures. In *1992 Workshop on Volume Visualization*, pages 17–24. ACM, 1992.

[Nor82] Alan Norton. Generation and Display of Geometric Fractals in 3-D. In *Proceedings of SIGGRAPH '82*, 1982.

[OUT85] Toshiaki Ohashi, Tetsuya Uchiki, and Mario Tokoro. A Three-Dimensional Shaded Display Method for Voxel-Based Representations. In *Proceedings of EUROGRAPHICS 1985*, pages 221–232. National Computer Graphics Association, September 1985.

[PD84] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, 1984. Proceedings of SIGGRAPH '84.

[PFTV86] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes - The Art of Scientific Computing*. Cambridge University Press, 1986.

[plo89] PLOT3D User's Manual. National Aeronautics and Space Administration, Fluid Dynamics Division, NASA Ames Research Center, 1989.

[PWD93] Richard J. Procassini, Scott R. Whitman, and William P. Dannevik. Porting a Global Ocean Model Onto a Shared-Memory Multiprocessor: Observations and Guidelines. *Journal of Supercomputing*, 1993. to appear.

[Ram91] Shankar Ramamoorthy. Curvilinear Grids. In *Course Notes 8: State of the Art in Volume Visualization*. ACM Siggraph '91 Conference, 1991.

[RKK86] S. E. Rogers, D. Kwak, and U. K. Kaul. A Numerical Study of Three-Dimensional Incompressible Flow Around Multiple Posts, 1986. AIAA Paper 86-0353, Reno, Nevada.

[RW92] Shankar Ramamoorthy and Jane Wilhelms. An Analysis of Approaches to Ray-Tracing Curvilinear Grids. Technical Report UCSC-CRL-92-07, University of California, Santa Cruz, 1992.

[Sab88] Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Computer Graphics*, 22(4):51–58, 1988. Proceedings of SIGGRAPH '88.

[SG91] Georgios Sakas and Matthias Gerth. Sampling and Anti-Aliasing of Discrete 3-D Volume Density Textures. In *Proceedings of Eurographics '91*, pages 87–102, 1991.

[SH92] Georgios Sakas and Jochen Hartig. Interactive Visualization of Large Scalar Voxel Fields. In *Visualization '92*, pages 29–36. IEEE, October 1992.

[SK90] Don Speray and Steve Kennon. Volume Probes: Interactive Data Exploration on Arbitrary Grids. *Computer Graphics*, 24(5):5–12, November 1990. Proceedings of the San Diego Workshop on Volume Visualization.

[SM92] Reese L. Sorenson and Karen McCann. Grapevine: Grids About Anything by Poisson's Equation in a Visually Interactive Networking Environment. In *Computational Aerosciences Conference Compendium of Abstracts*, 1992. NASA Ames Research Center.

[SS89a] Carlo H. Sequin and Eliot K. Smyrl. Parameterized Ray Tracing. *Computer Graphics*, 23(3):307–314, July 1989. Proceedings of SIGGRAPH '89.

[SS89b] J. A. Sethian and James B. Salem. Animation of Interactive Fluid Flow Visualization Tools on a Data Parallel Machine. *The International Journal of Supercomputer Applications*, 3(2):10–39, 1989.

[SS91] Peter Schröder and James B. Salem. Fast Rotation of Volume Data on Data Parallel Architectures. In *Course Notes 8: State of the Art in Volume Visualization*. ACM Siggraph '91 Conference, 1991.

[SS92] Peter Schöder and Gordon Stoll. Data Parallel Volume Rendering as Line Drawing. In *1992 Workshop on Volume Visualization*, pages 25–32. ACM, 1992.

[ST90] Peter Shirley and Alan Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics*, 24(5):63–70, November 1990. Proceedings of the San Diego Workshop on Volume Visualization.

[TG88] Filippo Tampieri and Donald Greenberg. Experimental Distributed Processing System for Global Illumination Algorithms. Technical report, Cornell University, 1988.

[TL93] Takashi Totsuka and Marc Levoy. Frequency Domain Volume Rendering. In *Proceedings of SIGGRAPH '93*, pages 271–278, 1993.

[UK88] Craig Upson and Michael Keeler. VBUFFER: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, 1988. Proceedings of SIGGRAPH '88.

[VFR92] Guy Vézina, Peter A. Fletcher, and Philip K. Robertson. Volume Rendering on the MasPar MP-1. In *1992 Workshop on Volume Visualization*, pages 3–8. ACM, 1992.

[VW93] Allen Van Gelder and Jane Wilhelms. Rapid Exploration of Curvilinear Grids Using Direct Volume Rendering. In *Proceedings of Visualization '93*. IEEE, October 1993. to appear.

[WCA$^+$90] Jane Wilhelms, Judy Challinger, Naim Alper, Shankar Ramamoorthy, and Arsi Vaziri. Direct Volume Rendering of Curvilinear Volumes. *Computer Graphics*, 24(5):41–47, November 1990. Proceedings of the San Diego Workshop on Volume Visualization.

[WCH$^+$87] Karl-Heinz A. Winkler, Jay W. Chalmers, Stephen W. Hodson, Paul R. Woodward, and Norman J. Zabusky. A Numerical Laboratory. *Physics Today*, pages 28–37, October 1987.

[Wes89] Lee Westover. Interactive Volume Rendering. In *Conference Proceedings of the Chapel Hill Workshop on Volume Visualization*. Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[Wes90] Lee Westover. Footprint Evaluation for Volume Rendering. *Computer Graphics*, 24(4), 1990. Proceedings of SIGGRAPH '90.

[WH79] Thomas Wright and John Humbrecht. ISOSURF - An Algorithm for Plotting Iso-Valued Surfaces of a Function of Three Variables. In *Proceedings of SIGGRAPH '79*, pages 182–189, 1979.

[Whi92] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. Jones and Bartlett, 1992.

[Wil92a] Peter L. Williams. Interactive Splatting of Nonrectilinear Volumes. In *Visualization '92*, pages 37–44. IEEE, October 1992.

[Wil92b] Peter L. Williams. Visiblity Ordering Meshed Polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, April 1992.

[Wil92c] Peter Lawrence Williams. *Interactive Direct Volume Rendering of Curvilinear and Unstructured Data*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[WM92] Peter L. Williams and Nelson Max. A Volume Density Optical Model. In *1992 Workshop on Volume Visualization*, pages 61–68. ACM, 1992.

[WMW86] G. Wyville, C. McPheeters, and B. Wyville. Data Structures for Soft Objects. *The Visual Computer*, 2(4):227–234, 1986.

[WV91] Jane Wilhelms and Allen Van Gelder. A Coherent Projection Approach for Direct Volume Rendering. *Computer Graphics*, 25(4), 1991. Proceedings of SIGGRAPH '91.

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