5-10-2003

Parzsweep: A Novel Parallel Algorithm for Volume Rendering of Regular Datasets

Lakshmy Ramswamy

PARZSWEEP: A NOVEL PARALLEL ALGORITHM FOR VOLUME RENDERING OF REGULAR DATASETS

By

Lakshmy Ramaswamy

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2003

PARZSWEEP: A NOVEL PARALLEL ALGORITHM FOR

VOLUME RENDERING OF REGULAR DATASETS

By

Lakshmy Ramaswamy

Approved:

Susan M. Bridges Professor of Computer Science and Engineering (Major Professor)	Ricardo Farias Adjunct Assistant Professor of Computer Science and Engineering (Thesis Director)
David Dampier Assistant Professor of Computer Science and Engineering (Committee Member)	Thomas Philip Professor of Computer Science and Engineering (Committee Member)
Susan M. Bridges Professor of Computer Science and Engineering Graduate Coordinator Department of Computer Science and Engineering	A. Wayne Bennett Dean of the College of Engineering

Name: Lakshmy Ramaswamy

Date of Degree: May 10, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Susan Bridges

Director of Thesis: Dr. Ricardo Farias

Title of Study: PARZSWEEP: A NOVEL PARALLEL ALGORITHM FOR VOLUME RENDERING OF REGULAR DATASETS

Pages in Study: 53

Candidate for Degree of Master of Science

The sweep paradigm for volume rendering has previously been successfully applied with irregular grids. This thesis describes a parallel volume rendering algorithm called PARZSweep for regular grids that utilizes the sweep paradigm. The sweep paradigm is a concept where a plane sweeps the data volume parallel to the viewing direction. As the sweeping proceeds in the increasing order of z, the faces incident on the vertices are projected onto the viewing volume to constitute to the image. The sweeping ensures that all faces are projected in the correct order and the image thus obtained is very accurate in its details. PARZSweep is an extension of a serial algorithm for regular grids called RZSweep. The hypothesis of this research is that a parallel version of RZSweep can be designed and implemented which will utilize multiple processors to reduce rendering times. PARZSweep follows an approach called image-based task scheduling or *tiling*. This ap-

proach divides the image space into tiles and allocates each tile to a processor for individual rendering. The sub images are composite to form a complete final image. PARZSweep uses a shared memory architecture in order to take advantage of inherent cache coherency for faster communication between processor. Experiments were conducted comparing RZSweep and PARZSweep with respect to prerendering times, rendering times and image quality. RZSweep and PARZSweep have approximately the same prerendering costs, produce exactly the same images and PARZSweep substantially reduced rendering times. PARZSweep was evaluated for scalability with respect to the number of tiles and number of processors. Scalability results were disappointing due to uneven data distribution.

DEDICATION

To my parents and my sister.

ACKNOWLEDGMENTS

I would like take this opportunity to thank a few people who have contributed towards the success of my masters thesis.

I would first like to extend my gratitude to my co-researcher Gautam Chaudhary, who has always been a strength in this endeavour. I would also like to thank all my friends who have been very supportive of me during my thesis.

I would like to extend heart felt gratitude to my thesis director Dr. Ricardo Farias who has guided me in my research and my life at MSU. I would like to thank my advisor Dr. Susan Bridges without whose guidance I would not have been able to finish my thesis. She has been extremely understanding and supportive during my thesis documentation. I would also like to thank Dr. Joerg Meyer, who has always been there to help me through questions and provide support.

I thank my committee members for their valuable comments on this thesis, and helping me to make my thesis a success.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Most phenomena in the world are inherently three dimensional in nature. It is apparent that visualizing these phenomena in two dimensions will not be intuitive to scientists. Hence, the concept of involving the volume of the object to be visualized is vital. Volume rendering is a computationally expensive technique for visualizing data samples that are three dimensional in nature. This thesis describes previous approaches to volume rendering and a new approach called RZSweep. The basis for RZSweep is the ZSweep algorithm of Farias et al. [13]. The RZSweep algorithm yields good quality images in acceptable times in its current state. The focus of this thesis is a parallel version of the RZSweep algorithm called PARZSweep that has been developed in order to improve performance.

1.1 Volume Rendering

Volume rendering is a visualization method used primarily for medical and scientific 3-dimensional (3D) data where the entire volume of the object is used to create high quality images. The images thus rendered show information about the interior of the data, giving a clearer understanding on the nature of the data. Data for volume rendering is usually collected through high-quality CAT Scan(CT) or Magnetic Resonance Imaging(MRI) scan-

ners. The size of the data sets collected is usually very high, since these scanners acquire very detailed information. Medical imaging, finite element modeling, large-scale geospatial simulations, molecular microscopy, and non-destructive material testing are some of the fields in which the volume data are collected [44]. Each data element is a scalar value representing a volumetric data element (also called a voxel). Data collected typically includes a volume of 256^3 data points. Due to the vast volume of the data collected, volume rendering is a very expensive visualization method. New techniques are being explored to improve the rendering speed.

1.2 Classification of Datasets

Volumetric data can be classified based on its organization as regular or irregular grids. In a regular grid the data points are placed at uniform distances from each other. A regular grid can be visualized as a lattice where the data points are placed at each vertex (voxel) of the grid. An irregular grid has no specific order for placement of data points. The data points are displaced in space at no specific intervals. Volume rendering algorithms are based on the type of grid structure that they handle and are classified as regular grid algorithms or irregular grids algorithm. This research focuses on the development of a parallel algorithm for rendering regular grid data. The method presented here is based on the sweep paradigm.

1.3 Sweep Paradigm

Volume rendering is typically based on one of two techniques: indirect volume rendering (IVR) or direct volume rendering (DVR). In IVR, an intermediate representation of the volume data is created [31]. Direct volume rendering (DVR) uses no intermediate representation of the volume and the data is swept as a whole at a single step without any transitional state [11]. A DVR based algorithm has been developed where the scalar values are rendered using a sweep paradigm. The idea is based on the ZSweep algorithm for irregular grids presented by Farias et al. [13]. In the sweep paradigm, an imaginary plane sweeps the data volume parallel to the viewing direction. As it sweeps the data in increasing order of z, it projects the implicit faces incident on the vertices that the sweep plane encounters. To get the correct resultant image, it is important that the vertices are projected in proper order. Since the data visualized is a regular grid, no auxiliary data structure is maintained to store the order of the vertices. The simplicity of the algorithm is that it exploits the implicit nature of the grid and uses only a heap to sort the vertices encountered during the sweeping process. This has enabled implementation of a version of the algorithm that uses graphics hardware as well as a hardware independent version.

1.4 Parallel Techniques for Volume Rendering

In general, algorithms that use a distributed memory architecture are more scalable than the algorithms based on a serial architecture. However, the complexity of distributed schemes produces problems such as lack of memory coherence, latency, bandwidth, prior-

ity scheduling and data distribution. Since the distributed scheme does not use a common data bank, one processor often needs data from another processor, causing memory access problems. A high degree of cache coherence is required to resolve unnecessary data sharing problems. Many approaches have been taken to solve these problems including large caches and memory pre-fetching techniques. But still, the algorithm must collaborate with the hardware to achieve the best performance [14]. Many popular parallel algorithms have been developed using this scheme and these algorithms have proven to be highly scalable. Another popular approach is to use a shared memory architecture. In a shared memory parallel architecture, the tasks can easily be assigned to processors dynamically by maintaining a common pool of tasks from which available processors claim work to do. Also, each processor can have a local memory cache for its individual memory requirement. Typical problems of bandwidth are avoided since the access to the memory bank is done with bus or switching networks. Algorithms that follow this method are not very complex but the nature of the shared memory architecture limits the scalability of the algorithms. Load balancing strategies are critical to ensure good performance in shared memory parallel schemes.

1.5 Parallelization of RZSweep

In this thesis, a parallel version of the RZSweep algorithm is described that uses a shared memory architecture. The approach is similar to the parallel version of the ZSweep algorithm. Image space partitioning is used to assign jobs to the various processors. Al-

gorithms developed by Nieh and Levoy [34] and the parallel ZSweep algorithm of Farias et al. have successfully used this method (also called *tiling*). This method partitions the screen space into tiles and assigns the tiles to the processors in a dynamic fashion to render the final image. The scheme described preserves the simplicity of the serial algorithm. The implicit regularity of the grid facilitates the use of only image space partitioning to perform the parallelization. A heap is used because the rendering function needs the vertices swept to be in order. However, the need for the octree in the parallel ZSweep [14] is eliminated due to the regularity of the dataset. Hence no explicit partitioning of object space is needed and this improves the space complexity of the algorithm.

The hypothesis of this research is that a parallel algorithm of RZSweep, PARZSweep, can be designed and implemented which will utilize multiple processors to reduce rendering times. Chapter II describes the literature survey conducted in volume rendering algorithms on regular data sets and parallel volume rendering algorithms. The serial algorithm RZSweep is briefly discussed in Chapter III. Chapter IV describes the design, implementation and details of the parallel algorithm PARZSweep. Chapter V describes the experiments conducted to test the efficiency and performance of the parallel algorithm in a shared memory architecture.

CHAPTER II

LITERATURE SURVEY

Volume rendering algorithms for regular grids can be categorized into 4 groups.

1. ray casting [23, 40]
2. splatting [41]
3. shear warp [22]
4. 3D texture mapping [6].

Each of these methods is reviewed below.

Ray casting is an image order volume rendering method where rays are shot through the screen space to intersect the volume data. The intersections with the volume are interpolated to calculate the final color of the pixels of the image. It is a time consuming method since shooting rays through every pixel takes time, but the quality of the resulting image is very accurate. There are two main approaches used with ray casting algorithms.

- The first changes the color and the opacity and is also called pre-direct volume rendering integral [23, 24].
- The second generates the density and gradient attributes for each point, and is called post-direct volume rendering integral [18, 38, 4, 39].

Ray casting involves all the voxels of the data set for generation of the images and hence it is computationally expensive. This is the major drawback of ray casting. There are several

optimization techniques which are used for improving the efficiency. One of the optimizations is encoding coherence in volume data. Many voxels in a dataset are *"empty"*(which means that the opacity value of that voxel is zero). It has been observed that such empty voxels are often found in coherent regions of the dataset. Such regions can be encoded for optimization purposes using *octree hierarchical spatial enumeration*[29], *polygonal representation of bounding surfaces*[15] and *octree representation of bounding surfaces*[16]. Extensive research has been done in applying the technique of traversing and skipping empty space for faster rendering [11, 47]. This is also called Space Leaping [45].

Another common optimization technique in ray casting involves the concept of early ray termination or adaptive ray termination. This idea was first proposed by Whitted [43] in 1980. If a ray strikes an opaque object or if it traverses through a volume for a period of time, then the further contribution of that ray towards the color of the object becomes minimal. Since data volumes are very huge in size, early termination of such rays saves precious time and cost. Most rays are terminated when the opacity reaches a user specified threshold. Ray casting produces high quality accurate results and can be used to create images without specifically outlining the surface geometry.

Splatting, designed by Westover [41], increases the speed of volume rendering over ray casting but sacrifices image quality. It is an approximation algorithm that considers the entire volume as an array of basis functions centered on each data voxel. Each basis function defines a footprint of a simple shape that is projected on the screen following a depth order. The efficiency of the algorithm depends heavily on the complexity of the

shape chosen for such footprints. In one approach, all voxels are projected without any consideration of their associated values. Further speed-ups have been achieved by avoiding the projection of voxels associated with low scalar values. Following the idea of early ray termination used in ray casting, early splat elimination has been applied, avoiding further splatting on opaque regions of the screen. Later, Mueller et al. [33] developed image-aligned splatting to enable animation.

Shear warp, developed by Lacroute and Levoy [22], greatly increased the speed of rendering. Even though it is an approximate algorithm, it is known as the fastest volume-rendering algorithm to date. The algorithm is a hybrid of both image order and object order algorithms. The main idea is to factorize the view planes in 3D slices which are parallel to the volume slices, apply a projection to form the temporary image and to apply a two dimensional shear to obtain the final image. However the quality of the images produced deteriorates as the size of the viewport increases. Other limitations of this algorithm include the memory required to keep the intermediate planes it produces. This memory requirement increases as higher quality images are desired and it is dependent on the amount of texture memory of the graphics hardware. Regardless, it is still a good choice if all requirements are satisfied and approximated images are acceptable. A package named VolPack has been designed at Stanford [1] based on shear warp.

The 3D texture mapping algorithm relies completely on the graphics hardware capabilities. The idea, developed by Cabral [6], is intended solely for non-shaded rendering. Substantial subsequent work has been done and shading capabilities have been added [10, 30].

The textures of the volume are uploaded to the graphics hardware and the hardware does the rasterization to perform the rendering to get the final image. The rendering normally does not check for any early termination criteria. Although the method is very fast due to the availability of the high end graphics hardware, it faces three main bottlenecks:

1. dependence on the graphics hardware which is extremely costly,
2. limitation to the texture memory, and
3. dependence on the swap buffer for swapping textures in and out.

The plane sweep paradigm has been widely discussed in the area of computational geometry [35]. The sweeping paradigm has been used in some algorithms, primarily applied to irregular grids datasets. Girsten [17] was the first to use the concept in volume rendering. Yagel [46] and Silva [37] furthered the work on sweeping algorithms. The most recent work based on the sweep paradigm was the ZSweep algorithm developed by Farias [13]. In ZSweep, the sweep plane is a virtual plane that sweeps the data volume in a direction parallel to the viewing plane. The algorithm projects faces of cells, incident on the vertices that are encountered by the sweep plane. Certain data structures like a vertex array and a cell array are used to avoid double projection of internal faces and to assure correctness in the order of projection. At a given time information of only a few slices of the data set needs to be stored. This reduces the memory needs of the algorithm and hence it is memory efficient. The algorithm is hardware independent and has achieved good speedup compared to its predecessor, the lazy sweep algorithm [37].

The lazy sweep algorithm precedes ZSweep in using the sweep paradigm. Since the algorithm has been implemented for irregular grids, it is not directly relevant to RZSweep.

However, since the algorithm uses the sweep paradigm which is the basis of this work, it is worth a mention. The main difference in the sweep paradigm used in lazy sweep as opposed to ZSweep is the direction of sweep. In lazy sweep, the sweep plane is parallel to the x-z plane whereas in ZSweep and RZSweep, the sweep plane is orthogonal to the viewing(x-z) direction [37].

The following sections describe the literature on parallel approaches for volume-rendering algorithms. There have been many approaches for parallelizing ray-casting algorithms specifically due to the simplicity of the algorithm. The aspect that has received the most attention is minimizing the redistribution cost of the volume data. The typical solution simply distributes the volume data into the processing nodes and lets each of the processing nodes generate a partial image. Each node will have part of the data, which it will use to generate images for all the frames and orientations. The final image will be a composite image of all the partial images generated by the processors. The final composition can also be made parallel or can be performed by the master processor [19, 26, 27, 32].

Better algorithms have been developed which exploit the nuances of the parallel approach to speed up ray casting. Tasks are created which are obtained by partitioning the data space into square tiles or adjacent scan lines [7]. These tasks are then assigned to processors which perform their respective rendering routines. Several constraints have been placed on these job queues to make the rendering faster. One constraint is to keep the queue in a sorted order so that the allocation of jobs to the processors will be in an orderly fashion until the queue is exhausted [7]. Neih and Levoy [34] and Whitman [42] proposed

another method where there is a sharing of the jobs between processors. If processor A has finished the task assigned to it, it would then assist processor B by accepting smaller jobs from processor B. An interesting approach is followed by Corrie and Mackerras [9] where time is a key factor. For a processor, a time stamp is the time taken for that processor to finish its task. The method followed is that if the processor fails to finish the assigned task before the time stamp expires, then the remaining task is taken as a whole job, re-divided and re-assigned to the various processors.

Shear warp [22] is another algorithm to which various researchers have devoted time and effort to make it faster and better using parallel approaches. Both distributed and shared memory architectures have been used. Amin et al. [3] and Sano et al. [36] have done notable work using a distributed memory architecture with shear warp. Amin et al. [3] used adaptive load balancing techniques coupled with partitioning in the sheared space for parallelization. Sano et al. [36], on the other hand, used sheared space partitioning to create volumes, which were parallel to the intermediate image plane in the shear warp algorithm.

Algorithms involving shared memory architectures include the parallel shear warp by Lacroute [21]. Lacroute follows the approach of using optimized codes for improving frame rates. Since optimized codes use complex preprocessing and have high data dependency, they cannot be mapped onto simple SIMD processors. Hence there is dependency on high performance MIMD multiprocessors. Three major factors are attributed to the increase in frame rates in shear warp [21]. First is the use of a fast serial algorithm for

shear warp that is a hybrid order algorithm using both the volume and the image space in the rendering process. The parallelization described uses image-based partitioning of the data and exploits the optimizations in the algorithm to enhance the performance [22].

Splatting is an object order projection based volume rendering technique. The main idea involves projecting the volume data over a three dimensional kernel and then accumulating the projections over the image planes with a two dimensional kernel called *splats*. The advantage of splatting over other algorithms like shear warp and ray casting is the flexibility to choose the reconstruction kernels. There are several improvements in splatting since Li and Whitman [25]. Parallel approaches have been developed for speeding up the rendering process. The job of data distribution along the processors is done using axis aligned planes (also called slices) [12] or blocks to processing nodes [25]. The images are then rendered individually in parallel, and then composited together to get the final image. The work of Machiraju and Yagel [28] assigns the data volume to the processors in batches or sub-volumes. The processors each get a batch of volume to process and the final image is then again composite in parallel in a depth order fashion. Certain hierarchical data structures like a k-d tree have been used to accelerate the rendering procedure. Occlusion culling has been used by Huang et al. [20] to speed up the rendering process. Also previous splatting methods have artifacts because of the separation of volume integration and volume reconstruction. Huang's method [20] overcomes those aberrations by using an image-aligned sheet where the voxels are collected in planes parallel to the image plane. Data distribution to the processors is done by assigning the data closest to the image plane.

Then an occlusion map is applied to cull data that is inconspicuous and has minimal effect to the resultant image.

A lot of additional research has been done on distributed SIMD and MIMD architectures, but those papers are not mentioned here because they have limited relevance to work with a shared memory architecture. The following chapters first describe the RZSweep algorithm, then the parallel implementation, PARZSweep. The hypothesis is that the parallel version provides a significant gain in performance versus the serial version for typical volume datasets with identical image quality.

CHAPTER III

RZSWEEP

The research described in this paper is the development of a parallel version of RZSweep. The parallel algorithm makes use of a shared memory architecture and is based on an image partitioning technique. An introduction to the serial RZSweep algorithm is provided as background.

3.1 Significance of Work

The sweep paradigm is a unique concept of traversing an entire dataset in an orderly fashion. The technique has been explored for volume rendering of irregular grids by several algorithms in the past. RZSweep is an attempt to explore the capacity of the sweep paradigm for sweeping the regular data sets. Hence the algorithm is a novel approach for volume rendering of regular data sets. Since PARZSweep is a parallel version of RZSweep, the following chapter briefly describes the serial RZSweep algorithm. More details of RZSweep can be found in [8].

3.2 RZSweep Algorithm

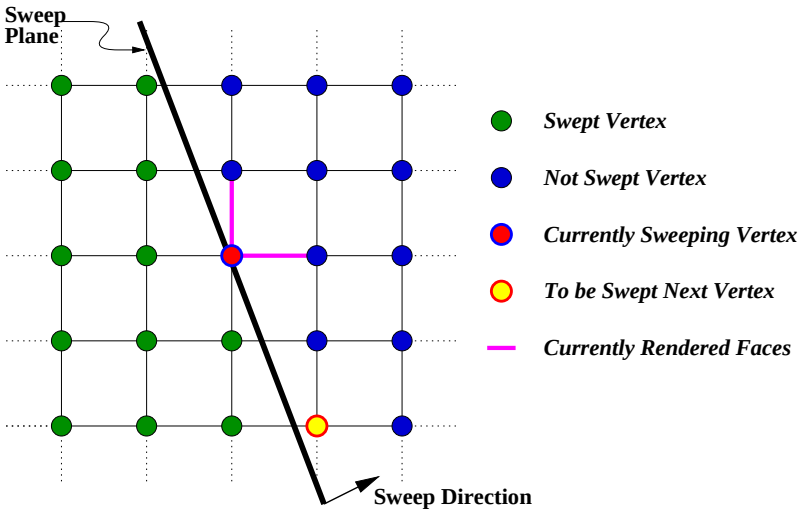


Figure 3.1 Sweep Process (adapted from [8])

Rectilinear data sets are assumed to be part of a lattice where the data points are the vertices of the lattice. The initial requirement is to convert this lattice into real world coordinates. We consider that there are three unit vectors, v_0, $v_1$and v_2 corresponding to the real world axes *x, y* and *z*. The necessary transformations are combined into a single transformation matrix and then applied to these unit vectors. The unit vectors are responsible for converting the imaginary grid point (*i, j, k*) into the real world coordinate system.

The RZSweep algorithm follows the sweep paradigm used in ZSweep [13]. The sweeping is performed when an imaginary plane sweeps through the data volume orthogo-

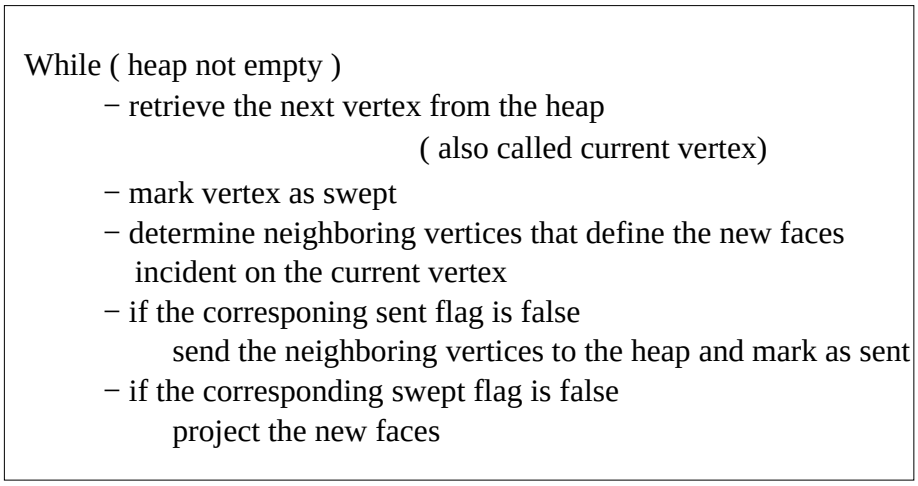


Figure 3.2 RZSweep Algorithm

nal to the z direction. All vertices are initially unsent and unswept. When the sweep plane touches a vertex, it is marked as *swept*. Only the neighboring vertices that have not been marked as *sent* are inserted into the heap to avoid multiple insertion. These neighboring vertices are then marked as *sent* . The sweeping uses a heap sort (called *heap*) to order the projection of the vertices. The vertices that are encountered by the sweep plane are sent to the heap. The sweeping continues until the heap becomes empty (Figure 3.1). The main loop of the algorithm is given in Figure 3.2.

A face is comprised of four vertices. A face is projected onto the display only if all four vertices of that face are marked as unswept. Only data points that fall in a user-specified scalar range are considered for sweeping. To accomplish this, a flagging routine is carried out in the preprocessing stage that determines which data points fall in the scalar range.

Further details describing the implementation details, optimizations and implementations of lighting, transfer functions and other opacity functions can be found in [8].

CHAPTER IV

PARZSWEEP

4.1 PARZSweep

A parallelization of the serial algorithm described in chapter III has been developed and tested. RZSweep was originally designed as a single processor serial algorithm. To exploit the computational capabilities of a cluster of processors, a parallel version was developed. The parallelization is for a multiprocessor shared memory architecture like that found in SGI machines. The hypothesis of this thesis is that a parallel version of RZSweep can be designed and implemented which will utilize multiple processors to reduce rendering times. In this chapter, the design decisions that were made in the development of PARZSweep, detailed description of the algorithm, implementation issues and images generated by PARZSweep are presented. Experiments and results designed to evaluate the performance of PARZSweep are given in chapter V.

4.1.1 *Parallel Architectures*

There are basically two types of memory architectures for parallel systems: distributed memory architecture and shared memory architecture. Eventually, the goal is to implement PARZSweep for both architectures.

In a distributed memory architecture, each processor has a local copy of all the data structures. Each processor has individual processing capabilities and its own memory and caches. No processor can access another processor's memory directly. The communication of results and data between processors is done using a network interface. A schematic diagram showing this type of architecture is given in Figure 4.1. *C* denotes the cache and *M* denotes the memory of each processor.

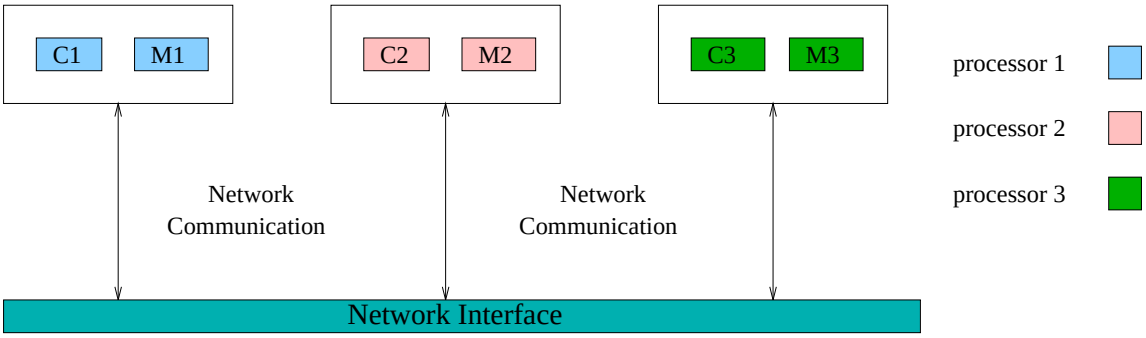


Figure 4.1 Distributed memory architecture

A shared memory architecture uses a common pool of memory that is shared by many processors. There is no need for communication through any interface since every processor can access every other processor's memory (see Figure 4.2). This saves on communication time. But there is a problem if two processors try to access the same memory location at the same time. Switching circuits are used to solve this problem. The switches control the routing of the messages and prevent memory conflicts. This is depicted in Figure 4.3.

With a shared memory architecture there is no need to keep multiple copies of the same data. Hence memory consumption is saved. The disadvantage of the shared memory architecture is the limitation to scaling of the processors. It has been shown that when the number of processes is greater than 24, there is a decrease in the efficiency and scaling of the program. This is due to the fact that the communication overhead becomes large than the performance gain.

A shared memory architecture was chosen for the first implementation of a parallel version of RZSweep because it is typically easier to develop parallel algorithm for this architecture. This architecture has favorable memory usage characteristics, and it is the typical architecture used for SGI graphics workstations.

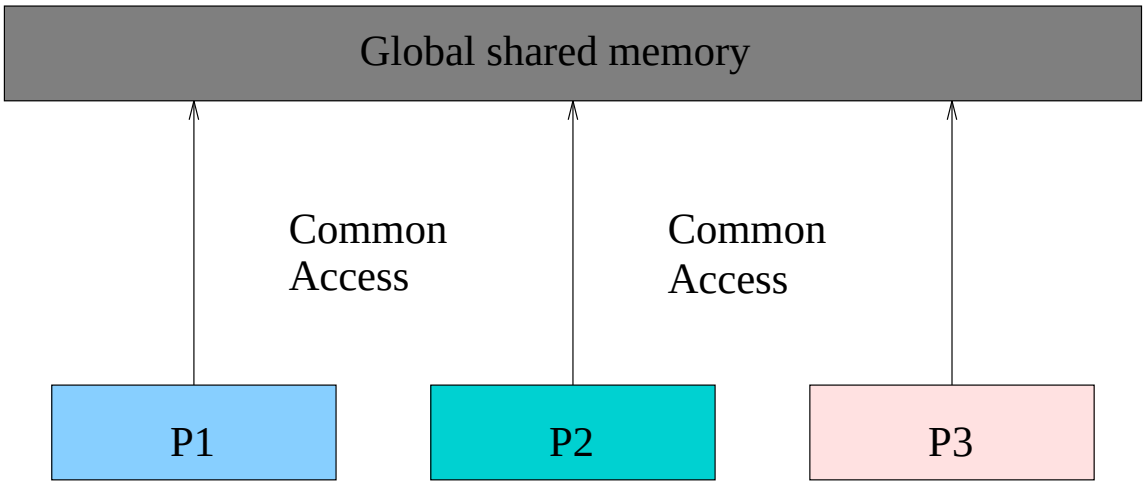


Figure 4.2 Shared memory architecture

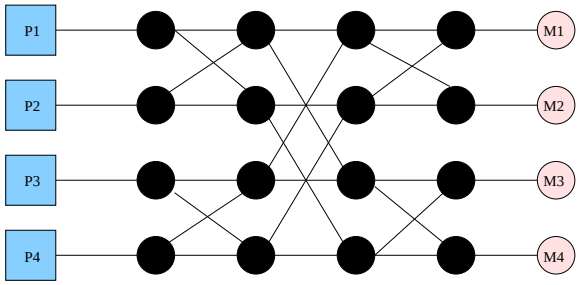


Figure 4.3 Switching circuits to avoid conflicts (adapted from [5])

4.1.2 *Parallelization Process*

The following steps were followed for the parallelization.

- A parallel extension to the serial algorithm was developed that uses an image partitioning technique called *tiling*. The approach is similar to the parallel ZSweep algorithm by Farias et al. [14].
- The screen space is divided into tiles, and each tile is dynamically assigned to the processors until all the tiles are exhausted.
- The rendering is done individually by each processor. Each processor has an image space of its own where it renders its sub image. The final image is composite of all these sub images.
- No special data structures are required to store the data, since the shared memory architecture allows access from a single data bank. All the processors access data from the single pool of data. This eliminates any data replication and hence reduces the memory requirement.
- The main loop of the algorithm remains unaltered. Each processor has the main algorithm running independently.
- Experiments have been conducted on a multi-processor SGI machines to test the efficiency of PARZSweep in terms of speed and image quality.

4.1.3 *Pseudo Code of Algorithm*

The PARZSweep algorithm has been designed based on the approach of Neih and Levoy [34]. The algorithm uses an adaptive image based task scheduling approach. It has a simple logic of assigning tiles to each processor and allowing the processor to render the tile and generate the final sub image. The algorithm is presented in Figure 4.4

```
Divide screen space into tiles
Determine number of processors
While ( tiles not empty )
  − assign next available tile to processor dynamically
  − set tile to rendered

      while ( heap not empty )
        − retrieve next vertex from heap
        − set as swept
        − determine neighboring vertices
              that define new faces incident on
              the current vertex
        − if the corresponding sent flag is false
              send neighboring vertices to heap
              and set as sent
        − if the corresponding swept flag is false
              project the new faces
```

Figure 4.4 PARZSweep algorithm

The cost of rendering using PARZSweep on multiple processors is generally less than the total cost of the RZSweep algorithm. The rendering cost for each tile depends on the

area of the tile and the number of data points present on that tile. If the number of tiles is equal to the number of processors, the total rendering tile for the image is approximately equal to the longest rendering time for a tile.

4.1.4 *Tiling*

One of the main issues of parallelization is deciding the work distribution among the processors. The work division can be approached either from object space or from image space. Object space distribution involves dividing the volume into chunks and assigning each chunk to a different processor. Alternatively, image-based distribution divides the image of the projected volume and assigns the corresponding sub volume to a processor as a job.

PARZSweep follows an approach called image based task scheduling or tiling for parallelization. The approach is very simple in its nature and yields good results. The idea is to divide the screen into tiles, and place them into a work queue. The tiles are user specified and each tile qualifies for a job. The processors are assigned these tiles in a dynamic fashion and each processor renders the tile independently to yield an image. This process continues until there are no more tiles to render. The final image consists of all these images pooled together (see Figure 4.5).

To perform the rendering it is important to know the bounds of each tile in object space. Each of the tiles in the screen is extruded into the implicit grid of the regular data set. This lattice is in the world coordinate system whereas the tiles are in the view

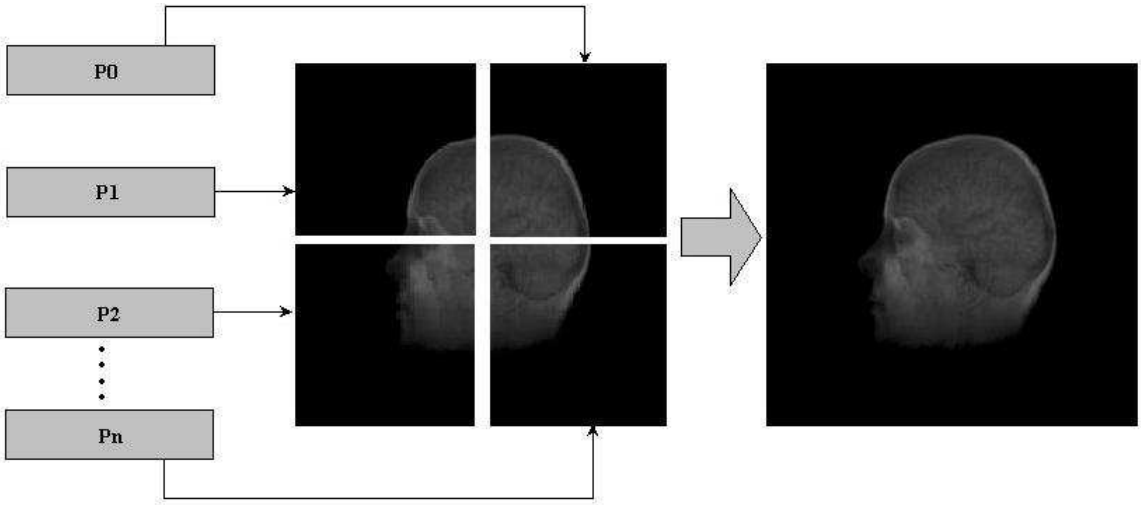


Figure 4.5 Image based task scheduling or tiling

coordinate system. Hence there is a conversion from the screen to the world coordinate system to determine the vertices which form the bounding box for each tile (Figure 4.6). Each of the bound volumes is swept separately using the RZSweep algorithm described above.

4.1.5 *PARZSweep*

This section describes the algorithm PARZSweep in detail. PARZSweep performs RZSweep on each of the tiles obtained after image partition. As stated above, the tiling process is performed on the screen to obtain the image partitions. Each tile is defined by 4 vertices which form its boundary. The vertices that bound each tile are then converted into the world coordinate system by performing a screen to world conversion. The vertices thus obtained are imposed on the lattice of the rotated regular grid (described in section 3.2) to

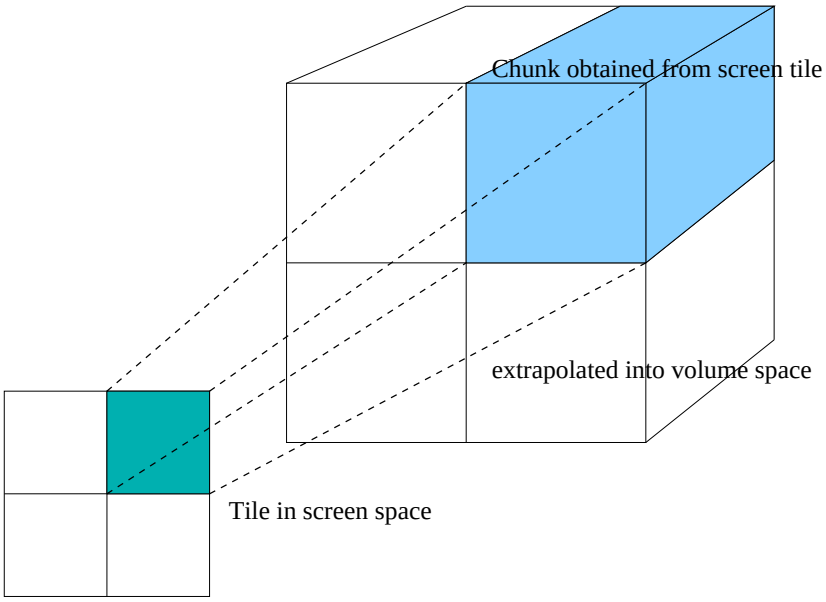
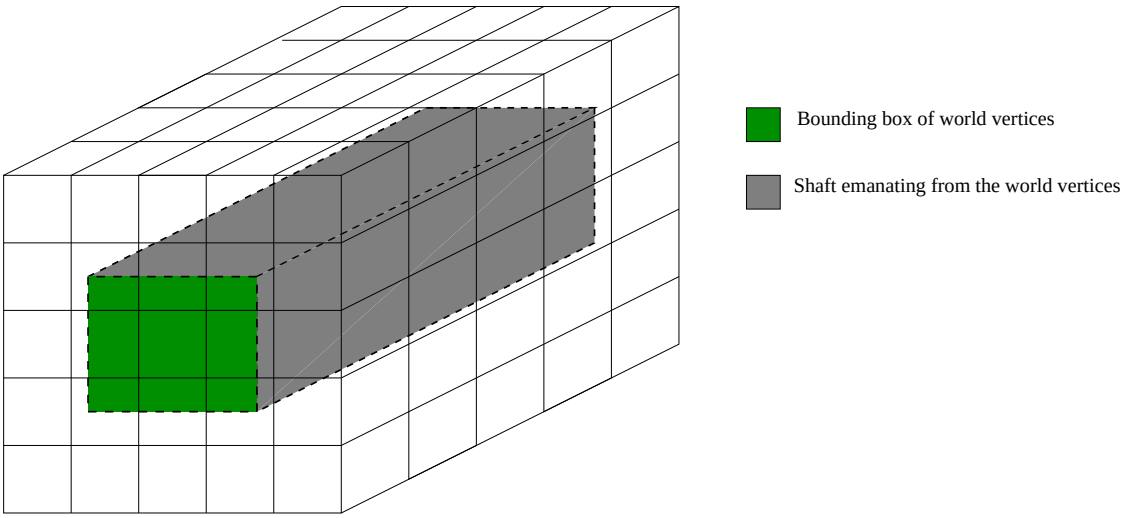


Figure 4.6 Extrusion of tiles to find the bound volumes

obtain the intersections with the data volume. These vertices are called world vertices. It is important to note that the grid described is rotated since the user can specify any rotation. The rotation specified by the user is applied to the implicit grid and a rotated grid is obtained. This rotated grid is used to to obtain the world vertices. These world vertices may not be perfectly aligned with the regular grid due to the randomness in the tile creation. Also screen space is not divided in the same units as the regular grid giving rise to the possibility that the world vertices will not fit exactly into the grid as shown in Figure 4.7.

The intersections with the regular grid are obtained by considering the nearest grid point compared to the world intersection (Figure 4.8). These grid intersections are the bounds of the ’shafts’ emanating from each tile.



The world vertices do not always fit in the regular lattice. Hence the cubical volume does not align with the grid and the exact grid intersections need to be found.

Figure 4.7 Mismatch of world and screen grid

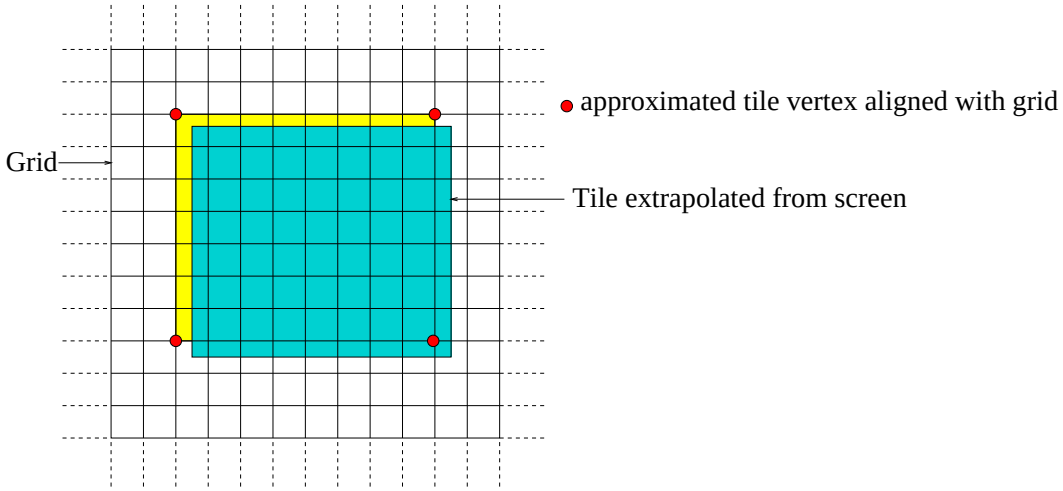


Figure 4.8 Aligning tile from screen with grid points

These shafts divide the data volume into cubes of sub volumes as described in Figure 4.6 and Figure 4.7. Each of the sub volumes are treated as separate jobs which need to be rendered to yield the result. Hence every cubic volume is rendered using the RZSweep algorithm described before. The order of assigning jobs to the processors is a significant decision. In PARZSweep, there is no priority implemented for assigning the jobs to the processors since each of the cubic volumes is given the same importance. Therefore the processors are assigned the jobs in a dynamic fashion. There is a race between the processors to grab the jobs placed in the job queue. Conflicts can arise during the job assignment between the processors. To avoid any conflict, multiple rendering locks are implemented during the distribution. A lock is a command that forces all the parallel work to be done into a serial queue. Once a lock command is issued at a specific place in the code, the processors wait until every single processor gets to that specific place in the code. So synchronization of all the procesors is done at the lock. This breaks the parallel nature of the algorithm. Usage of the lock prevents all the processors from working in parallel and places them in a serial queue. This queue is managed in a *'first in first out(FIFO)'* basis. Since locks break the parallel processing and force serial implementation, minimum usage of locks is advisable (Figure 4.9). PARZSweep uses just one lock for the work distribution.

As mentioned before, rendering of each cubic volume is done using RZSweep. Each tile yields a sub image of its own. The final image is a composite of all these sub images. An important issue is to handle the border of these images properly. As described before the grid and the tiles do not necessarily align perfectly. In such a case there may be a

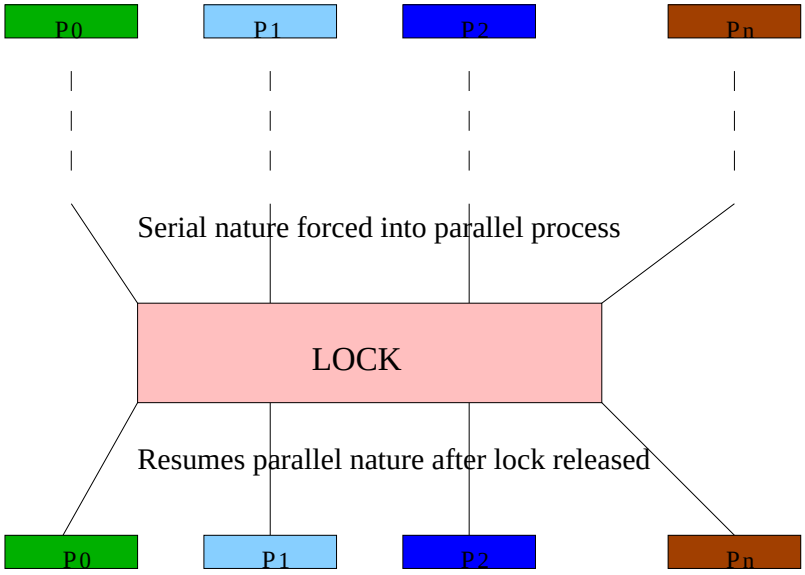


Figure 4.9 Lock implementation forces serial behavior

face that is intersected by two tiles (Figure 4.10). Care must be taken to avoid multi projection of these faces. Such faces can cause unnecessary time consumption and can result in incorrect images. PARZSweep takes care of this problem and does not perform multi projection of such faces. This is accomplished by storing the values of the boundary vertices in a structure in the preprocessing step. During the composition step, if the vertices are boundary vertices, this structure is accessed and the composition is done until the pixels reach the boundary values.

4.1.6 *Implementation Issues of PARZSweep*

A number of implementation issues must be addressed in PARZSweep. These include data structures, memory issues, lock usage, and image to world conversion.

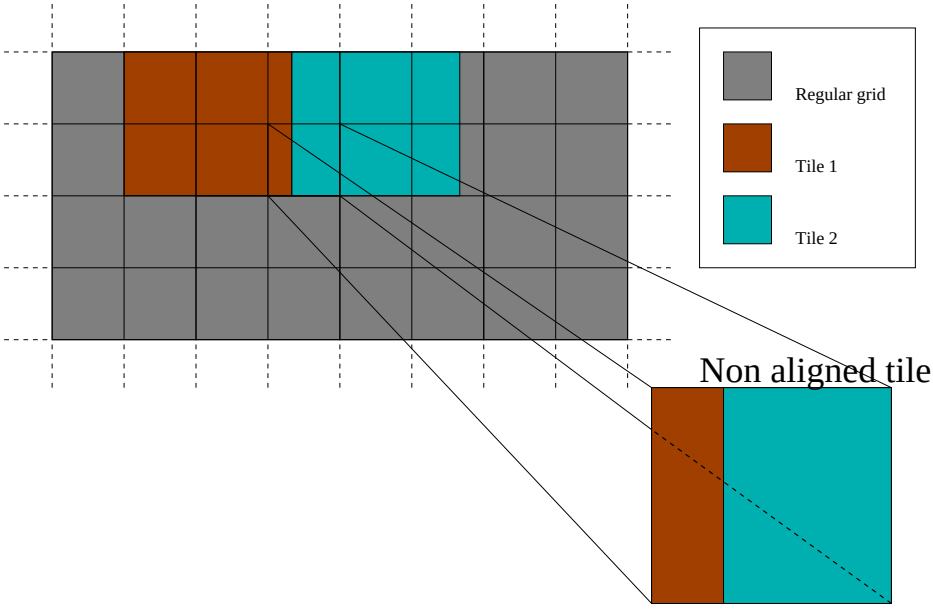


Figure 4.10 Tiles do not align with the grid causing the face to split into two

4.1.6.1 Changes in data structures

No changes were made in the data structures previously used in RZSweep. Each tile creates a copy of its own heap to sort the vertices that belong to it. The size of the heap is equal to the chunk of the data volume that the tile projects into. Hence PARZSweep is not very memory intensive. A new data structure is used to store the values of the vertices defining each tile. The size of the data structure depends on the number of tiles created which typically does not exceed a few hundred.

One of the components of the data structure is the values of the boundary of the tiles in the image space. These boundary values are used to avoid multiple projection. During the composition stage, these values are used for limiting the scan conversion of the faces.

Hence, when the faces do not align with the grid (see Figure 4.10 and Figure 4.8) the faces are rendered until the end of each tile. This limitation is brought about by storing the boundary values of each tile. This needs to be done in the image space by storing the interger coordinates of the pixels.

4.1.6.2 Memory issues

Since no special data structures are used, the memory requirement of PARZSweep is similar to RZSweep.

The shared memory architecture allows pooling of the memory for availability to the various processors. Hence many data structures can be shared between processors, and this makes the algorithm less memory intensive. There exists a single copy of the data volume shared by all processors. Since all the processors only perform read operations on the data, a single copy is sufficient for all the processors.

As discussed in [8], an attributes flag array is maintained to mark the vertices as sent and swept. This array has the size of the entire data set, since each vertex needs to be flagged. A write operation into this array is performed for each vertex. However, only a single copy of this attribute flag array is used in PARZSweep in order to increase memory efficiency. Since each chunk of data has a corresponding chunk in the attribute flag array, every processor accesses only a part of the flag array and there are no clashes among write operations as shown in Figure 4.11. However, this is not true for the boundary vertices, since each boundary vertex would be accessed by more than one processor. This is solved

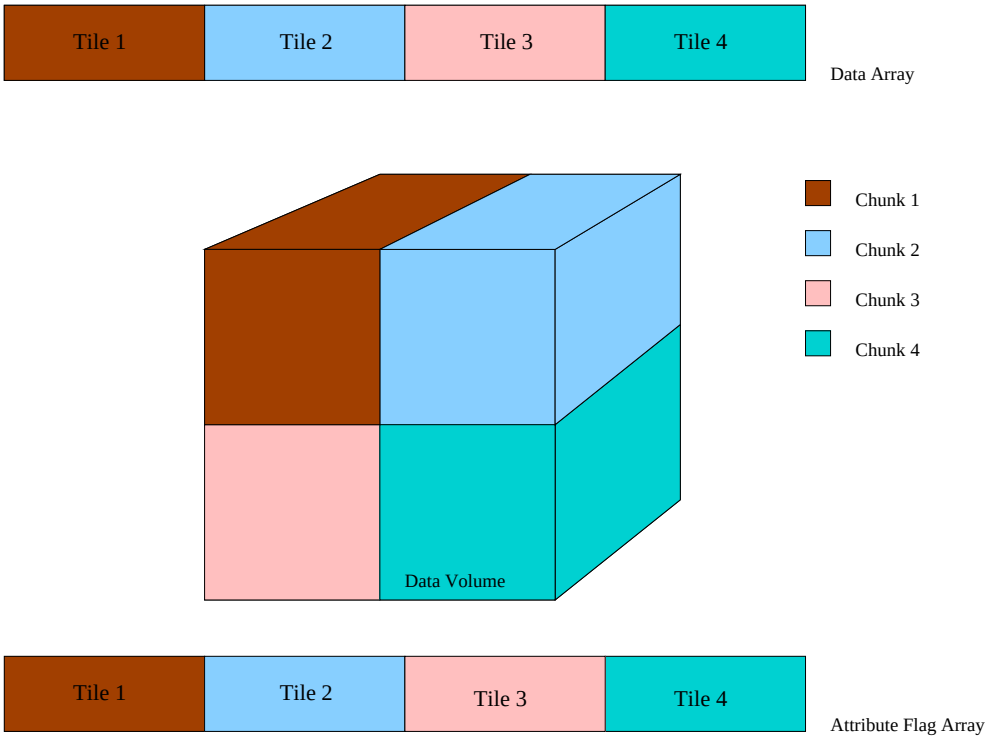


Figure 4.11 Memory access by each processor

by applying a simple yet effective strategy. The vertices in the boundary are modified by each processor during its individual execution of the algorithm. But as soon as the processor completes its execution, the attribute flag bits for the boundary vertices are reset into the original values. Hence for the next processor, these vertices are again ready to be modified according to the sweeping process of that processor.

4.1.6.3 Lock usage

PARZSweep uses only one lock in order to preserve the parallel nature of the algorithm. The lock is used in the module that assigns the next tile to an available processor. Processing of all the currently running jobs is frozen by the lock and the tile count is incremented. The next tile is then assigned to a free processor requesting a job. This functionality ensures that there is no conflict in the tile assignment and no repetition in the tile distribution.

In parallel processing, write operations for shared variables need to be monitored to avoid conflicts between processors. The shared variables in PARZSweep are the data and the attribute flag array.

The data array is used only to access the scalar value of each voxel. Hence only read operations are performed on the data and this does not require a locking mechanism. The attribute flag bits array, on the other hand, has write operations that can cause conflicts. The PARZSweep algorithm avoids the use of inefficient locks to address this problem. Instead of using the locks for this purpose, the attribute flag bits are monitored for conflict areas and reset back to original values as described in section 4.1.6.2.

4.1.6.4 Image to world conversion

PARZSweep is based on image-based task distribution. Hence it starts from the view plane and extends to the data volume to create chunks of volume that are assigned to each processors to render. The image space and the object space do not align perfectly as discussed before. Hence a conversion from the image to the world coordinate system is performed.

4.1.7 *PARZSweep in action*

Figure 4.12 and Figure 4.13 give a sequence of images that have been captured to illustrate the working of PARZSweep. All these images have been rendered using a single processor. The dataset used in Figure 4.12 is a fuel dataset with a scalar threshold of 75. The dataset used in Figure 4.13 is a CT scan of a lobster with a scalar threshold of 75. Notice that the complete image is a composite of all the four tiles. Each tile is rendered in the same image buffer, saving reconstruction cost.

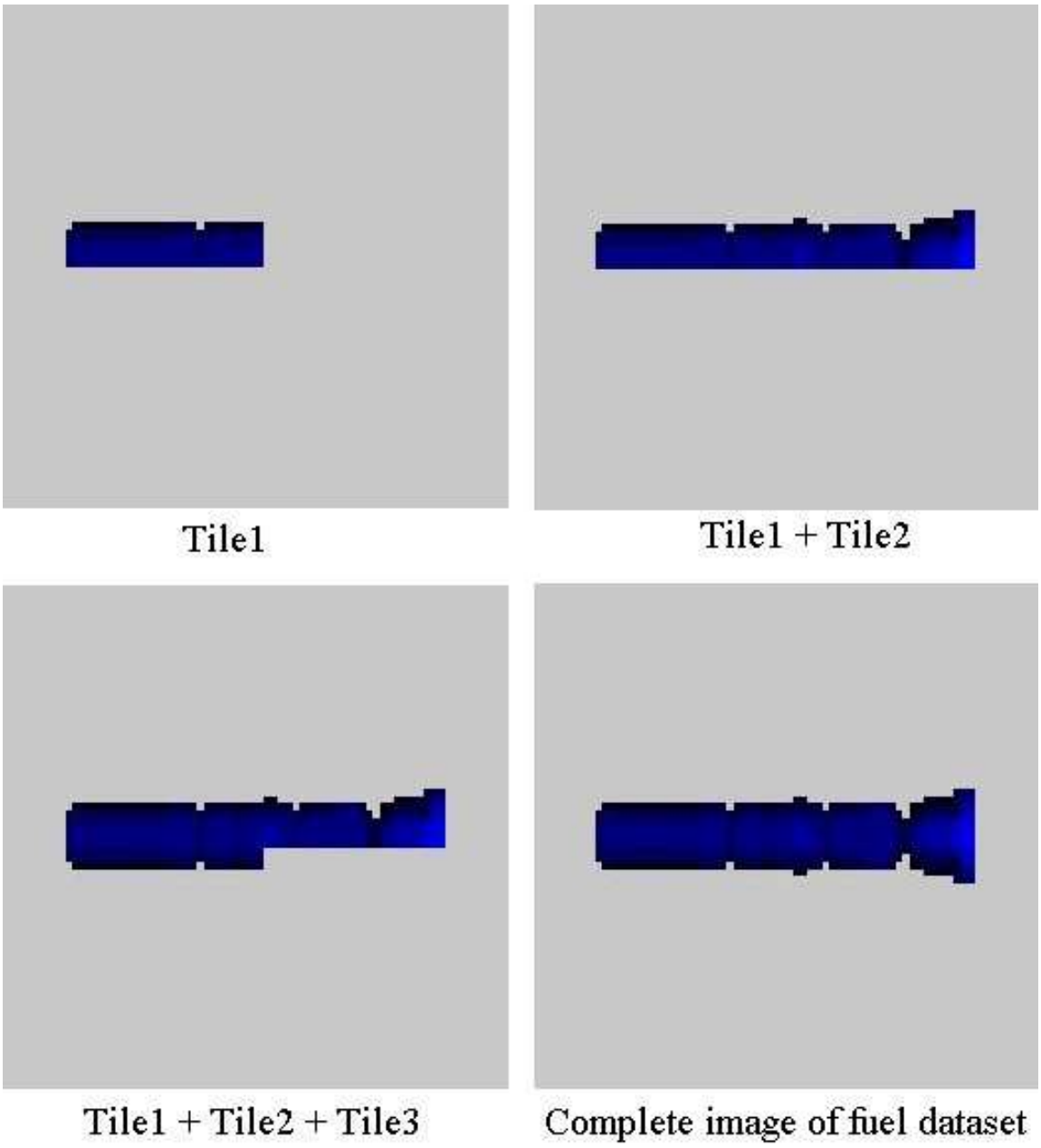


Figure 4.12 PARZSweep processing of fuel dataset

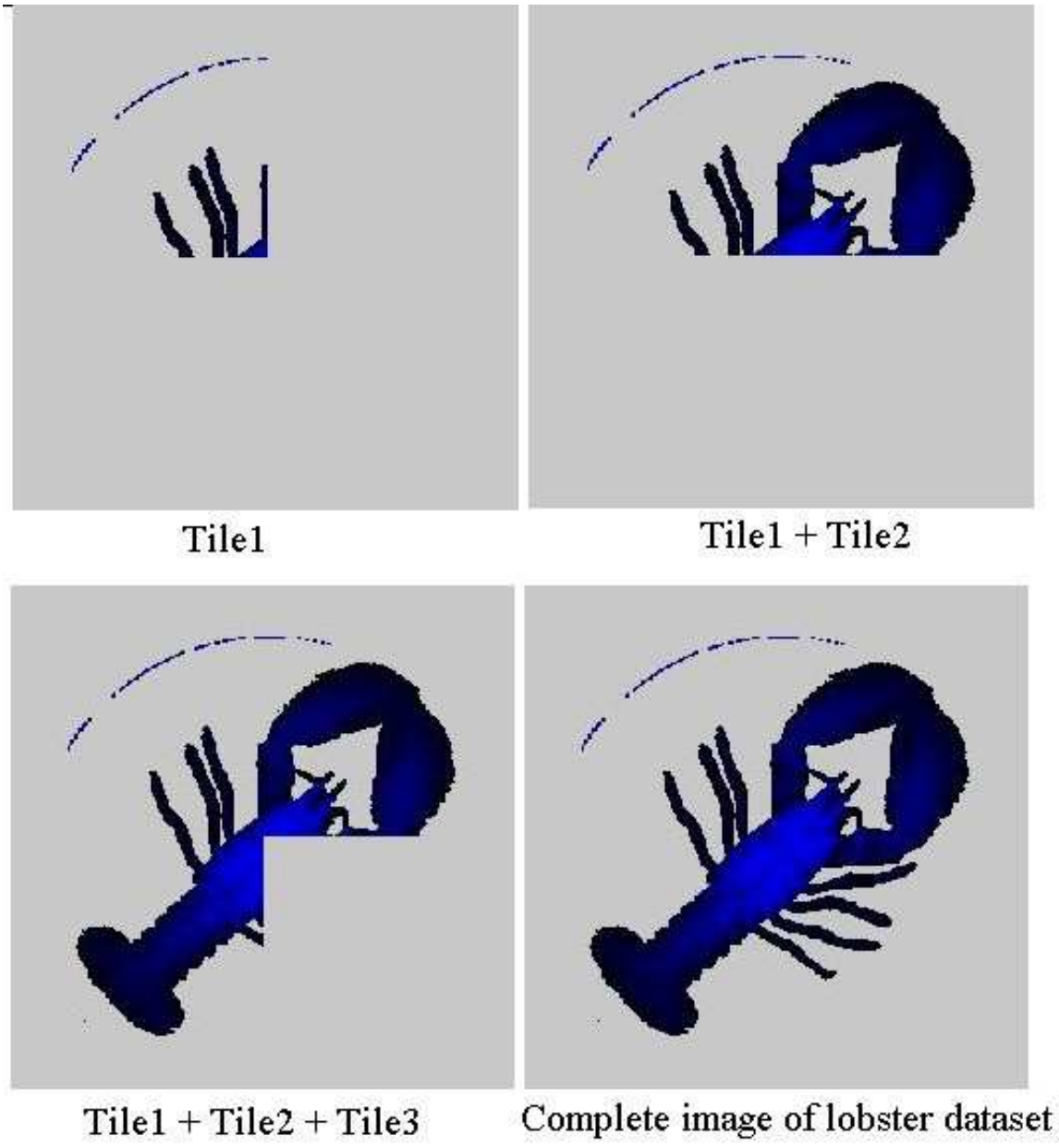


Figure 4.13 PARZSweep processing of lobster dataset

CHAPTER V

EXPERIMENTAL RESULTS

This section describes a set of experiments that were conducted to evaluate the performance of PARZSweep. RZSweep and PARZSweep are compared based on the prerendering times, run-times and the image quality produced. The performance of PARZSweep is evaluated with increasing number of tiles and processors.

5.1 Experimental Conditions

Most experiments were performed on an SGI machine that is a 4 processor shared memory architecture machine. Details about the machine is given in the next section5.1.1.

One would typically expect to use a number of tiles equal to the number of processors. In this case, the timing of the most expensive tile is taken as the time to render that dataset. Results given in section 5.1.2 indicate that the preprocessing time is insignificant compared to the rendering time. The times reported in the thesis have been averaged over 5 iterations. Standard deviation has been provided wherever applicable. In most of the tables, a minimum time and a maximum time is presented. The minimum time is the time taken for the least expensive tile to be rendered. The maximum time is the time taken for the most expensive tile to be rendered.

Table 5.1 describes the datasets used. Datasets have been down-loaded from [2].

Table 5.1 Size of the datasets used

Dataset	Size (bytes)
fuel	64^3
lobster	$301 \times 324 \times 56$
MRI	$256^2 \times 111$
CT	$512 \times 256 \times 512$

5.1.1 *Machine Details*

The following is the architecture of the machine on which all the experiments have been conducted.

- 4 400 MHZ IP27 Processors
- CPU: MIPS R12000 Processor Chip Revision: 3.5
- FPU: MIPS R12010 Floating Point Chip Revision: 3.5
- Main memory size: 4096 Mbytes
- Instruction cache size: 32 Kbytes
- Data cache size: 32 Kbytes
- Secondary unified instruction/data cache size: 8 Mbytes
- Integral SCSI controller 2: Version QL1040B (rev. 2), single ended
- Integral SCSI controller 3: Version QL1040B (rev. 2), differential
- Integral SCSI controller 4: Version QL1040B (rev. 2), differential
- Integral SCSI controller 5: Version QL1040B (rev. 2), differential
- Integral SCSI controller 1: Version QL1040B (rev. 2), single ended

- Integral SCSI controller 0: Version QL1040B (rev. 2), single ended
- Disk drive: unit 1 on SCSI controller 0
- CDROM: unit 6 on SCSI controller 0
- IOC3 serial port: tty1
- IOC3 serial port: tty2
- IOC3 serial port: tty3
- IOC3 serial port: tty4
- IOC3 parallel port: plp1
- Graphics board: InfiniteReality3
- Integral Fast Ethernet: ef0, version 1, module 1, slot io1, pci 2
- ATM PCA-200E OC-3: module 1, xio_slot 2, pci_slot 0, unit 0
- Iris Audio Processor: version RAD revision 7.0, number 1
- Origin MSCSI board, module 1 slot 7: Revision 4
- Origin BASEIO board, module 1 slot 1: Revision 4
- Origin PCI XIO board, module 1 slot 2: Revision 4
- IOC3 external interrupts: 1

5.1.2 *Prerendering Times*

This section describes the times taken by PARZSweep prior to its rendering routine for each dataset. The steps included in prerendering are the division of the screen, reading of the data file, storing into the data structures and calculating the visible faces to be drawn. Comparison of the times required by PARZSweep and RZSweep for prerendering are given in Table 5.2. The times for prerendering required by RZSweep and PARZSweep are similar. This shows that the parallel implementation has not increased the memory

allocation overhead and and the time complexity for prerendering remains the same for the parallel version.

Table 5.2 Time required for preprocessing in PARZSweep

Dataset	Threshold	PARZSweep Times (sec)	RZSweep Times (sec)
fuel	75	0.1277	0.127
lobster	27	4.023	4.130
MRI	40	5.088	5.053
CT	180	38.55	41.097

5.1.3 *PARZSweep vs. RZSweep*

The times required for volume rendering by PARZSweep and RZSweep were compared and timing results are discussed in Table 5.3. The times obtained for PARZSweep correspond to the timings taken for 4 processors and 4 tiles (2×2) tiles. The run-times for RZSweep for lobster and CTbrain datasets are quite variable as shown by the large standard deviation. For all datasets, PARZSweep running on 4 processors result in times that are at most one third of the times obtained by RZSweep. This follows the expectation, since parallelization should result in a decrease in rendering times. In the case of fuel dataset, super linear speed up is consistently obtained. This super linear speed up can be attributed to better cache hits. During the process of rendering, the operating system prefetches data that are in the vicinity of the current vertex. This prefetched data is stored in the cache of every processor. When multiple processors are used, more than one cache

is being exploited at a single time. So more data is prefetched in a single run. PARZSweep appears to utilize this cache prefetching to reduce the total rendering time.

Table 5.3 Computational times of PARZSweep vs. RZSweep

Dataset	PARZSweep (sec)(μ)	PARZSweep (sec)(σ)	RZSweep (sec)(μ)	RZSweep (sec)(σ)
fuel	0.085	$6 \times 10^{-}6$	0.26	$5.6 \times 10^{-}4$
lobster	13.21	0.508	58.14	24.35
MRI	25.20	0.88	75.10	4.54
CTbrain	91.75	04.72	350.11	176.19

5.1.4 *Image quality comparison between PARZSweep and RZSweep*

Figure 5.1 through Figure 5.4 show images of datasets rendered by PARZSweep using 4 tiles and 1 processor. A per pixel comparison was done between each of these images and the corresponding image rendered by RZSweep. This demonstrates that PARZSweep maintains the image quality obtained by RZSweep. There was no difference between the images generated by the parallel version and the serial version as seen in Table 5.4.

Table 5.4 Per pixel difference between RZSweep and PARZSweep Images

Dataset	Numerical Diff.
fuel	0
lobster	0
MRI	0
CT	0

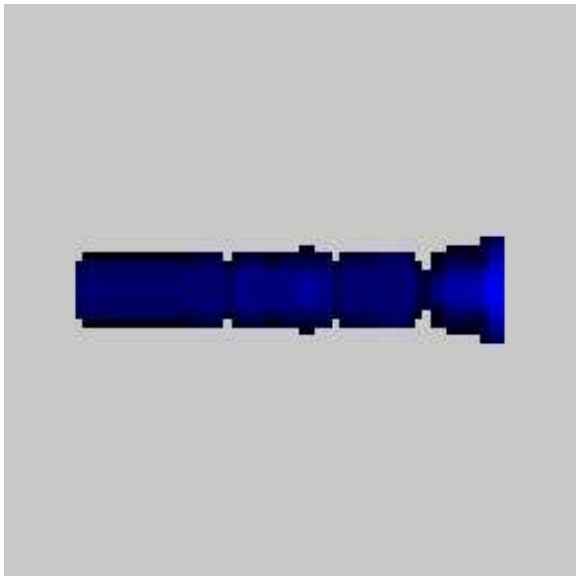


Figure 5.1 Fuel dataset with a threshold of 75

5.1.5 *Increasing tile numbers*

The goal of parallelization is to divide the rendering task among multiple processors. We compared the maximum time taken to render a tile when the images were divided into different number of tiles. We expected the maximum time taken to render a single tile to decrease with an increasing number of tiles since the amount of data per tile decreases. As seen in Table 5.5, we observed an unexpected increase in the maximum time when going from 4 tiles (2×2) to 16 tiles (4×4). The amount of time taken remain fairly stable with further increases in the number of tiles. This can be explained by the concentration of data in certain regions of the dataset. Table 5.6 shows that the maximum number of points obtained as the number of tiles increases decreases very slowly. This indicates that

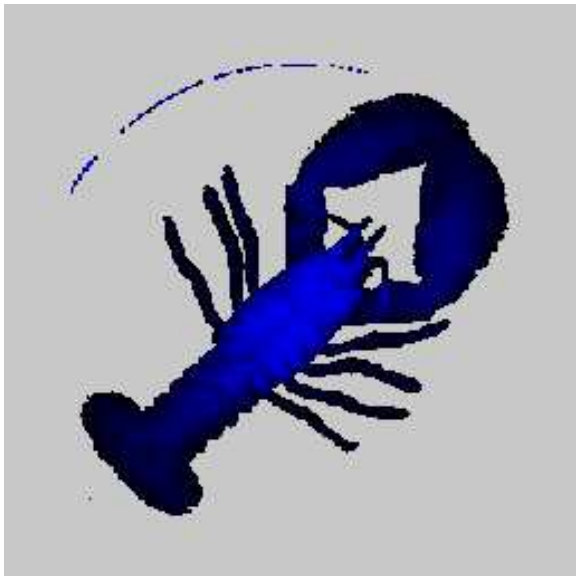


Figure 5.2 Lobster dataset with a threshold of 27

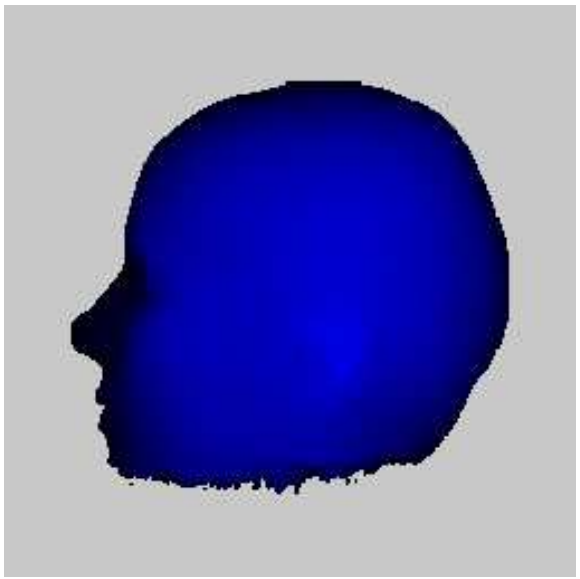


Figure 5.3 MRI dataset with a threshold of 40

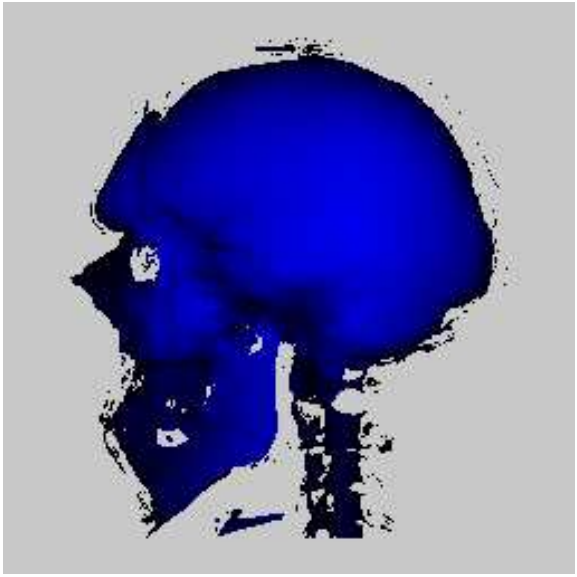


Figure 5.4 CT dataset with a threshold of 180

splitting the image into more tiles does not necessarily substantially reduce the number of points in every table. This indicates that the naive task scheduling algorithm used is not sufficient and further work is required to exploit the maximum potential of the algorithm. The increase in time required when going from 4 tiles to 16 tiles cannot currently be explained and requires further investigation.

There is a clear difference in the minimum and maximum times required to render tiles, indicating that there needs to be load balancing of the process. It is apparent that the amount of data contained in the tiles varies greatly. Several load balancing strategies can be applied to bring down the difference such as [42] and [9].

Table 5.5 Computational times as a function of number of tiles with the fuel dataset

Tiles	Min time (sec)(μ)	Min time (sec)(σ)	Max time (sec)(μ)	Max time (sec)(σ)
2×2	0.047	1.1×10^{-5}	0.092	1.8×10^{-3}
4×4	1.28×10^{-5}	7.0×10^{-9}	0.26	7.0×10^{-4}
8×8	1.20×10^{-5}	1.5×10^{-8}	0.26	3.0×10^{-4}
16×16	1.19×10^{-5}	1.0×10^{-8}	0.27	3.0×10^{-4}

Table 5.6 Number of points projected during rendering as a function of number of tiles with the fuel dataset

Tiles	Minimum points	Maximum points
2×2	406	1081
4×4	0	876
8×8	0	784
16×16	0	723

5.1.6 *Increasing number of processes*

Table 5.7 gives a comparison of the times for PARZSweep as the number of processors increases for the fuel dataset. The limitation of available hardware has restricted the number of processors to a maximum of four. The unexpected increase in time required for rendering when going from 4 to 16 tiles was observed again. The times for rendering a tile with increasing processor number is almost stable.

This is attributed to uneven division of work among processors. The goal behind increasing the number of tiles is to reduce the work load in each tile and hence reduce the work load for each processor. But the results show that the number of points in each tile has not been significantly reduced. So, the maximum time for rendering a tile has not been

Table 5.7 Computational times as a function of number of processes

No. of procs.	2×2		4×4		8×8		16×16	
	min	max	min	max	min	max	min	max
1	0.04	0.08	1.20×10^{-5}	0.25	1.20×10^{-5}	0.26	1.20×10^{-5}	0.26
2	0.05	0.11	1.50×10^{-5}	0.26	1.20×10^{-5}	0.26	1.30×10^{-5}	0.26
3	0.05	0.12	2.10×10^{-5}	0.26	2.30×10^{-5}	0.26	1.80×10^{-5}	0.26
4	0.07	0.12	2.60×10^{-5}	0.26	2.10×10^{-5}	0.26	1.50×10^{-5}	0.26

reduced. This indicates that the concentration of data in regular datasets is such that there is more data in regions of the data volume than in the others. Those concentrated regions tend to stay on in a single tile even when the tiles are subdivided. Hence the maximum time taken for rendering a tile remains the same.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The focus of this research was to develop a parallel version for a new volume rendering algorithm for rectilinear datasets. The hypothesis of this thesis research was that a parallel version of RZSweep can be designed and implemented which utilizes multiple processors to reduce rendering times. This hypothesis was confirmed. The novelty of the RZSweep algorithm is that for the first time the sweep paradigm has been adapted for rendering of regular grids. The serial version of the algorithm, RZSweep, successfully rendered rectilinear datasets resulting in images with good quality. The implicit regularity of rectilinear datasets has been exploited in the algorithm to find out the adjacency information for all vertices. Hence no additional memory has been used to store the vertices information. The memory requirement of the algorithm was also less resulting in good space complexity. Although, emphasis was given on image quality, the rendering speed was also satisfactory. Further work has been done in the serial version to implement lighting and transfer functions to make the resulting images more realistic. More details on this can be found in [8].

The Parallel capacity of the algorithm was explored in this thesis. PARZSweep, a parallel version of RZSweep, has been developed for rendering rectilinear datasets, for the shared memory architecture. Data distribution has been done using the image space partitioning technique, also called *tiling*. This technique divides the screen space into tiles and each tile is dynamically assigned to a processor for rendering of sub images. The volume space is divided by extrapolating the tiles into the object space. Each tile results in sub volumes. Each sub volume is then assigned dynamically to separate processors. Each processor separately renders its own data resulting in sub images. The final image is a composite of these sub images. This partitioning technique is a very simple space distribution scheme. Since a shared memory architecture is used, only a single copy of the data volume must be kept. Hence, the memory efficiency of the serial code has been preserved successfully in the parallel version. No additional data structures have been used in the parallel version maintaining the same space complexity as the serial version. Simple *mfork* commands have been used to spawn new processes. No special forking techniques such as *p-threads* have been used.

This was a first attempt to test the capability of the sweep paradigm for rendering regular datasets in a parallel fashion. PARZSweep has achieved great speedup in the rendering of regular grids over its serial predecessor RZSweep when using 4 tiles and 4 processors. The speedup has been more than 50% of the rendering speed as seen in the results section. This speedup is probably because there are good cache hits in the code and efficiency in prefetching of data has been increased. Although there is a considerable amount of speed

up in the parallel code in the rendering time when compared to the serial version, the scalability of PARZSweep has turned out to be poor. The poor scalability is due to improper load balancing of the jobs. This results from an uneven distribution of data amongst tiles. This causes certain processors to do more of the rendering than others. Hence this affects the efficiency as the number of processes increase. Another cause may be weak task scheduling. Image-based task scheduling is a very simple and basic type of task distribution scheme. No special priority queues has been implemented in this first approach. Priority queues can increment the performance factor significantly since the tiles that have more data would be rendered with a higher priority than other tiles. More research needs to be done in this area. A more complex task scheduling scheme needs to be implemented to make the code scalable.

6.2 Future work

This research has demonstrated the potential of PARZSweep for parallel rendering of regular grids. However, further work needs to be done to improve performance. Methods for achieving a more even division of data among tasks need to be explored as well as more sophisticated methods of scheduling tasks.

The results show that the code is extremely load imbalanced. So load balancing schedules need to be worked out. Various load balancing schemas need to be tested and the most efficient one needs to be implemented. The results show that PARZSweep is faster than

RZSweep. Hence the expectation is that since there has been a speed up in the timings from the serial version, load balancing would bring about scaling in the code.

A new approach of object-based task scheduling could be implemented where the object space would be partitioned depending on the processes required by the user. Each volume is given to the corresponding processor to render the sub image.

Priority queues to order the tasks based on the load could also be implemented. The current implementation assigns the jobs to the processor in a dynamic fashion. This type can be classified as *first come first serve* basis. The processes compete with each other to get the next job based on the time taken to complete the previous task. But that has proven to be a weak scheme. Priority queues would prioritize the job queue instead of dynamically assigning the jobs. Priority can be based on the work load or time taken to complete the existing jobs. Time stamps can also be used to further the efficiency of the queues.

This version of PARZSweep has been developed primarily for shared memory architectures. Part of the future work is to advance it to utilize the capacity of distributed memory architectures. Research needs to be done concerning the methodology to carry out the task distribution on a distributed architecture.

REFERENCES

[1] "VOLPACK," http://www-graphics.stanford.edu/software/volpack (current August 10, 2002).

[2] "Volvis," http://www.volvis.org (current 7 Dec. 2002).

[3] M. Amin, A. Grama, and V. Singh, "Fast volume rendering using an efficient, scalable parallel formulation of the shear–warp algorithm," *Proceedings of Parallel Rendering Symposium*, Atlanta, GA, 1995, pp. 7–14.

[4] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang, "Volvis: A diversified volume visualization system," *Proceeding of IEEE Visualization 1994*, Washington, DC, Oct 17 – 21 1994, pp. 31–38.

[5] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation*, Prentice Hall, Englewood cliffs, New Jersey 07632, 1989.

[6] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," *Proceedings of 1994 Symposium on Volume Visualization*, Tysons corner, Virginia, United States, Oct 17–18 1994, pp. 91–98.

[7] J. Challinger, "Scalable parallel volume raycasting for non rectilinear computational grids," *Proceedings of Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 81–88.

[8] G. Chaudhary, *RZSWeep: A new volume-rendering technique for uniform rectilinear datsets*, master's thesis, Mississippi State University, Department of computer science, Mississippi state university, Mississippi state, 39762, May 2003.

[9] B. Corrie and P. Mackerras, "Parallel volume rendering and data coherence," *Proceeding of Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 27–34.

[10] F. Dachille, K. Kreeer, B. Chen, I. Bilter, and A. Kaufman, "High quality volume rendering using texture mapping hardware," *Proceedings of the 1998 SIGGRAPH/ EUROGRAPHICS Workshop on Graphics Hardware*, Libson, Portugal, Aug 31–Sept 01 1998, pp. 67–76.

[11] J. Danskin and P. Hanrahan, "Fast algorithm for volume ray tracing," *Proceedings of 1992 Workshop on Volume Visualization*, Boston, Massachusetts, Oct 19–20 1992, pp. 91–98.

[12] T. Elvins, "Volume rendering on a distributed memory parallel computer," *Proceedings of Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 93–98.

[13] R. Farias, J. S. B. Mitchell, and C. Silva, "ZSweep: An efficient and exact projection algorithm for unstructured volume rendering," *Proceedings of ACM/IEEE Symposium on Volume Visualization 2000*, Salt Lake City, Utah, Oct 9–10 2000, pp. 91–99.

[14] R. Farias and C. Silva, "Parallelizing the ZSweep algorithm for distributed–shared memory architectures," *Proceedings of ACM/IEEE International Workshop on Volume Graphics 2001*, Stony Brook, New York, Jul 21–22 2001, pp. 59–66.

[15] H. Fuchus, Z. M. Kedem, and S. P. Uselton, "Optimal surface reconstruction from planar contours," *Communication of the ACM*, vol. 20, no. 10, Oct 1982, pp. 693–702.

[16] I. Gargantini, T. R. S. Walsh, and O. L. Wu, "Displaying a voxel-based object via linear octtrees," *Proceeding of SPIE 626*, Jul 1986, pp. 460–466.

[17] C. Girsten, "Volume visualization of sparse irregular meshes," *IEEE Computer Graphics and Applications*, vol. 12, no. 2, 1992, pp. 40–48.

[18] K. H. Hoehne, B. Pfiesser, A. Pommet, R. S. M. Riemer, T. Schiemann, and T. U, "A virtual body model for surgical education and rehearsal," *IEEE Computer Graphics and Applications*, vol. 29, no. 1, 1996, pp. 25–31.

[19] W. Hsu, "Segmented ray casting for data parallel volume rendering," *Proceeding Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 93–98.

[20] J. Huang, N. Shareef, R. Crawfis, P. Sadayappan, and K. Mueller, "A parallel splatting algorithm with occlusion culling," *3rd Eurographics Workshop on Parallel Graphics and Visualization*, Girona, Spain, Sept 2000.

[21] P. Lacroute, "Real time volume rendering on shared memory multiprocessors using shear warp factorization," *Proceeding of Parallel Rendering Symposium*, Phoenix, AZ, 1995.

[22] P. Lacroute and M. Levoy, "Fast volume rendering using a shear warp factorization of the viewing transformation," *Proceedings of SIGGRAPH '94*, Orlando, Florida, Jul 1994, pp. 451–458.

[23] M. Levoy, "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 5, 1988, pp. 29–37.

[24] M. Levoy, "Efficient ray tracing of volume data," *ACM Transaction Computer Graphics*, vol. 9, no. 3, 1990, pp. 245–261.

[25] P. Li, S. Whitman, R. Mendoza, and J. Tsiao, "Prefix– A parallel splatting volume rendering system for distributed visualization," *Proceeding of Parallel Rendering Symposium*, Phoenix, AZ, 1997.

[26] K. Ma, "Parallel volume ray casting for unstructured grid data on distributed memory architecture," *Proceeding of Parallel Rendering Symposium*, Atlanta, GA, 1995, pp. 23–30.

[27] K. Ma and T. Crockett, "A scalable parallel cell projection volume rendering algorithm for three dimensional unstructured data," *Proceeding Parallel Rendering Symposium*, Phoenix, AZ, 1997, pp. 23–30.

[28] R. Machiraju and R. Yagel, "Efficient feed forward volume rendering techniques for vector and parallel processors," *Proc. of SUPERCOMPUTING 93*, Portland, Oregon, Nov 1993, pp. 699–708.

[29] D. Meagher, "Geometric modeling using octree encoding," *Computer Graphics and Image Processing*, vol. 20, 1982, pp. 129–147.

[30] M. Meissner, U. Hoffman, and W. Strassner, "Enabling classification and shading for 3D texture mapping based volume rendering," *Proceeding of IEEE Visualization '99*, San Francisco, CA, Oct 24–29 1999, pp. 207–214.

[31] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfish, "A practical evaluation of popular volume rendering algorithms," *Proceedings of the 2000 IEEE Symposium on Volume Visualization 2000*, Salt Lake City, Utah, Oct 9–10 2000, pp. 81–90.

[32] C. Montani, R. Perego, and R. Scopigno, "Parallel volume visualization on a hypercube architecture," *Proceeding of Parallel Rendering Symposium*, Boston, MA, 1997, pp. 9–16.

[33] K. Mueller and R. Crawfish, "Eliminating popping artifacts in sheet buffer-based splatting," *Proceeding of the Conference on Visualization '98*, Research Triangle Park, North Carolina, Oct 18–23 1998, pp. 239–245.

[34] J. Neih and M. Levoy, "Volume rendering on scalable shared-memory mimd architecture," *1992 Workshop on Volume Visualization Proceedings*, Boston, New York, Oct 1992, pp. 17–24.

[35] F. Preperata and M. Shamos, *Computational geometry: an introduction*, Springler verlag, New york, 1985.

[36] K. Sano, H. Kitajima, H. Kobayashi, and T. Nakamura, "Parallel processing of the shear warp factorization with the binary swap method on a distributed memory multiprocessor system," *Proceeding of Parallel Rendering Symposium*, Phoenix, AZ, 1995.

[37] C. Silva and J. Mitchell, "The lazy sweep raycasting algorithm for rendering irregular grids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 5, 1998, pp. 142–157.

[38] U. Tiede, K. H. Hoehne, M. Bomans, A. Pommert, M. Riemer, and G. Weibecke, "Investigation of medical 3D rendering algorithms," *IEEE Computer Graphics and Applications*, vol. 10, no. 2, 1990, pp. 41–53.

[39] U. Tiede, T. Schiemann, and K. H. Hoehne, "High quality rendering of attributed volume data.," *Proceeding of IEEE Visualization '98*. 1998, pp. 255–262, Springer-Verlag, New York.

[40] H. Tuy and L. Tuy, "Direct 2D display of 3D objects," *IEEE Computer Graphics and Applications*, vol. 8, no. 5, 1988, pp. 29–33.

[41] L. Westover, "Footprint evaluation for volume rendering," *Computer Graphics (Proc. SIGGRAPH)*, vol. 24, no. 4, Aug 1990, pp. 367–376.

[42] S. Whitman, "A task adaptive parallel graphics renderer," *Proceeding of Parallel Rendering Symposium*, San Jose, CA, 1993, pp. 27–34.

[43] T. Whitted, "An improved illumination model for shaded display," *Communication of the ACM*, vol. 23, no. 6, 1980, pp. 343–349.

[44] C. M. Whittenbrink, *Designing optimal parallel volume rendering algorithms*, doctoral dissertation, University of Washington, 1993.

[45] R. Yagel, "Towards real time volume rendering," *Proceedings of GRAPHICON '96*, vol. 1, Jul 1996, pp. 230–241.

[46] R. Yagel, D. Reed, A. Law, P. W. Shih, and N. Shareef, "Hardware assisted volume rendering of unstructured grids by incremental slicing," *Proceedings 1996 Symposium on Volume Visualization*, San Francisco, CA, Sept 1996, pp. 55–62.

[47] R. Yagel and Z. Shi, "Accelerating volume animation by space leaping," *Proceeding of Visualization '93*, San Jost, California, Oct 1993, pp. 62–69.