

## University of New Hampshire University of New Hampshire Scholars' Repository

---

Master's Theses and Capstones

Student Scholarship

---

Winter 2006

# Volume visualization of time-varying data using parallel, multiresolution and adaptive-resolution techniques

Sadaf Shams

*University of New Hampshire, Durham*

Follow this and additional works at: <https://scholars.unh.edu/thesis>

---

### Recommended Citation

Shams, Sadaf, "Volume visualization of time-varying data using parallel, multiresolution and adaptive-resolution techniques" (2006).  
*Master's Theses and Capstones*. 246.  
<https://scholars.unh.edu/thesis/246>

This Thesis is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Master's Theses and Capstones by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact [nicole.hentz@unh.edu](mailto:nicole.hentz@unh.edu).

**VOLUME VISUALIZATION OF TIME-VARYING DATA  
USING PARALLEL, MULTIREOLUTION AND  
ADAPTIVE-RESOLUTION TECHNIQUES**

BY

**SADAF SHAMS**

B.S. in Computer Science, Lahore University of Management Sciences, 2003

THESIS

Submitted to the University of New Hampshire

In Partial Fulfillment of

The Requirements for the Degree of

Master of Science

In

Computer Science

December, 2006

UMI Number: 1439292

### INFORMATION TO USERS

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

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

**UMI<sup>®</sup>**

---

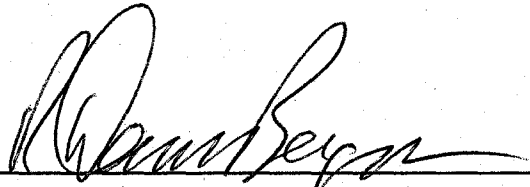
UMI Microform 1439292

Copyright 2007 by ProQuest Information and Learning Company.

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

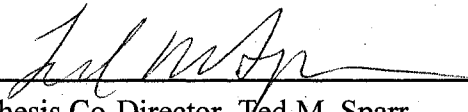
ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

This thesis has been examined and approved.



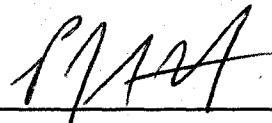
---

Thesis Co-Director, R. Daniel Bergeron,  
Professor of Computer Science



---

Thesis Co-Director, Ted M. Sparr,  
Professor of Computer Science



---

Philip J. Hatcher  
Professor of Computer Science

21 November 2006

Date

## **ACKNOWLEDGEMENTS**

I would like to thank Dr. R. Daniel Bergeron, Dr. Philip J. Hatcher, Dr. Ted M. Sparr,  
and Andrew Foulks for their assistance and guidance.

# TABLE OF CONTENTS

<u>ACKNOWLEDGEMENTS.....</u>	<u>iii</u>
<u>LIST OF TABLES.....</u>	<u>vi</u>
<u>LIST OF FIGURES .....</u>	<u>vii</u>
<u>ABSTRACT.....</u>	<u>ix</u>

CHAPTER	PAGE
<u>INTRODUCTION .....</u>	<u>1</u>
<u>1 BACKGROUND AND RELATED WORK .....</u>	<u>3</u>
<u>1.1 Grid Visualization.....</u>	<u>3</u>
<u>1.2 Load Distribution Issues .....</u>	<u>4</u>
<u>1.3 Parallel Rendering.....</u>	<u>5</u>
<u>2 PARALLEL VISUALIZATION .....</u>	<u>6</u>
<u>2.1 Application Architecture.....</u>	<u>7</u>
<u>2.1.1 Test Data .....</u>	<u>7</u>
<u>2.1.2 Temporal Decomposition.....</u>	<u>7</u>
<u>2.1.3 Spatial Decomposition.....</u>	<u>8</u>
<u>2.2 Data Division .....</u>	<u>9</u>
<u>2.2.1 Temporal Division .....</u>	<u>9</u>
<u>2.2.2 Spatial Division.....</u>	<u>11</u>

2.3	Rendering.....	14
2.3.1	Temporal Rendering .....	15
2.3.2	Spatial Rendering.....	17
2.4	Assembly.....	17
2.4.1	Temporal Sequencing .....	18
2.4.2	Spatial Compositing.....	19
3	MULTIRESOLUTION VISUALIZATION .....	22
3.1	Space Resolution.....	22
3.2	Time Resolution.....	26
4	ADAPTIVE-RESOLUTION VISUALIZATION (SPATIAL PARALLEL).....	29
4.1	Adaptive-resolution Data Format .....	30
4.2	Data Division .....	33
4.3	Rendering.....	38
4.4	Assembly.....	39
5	PERFORMANCE ANALYSIS .....	51
6	CONCLUSIONS.....	58
	LIST OF REFERENCES .....	60

## LIST OF TABLES

<u>Table 1: Data division that causes boundary artifacts .....</u>	<u>21</u>
<u>Table 2: Example with 3 levels of time resolutions.....</u>	<u>26</u>
<u>Table 3: Test machine specifications .....</u>	<u>28</u>
<u>Table 4: Adaptive-resolution data blocks and their cell counts.....</u>	<u>36</u>
<u>Table 5: Sorted adaptive-resolution data blocks with assigned processors .....</u>	<u>37</u>
<u>Table 6: Blocks and number of cells assigned to each processor .....</u>	<u>37</u>
<u>Table 7: Comparison of equal work division and equal data division approach.....</u>	<u>38</u>
<u>Table 8: Summary of performance analysis .....</u>	<u>57</u>



## LIST OF FIGURES

<u>Figure 1: Basic architecture .....</u>	<u>7</u>
<u>Figure 2: GUI and output (time) .....</u>	<u>8</u>
<u>Figure 3: GUI and output (space) .....</u>	<u>9</u>
<u>Figure 4: Time/frame decomposition.....</u>	<u>10</u>
<u>Figure 5: Multiresolution time/frame decomposition .....</u>	<u>11</u>
<u>Figure 6: Data decomposition .....</u>	<u>12</u>
<u>Figure 7: Time frames with 6 processors.....</u>	<u>16</u>
<u>Figure 8: Partial images with 2 processors .....</u>	<u>17</u>
<u>Figure 9: Partial images with 4 processors .....</u>	<u>17</u>
<u>Figure 10: Frame sequencing.....</u>	<u>18</u>
<u>Figure 11: Boundary artifacts when compositing 10 sub-images.....</u>	<u>20</u>
<u>Figure 12: Data division that causes boundary artifacts .....</u>	<u>20</u>
<u>Figure 13: Data division that prevents boundary artifacts .....</u>	<u>21</u>
<u>Figure 14: Rendering time per frame using 3 space resolution levels.....</u>	<u>24</u>
<u>Figure 15: Difference in quality between low, medium and high resolutions.....</u>	<u>25</u>
<u>Figure 16: Time taken for I/O-intensive verses computation-intensive rendering .....</u>	<u>27</u>
<u>Figure 17: Adaptive-resolution data .....</u>	<u>30</u>
<u>Figure 18: Data format.....</u>	<u>31</u>
<u>Figure 19: Example data description file.....</u>	<u>32</u>

<u>Figure 20: Data decomposition options .....</u>	<u>34</u>
<u>Figure 21: Adaptive-resolution data segmented into 8 blocks.....</u>	<u>36</u>
<u>Figure 22: Calculating optimal number of cells per processor .....</u>	<u>36</u>
<u>Figure 23: Partial images generated for each data block using data in figure 19 .....</u>	<u>39</u>
<u>Figure 24: Expansion of partial images of different resolutions.....</u>	<u>41</u>
<u>Figure 25: IDs assigned to each block for composite time ordering .....</u>	<u>42</u>
<u>Figure 26: Resolving boundary artifacts by sharing x, y and z slices.....</u>	<u>44</u>
<u>Figure 27: Compositing ordered images.....</u>	<u>46</u>
<u>Figure 28: Adaptive-resolution image compared to uniform resolution image.....</u>	<u>47</u>
<u>Figure 29: Adaptive-resolution performance verses uniform resolution performance.....</u>	<u>48</u>
<u>Figure 30: Adaptive-resolution images: Different choices of high-resolution blocks.....</u>	<u>49</u>
<u>Figure 31: Adaptive-resolution: Different choices of high-resolution blocks .....</u>	<u>50</u>
<u>Figure 32: Specifications of Zaphod (cluster) .....</u>	<u>52</u>
<u>Figure 33: main.cpp: Creating processes using MPI for parallel temporal visualization. 53</u>	
<u>Figure 34: main.cpp: Creating processes using MPI for parallel temporal visualization. 54</u>	
<u>Figure 35: Parallel temporal visualization: 40 high resolution frames .....</u>	<u>55</u>
<u>Figure 36: Parallel temporal visualization: 40 medium resolution frames .....</u>	<u>56</u>
<u>Figure 37: Parallel temporal visualization: 40 low resolution frames .....</u>	<u>56</u>

# **ABSTRACT**

## **VOLUME VISUALIZATION OF TIME-VARYING DATA USING PARALLEL, MULTIRESOLUTION AND ADAPTIVE-RESOLUTION TECHNIQUES**

by

Sadaf Shams

University of New Hampshire, December, 2006

This paper presents a parallel rendering approach that allows high-quality visualization of large time-varying volume datasets. Multiresolution and adaptive-resolution techniques are also incorporated to improve the efficiency of the rendering. Three basic steps are needed to implement this kind of an application. First we divide the task through decomposition of data. This decomposition can be either temporal or spatial or a mix of both. After data has been divided, each of the data portions is rendered by a separate processor to create sub-images or frames. Finally these sub-images or frames are assembled together into a final image or animation. After developing this application, several experiments were performed to show that this approach indeed saves time when a reasonable number of processors are used. Also, we conclude that the optimal number of processors is dependent on the size of the dataset used.

# INTRODUCTION

A huge number of applications exist for volume visualization of static data sets like CT/MR scans, or time-varying data sets like pressure and temperature. Most of these are serial and work on stand-alone computers using full-data resolution. These applications may take up a large amount of time for rendering big datasets and are unable to take advantage of multi-processor machines or computer clusters. This problem can be solved by using volume visualization software that is capable of operating under parallel computing environments. Parallel computing allows the computing power of a large number of machines to be harnessed, which allows problems of much greater complexity to be solved at very low cost using existing resources. Parallel computing software, such as MPI[10] can achieve high processing speeds for large, distributed datasets without compromising quality. Also, the datasets for time-varying volume visualization can be very large and hence they might not fit into the memory of one processor. By dividing the data up over several processors for rendering we can reduce the memory demands on each processor.

We devised a parallel visualization technique for multiresolution and adaptive-resolution data. Resolution is an important factor to be considered since often it is sufficient to have an overall low-resolution visualization with the option of zooming into higher resolution when and where needed. It might also be the case that the data is too large for interactive visualization at full resolution and hence a lower resolution is needed for that purpose. Sometimes only a portion of the higher resolution is needed as the *focus*

region and the rest can be viewed at a lower resolution as the *context*. This technique coupled with parallel visualization can increase efficiency and allow much larger data sets to be rendered interactively. Both spatial and temporal parallel visualization techniques have been implemented that include variations in both space resolution and time resolution. Furthermore, these parallel techniques have been modified to take into account adaptive-resolution data as well.

This project aims at speeding up the visualization process by providing a parallel rendering framework that allows distributing the processing and data over a cluster of machines or several processors of a multiprocessor system. Given a data set, its format and the number of processors available for computational purposes, this application performs fast, high-quality rendering by distributing the workload across processors. Each processor renders its own data portion and the results are then assembled together.

The goal of this project is to render data fast enough to allow for interactive visualization. The options to run the visualization using a single processor or multiple processors are both available since for small datasets visualization may be faster using a single processor if the time saved by parallel rendering is small compared to the compositing overhead of the parallel approach.

# CHAPTER 1

## BACKGROUND AND RELATED WORK

### 1.1 Grid Visualization

Recently, there has been renewed interest in parallel visualization algorithms because of the availability of commodity computer clusters such as Beowulf [7] and the rapid rise of Grid Computing [12]. Shalf and Bethel [9] were among the first to present the vision of a grid-based visualization system. They state that the basic issue that prevents the grid from being used by current visualization systems is the nature of the existing visualization applications. These applications are designed to work on a serial system and cannot take advantage of the grid. Therefore the first step to take while moving to the grid-based system is to create applications that allow for parallel rendering. Brodlie *et al* [3] modified an existing visualization system IRIS Explorer, to allow it to work on the grid. They also demonstrated how this tool would be useful to scientists through two applications: the pollution dispersion visualizer and the PSE for elasto-hydrodynamic lubrication. Bhaniramka *et al.* [2] very recently developed the OpenGL Multipipe SDK called MPK. MPK is a toolkit that allows the creation of scalable parallel applications using OpenGL. It provides a flexible distribution approach by allowing users to choose from a range of decomposition strategies such as data decomposition, screen decomposition and eye decomposition.

## 1.2 Load Distribution Issues

The first step to develop a parallel algorithm is to divide the tasks among various processors. Basically three types of parallel rendering techniques exist: *sort-first*, *sort-middle* and *sort-last* as identified by Molnar *et al* [8]. In each of these the sort from object space to screen space occurs at a different point.

The *sort-first* approach divides the screen up into portions and renders each portion separately. Eventually all rendered portions simply need to be pasted together on the screen. The Chromium System uses the *sort-first* approach to distribute rendering work to the different nodes in a cluster as discussed by Bethel [1]. Chromium is used to drive multi-projector displays on clusters of computers and it initially used *sort first* to distribute the graphics primitives over the nodes in the cluster before the transformation and lighting stages of rendering.

The *sort-middle* approach distributes the primitives between the transformation/lighting stage and the rasterization stage of rendering. Williams and Hiromoto [13] modified the Chromium system to use the *sort-middle* approach. Although, as discussed earlier, Chromium initially used the *sort-first* approach, network delays cause this approach to be inefficient. Hence William and Hiromoto [9] came up with a *sort-middle* approach that allows the Chromium system to have a frame-rate that is twice as large as the frame-rate attained by the *sort-first* approach.

The *sort-last* approach divides the dataset into portions rendered separately and all sub-images are composited at the end. Cavin, Mion and Filbois [4] use the *sort-last* approach to visualize large datasets on commodity off-the-shelf (COTS) clusters. The data is divided up and distributed among different nodes for rendering. Then they use

parallel compositing techniques to reduce the compositing overhead that has to be incurred for the *sort-last* approach.

The schemes described above are all based on *spatial* decomposition. *Temporal* decomposition schemes also exist. Bhaniramka *et al* [2] describe two temporal techniques: *frame multiplexing* and *data streaming*. In frame multiplexing, each processor is assigned a group of unique time steps to render. The division is such that all processors are kept busy and the frame generation rate matches the frame display rate. In data streaming, all processors work jointly on each time step. Each processor adds a little to the frame for one time step and then moves on to the next time step.

### 1.3 **Parallel Rendering**

The second step towards creating parallel applications is to develop a parallel rendering algorithm. For this purpose ray tracing has been the most popular algorithm since it can be conveniently altered to work on a parallel system. Ma *et al.* [6] were among the first who presented a divide and conquer ray-traced volume rendering algorithm. They used the existing volume ray-tracing scheme presented by Levoy [5] and modified it to break each ray into segments. Each segment is processed separately in parallel. Eventually all segments are combined together in the compositing step.

Some very recent studies have tried to add to the benefit of parallel rendering by using multiresolution techniques. Wang *et al.* [11] describe a parallel multiresolution volume-rendering framework that uses a wavelet-based time-space partitioning tree. Each processor has a copy of this tree and is pre-assigned data blocks from the tree to render. After all sub-images have been rendered, they are composited into the final image.



## CHAPTER 2

### PARALLEL VISUALIZATION

Our application, *STARVolume*, supports parallel rendering and hence can take advantage of the resources available in a parallel computing environment. For spatial parallel rendering, *STARVolume* uses the sort-last approach discussed by Cavin, Mion and Filbois [3]. This approach allows huge datasets to be visualized at high speeds and it is possible to divide the workload equally among processors. For temporal parallel rendering, *STARVolume* uses frame multiplexing discussed by Bhaniramka *et al* [1] since that allows each time step to be needed by only one processor. Ray tracing is used for rendering since it is inherently parallel in nature and can take effective advantage of the parallel computing environment. Our parallel ray tracing algorithm was based on the approach discussed by Ma *et al.* [5].

*STARVolume*'s parallel visualization component is comprised of three basic modules. The *data division* module specifies how the data is divided among the different processors. The *Renderer* module allows each thread/processor to render a certain portion of the data. The *Assembly* module encapsulates the conversion of sub-images into a final image or frames into an animation. Next we discuss the application architecture, followed by the three modules.

## 2.1 Application Architecture

Figure 1 shows the basic architecture. A number of processors are available and one processor acts as the controller. The controller divides data and sends portions of it to other processors for rendering. When the other processors are done, they send back the images to the controller for assembling.

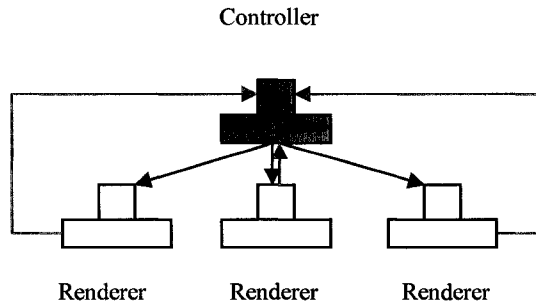


Figure 1: Basic architecture

### 2.1.1 Test Data

We use two different data sets to demonstrate our algorithms. To show the temporal decomposition algorithms, we use time-varying magneto-hydrodynamics data (MHD) produced by numerical simulation. This data was generated from research into solar wind activity done at the Space Science Center at the University of New Hampshire. The simulation records many physical attributes, such as particle velocity, current density, magnetic field, and pressure.

To show the spatial decomposition approach, we used a static 3D dataset where each byte represents the density of a point of the MR scan of human head and brain.

### 2.1.2 Temporal Decomposition

Figure 2 shows the output when using the MHD pressure data at the lowest resolution (98\*28\*28). This shows the first time step. The data source that this application uses has data arranged in a space-time tree of varying resolutions. The

application starts by using the lowest resolution data from the tree and the user can jump to higher resolutions in space or time by using the appropriate resolution buttons on the GUI. Through this scheme, the user gets both fast renderings at low resolutions and high-quality renderings using high resolution. It also provides the user with a lot of flexibility in terms of features that are important (time or space). For instance, if time is less important, the user can choose to view the data at the lowest time resolution but higher space resolution.

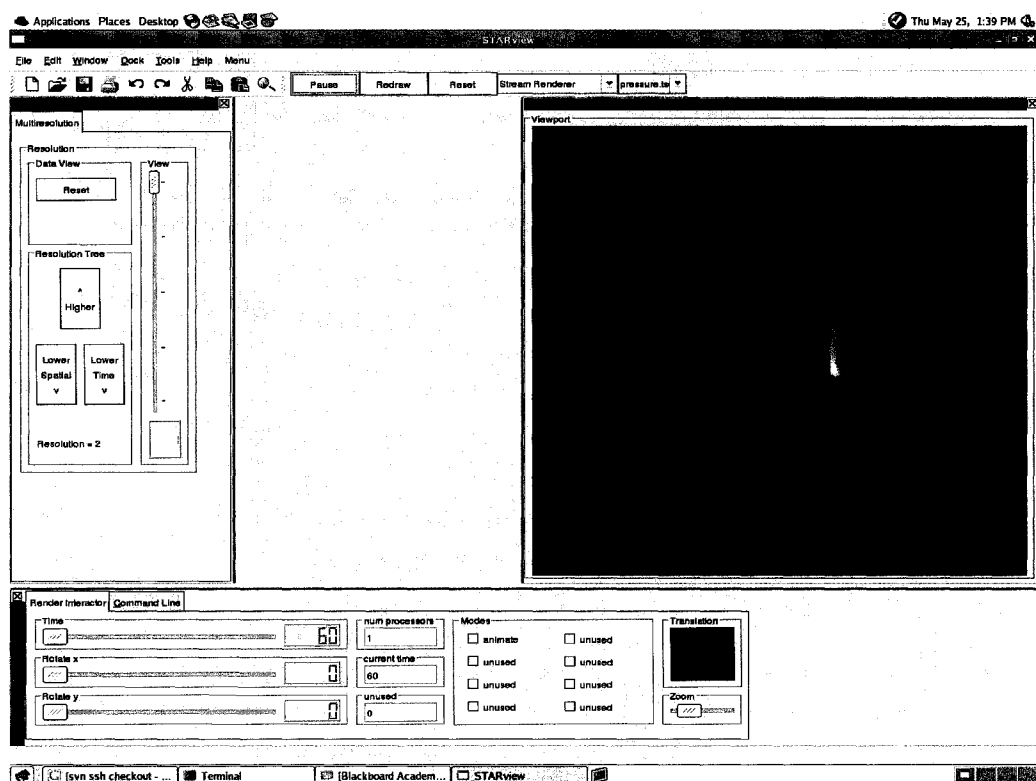


Figure 2: GUI and output (time)

### 2.1.3 Spatial Decomposition

Figure 3 shows the output of this application using spatial decomposition of the MR scan data comprised of 84 slices of  $128 * 128$  bytes. The visualization of this data with this parallel algorithm appears to be exactly the same as that of the existing serial approach; the user can see no difference.

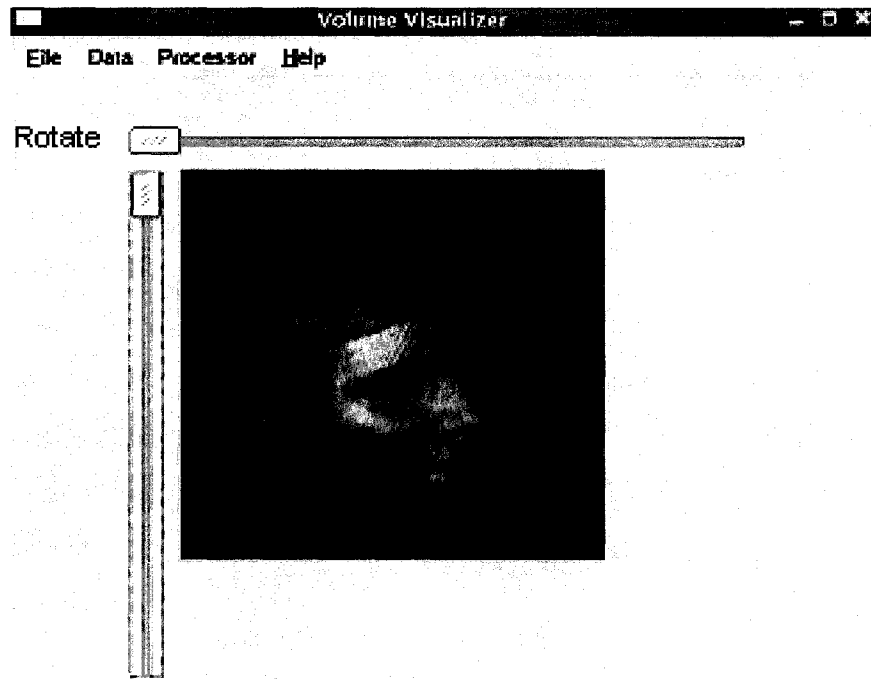


Figure 3: GUI and output (space)

## 2.2 Data Division

The first module of this application is the data decomposition module. We can either divide the data on the temporal domain only or on the spatial domain only or a combination of both. Temporal division can be used for volumes that have more than one time step. For mixing temporal and spatial division, two possibilities exist: spatial distribution within each time step or spatial distribution over time. We have implemented temporal only and spatial only divisions and these are described below.

### 2.2.1 Temporal Division

#### 2.2.1.1 Temporal Division of Uniform Resolution Data

Temporal decomposition is usually very efficient because data for different time steps often exists in different files. Each processor can be assigned the responsibility to

render a different set of files. Unlike spatial decomposition, we do not need to break up a single file between processors. Hence the preprocessing step requires less work. It is possible for different time steps to be placed on different processors. Hence in this case very large datasets that do not fit in the memory of one processor can be rendered by placing only a few time steps out of the total time steps on each computer.

Each frame represents a time step and these frames have to be displayed in sequence to generate the animation. Figure 4 shows how the time steps are divided among processors. In this case there are nine time steps and time steps 1,4,7 are assigned to the first processor, time steps 2,5,8 to the second processor and time steps 3,6,9 to the third processor. The first processor displays frame 1 and then starts work on frame 4. By this time the second processor has completed rendering frame 2 so it displays this frame and starts work on frame 5. Similarly now the third processor is done with frame 3 and it displays it and starts work on processor 6. Hopefully, the first processor should now be done with frame 4 and hence frame 4 is displayed. This cycle continues in a similar manner. Ideally, there would be an initial time lag to generate the first frame but after that all frames should be ready before they are needed and hence can be displayed without any lag.

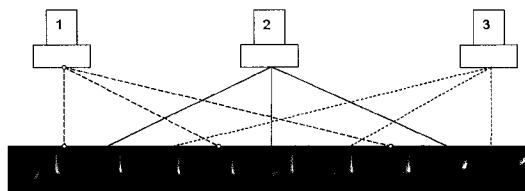


Figure 4: Time/frame decomposition

### 2.2.1.2 Temporal Distribution of Adaptive-Resolution Data

The division described above ensures that all processors have equal work to do if all time steps are of the same resolution. This is not the case for adaptive-resolution data. If one time step is of higher resolution while another one is of lower resolution or if resolutions vary within each time step, this scheme may not be very efficient. In this case, new schemes are required that assign time steps to each processor taking into account the resolutions of these time steps so that the processors assigned low resolution data get more time steps while the ones assigned high resolution data get fewer time steps. Figure 5 shows this kind of decomposition. The first processor is assigned the high resolution frame 1. Since it takes longer to generate the high resolution frames, processor 2 is assigned the next three consecutive low resolution frames. After that, processor 1 gets the fourth frame. In total processor 1 gets three high resolution frames to work on and processor 2 gets six low resolution frames, so that work is divided more equally between the two processors.

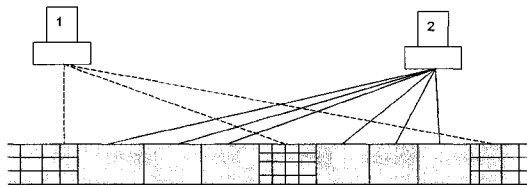


Figure 5: Multiresolution time/frame decomposition

## 2.2.2 Spatial Division

### 2.2.2.1 Spatial Distribution of Uniform Resolution Data

The spatial option divides one data file (a single time step) into portions so that each portion can be rendered separately as displayed in figure 6. A number of methods exist for partitioning the data. We have chosen to partition it along the z-axis so that each processor gets a fixed number of slices to render. This is a relatively easy and efficient approach. It makes reading the data from the file very efficient when slice order equals storage order since each slice is contiguous in the file and its starting seek point in the file is easily determined. Also, this technique allows for efficient compositing as well if the slice order is the same as storage order. Since a very clear front-to-back ordering can be established, the ray segments can be combined without any additional mappings.

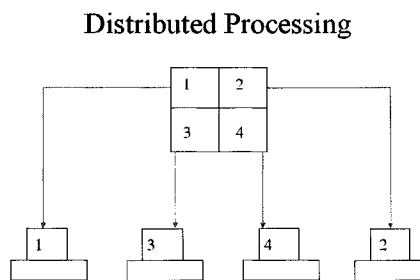


Figure 6: Data decomposition

An attempt has been made to divide data as equally as possible but this cannot be achieved at all times. If the number of processors available is not a multiple of the data slices, equal sized data portions cannot exist. To resolve this issue some processors may receive a little more or less data at times. If the data has uniform resolution, dividing it equally by slice ensures that all processors finish up at almost the same point in time and hence any one processor would not cause the compositor to wait.

The algorithm that can be used to accomplish almost equal data division is described below.

s: number of slices

n: number of processors

$P_1 - P_n$ : processors available

<u>Processor</u>	<u># of Slices</u>
------------------	--------------------

$P_1 - P_{s \% n}$ :	$\lfloor s / n \rfloor + 1$
----------------------	-----------------------------

$P_{(s \% n) + 1} - P_n$ :	$\lfloor s / n \rfloor$
----------------------------	-------------------------

As an example, if we have 31 slices and 5 processors, this algorithm assigns 7 slices to the first processor and 6 slices to all the others. For the 34 slices/5 processor example, it assigns 7 slices to the first four processors and 6 slices to the last one. This algorithm ensures that the number of slices assigned to each processor would not differ by more than one.

When the data is divided into portions that do not overlap, a naïve solution for most volume rendering algorithms could result in boundary artifacts appearing while compositing. For example, a black stripe could appear at the boundary of each sub-image in the final composited image. This problem occurs since the opacity and color calculations do not take place at the boundary where the data division occurs. Hence we need to have a small overlap of data among adjacent data sets to remove this artifact. Before compositing, these extra/duplicated slices of data are removed to prevent the image from being stretched due to the addition of extra slices.

#### **2.2.2.2 Spatial Distribution of Adaptive-Resolution Data**

For adaptive-resolution data, rather than dividing the data into equal portions, *workload* needs to be divided equally among the processors. Spatial adaptive-resolution



data sets have certain portions (blocks) that have high resolution while others that have lower resolutions. Dividing this kind of a data set equally by spatial extent would mean that processors working on low-resolution portions would complete their work before those working on high-resolution portions and would sit idle after that. To resolve this issue, the processors that are assigned high-resolution blocks should be assigned fewer blocks than those assigned low-resolution blocks or a good mix of low-resolution and high-resolution blocks should be assigned to each processor so that they complete their work at almost the same time. Detailed techniques for implementing this approach are discussed in section 5.

### **2.3 Rendering**

The data portion passed to the renderer is a three-dimensional array of data. Each element of the array may represent any data field such as tissue density or temperature. One common volume rendering algorithm is described below. This algorithm based on ray tracing is currently being used but it should be simple to use other rendering algorithms instead to produce similar results. For the ray-tracing algorithm, first each element is converted into a voxel. Next each voxel is classified by assigning opacity to it. A transparent voxel has the opacity of zero while a completely opaque voxel has the opacity of one. High opacities are assigned to those voxels that should be visible and low opacities are assigned to those that should be hidden. Classification is done using lookup tables and transfer functions that map the scalar data to opacity. Next shading/lighting is done which calculates a color for each voxel. The standard Phong shading equation has been used for this purpose. A gradient vector is calculated for each voxel that is then used to compute the direction in which light is reflected from a voxel. Finally a set of parallel

rays is cast from the eye to the voxels and the voxel values are projected to the view plane to produce a two-dimensional image dataset. During this process, the different processors do not need to communicate with each other so there are no communication overheads or network delays. The VolPack library functions (<http://www-graphics.stanford.edu/software/volpack/>) have been called for classification, shading and some other well-known rendering procedures.

### **2.3.1 Temporal Rendering**

For temporal rendering, each processor generates the final image for its own time step. This image represents a frame of the animation. Unlike spatial rendering, this does not generate partial images but each image is complete in itself and is ready to be displayed. Figure 7 shows the filmstrip generated for the animation that shows how pressure changes over time. Six processors are used in this case and each picture in this strip is rendered by a separate processor.

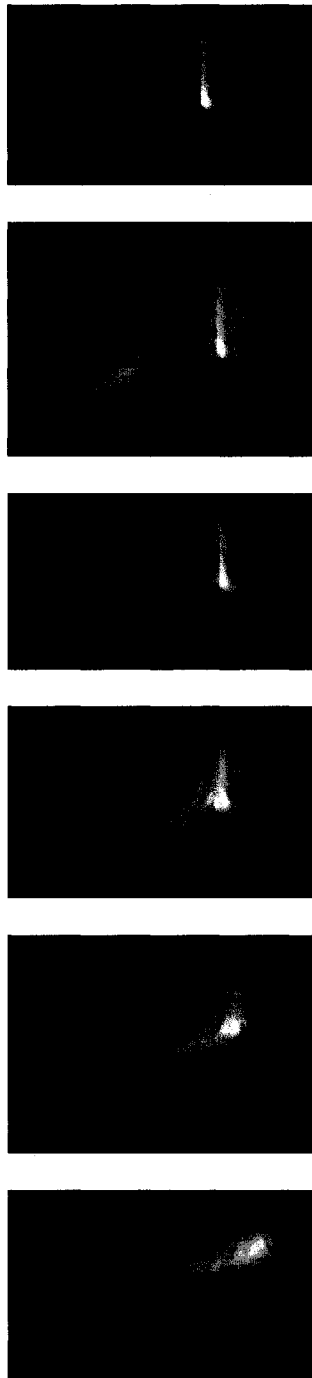


Figure 7: Time frames with 6 processors

### **2.3.2 Spatial Rendering**

Figure 8 shows the results of rendering when the data is divided into two portions and two threads are created, one for rendering each portion of the data. Thread 1 produced the right side of the head while thread 2 produced the left side. In these partial renderings some of those features are visible that would not be present in the final image. For instance the image on the right shows the interior of the brain. The compositor would eventually hide this information when it realizes that some other thread has produced an opaque image that lies in front of the brain segment.

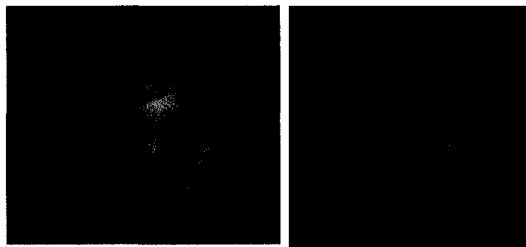


Figure 8: Partial images with 2 processors

Figure 9 shows the partial images when the data is divided into four portions. Each thread renders 21 slices of data in this case.

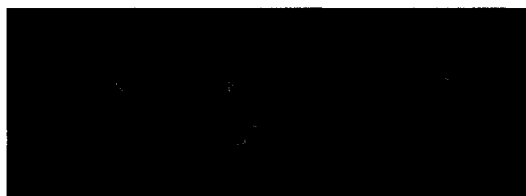


Figure 9: Partial images with 4 processors

### **2.4 Assembly**

The results from the different processors need to be assembled together before displaying them. For temporal division this involves a simple sequencing of frames while for spatial decomposition, we need more complex compositing algorithms.

### 2.4.1 Temporal Sequencing

For displaying animation, frames need to be presented sequenced according to increasing time. When a processor has completed its time step, it simply adds the image to the pool of frames and moves on to the next time step assigned to it. In this pool of frames the time steps can come in any order depending on processor speed and the division of time steps among the processors. The sequencing step starts by picking up and displaying the frame for the first time step from the frame pool and then it increments the time and looks for and displays the frame for this new time from the frame pool. This cycle continues until all frames are displayed. Figure 10 shows the frame sequencing process. There are three processors and each processor is given three frames to render. Once each processor is done rendering its frame, it sends it to the frame pool. This diagram shows a situation where perfect synchronization has been achieved; as soon as frame 1 is displayed, frame 2 is ready, and as soon as frame 2 has been displayed, frame 3 is ready.

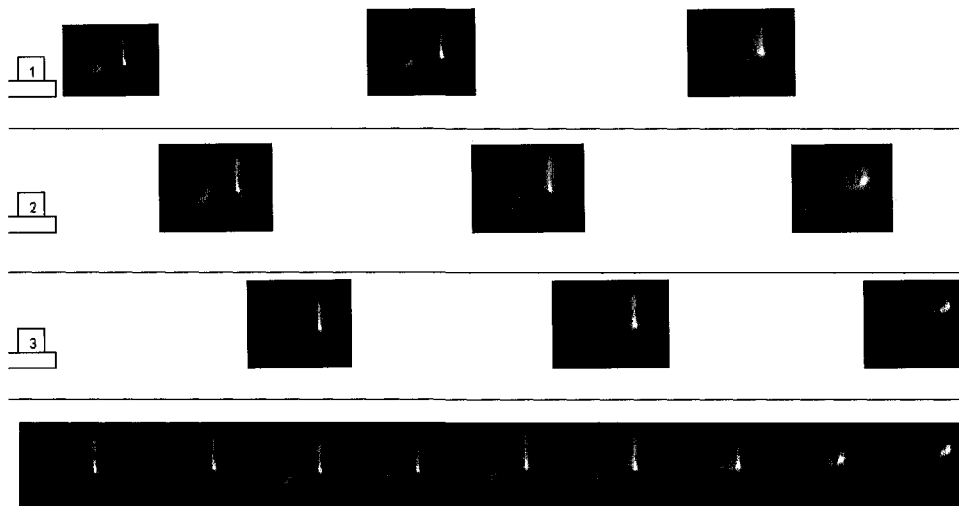


Figure 10: Frame sequencing

There are issues that may arise with this scheme. It is possible that if one processor slows down for some reason, a particular frame may arrive late and the animation has to wait for it. In this situation it would seem as if the speed of the animation is fluctuating. Sometimes the frame-rate is very fast and at times it slows down and then speeds up again. To resolve this problem some time lag can be introduced at the start to make sure that there are always enough frames in the pool to avoid any wait. Inconsistent frame-rates can also occur with multiresolution and adaptive-resolution data. In this case the frame generated from higher resolution data may take more time to display (due to larger image size for instance) than the frame with smaller resolution, again creating animations that are not very smooth. To resolve this problem, time lags can be introduced so that each frame takes a fixed time to display and this fixed time can be set to the time needed by the highest resolution frame to display.

This sequencing phase is very easy to implement and has very little additional overhead unlike the huge overhead of the compositing phase needed by spatial decomposition.

#### **2.4.2 Spatial Compositing**

Once each thread has completely processed its own data portion, the sub-images are collected by the main thread and composited together into one final image. The image compositing process merges all separate ray segments obtained from the different threads. The colors and opacities of each of the sub-images are merged together to render a final image. When all sub-images are ready they are composited in a front-to-back order. The compositing is currently done by the main thread after all the other threads

finish and hence might impose some overhead. But the overall structure of the application allows for a shift to more efficient parallel compositing techniques.

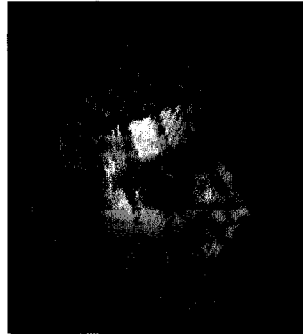


Figure 11: Boundary artifacts when compositing 10 sub-images

The boundary artifacts displayed in figure 11 were visible when using 10 processors with a naïve compositing approach. In this approach the data was divided into disconnected portions and the sub-images produced by each portion were composited without any editing. The new approach for removing this artifact uses data with shared slices and removes the extra slices before compositing.

While compositing, boundary lines of each sub-image are visible if the data is divided into discrete portions. For instance figure 12 shows the first three slices go to processor 1, the next three slices go to processor 2 and the last three slices go to processor 3. This kind of division shows the boundaries of the sub-images.

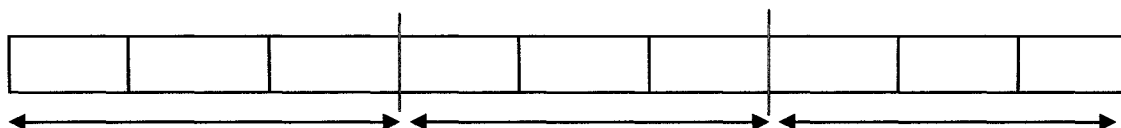


Figure 12: Data division that causes boundary artifacts

To resolve this problem, we need to have overlapping data boundaries. Each processor reads one extra slice at the start and one slice at the end. Later after assigning opacities and colors to all slices; it drops these extra slices. The first and the last processor are two exceptional cases. The first processor reads only one extra slice at the end while the last processor reads only one extra slice at the start. Figure 13 shows this division for the 9-slices, 3-processors example. Processor 1 gets four slices, processor 2 gets five slices while processor 3 gets four slices.

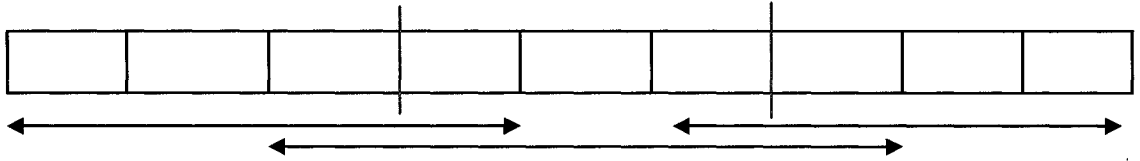


Figure 13: Data division that prevents boundary artifacts

In general the following formulas can be applied, assuming that  $n$  is the number of processors and  $s$  is the number of slices that each processor would get with no overlap ( $s=3$  in table 1). When using this approach the composited image displays no boundaries and looks the same as the image generated without using parallel rendering.

Processor	First Slice	Last Slice
$P_1$	0	$s + 1$
$P_i (i = 2 \text{ to } n-1)$	$(i - 1) * s - 1$	$(i * s) + 1$
$P_n$	$(i - 1) * s - 1$	$i * s$

Table 1: Data division that causes boundary artifacts



## CHAPTER 3

### MULTIRESOLUTION VISUALIZATION

In addition to parallel visualization, another technique that can be used for efficient rendering of time-varying data is multiresolution visualization. To implement multiresolution visualization we usually convert the data to lower resolution levels. For time-varying data, we can either use space-resolution or time-resolution. Consider the example of a data set with 100 time steps (1,2...100) where each time step has dimensions  $256*256*256$ . A lower *space* resolution for this data could be 100 time steps (1,2...100) where each time step has its dimensions cut down to  $128*128*128$ . Here the dimensions of each time step are changed. Whereas a lower *time* resolution for this data would be 50 time steps (1,3,5...99) where each time step has dimensions  $256*256*256$ . Here the number of time steps has changed.

#### 3.1 Space Resolution

Using lower space resolution enables efficient rendering by decreasing the time it takes to render each time step since each time step now has smaller dimensions. Low-resolution data sets render much faster since less I/O needs to be done for reading the smaller data sets and less computation needs to be done to render smaller data. While rendering, the lowest resolution level is often used for the initial view. Although this

decreases the quality of the image, the rendering is faster. During the interaction, a user can identify an interesting region of the data and “zoom” into that region to display it at higher resolution. This increases the quality of the image but may slow down the animation if the small spatial extent of the view does not offset the higher resolution of the data.

Performance analysis using different resolution levels was conducted on a single-processor machine. Pressure data was used with original resolution of  $392 * 112 * 112$ . Two lower resolutions were created using this data. This was done by generating a multiresolution hierarchy based on a wavelet transformation to produce lower resolution data and error, which is stored on the disk in a directory tree. For our discussion, we label the three resolution levels as follows:

High (original):  $392 * 112 * 112$

Medium:  $196 * 56 * 56$

Low:  $98 * 28 * 28$

Figure 14 shows the average time it took to render a frame using the three resolution levels (on a test machine with specifications listed in table 3). Rendering the high resolution (original) took 55.7 seconds; the medium resolution took 6 seconds while the low resolution took only 1.4 seconds. This shows that using the medium resolution saved around 49.7 seconds, while using the low resolution saved 54.3 seconds.

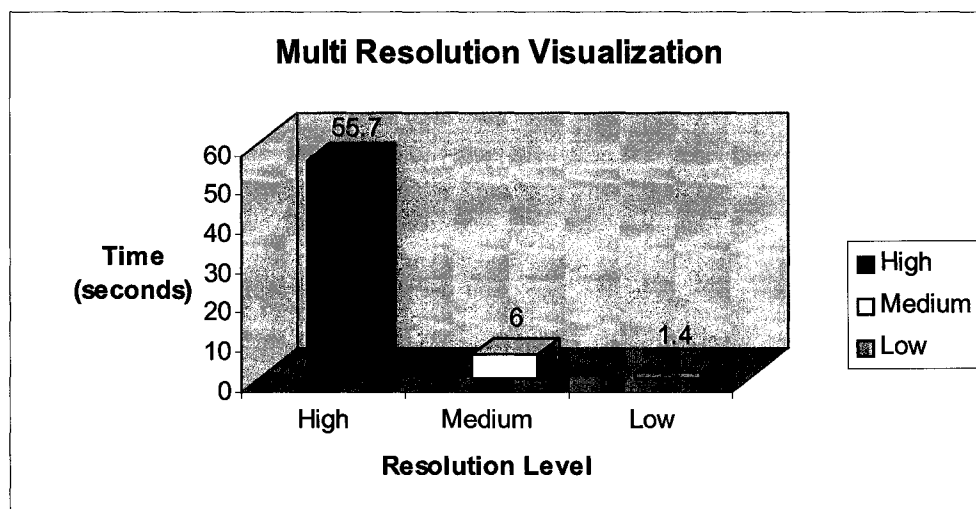
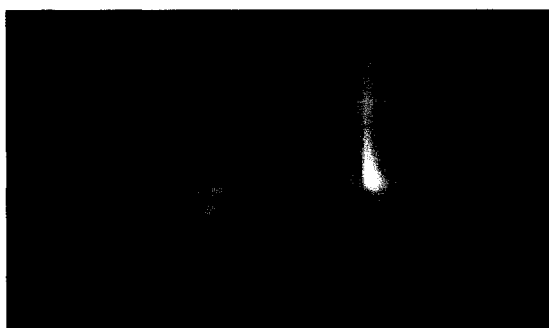


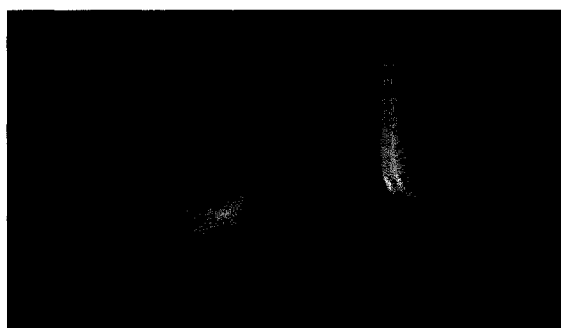
Figure 14: Rendering time per frame using 3 space resolution levels

Figure 15 shows the difference in quality between the images generated by the three resolution levels. We can observe that although the time saved is huge, the quality compromised might be acceptable for less important regions. So it would be best to use one of the lower resolution levels as default visualization and jump up to higher resolution for important/interesting regions.

Low Resolution



Medium Resolution



High Resolution



Figure 15: Difference in quality between low, medium and high resolutions

### 3.2 Time Resolution

Using a different time resolution means increasing or decreasing the number of time steps that are rendered. Whereas when using space resolution we can only decrease the resolution down from the original, when using time resolution we can increase the resolution as well. Also, decreasing time resolution can be done on the fly while decreasing space resolution requires pre-processing. Consider an experiment that generates 10 frames at 2-second intervals. This experiment would have 10 time steps labeled 0,2,4...18 where the labels represent the time in seconds when the result was generated. The original resolution of this data is 10 since there are 10 time steps. To decrease the time resolution of this data we can simply discard every other time step to get 5 time steps labeled 0, 4, 8 ... 16. Another approach is to average several time steps by using a wavelet transform and use this average instead of the original time steps. This loses less information than simply discarding half the data. For increasing the time resolution, we can interpolate between every two time steps to get a new time step. This would generate 20 time steps labeled 0,1,2,3...19. In this case time step 1 for instance has been calculated by interpolating between the values in time step 0 and 2. Figure 2 summarizes this.

Time Resolution	# of Frames	Labels of Frames
High	20	0,1,2,3...19
Medium (original)	10	0,2,4...18
Low	5	0, 4, 8 ... 16

Table 2: Example with 3 levels of time resolutions

Decreasing the time resolution would help speed up the rendering since fewer frames need to be rendered and this can be done through our application by specifying a time increment. For instance the time increment would be 4 for the example discussed above.

Increasing the time resolution helps obtain a smoother animation because it calculates the intermediate time steps that are not present in the original data. Also, we can save disk space by storing fewer time steps and calculating the intermediate time steps by interpolating between the existing ones. For the example above, time resolution can be increased in our application by specifying an increment of 1.

As the time resolution is increased beyond the original resolution, less I/O is needed but more CPU time is needed to compute the missing time step values. Using the original or lower time resolutions increases the I/O demands while decreasing CPU workload. A good balance of I/O and CPU workload is needed to reach an acceptable computation time, obtain a smooth animation and use disk space efficiently.

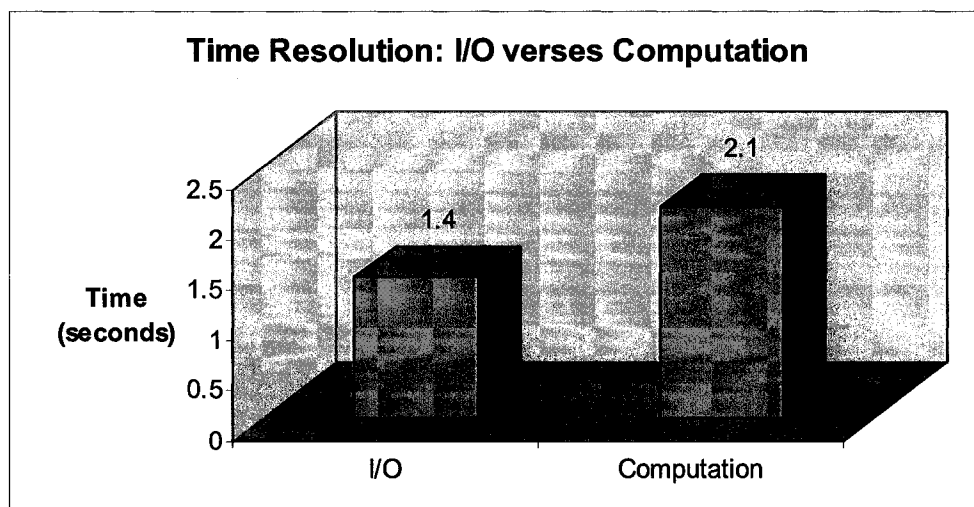


Figure 16: Time taken for I/O-intensive verses computation-intensive rendering

<b>Test Machine Specifications</b>	
vendor_id	GenuineIntel
cpu family	15
model	2
model name	Intel(R) Pentium(R) 4 CPU 2.40GHz
cpu MHz	2399.916
cache size	512 KB
total memory	514116 kB

Table 3: Test machine specifications

Figure 16 shows that it takes 1.4 seconds on average to render a time step (on the machine with specifications listed in table 3) that needs to be read in from a file while it takes 2.1 seconds to render a time step that has to be calculated by using the two surrounding time steps. This increase in time might be acceptable when the user has limited space or is interested in a smoother animation.

These numbers represent the case where data is located on the local machine and would not necessarily hold true if the data is being accessed remotely. Remote access might be used because the local machine does not have enough space or because the data is being shared rather than replicated for every user. If data is being accessed remotely, then I/O would take longer and computation would be relatively faster. In this case interpolating could make more sense than reading each time step in.

## **CHAPTER 4**

### **ADAPTIVE-RESOLUTION VISUALIZATION (SPATIAL PARALLEL)**

Decreased rendering time can also be achieved by using adaptive-resolution data. We only discuss spatial adaptive-resolution here. Whereas in multiresolution, the resolution (dimensions) of the whole data set is changed to a lower one, for adaptive-resolution, portions within a data set can exist at different resolution levels. Hence interesting portions can be viewed at a higher resolution while uninteresting ones can be viewed at lower resolution levels to speed up rendering. Figure 17 shows the image generated using an actual adaptive-resolution data set where portions of interest have higher resolutions (more grid cells) and others have lower resolutions. The portion on the top left corner is rendered at a low resolution since it is simply empty space and very little information is lost by converting it into lower resolution. While the bottom right portion of the image is rendered at higher resolution since that portion is interesting and more information could be lost by converting it to a lower resolution.





Figure 17: Adaptive-resolution data

This section discusses how we can effectively render adaptive-resolution data sets using the spatial parallel approach, so that we can benefit from both parallel visualization and adaptive-resolution.

#### 4.1 Adaptive-resolution Data Format

For parallel visualization of spatial adaptive-resolution data, we use the MHD pressure data. The goal of this project was not to create the AR Data therefore for temporary use we created our own dummy AR Data Representation. This adaptive data set has portions (blocks) with three different resolutions:

High =  $(392 * 112 * 112)$ ;

Medium =  $(196 * 56 * 56)$ ;

Low =  $(98 * 28 * 28)$ .

The user needs to create a data description file that shows how to divide the data up into portions (blocks) of different resolution. Figure 18 shows the format of this file. The first line of this file specifies the total number of blocks. Each of the remaining lines

gives the starting and ending x, y and z values and the resolution for that block. The resolution can have one of the following three values:

high = “/”

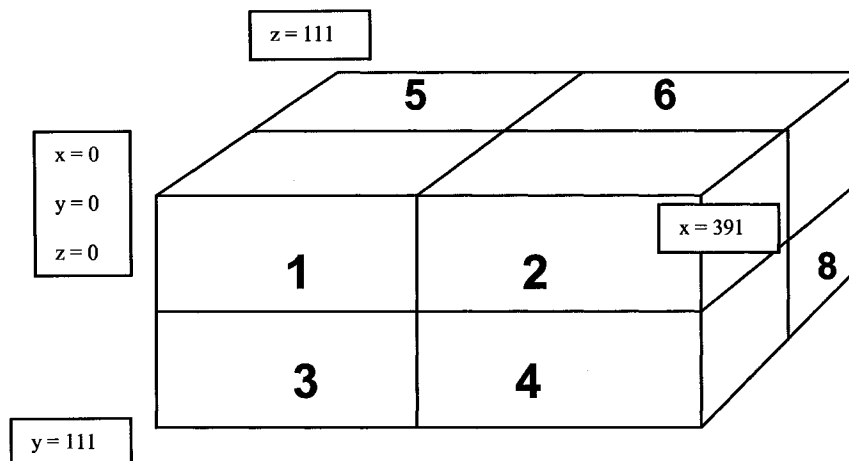
medium = “/s”

low = “/s/s”.

Total Number of Blocks						
Block-1-Start-X	Block1-Start-Y	Block1-Start-Z	Block-1-End-X	Block-1-End-Y	Block-1-End-Z	Resolution
...						
...						

Figure 18: Data format

Figure 19 shows an example data description file and a visual representation of the data blocks. The example uses  $392 * 112 * 112$  data that is divided into 8 equal portions (blocks) where each portions has dimension of  $196 * 56 * 56$ . These portions are block-based rather than slice-based because regions of importance often exist as blocks so it makes more sense to use blocks for adaptive-resolution. The first row in the data represents the total number of blocks (i.e., 8). The second row represents block 1 in the picture and has x values from 0 to 198, y values from 0 to 55 and z values from 0 to 55. This block should be rendered at high resolution. The third row represents blocks 2 with starting values (196, 0, 0) and ending ones (391, 55, 55). This block should be rendered at medium resolution. The rest of the rows represent blocks 3, 4, 5, 6, 7 and 8 in a similar manner.



8						
0	0	0	195	55	55	"/"
196	0	0	391	55	55	"/s"
0	56	0	195	111	55	"/s/s"
196	56	0	391	111	55	"/s/s"
0	0	56	195	55	111	"/"
196	0	56	391	55	111	"/s"
0	56	56	195	111	111	"/s/s"
196	56	56	391	111	111	"/s/s"

Figure 19: Example data description file

This data description format provides the user with a lot of flexibility in terms of portions that are important and should be viewed at higher resolution than others. The user can either use an error-based approach to come up with the appropriate resolution and dimensions for each block or can estimate these values based on previous experience with this data.

After the data description file has been read in, the main processor uses it to distribute the data amongst the other processors. Each of the other processors renders its own data portions and returns the results back to the main processor which then composites these results into a final image. Similar to spatial parallel visualization, this

application also has the three basic components: data division, rendering and compositing. These three components are discussed next.

## **4.2 Data Division**

In case of adaptive-resolution data, the data-description file has already divided the data into distinct blocks that have different resolutions. Hence, in this case the data division component only needs to decide which blocks to assign to a particular processor. Figure 20a shows the data division approach that we used for spatial parallel visualization. This approach divides the data into equal portions according to the number of processors available. In this case, each processor gets only one data block and each block is of almost equal size. In the case of adaptive-resolution data sets, the data description file might have more or fewer blocks than the processors available. If the number of processors is more than number of blocks, the blocks can be subdivided or the rest of the extra processors can be ignored. If the number of processors is less than the number of blocks, we need to decide which blocks to assign to each processor. Assigning an equal number of blocks is not a good solution since low resolution blocks can be rendered much faster than high resolution blocks. To resolve this issue, the processors that are assigned high resolution blocks should be assigned fewer blocks than those assigned low resolution blocks or a good mix of low and high resolution blocks should be assigned to each processor so that they complete their work at almost the same time.

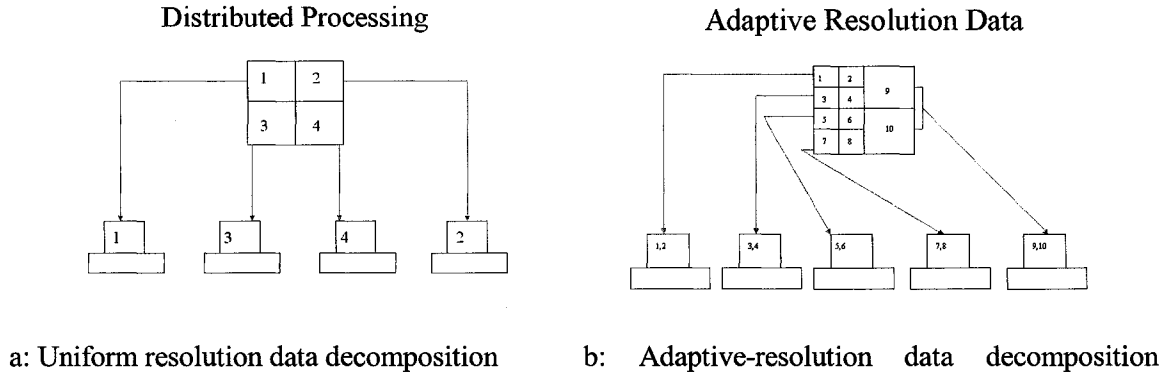


Figure 20: Data decomposition options

Therefore for adaptive-resolution data, rather than dividing the data into equal portions based on blocks, *workload* needs to be divided equally among the processors as displayed in figure 20b. In this figure, blocks 9 and 10 have a low resolution therefore they need less work and are therefore assigned to a single processor. Because blocks 1 to 8 are high resolution, they are assigned in pairs to each of the remaining processors.

We have devised the following algorithm to take into account adaptive-resolution while dividing data among threads. This algorithm divides the data into blocks based on resolution. Each block has a consistent resolution and this resolution is represented by the number of cells in that block. Counting the total number of cells in the data and dividing them by the number of processors give the number of cells each processor should render. But we can't allocate that exact number to each processor since we want each processor to have a block of consistent resolution. Therefore we try to allocate blocks so that the total number of cells that a processor gets is almost equal to the optimal number of cells per processor. This can be done by sorting the blocks according to the cell count of each block in descending order. Next the shuttle algorithm is used where one block is assigned

to each processor first in forward order starting from the first processor to the last processor. The remaining blocks are assigned in backwards order from the last processor to first processor. If there are still more blocks the shuttle algorithm is initiated again. This algorithm continues until there are no more blocks to assign. As soon as a processor is assigned the optimal number of cells or more, it is removed from the list.

# of Processors:  $i$   
Processors:  $p_1 \dots p_i$

Blocks:  $B_1, B_2, \dots B_n$   
Resolutions:  $R_1, R_2, \dots R_n$   
Cell Count:  $C_1, C_2 \dots C_n$

Total number of cells:  $\sum_{(x=1 \dots n)} C_x$   
optimal\_cells\_per\_processor:  $\sum_{(x=1 \dots n)} C_x / i$

#### **Cell Count Algorithm:**

```
sort_blocks_by_cellcount_descending();

while( more_blocks )
{
    current_processor = processor_list.get_next();
    current_block = get_next_block_from_sorted_blocks();

    assign_block_to_processor( current_processor, current_block );

    if( current_processor.cellcount >= optimal_cells_per_processor )
        processor_list.mark_processor_full( current_processor );
}
```

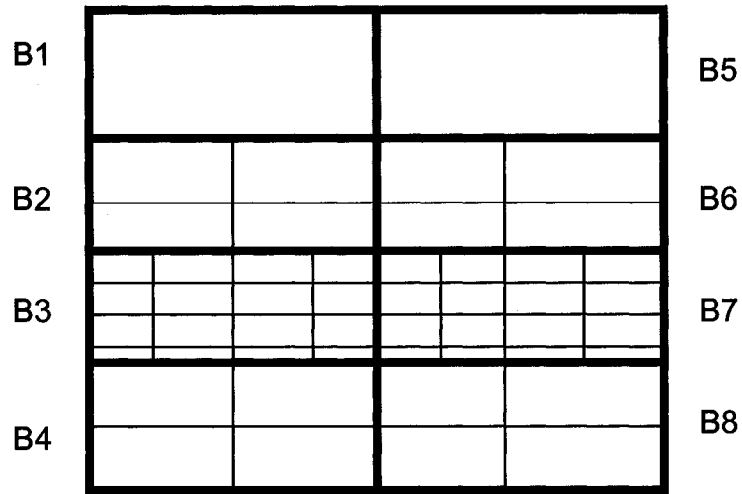


Figure 21: Adaptive-resolution data segmented into 8 blocks

Applying this algorithm to the data in figure 21 works as follows. The first step is the calculation of total cell count as displayed in table 4.

Blocks	B1	B2	B3	B4	B5	B6	B7	B8
Cell Count	1	4	16	4	1	4	16	4

Table 4: Adaptive-resolution data blocks and their cell counts

Next, the total number of cells is calculated and then cells per processor are calculated by dividing this total by the number of processors as displayed in figure 22.

# of Processors: 4 Processors: p1...p4  Total number of cells: 50 cells_per_processor: $50/4 = 12.5$ (round off to 13)
--

Figure 22: Calculating optimal number of cells per processor

After this, blocks are sorted by cell count and assigned processors in forward/backward cycles, removing processors that get 13 or more cells. This is displayed in table 5.

Blocks	B3	B7	B2	B4	B6	B8	B1	B5
Cell Count	16	16	4	4	4	4	1	1
Processor	P1	P2	P3	P4	P4	P3	P3	P4
Processor Status	removed	removed						

Table 5: Sorted adaptive-resolution data blocks with assigned processors

Each processor is assigned the number of cells listed in table 6 at the end of the algorithm, which is not optimal (12 or 13 would have been optimal) but is the best that could have been done keeping resolution consistent within each block.

Processor	Number of cells assigned	Blocks assigned
P1	16	B3
P2	16	B7
P3	9	B2, B8, B1
P4	9	B4, B6, B5

Table 6: Blocks and number of cells assigned to each processor

Also, this is better than the number of blocks that would have been assigned by the equal data division approach as displayed in table 7. For instance, equal data block division would only assign 2 cells to processor P1. The numbers in the division of



(16,16,9,9) is much closer to the optimal number 13 than the ones in the division of (2,8,32,8).

Processor	# of cells assigned by cell count algorithm (equal work division)	# of cells assigned by slicing algorithm (equal data division)
P1	16	2
P2	16	8
P3	9	32
P4	9	8

Table 7: Comparison of equal work division and equal data division approach

After the data has been divided by the main processor using this cell count algorithm, each processor is passed the information regarding the portions it needs to render. Next each processor renders the data blocks assigned to it. This rendering step is discussed in the next section.

### 4.3 **Rendering**

In case of normal spatial parallel visualization, each processor receives one data portion to render and the processor returns the results for that portion to the main processor. In the case of adaptive visualization, each processor receives multiple data portions to render since the data description file for adaptive data may have more blocks than the number of processors available. In this case a processor renders the blocks assigned to it one by one and returns the results to the main processor as each block is done.

Other than that, the rendering techniques for adaptive visualization are similar to that of normal spatial parallel visualization. The image in figure 23 shows the results of

rendering the data described in figure 19. In this case we have eight partial images generated as a result of block wise data division. Images 1, 3, 5 and 7 have been generated from high resolution data since they have a lot of important information. Images 2, 4, 6 and 8 have been rendered from medium resolution data since these mostly have empty space with a small portion of interesting region at one corner. After rendering, these partial images need to be composited together to form the final image.

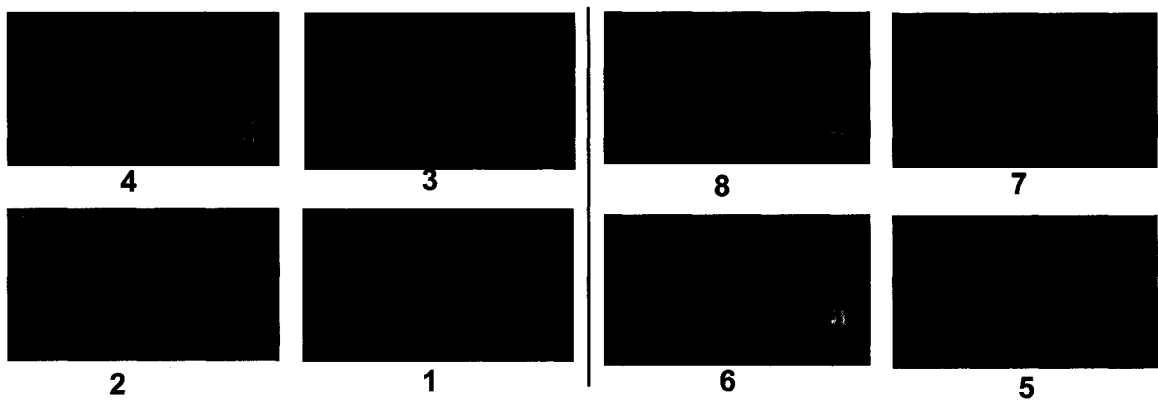


Figure 23: Partial images generated for each data block using data in figure 19

#### 4.4 Assembly

Once all the partial images have been generated and returned, the main processor needs to composite them into one image. Unlike normal spatial data, adaptive spatial data requires compositing even if we have only one processor, since a single processor might be rendering multiple blocks of different resolutions.

Also, the subimages generated may be of different sizes since the data for them had different resolutions. To understand this, we need to differentiate between virtual dimensions and actual dimensions of a data portion.  $196 * 56 * 56$  is the virtual dimension of each block shown in figure 19, since this represents the dimensions of each

block at high (original) resolution. But in reality the data blocks exist in different resolutions and hence have different numbers of data points in them. The total number of these actual data points make up the actual dimensions. Hence in this example the actual dimensions for high resolution blocks would be  $196 * 56 * 56$ , for medium resolution blocks would be  $98 * 28 * 28$  and for low resolution blocks would be  $49 * 14 * 14$ . Since these dimensions are different for each block, the number of points that need to be composited within each subimage also differ.

To overcome this problem, we expand each subimage to its virtual dimensions. An efficient approach would be to set the virtual dimension to the dimension of the largest subimage. For instance if no subimage has a resolution greater than medium, then the dimensions for medium could be set as the virtual dimensions for compositing. These expanded partial images can then be easily composited since they now have the same number of points/cells in them. Figure 24 illustrates this expansion process.

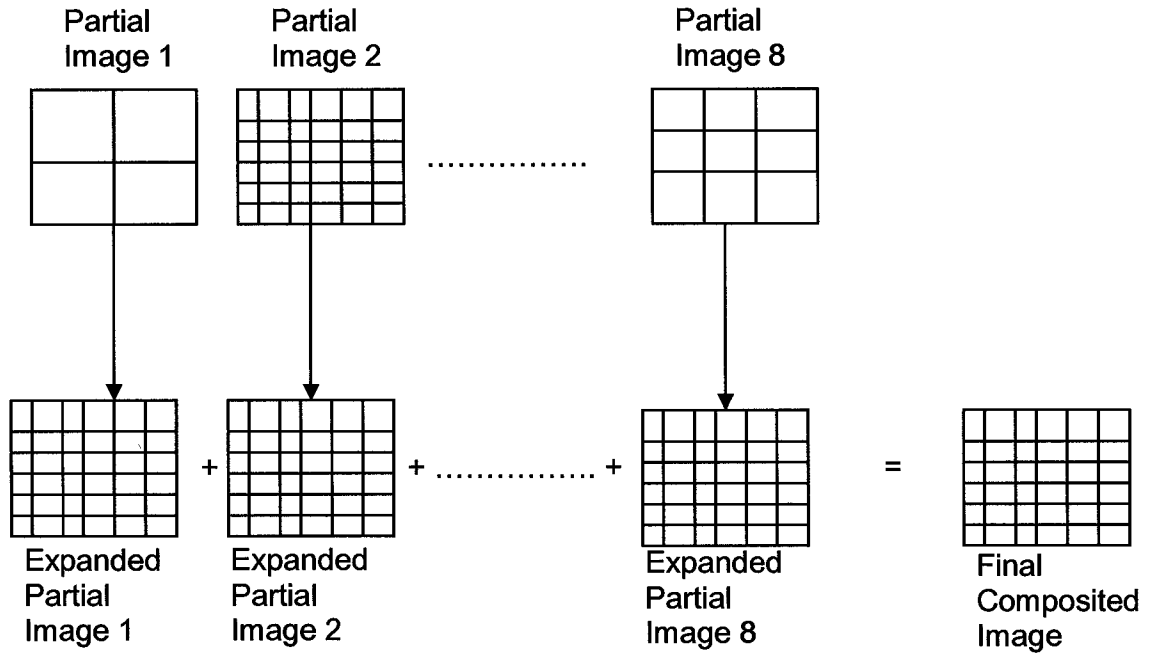


Figure 24: Expansion of partial images of different resolutions

Once all partial images have been expanded, they need to be ordered for compositing. This ordering can be determined by assigning IDs to each block during the data division phase. The approach we used to assign ids was to start from the block at (0, 0, 0) co-ordinates and assign it an ID of 1. Then pick up the next block in the x direction and assign it an ID of 2. In a similar manner IDs are assigned to all the blocks in the x direction until there are no more. After that again start from  $x = 0$  but with the next y block and this process continues until the whole y dimension is covered. Then again start from  $x=0$  and  $y=0$  with the next z block and continue this processes until all blocks have an ID. Figure 25 shows the IDs assigned using this process to blocks for a data set that has 8 blocks. For composite time ordering each partial image gets the ID corresponding to its data block.

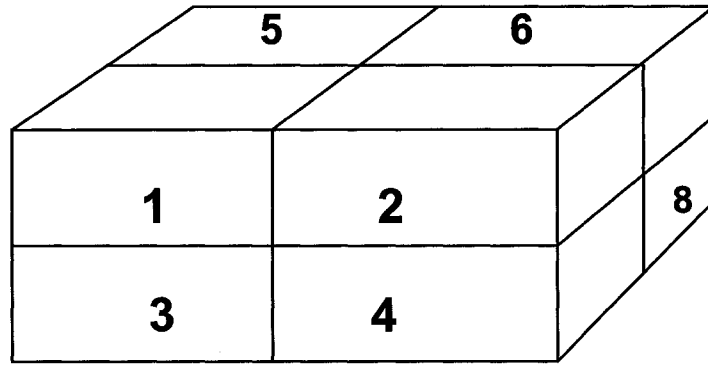


Figure 25: IDs assigned to each block for composite time ordering

Similar to the spatial parallel approach, boundary artifacts were observed during compositing in adaptive visualization too. These boundary artifacts are displayed in the first image in figure 26. This image was generated using a data set with eight blocks. These boundary artifacts occurred because the data was divided into disconnected portions and the sub-images produced by each portion were composited without any editing. To remove this artifact we can use a boundary sharing approach similar to the one discussed earlier in section 3.4.2. There is some added complexity in this case because we divide data by blocks rather than slices. For slicing we had to share the first and the last slice of each portion only but for blocks we need to share slices on each side of the block. For instance a block at the center of the data would need to share 6 slices (left , right , front , back , top , bottom) while the top right corner block might need to share only 3 slices ( bottom , left , back ). A simple algorithm can be used to determine which sides need to be shared. The sides of a data block that coincide with that of the actual data are ignored and the rest are shared.

Figure 26 shows how the boundary artifacts are removed by sharing x, y and z slices. The first image is generated without any boundary sharing. The boundary lines of the eight blocks are obvious in this image. The second image is generated when the x slice is shared by all blocks. This removes the x boundary artifact. The third image is generated when the y slice is shared too in addition to the x slice. This removes the y boundary artifact. Finally the last image removes the z boundary artifact by sharing the z slice as well. The final image generated now has no boundary artifacts.

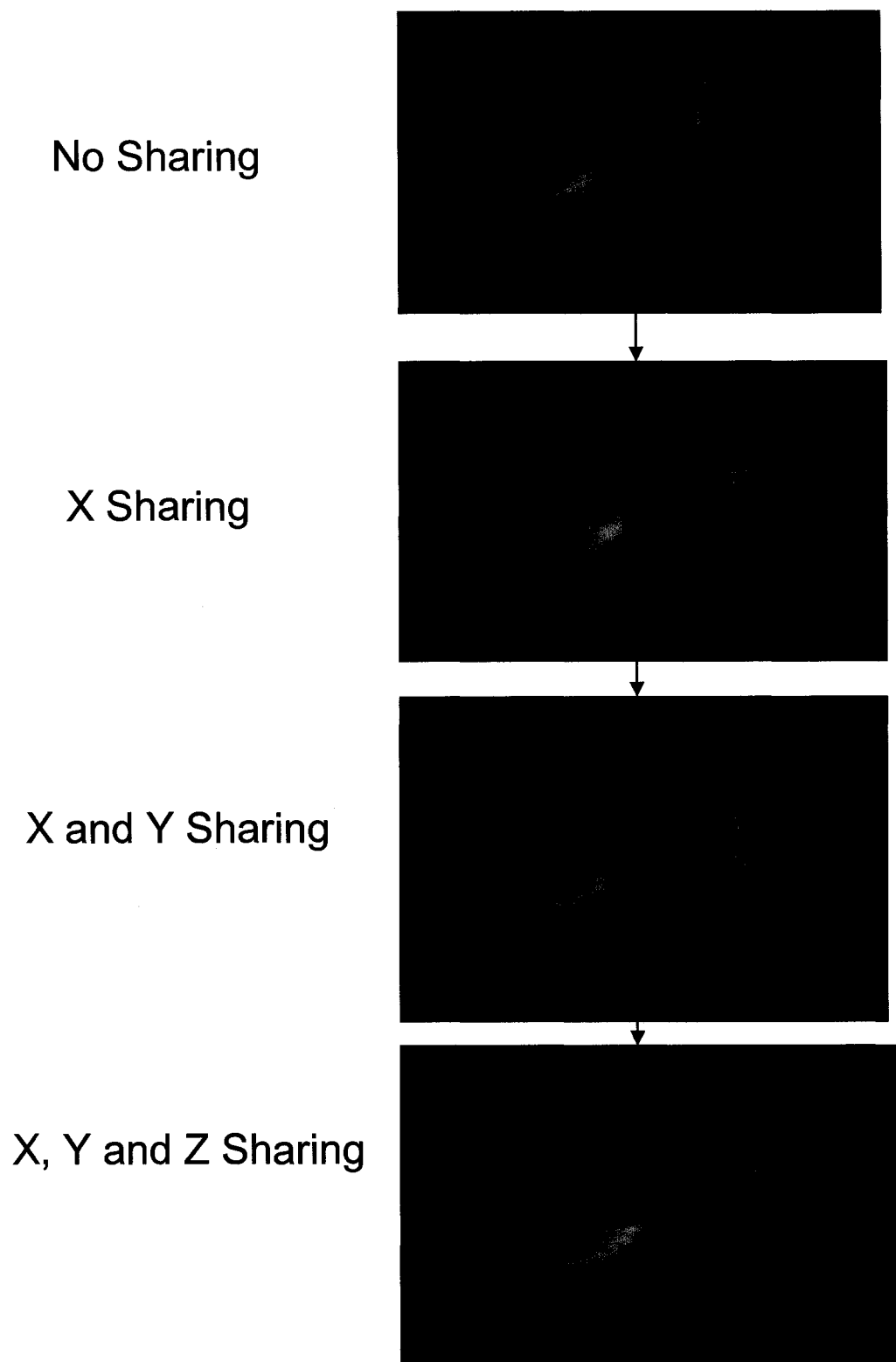


Figure 26: Resolving boundary artifacts by sharing x, y and z slices

During boundary sharing, further complexity arises due to the existence of multiple resolutions in an adaptive-resolution data set. Since each block may have a different resolution, the shared slice needs to adapt its resolution to the block it is a part of. We need to expand the resolution of a slice if it is going to be shared by a higher resolution block or contract the resolution if needs to be shared with a lower resolution block since the resolution within a block must be consistent for rendering. Therefore the same shared slice might exist as a lower resolution for one block and as a higher resolution for an adjoining high resolution block.

Once boundary sharing has been enabled, the partial images can be easily composited together. Figure 27 clarifies how the partial images contribute to the final composited image. The images in the first column are the partial images generated by rendering each block of the adaptive-resolution data. The second column shows the images in column one composited in pairs. The images in the second column are composited again in pairs to generate the images in the next column and so on. The final column shows the composited image for all 8 blocks.



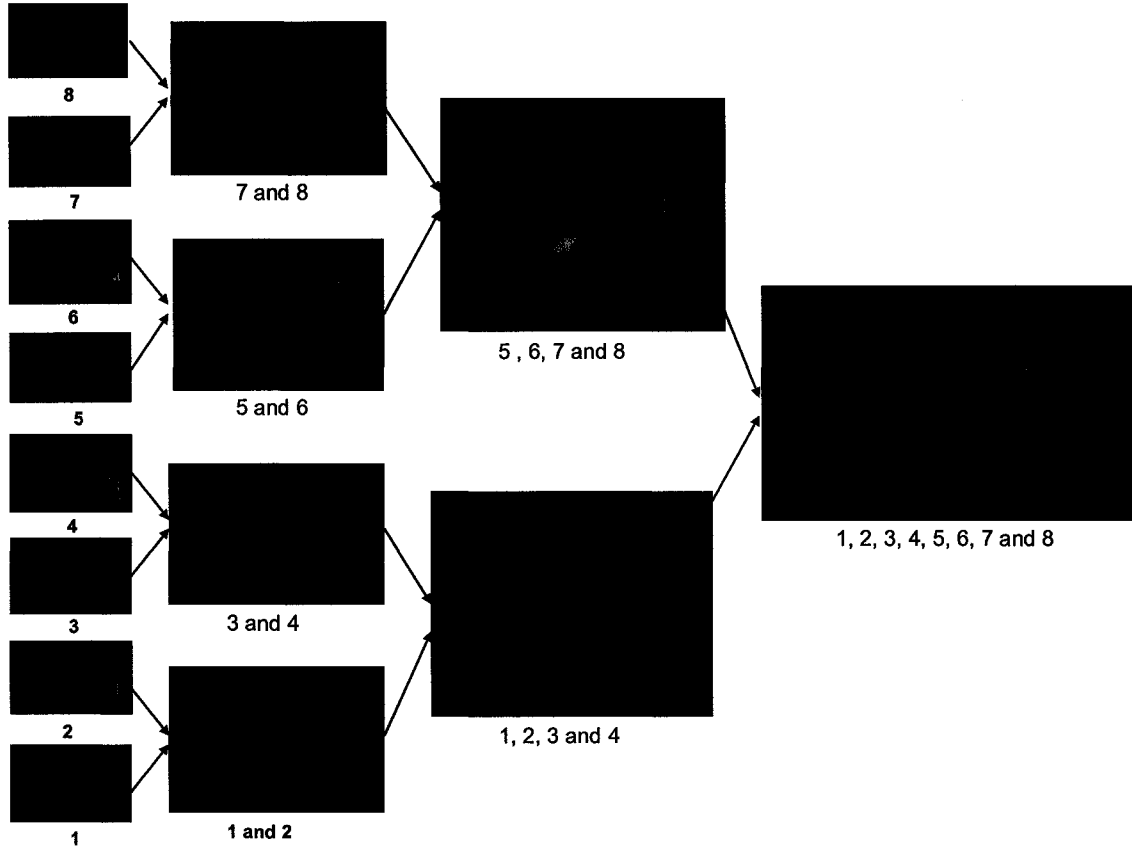


Figure 27: Compositing ordered images

Next we compare the quality of adaptive-resolution images to that of a uniform resolution image. Figure 28 shows three images generated using a dataset divided into eight blocks. The first image in this figure is a uniform resolution image since all blocks in this case had high resolution. The next two are adaptive-resolution images. The second image was generated from four blocks of medium resolution and four blocks of high resolution while the third image was generated from 4 blocks of low resolution and 4 blocks of high resolution. We can notice a slight blur at the tail (right end) of the adaptive-resolution images. These blurry portions are generated from the medium or low resolution data blocks while the rest of the image uses high resolution data. We observe little difference in quality, especially between the first and the second images, although

the rendering time of the adaptive-resolution images was much less than that of the uniform resolution image. Figure 29 shows that it took 78.9 seconds to render the uniform resolution image while it took 51.5 seconds to render the medium/high resolution image and 46.2 seconds to render the low/high resolution image. Thus if good resolution choices are made, going for adaptive visualization may increase efficiency in terms of time with very little compromise on quality.

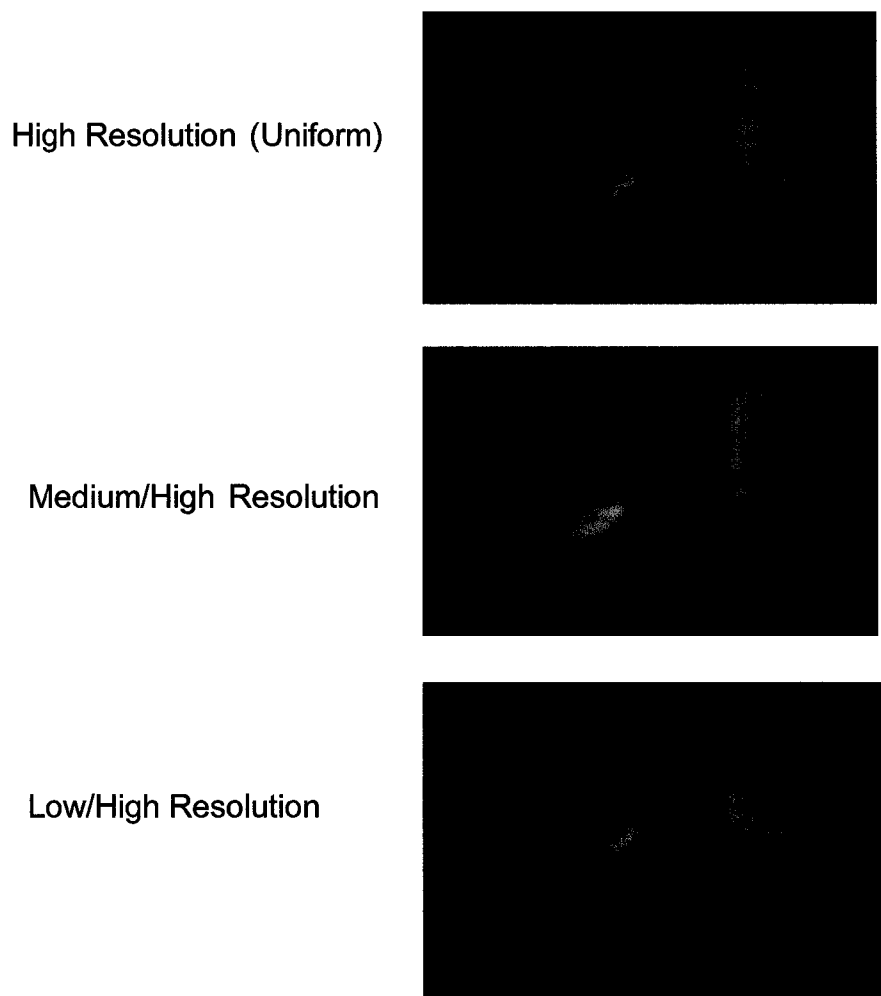


Figure 28: Adaptive-resolution image compared to uniform resolution image

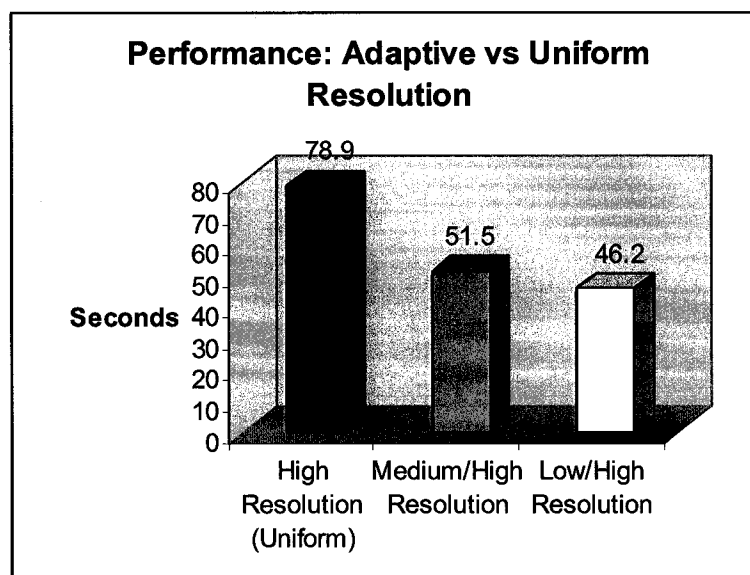


Figure 29: Adaptive-resolution performance verses uniform resolution performance

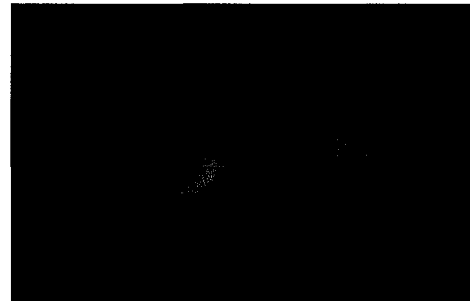
Next, we compare the quality of adaptive-resolution images when good and bad choices for block resolutions are made. Figure 30 shows three images where the first image is a uniform resolution image, and the second and third images are adaptive-resolution images. We can observe by comparing the second and third images that the second image has a much better quality (looks closer to the uniform resolution image) than the third image. Even though both of these images take almost the same amount of time to render (figure 31) since each has four blocks of high resolution and four blocks of medium resolution, the choice of blocks gives one image a much better quality than the other. The second image was generated by rendering blocks 2,4,6,8 at a medium resolution and rest at high resolution and since the blocks 2,4,6,8 have mostly empty space and very little data of importance, the quality of the image is not reduced greatly.

The third image was generated by rendering blocks 1,3,5,7 (important portions) at medium resolution and less important ones at high resolution hence the quality is greatly reduced. This shows that the choice of resolutions for different data blocks greatly impacts the success of adaptive-resolution visualization.

High Resolution (Uniform)  
All high resolution blocks



High/medium Resolution  
Good choice of high-resolution blocks



High/medium Resolution  
Bad choice of high-resolution blocks

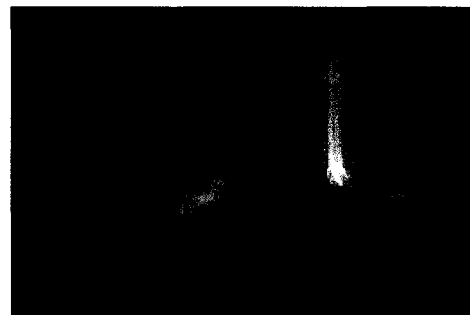


Figure 30: Adaptive-resolution images: Different choices of high-resolution blocks

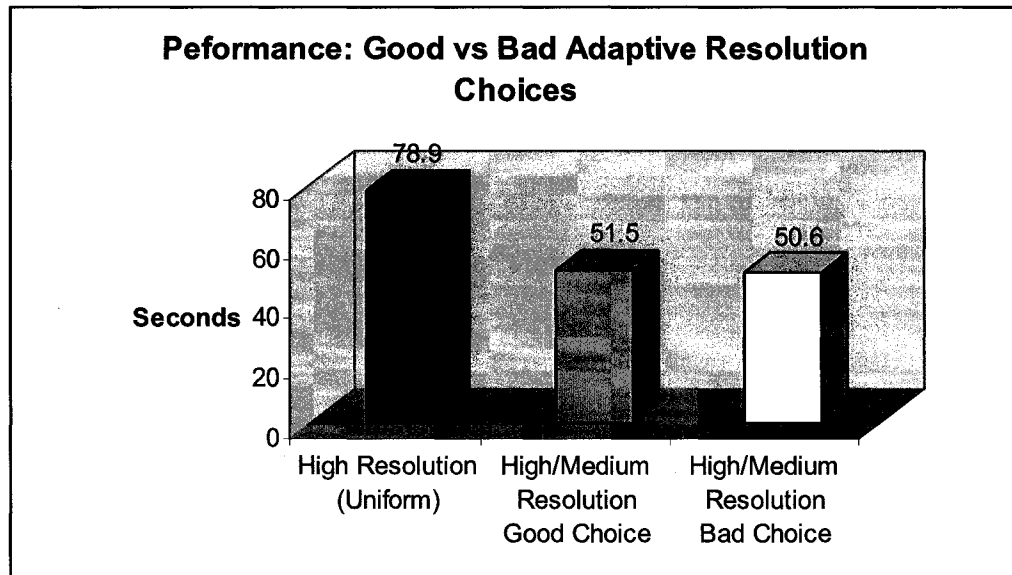


Figure 31: Adaptive-resolution: Different choices of high-resolution blocks

## **CHAPTER 5**

### **PERFORMANCE ANALYSIS**

The goal of this research is to enable interactive visualization of huge datasets. To enable interactive visualization we devised temporal/spatial parallel visualization techniques, multiresolution techniques and adaptive-resolution techniques. The performance of multiresolution and adaptive-resolution techniques could easily be tested on a single processor machine and we saw improvement in terms of rendering time when these techniques were used. The performance of spatial parallel visualization could be tested on a multiprocessor machine and that showed some performance improvement too. But temporal parallel visualization did not show much improvement on a multiprocessor since it is very I/O intensive and I/O created a bottleneck. Therefore, we decided to enable temporal parallel visualization to run on a cluster of machines that would allow parallel I/O.

The cluster that we used is called Zaphod. Zaphod is a Beowulf cluster located in the Research Computing Center (RCC) of the Institute for the Study of Earth, Oceans, and Space (EOS) at UNH. The specifications of this cluster are listed in figure 32. The figure shows that this cluster has 160 compute nodes where each node (machine) has a dual processor. The system uses RAID storage to enable parallel I/O. For networking it can either use Gigabit Ethernet or Myrinet. For our experiments we used a maximum of

40 compute nodes and for inter-process communication between these nodes we used Myrinet.

To enable our code to work on this cluster, we had to replace our threads with processes and use inter-process communication in place of memory sharing. For setting up processes and inter-process communication, MPI was used. Wikipedia ([www.Wikipedia.org](http://www.Wikipedia.org)) defines MPI as follows:

The Message Passing Interface (MPI) is a computer communications protocol. It is a de facto standard for communication among the nodes running a parallel program on a distributed memory system. MPI implementations consist of a library of routines that can be called from Fortran, C, C++ and Ada programs. The advantage of MPI over older message passing libraries is that it is both portable (because MPI has been implemented for almost every distributed memory architecture) and fast (because each implementation is optimized for the hardware on which it runs).

**160 compute nodes:**

*processors:* dual **Opteron** 246

*memory:* 4 GB

*storage:* 120 GB, single disk

**2 head nodes for interactive access:**

*processors:* dual **Opteron** 250

*memory:* 4 GB

*storage:* 2.7 TB RAID5 storage

**6 post-processing and storage nodes:**

*processors:* dual or quad **Opteron**

*memory:* up to 16 GB

*storage:* 12 TB SCSI/SATA RAID storage

**Networking:**

Gigabit Ethernet (all nodes)

**Myrinet** (storage nodes & 122 compute nodes)

Figure 32: Specifications of Zaphod (cluster)

The skeleton code `main.cpp` in figure 33 shows how processes were created for parallel Temporal Visualization using MPI on the cluster. In this code, *MPI\_Comm\_size* returns the number of processors in the variable *numprocs* and *MPI\_Comm\_rank* returns the id of the current process in the variable *myid*. Then process 0 works as the Controller Node for parallel temporal visualization that picks up the images once they are generated and all other processes work as Renderer Nodes that generate those images.

```
#include <mpi.h>

int main(int argc, char *argv[] )
{
    int numprocs;
    int myid;
    MPI_Status stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if(myid == 0)
    {
        parallelTemporalVisualizationWithinControllerNode()
    }
    else
    {
        parallelTemporalVisualizationWithinAllRendererNodes();
    }

    MPI_Finalize();
    return 0;
}
```

Figure 33: `main.cpp`: Creating processes using MPI for parallel temporal visualization



Next, we used the batch script displayed in figure 34 to run this code on Zaphod. The line “#PBS -l nodes=4:ppn=2” means that we want 4 nodes with 2 processors per node. This means that our program main.cpp will create 8 processes.

```
#!/bin/bash

# Define qsub command options here. Each line
# processed by PBS/TORQUE begins with "#PBS"
#
#PBS -l nodes=4:ppn=2:myri
#PBS -l walltime=00:01:00

cd $PBS_O_WORKDIR
mpiexec ./main
```

Figure 34: main.cpp: Creating processes using MPI for parallel temporal visualization

After submitting the job, the images generated are stored in ppm image files. The user can either open these files to view the images or use a simple OpenGL program to read and display these images.

After enabling and testing parallel temporal visualization on Zaphod, the program was run with the number of nodes ranging from 1 to 40 for performance evaluation purposes. For these experiments we used a total of 40 frames; therefore a maximum of 40 nodes were used. Even though each node in the case of Zaphod is a multiprocessor, we expect only one processor of the node to be fully utilized since our code is not multithreaded. Figure 35 shows the results using the original (high resolution) dataset. When a single node was used (serial visualization) it took around 986 seconds to generate 40 frames. As the number of nodes was increased (parallel visualization) the rendering

speed also increased. In the case when 40 nodes were being used, each node rendered a single frame only and the total rendering time dropped down to 28 seconds.

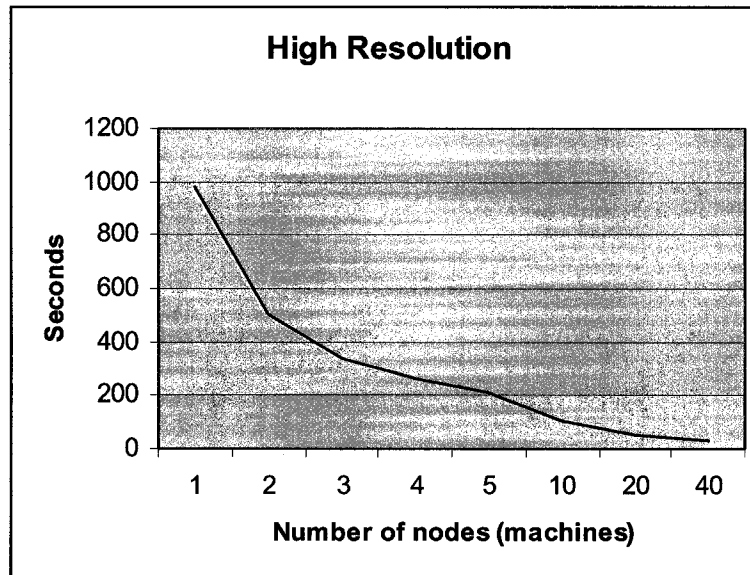


Figure 35: Parallel temporal visualization: 40 high resolution frames

Even though we saw a significant improvement in terms of time, the results are still not very good in terms of interactive rendering. This is the best that can be done using the pure temporal approach, to further speed up the visualization we can enable parallel spatial visualization within each node and since each node in this system is a multiprocessor this should further speed up the visualization of each frame. We can also combine adaptive-resolution or multiresolution techniques with parallel visualization to further increase the rendering speed. Figure 36 and 37 show the results when parallel temporal approach is mixed with the multiresolution approach. Figure 36 shows the results for the medium resolution data. In this case, when 40 nodes are used, the rendering time (for 40 frames) drops down to 5 seconds.

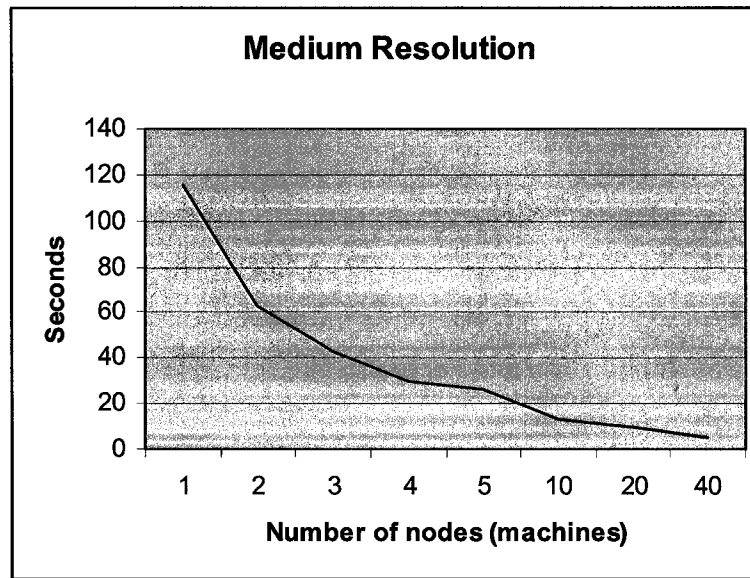


Figure 36: Parallel temporal visualization: 40 medium resolution frames

Figure 37 shows the results when low resolution is used and here the rendering time (for 40 frames) further dropped down to 0.9 seconds when 40 nodes were used.

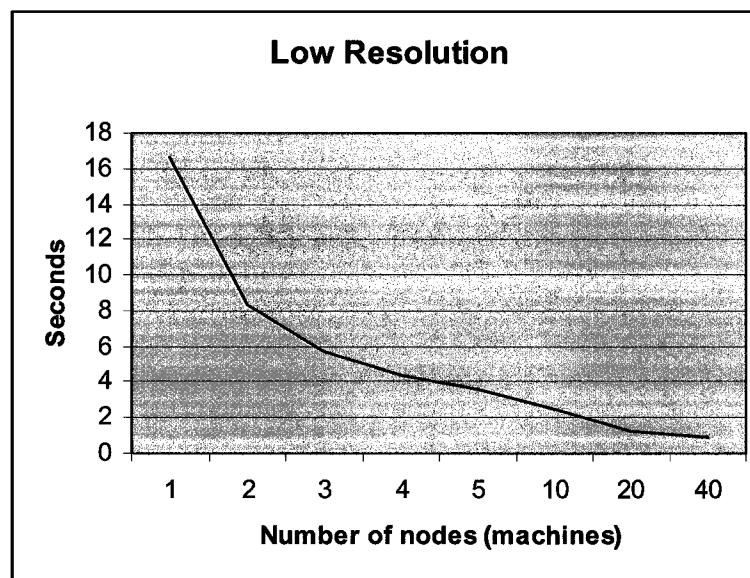


Figure 37: Parallel temporal visualization: 40 low resolution frames

Table 8 summarizes these results. We started off with the Single Node/High resolution approach which took 986 seconds for 40 frames and after using a mix of the parallel temporal approach and multiresolution approach, we improved the rendering time to 0.9 seconds for 40 frames which allows for interactive visualization.

	Single Node (Serial Approach) (Time in seconds to render 40 frames)	Multiple Nodes(Multiple Approach) (Time in seconds to render 40 frames)
High Resolution	986	28
Med Resolution	116	5
Low Resolution	16.7	0.9

Table 8: Summary of performance analysis

## CHAPTER 6

### CONCLUSIONS

We have implemented (temporal and spatial) parallel visualization techniques for time-varying (temporal and spatial) multiresolution data that can work on shared memory multiprocessors.

We have also devised parallel rendering techniques for adaptive-resolution data. For this we used the cell-count algorithm. This enables spatial adaptive-resolution data to be rendered effectively using parallel visualization. Also, for the cell-count algorithm block-wise spatial data division was implemented instead of slicing.

Next we enabled the temporal parallel visualization module to run on Zaphod (Cluster). The data was distributed temporally over the different nodes of the cluster. The goal of this task was to allow interactive visualization of large data sets by distributing work among the nodes. The performance analysis showed that mixing the temporal approach with the multiresolution approach did indeed result in interactive visualization.

We discovered that moving to the parallel and adaptive/multiresolution visualization systems is a challenging task. There are several issues that must be taken into consideration. First, we need to carefully analyze performance to determine the degree of parallelism needed for a particular task. Next, it is difficult to conclude what mix of spatial/temporal data division would present the most efficient solution.

Furthermore, it needs to be determined whether a dataset is large enough to justify the use of parallel or low-resolution rendering. Also, in case of spatial decomposition, the time saved by parallel rendering must exceed the time lost due to compositing. If these issues are resolved then a considerable amount of efficiency gain can be achieved through the use of parallel visualization and adaptive/multiresolution systems.

## **LIST OF REFERENCES**

[1] Bethel E. Wes, White Paper: Sort-First Distributed Memory Parallel Visualization and Rendering with OpenRM Scene Graph and Chromium, R3vis Corporation, page 10-14 July, 2003.

[2] Bhaniramka P., Robert P. C.D., and Eilemann S. OpenGL Multipipe SDK: A Toolkit for Scalable Parallel Rendering, *IEEE Visualization*, page 119-125 October, 2005.

[3] Brodlie K.,Duce D.,Gallop J. Sagar M., Walton J. and Wood J. Visualization in Grid Computing Environments, *IEEE Visualization*, page 155-162 October, 2004.

[4] Cavin X., Mion C, and Filbois A. COTS Cluster-based Sort-last Rendering: Performance Evaluation and Pipelined Implementation, *IEEE Visualization*, page 111-118 October, 2005.

[5] Levoy, M. Display of Surfaces from Volume Data., *IEEE Computer Graphics and Applications* (May 1988), 29-37.

[6] Ma K., Painter J.S., Hansen C.D. and Krogh M.F. A Data Distributed, Parallel Algorithm for Ray-Traced Volume Rendering, *Proceedings of the 1993 Symposium on Parallel Rendering*, page 15-22, 1993.

[7] Merkey, P., *Beowulf Project Overview*, <http://www.beowulf.org/overview/index.html>, Beowulf.org, 2004-2005.

[8] Molnar, S. Eyles J., and Poulton. J. PixelFlow: High-Speed Rendering Using Image Compositon, *Proceedings of SIGGRAPH 97*, pages 231-240, August 1992.

[9] Shalf, J. and Bethel, E.W. 2003. The Grid and Future Visualization System Architectures, *IEEE Computer Graphics and Applications* 23, 2, 6-9.

[10] University of Tennessee, *MPI-2: Extensions to the Message-Passing Interface*, <http://www-unix.mcs.anl.gov/mpi/standard.html>, University of Tennessee, Knoxville, Tennessee, 1997.



[11] Wang C., Gao., Li L., Shen H.W A Multiresolution Volume Rendering Framework for Large-Scale Time-Varying Data Visualizations, *The Eurographics Association*, 2005

[12] Wikipedia, *Grid Computing*, [en.wikipedia.org/wiki/Grid\\_computing](http://en.wikipedia.org/wiki/Grid_computing), Wikimedia Foundation, Inc., November 2006.

[13] Williams J.L. and Hiromoto R.E. Sort-Middle Multi-Projector Immediate-Mode Rendering in Chromium, *IEEE Visualization*, page 103-110 October, 2005.