Visualization and HPC

Jianqiang Liang

September 20, 2005

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2005
Authorship declaration

I, Jianqiang Liang, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;

2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

3. I have acknowledged all main sources of help;

4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

5. I have read and understand the penalties associated with plagiarism.

Signed:

Date: 20th, September, 2005

Matriculation no: 0455140
Abstract

Researchers and institutions always need simulation to help them gain a better understanding of physical phenomena. However visualization of extremely large and time-varying data sets, such as weather forecasting and nuclear explosions, is always a challenge. Applying parallelisms to visualization is an interesting approach to this problem. Here, we will use three kinds of parallelism to visualize an extremely large set of time-varying data. The visualization system we are using in this project is Visualization ToolKit (VTK), which is portable, general-purpose and freely available visualization system with open-source codes. At the end, we test the performance of each parallelism which is applied to real problems in the VTK system on cluster.

In order to test the performance of each parallelism, we create several visualization programs in C++ and apply them to a real problem which simulates a molecular dynamics (MD) system. The visualization programs are running in VTK system on a cluster, Sun Fire E15k server. After analysis the results we get, we find that these three parallelisms do speed up visualization processes. Moreover, we find that the pipeline parallelism gets very good parallelisation, but the other two parallel techniques do not provide very much speed up when parallelised.
Acknowledgements

First and foremost, I would like to thank my supervisors Adrian Jackson and Chris Johnson for their guidance and supports throughout this project.

I also like to thank the EPCC fellow students and my friends for their assistance.

Lastly, I would like to thank my parents and T. Tong, who kept me full of passion on the project over the last three months.

Jianqiang Liang

August 2005
# Contents

List of Figures ................................................................. vii  
List of Tables ................................................................. ix  
1. Introduction ........................................................................................................ 1  
2. Background Theory .......................................................................................... 7  
   2.1 Scientific Model .......................................................................................... 7  
   2.2 Visualization Theory ................................................................................. 8  
   2.3 Visualization ToolKit System ..................................................................... 11  
   2.4 Experiments environment ......................................................................... 13  
3. Methodology ...................................................................................................... 14  
4. Pipeline parallelism ........................................................................................... 19  
   4.1 Performance Analysis ............................................................................... 25  
   4.2 Conclusion .................................................................................................. 26  
   4.3 Compare with an official pipeline parallelism ......................................... 27  
5. Data parallelism ................................................................................................. 32  
   5.1 Performance Analysis ............................................................................... 37  
   5.2 Conclusion .................................................................................................. 43
6. Task parallelism ........................................................................................................... 45
   6.1 Performance Analysis ......................................................................................... 47
   6.2 Conclusion ........................................................................................................ 49

7. Conclusions and future work ............................................................................... 50
   7.1 Conclusions ................................................................................................... 50
   7.2 Future Work .................................................................................................. 51

Bibliography .................................................................................................................. 53
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Colliding disk galaxies simulated by GADGET-2</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Pipeline Parallelism</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Data Parallelism</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Task Parallelism</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Basic visualization pipeline</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Visualization methods combined: isosurface and vectors</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>One of the results attained form pipeline parallelism</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Pipeline Parallelism Approach</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Running time versus of pipeline and its modules Number of processors</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Speed Ups versus Number of processors</td>
<td>23</td>
</tr>
<tr>
<td>4.4</td>
<td>Efficiency versus Number of processors</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>The pipeline parallelism from VTK official document</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Data Parallelism Approach</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Result of the data parallel program</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Running time versus numbers of data objects via multiple processors</td>
<td>38</td>
</tr>
</tbody>
</table>
5.4 Running time versus numbers of processors processing multiple data objects

5.5 Speed ups versus numbers of processors processing multiple data objects

5.6 Efficiency versus numbers of processors processing multiple data objects

6.1 Task Parallelism Approach

6.2 Result of the task parallel program

6.3 Efficiency versus different complexity levels when running on nine processors
List of Tables

4.1 Table 4.1: Running time of program and its modules when processing 2000 data files on various number of processors ..................................................22

4.2 The percentages of running time of various modules in whole pipeline when running on different number of processors. ........................................24

4.3 The running time of one processor when processing multiple numbers of data files ........................................................................................................25

4.4 The running time of diverse numbers of processors when processing same number of data files ..........................................................................................................26

4.5 The pipeline’s details of official pipeline parallelism.................................28

4.6 The running time of official pipeline parallelism when processing different number of data files on two processors ..........................................................29

4.7 The running time of our pipeline parallelism when processing different number of data files on two processors ..........................................................29

4.8 The pipeline’s details of our pipeline parallelism .........................................30
4.9 The pipeline’s details of official pipeline parallelism considering communications


4.10 spending time of multiple modules of an independent pipeline

5.1 Running time of program when processing different numbers of data objects on various number of processors

5.2 The percentages of running time of various modules in whole pipeline when processing 10 thousand data objects on different number of processors

5.3 The percentages of running time of various modules in whole pipeline when processing 200 thousand data objects on different number of processors.

5.4 The percentages of running time of various modules in whole pipeline when processing 10 thousand data objects on different number of processor.

6.1 Running time of program when processing objects with different complexity levels on one and nine processors
Chapter 1

Introduction

"Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionizing the way scientists do science."


Scientific visualization involves multiple techniques integrated to simulate real physical systems which are built by scientists' formulas. In order to gain a better understanding of the physical world, the visualization systems needs to be set up by the application of display technology, computers, and interaction. Visualization can provide the most direct way for scientists to show the virtual world created from their algorithms. It relies on knowledge about human perception in order to make the best choices for displaying and collecting information. As simulation models develop, more features and details are added to the models to increasing the understanding of the problem. In more detail, the term scientific visualization means the integration of high-speed computation and colorful 3-D graphics. Three Dimensional models can help researchers have a better understanding of their mathematical models. Generally, scientific visualization is the art of making the unseen visible. Scientific visualisation has strengthened people's ability to perform scientific research, and accelerated the transformation process from technology to productivity. It profoundly changes human's methods of understanding and constructing the world. In many domains of science and engineering, visualisation can be used for
significant research or highly complicated engineering design. It provides an important instrument and theoretical foundation for science research and technical innovation.

![Image: Colliding disk galaxies simulated by GADGET-2 [2]](image.png)

Figure 1.1: Colliding disk galaxies simulated by GADGET-2 [2]

However, as scientists' research goes deeper it is necessary to build more and more complicated models, usually with millions of analyzed objects, pretty huge space or complex interactions included. For example, cosmologists usually build their mathematical models based on algorithms to simulate the complicated physical phenomenon of our universe, such as the merging of galaxies. As we know, a galaxy contains trillions of stars. It means that simulation of the interaction need to handle movements of trillion stars for each step (See as Figure 1.1).

Furthermore, what the scientists are concerned with is the process of the interaction, not just 3-D images. So a movie should be made of the interaction to show the details of the
process. This involves the creation of hundreds of images. Therefore, their requirements usually exceed the capabilities of desktop computers even though processor's speed is always increasing. So, nowadays, simple scientific visualization will normally contain hundreds of Gigabytes of data files. It will easily overload the memory, or take a long time to proceed on a normal PC. Applying a parallel algorithm on clusters or supercomputers is a good approach for scientists to achieve their results.

In order to obtain faster results, parallel computing is the simultaneous execution of the same task on multiple processors.[3] It is now being viewed as a very promising approach for radically decreasing execution time and speeding up demanding problems naturally. It is now being successfully applied in various academic and business areas for solving different computing problems. Parallel computing is implemented on "big machines". These machines are built in several kinds of structures in order to deal with diverse specified problems, but they usually contain thousands of processors and appropriate amounts of memory and storage space. This infrastructure can benefit scientists when simulating their complicated models by running their simulation system in high performance computers. Therefore, building a parallel visualization system is essential.

In this project, we are going to visualize a series of time-varying data sets which are created by a serial code from a previous project. The code is run to simulate a molecular dynamics (MD) system. In this MD system, thousands of particles contained in a box, interact with each other through a force equation. Our task is to build a visualization model based on the Visualization Toolkit software system (known as VTK). VTK is open-source software, which can be freely downloaded from its website [4]. The software is a "true visualization system" which supports a wide range of visualization algorithms, including vector, texture, volumetric, implicit modeling, polygon reduction, contouring and so on. We can even use it to integrate 2D images, 3D models and data into one visualization scene to simulate a complicated model.

Since the mathematic model which we need to simulate contains thousands of data files, and each data file contains tens of thousands of particles' coordinates, the simulation needs several Gigabytes of memory space. It is hard to be accomplished the job using a normal computer and visualisation system. We are going to build and test Parallel VTK
models which will benefit our visualization performance. It is an important extension of VTK containing many classes which make it easy to do parallel visualization. This extension is built on top of the MPI (Message Passing Interface). With these libraries' support, VTK can make a visualization pipeline which can be run over multiple processes. They can operate on pieces of data sets in distributed locations. Processes can also communicate with each other. The Parallel VTK is based on the master-slave conception. It always uses the first process as the master process to control interaction. Other process must be slaved to master. They wait for messages to trigger remote renders and then integrated together to make an image.

There are three kinds of parallel visualization approach we are going to apply on our problem and then we will carry out performance analysis. They are Pipeline Parallelism, Data Parallelism and Task Parallelism.

![Pipeline Parallelism](image)

Figure 1.2: Pipeline Parallelism

Pipeline parallelism is designed for the visualization of multiple time-varying data files. Shown in figure 1.2, each process does similar work in this parallelism. They do similar reading, mapping and rendering. That's because all the data files represent data from the same mathematic model. The differences in the files just reflect the diverse situations of
the model. So each thread just does the same work and creates an image of each simulation. Finally, in order to observe the change of the object over time, we also need to make a movie from the multiple images generated by each process.

Figure 1.3: Data Parallelism

Data parallelism is used for extremely large data sets. According to the basic pipeline of the visualization system, the reading module reads the data to be visualized from the file, then transfers this to a data stream and passes it to the mapping module. However, loading a huge data file into memory always adds huge overheads. In order to deal with this kind of file effectively, we can divide the data set into several parts, shown in figure 1.3. Each part of the data will be processed as an independent data set. After mapping, the main thread will collect all the results of the others and then render the result.
Task parallelism is the most complicated parallelism of the three. In this parallelism, the task is accomplished by one main thread and several collaborated threads. Each thread does its own task: read different data files, or create some objects directly, then does the corresponding mappings. However, after finishing mapping, the main thread has to wait for the results of the other collaborated threads. The collaborated threads need to transfer their outcomes, the geometric representations to the main thread. Then the main thread can run its rendering module to show the objects in an image. This parallelism is used for complex visualizations which contains several unique branches.

Figure 1.4: Task Parallelism
Chapter 2

Background Theory

2.1 Scientific Model

In scientific domains, generally, there are two ways for resolving problems: experimental demonstration and logical simulation. Usually, these two methods need to be integrated together. For example, we have theoretical physics and experimental physics. The scientists of theoretical physics get results from mathematical algorithms. The result then need to be compared with those experimental physics using multiple experiments. Experiment is the only way to examine whether the theory results are tenable or not. In the past, experiments are the most direct way to discover and explore the physical environment. However, as science progresses, direct observation of physical phenomena becomes more difficult as the studied objects become more abstract. As we know, no one has ever directly seen an "elementary" particle even by using advanced electron microscopes. Similarly, none of today's cosmologists are capable of manipulating our Milky Way to explore its typical behavior in various situations. Almost all the theories and results in these areas are obtained from the mathematical formulas. As results, experiments, which involve the notion of contract or manipulation of objects and phenomena, also becomes more difficult to perform and observe.

Increasingly, scientific computing is the most promising way for these abstract problems. Scientific computing is an emerging interdisciplinary study which follows the appearance of computer simulation and rapidly develops to wide spread applications. It is the essen-
tial link between mathematics and computer science realizing applications in the high tech area. Now, the scientific theory, experiment, and computation have become the foundations of scientific activities. More and more significant science and technology problems are hard to be solved by ratiocination and experiment, but maybe feasible by computational method.

Based on the scientific computation technique, it is feasible to build-up numerical simulation for physical systems. The simulation system is a virtual system which is expressed through some appropriate mathematic models based on algorithms. Through this system, we can simulate the behavior of corresponding physical systems as we want to study. That's what we call "visualization techniques".

2.2 Visualization Theory

Visualization is the main technology of computer science. It is a process of input and output. The task is showing pictures on the screen by processing data which is read from other applications. Normally, it contains three key stages, reading, mapping and rendering. These three stages build up the main pipeline of visualization.

![Basic Visualization Pipeline](figure)

Figure 2.1: Basic visualization pipeline
Figure 2.1 shows the sketch map of the visualization pipeline. In this pipeline, there should be something that needs to be simulated. Therefore, the object, which is usually stored as a file, should be available for the visualization system. At the start of the visualization system, the reading module does the first job. Generally it reads a data file from disk space. Since the data file is formatted by the application, sometimes the reading module needs to change it to another format which can be recognized by the visualization system. Subsequently, the reading module transfers the data from the file format into the stream (or derived data) which can be stored in memory space in order to transfer the data to the mapping module easily.

However, in the transfer procedure, some modules that may be needed to do some extraction or conversion jobs, such as shrinking and making contour surface. These kinds of modules are called filtering modules. Filtering modules are designed with various mathematic algorithms which are used for processing input data. So, different filters can output diverse results with the same input dataset. Usually there are plenty of filters contained in a visualization system in order to meet various requirements.

![Visualization methods combined: isosurface and vectors.][5]
Figure 2.2 is a typical sample of scientific visualization. It combines two of the most common visualization methods, vectors and isosurface, to simulate both volume and flowing directions of an object.

In the visualization process, it is necessary to integrate several filters to get a specified result. Using the appropriate filtering modules is an important visualization technique. Mapping is another key module. It takes charge of creating a geometrical representation from input derived data. After mapping, an abstracted visualization object is constructed. The object consists of an attribute field, in which the (simulation) data is mapped. Actually, not only geometry, but also time, color, transparency, luminosity, reflectance, and surface texture might be included in these attribute fields. [6] Finally, the system then starts the rendering work. It operates on the abstracted visualization object to make an image on the user's computer screen. This operation it makes the geometric drawing more concrete and then displays as an image.

That's the basic pipeline of a visualization system. To deal with real problems, there are lots of simulation objects which need to be processed. Different objects may be stored in different data format or need to be simulated using different algorithms. So we usually need to set up and integrate multiple pipelines to simulate multiple objects independently in one visualization task. For example, we can add more reading modules to read more data files, or create polygonal objects directly. Then correspondent filtering and mapping modules are needed to deal with different objects.

Adding more details to the simulation process will help us understand the real world better, but this requires more work. We need some techniques to finish the complicated job more effectively. Applying parallel models in a visualization system may be an interesting solution. Based on the basic pipeline of visualization model, there are three fundamental types of parallelism we can use. They are task parallelism, pipeline parallelism and data parallelism.

These three parallelisms are designed for dealing with different simulation problems, but sometimes, they're also needed to be incorporated together in a simulation task. For ex-
ample, a global weather simulation usually contains multiple time-varying datasets which present various weather situations in different specified times. In order to speed up this problem, we can use the pipeline parallelism. On the other hand, since each dataset presents diverse weather situation of global area and the more details used give better accuracy but larger datasets. Data parallelism then needs to be applied to get more efficiency. Therefore, different real problems may often benefit from multiple pipelines or different parallelisms in order to speed them up.

2.3 Visualization ToolKit System

The Visualization ToolKit (VTK) is an open source, freely shared software system for image processing, 3D computer graphics, and visualization. It supports a wide range of arithmetic algorithms including scalar, vector, tensor, texture, and volumetric methods; and advanced modeling techniques like implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation.[7] Furthermore, VTK is software with good portability features. It can be built on several operating systems: UNIX, Cygwin, Windows and Mac OS, etc. Since most of the codes and the patents are freely available, this toolkit has grown rapidly with world-wide user base. It has been utilized both in the commercial and research communities.

VTK contains hundreds of C++ class libraries with various algorithms for different kinds of simulations, and several interfaces to several kinds of computer languages, such as C++, Java, TCL and Python. Since VTK can be installed on multiple operating systems, the multiple language extensions allow VTK codes to run on different systems with VTK already installed. Consequently, this feature means our parallel visualisation model contains the same wide-spread portability provided a suitable parallel environment is i-
stalled. We can utilize our model on different platforms with diverse operating systems or hardware support, including distributed systems and shared memory systems. Portability is important for the development of visualisation work. Nowadays, with the development of the Grid, more and more scientists store their data files and other resources, such as computing resources, in a distributed fashion. As a result, they also need to simulate distributed data. It requires flexible and robust visualisation software systems to achieve this task. Portability is one of the critical features.

Another crucial feature of VTK is its parallel extension. VTK supports parallel algorithms with multiple threads on shared memory, multiple processes and distributed memory systems. In VTK, communication between processes uses a class that is implemented on top of MPI (Message Passing Interface) libraries. However, MPI is not the only one communication techniques can be used in VTK. It also has communication classes based on shared memory and sockets, either of which can be transparently substituted for MPI based class. [8]

Visualization programs are usually required on back-end of cluster. There are two parts of HPC services of one cluster, front-end and back-end. Whilst front-end is used for compiling, and testing jobs, back-end is used for production runs of medium to large size parallel jobs [9]. It provides a reliable environment for program running. However, it can only be accessed by batch jobs scripts from front-end. Off-line running is the most common way for batch jobs’ running. Therefore, real-time visualization is not appropriate for the back-end jobs’ running. Therefore off-screen visualization is necessary to be applied instead the real time one. Off-screen rendering means render an object without showing any windows and displaying. The rendering results are shown in virtual widows stored in memory stack. User can get this result by writing them as image files. The Mesa library is the most promising choice for off-screen rendering. The Mesa library is a 3-D graphics library with an API which is similar to that of OpenGL. [10] It makes running visualization programs without OpenGL hardware support possible. However, in some application, both Mesa and native OpenGL may be required at the same time. A function named Mangled-Mesa is designed to target this requirement. In VTK, Mesa library support is
available by switching on in the VTK compiling process. After replacing the original mapper, render and actor modules by the VTK-Mesa modules in the program, the off-screen visualization on backend can be run.

2.4 Experiments environment

The machine we used for the development and experiments in the project is Sun Fire E15k. Its HPC services are provided by 52 UltraSPARC III Cu 1.2 GHz processors. It is partitioned by two parts: 4-processor front-end (known as Lomond) and 48-processor back-end. Sun Fire E15k is running on UNIX operating system (Solaris 2.9). Multiple compilers, Fortran(f90), C(cc), C++(CC) and Java are all supported for both sequential and parallel application.

SUN MPI is the default MPI implementation of Sun Fire E15k. It is an integral part of the Forte programming environment. [9] It also beneficial for VTK to build up MPI extension modules.

All the visualization programs and experiments mentioned below are all processed in this computing environment. All results are average values of 30 timing iterations. Moreover, since each test result is not less than $10^{-1}$ second, so the precision and accuracy is high enough for all the records are double-float time values. The results are all reliable.
Chapter 3

Methodology

Whilst the overall aim of the project was to create an efficient parallel visualization model to simulate a real scientific problem, it was necessary to produce several VTK programs to achieve this model.

Since there are three kinds of parallelism that will be used, and different types of parallelism are suitable for different problems, we need to choose the best one for the specific problem. We can evaluate these methods of parallelism by creating a VTK pipeline. However, firstly, we should find out some details of the real problem which we need to simulate.

Firstly, we need to describe about the details of the mathematical model we need to visualize. The data files are created by a serial code from a previous project, which is written in C. It is a molecular dynamics (MD) simulation. Thousands of particles contained in a box interact with each other through force. "The force on each particle is accumulated throughout the force calculations, with the total force exerted on a particle being stored and used to update the particle's position"[11]. Since all particles are contained in a box, when they reach the boundaries will be bounced and reflected inside the box. The positions of particles in different loops are stored in different created files.
Using these data files, we can simulate the movements of the particles interacted in a cube. Figure 3.1 is the result we can attain.

Above all, the visualization platform we will work on is the Visualization ToolKit system, so what we need to do first is set up the VTK system on the Lomond. As stated before, VTK is freely shared software which is available for many operating systems, such as Mac, UNIX, Cygwin, Linux and Microsoft Windows. Several versions can be downloaded for every system platform. For the reason of stability and functionality, we decided to install the latest official release 4.2.

Apart from the Windows version, what the VTK bundle downloaded from its website is just a package of original codes, written and developed by thousands of users. So we need to compile on the operating system before use. VTK is built on the C++ computer language. The installation is the process of building thousands of VTK libraries and links, which can be applied for different languages. Each of the libraries represents diverse
module with specified algorithm for different usages.

The first job we need to do is set correct arguments for the VTK installation. Since the server we are going to use is running on the UNIX operating system, we need to set directories of multiple tools whose libraries and executable files are required, such as the paths of C++ compiler, the locations of OpenGL libraries, the directories of X-server libraries and so on. However, there are some advanced functions of VTK are disabled by default. They need to be activated manually by users when their usage is required. According to the task of the project, there are two advanced options needing to be used. One is the MPI extension; another is Mangled-Mesa libraries.

The MPI extension is the core of parallel VTK. It contains several communication modules built over MPI, the Message Passing Interface standard. The MPI libraries enabled VTK to run over multiple processors or multiple threads over one processor. The vtkMultiProcessController is the main module of this extension. vtkMultiProcessController is the module to control multiple processes in distributed systems or multiple threads in a shared memory environment. This module has multiple methods for executing single or multiple methods on multiple processors.[8] Similar with other MPI programs, the communication is done using the communicator, named vtkMPIcommunicator. Then sending and receiving data among multiple processors is possible by using the same communicator. Beside the MPI communicator, there are some methods that can be used for sharing data, such as vtkPDataSetWriter. It is a module used to divide a dataset stored in a file into pieces, which can be read by the other processors.

Another important extension is the Mangled-Mesa feature by using one free sharing library, Mesa. According to Mesa's introduction, Mesa is a 3-D graphics library which contains an API similar to that of OpenGL libraries. This interface makes Mesa to utilize the OpenGL command syntax or state machine.[10] Normally, the operating systems need OpenGL hardware supports for display images on screen. This makes the performance of
visualization programs mostly dependent on various hardware facilities. But the Mesa library can completely substitute OpenGL within pure software supports. This allows us to visualize on systems with no OpenGL support. One of the important features of the Mesa library is the facility named off-screen rendering. From the release 1.2.4, Mesa has introduced it to users. Normally, OpenGL library supports rendering an image by showing in a window on screen. However, sometimes it is needed to finish off-screen rendering for image buffers in memory without showing anything on screen. This facility makes visualization program be independent of various operating systems. It is especially critical for visualization software running in batch job mode. Batch jobs are always running in off-line, lacking display arguments or without X-server supports. Windows can not be created for displaying pictures. So off-screen rendering is required to create intermediate images, animations or high-quality pictures in any size.

After VTK installation with MPI and Mesa extension supports, we can write codes to perform any visualization jobs we want. However, firstly we need to analyze the data we need to simulate. For the given program we want to simulate, the output data file presents multiple particle interactions in a cube. The objects we need to simulate are points. But in VTK data models, there are several kinds of data structures, including Polydata, unstructured and structured grid and image. All of these formats are feasible for point's description. Here, we choose structured grid to describe the data since it is beneficial for future jobs such as data dividing. Based on this data format, we can choose an appropriate reader to read the data. Then the following is filtering, mapping and rendering as basic pipeline. In VTK, especially, there are multiple computer languages are available for programming. But we choose C++ in this project because it seems to be the most natural way for using MPI extension comparing to TCL or others, in experience. That’s because when we use TCL to implement VTK-MPI modules, we can not find an available way to initialize MPI. We have no idea how to transfer pointers to VTK library via TCL script file. However, C++ code does it much more directly and naturally. Then the C++ computer language is preferred.
Now we are concerned ourselves with the VTK programming process. Firstly, we should modify the given code to create data files in VTK file format by adding description header on the created data files. Then we can create a VTK pipeline to simulate the data. According to the basic visualization pipeline described above, a suitable reader is needed firstly. Since we stored the data as the Polydata format, we need to use the vtkPolyDataReader module to read the data and then convert it to a data stream. Since we are simulating the movements of particles, we build spheres to represent the particles. Then we can use the vtkGlyph3D filter module to copy geometric representations, the spheres, to every point of the input dataset. We also need to create an outline by the vtkOutlineFilter module as a box. After that, we can apply the VTK mapper module and render module, and then display an image displayed on the screen. As each data file represents locations of particles in each step of the MD simulation, we can create a set of images which show different movements of the particles. Because all the data files are in the same format, we can create a loop and make all VTK modules do the same work. With time-varying data files read by the reader module, we can create a set of time-varying image files.

From these images, a movie which shows the movements of particles can be created. There are several software available for making movie, such as mpeg2encode on UNIX or VideoMatch program from Gromada[12] on Microsoft Windows. The VideoMatch software makes movie from images in good performance but its demo version just can be freely used for thirty days. The mpeg2encode is free for use,[13] however it’s easy to run out of memory for thousands of images converting. Here, we divide the images into several parts then convert them one by one using mpeg2encode. At last, we use the same software to merge them together.

Above is the brief process of a pipeline in serial program. In order to speed up the visualization process, applying this program on an appropriate parallelism is what will be discussed next.

More information of the process has been written on a log file, which can be found on my website.[14]
Chapter 4

Pipeline parallelism

Scientists often build up many data files with similar contents. Each of these data files represents the same model or simulation system which is in diverse situation, with the files being sequential over the time variation of the simulation. To visualize the simulation these files are always required to be marshalled together and then shown as a movie. Then scientists can study the serial changes of in the model or simulation. According to these features, data sets have similar characteristics, containing the same number of objects which are expressed in the same format. Data files are also stored in same format. Their simulation processes are alike too. The visualization system just use the same pipeline to process these diverse files with same reader, filter, mapper and renderer. Therefore, the output images are also alike (i.e. the same size of scene, etc.). Now we can see that actually the visualization system just loops one pipeline to simulate diverse datasets. The differences of outputs just indicate different values of same object in various files. Based on the characteristics of this problem, now we can apply a parallel model to speed up the visualization process, called pipeline parallelism. Since each dataset from resources is simulated with same series of computing modules in visualization pipeline, we can easily set up a set of copies of this pipeline to do simulation work synchronously. The differences of their jobs are the diverse datasets or data files they handle. Exploiting these pipelines on multiple processes is the core of pipeline parallelism.
The Figure 4.1 shows details of the pipeline approach of our problem. As described above, what the mathematical model generates are a series of time-varying data files which indicate the various positions of thousands of particles. By modifying the given code, these files have been generated in a format which VTK can recognize. Then we can apply multiple VTK pipelines on these files. According to the data format, we use the vtkStructuredGridReader for importing data, then make sphere model and copy it to each location of points got from the data representing the particles. The Mesa extensive modules need to use for mapper, actor, render. Finally, we print out the virtual window on memory as an image and save in disk. We can find that the whole visualization pipeline is totally independent. It can be simply parallelised over multiple processors. These data files can be divided into several stacks. Every processor is responsible for exploiting relevant file stacks in a loop. Then multiple image files are created on disk to make a movie.
The mathematical model we were visualizing contains 1000 particles which are interacted with each other in a cube with length of 4. There are 2000 steps of movements of the particles to simulate, so there are 2000 images to be created. Each image presents a frame of whole movement. Normally, every 24 frames in a second can generate a smooth movie. Therefore, with 2000 frames, a movie of 83 seconds can be generated.

Before discussion of the results, we need to check their correctness. There are two ways to examine results, first is checking the figures of each image generated for each data file. Or we can simply check the size of each image file, since they are all made by same visualization system within same pipeline. However, it’s hard to check thousands of files’ differences just by eye. Moreover, if the data files contain millions particles, eye’s checking is impossible. So we prefer to check their correctness by their size. Once we ensure the correctness of images generated by one processor, we can make a list of image file names and their corresponding file size. Now the correctness of generated images can be examined simply by their size. Moreover, since images are final results of the visualization model, size checking can be used as a method for judging the feasibility of it.

In order to calculate different time spent on the pipeline and its various modules, we can add some statement in VTK code to elapse multiple timing. Table 1 shows each running time of the model when paralleled on different number of processors.
Table 4.1: Running time of program and its modules when processing 2000 data files on various number of processors

According to the table, we can find several critical points:

1. From each elapsed time of different pipelines, we find that the running time of the whole simulation is less and less as increasing of processes number. And the decreasing running time is not only found in the whole processing. We also find that the elapsed time of each module is less and less. Now, it is easy to calculate the speed up and the efficiency of the program. The figures of "Running Time Vs. Process Number" (Figure 4.2), "Speed up Vs. Process Number" (Figure 4.3) and "Efficiency Vs. Process Number" (Figure 4.4) are shown below. Figure 4.2 indicates various running time of multiple modules when the program is running on different number of processors. From this figure, we can find that all the running time decrease whilst the number of processors increases.
Figure 4.2: Running time of Pipeline and its modules versus Number of processors

Figure 4.3: Speed Ups of pipeline versus Number of processors
2. The diverse values of each module show their elapsed time running in different number of processors. From these figures, we also can make another table, Table 4.2. This table shows the percentages of running time of multiple modules in whole pipeline.

<table>
<thead>
<tr>
<th>Process Number</th>
<th>1</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading</td>
<td>5.92%</td>
<td>5.49%</td>
<td>5.35%</td>
<td>5.71%</td>
<td>5.31%</td>
</tr>
<tr>
<td>Filtering</td>
<td>13.44%</td>
<td>13.39%</td>
<td>13.57%</td>
<td>13.57%</td>
<td>13.46%</td>
</tr>
<tr>
<td>Mapping</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Rendering</td>
<td>33.29%</td>
<td>33.11%</td>
<td>33.07%</td>
<td>33.01%</td>
<td>33.28%</td>
</tr>
<tr>
<td>Writing</td>
<td>47.34%</td>
<td>47.98%</td>
<td>47.64%</td>
<td>47.68%</td>
<td>47.72%</td>
</tr>
</tbody>
</table>

Table 4.2: The percentages of running time of various modules in whole pipeline when running on different number of processors.
According to this table, we can clearly see that the percentages of each module are similar. Although the Reader, filter and mapper module just spend about twenty percent of running time of the whole pipeline, their percentages are still quite stable as Render and Writer module do. Their figures almost do not have any change when the process is running on multiple numbers of CPU. It means the running situations of different modules are quite stable even in the parallel approach. Different files’ imports do not make any difference in every pipeline. Each pipeline processes same number of data sets which are stored in same format. Using same modules of each pipeline makes visualization processes spend almost same time dealing with different data files. Now we can realize that each pipeline is so independent that can be parallelized very well.

### 4.1 Performance Analysis

From the figures of speed up and efficiency shown above, we clearly find that the visualization program is almost completely paralleled by multiple processors. As the Table 4.2 shown above, we realize that the pipeline is quite independent and the overheads of the parallelism are quite small, so that the visualization process can be parallelised very well. However, we need to find out whether the efficiency of multiple processors will drop and how many processes can be used to get best performance. There are two more tests we can do to explore these performance problems.

<table>
<thead>
<tr>
<th>Numbers of Data Files</th>
<th>200</th>
<th>400</th>
<th>800</th>
<th>1600</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time (seconds)</td>
<td>116.345271</td>
<td>233.212382</td>
<td>465.871076</td>
<td>929.255087</td>
<td>1137.412109</td>
</tr>
</tbody>
</table>

Table 4.3: The running time of one processor when processing multiple numbers of data files
<table>
<thead>
<tr>
<th>Numbers of Data Files</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Number</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Running Time (seconds)</td>
<td>0.702868</td>
<td>0.712912</td>
<td>0.716095</td>
<td>0.717970</td>
<td>0.715921</td>
</tr>
</tbody>
</table>

Table 4.4: The running time of diverse numbers of processors when processing same number of data files

Table 4.3 shown above indicates the spent time of one processor when visualizes several numbers of data files. Since the pipeline is serial on one process, from Table 4.3, we can find that the running time increases, and the increasing times increase in proportion to the number of data files. That's because pipeline exploits different data files in a loop. It repeats updating its modules on input and output streams, but the visualization process is same for every data file. So the running time is longer, and increases in proportion to the number of data files. On the other hand, the Table 4.4 shows the running time of diverse numbers of processors when visualize diverse numbers of data files. In this test, we are going to find out the overhead of this parallel approach. In the table, the amount of data files is the same as the number of running processes. The running times of different processors are very similar, meaning that the pipelines are really completely parallelised, and the overheads are quite small compared to the whole pipeline (i.e. this is an embarrassingly parallel problem).

### 4.2 Conclusion

Therefore, according to static number of data files, the increasing number of processors can speed up the visualization pipelines faster and faster. Although the value of efficiency is dropping in this process, the maximum of overhead is still quite small to a pipeline. The pipeline parallelism is keeping good performance when processors' number increase to the same number of data files. Therefore the best way to parallel the pipelines is apply-
ing same number of processors as data files in this visualization system if this is possible (i.e. if you can access that number of processors with your available resources).

4.3 Compare with an official pipeline parallelism

According to the official document of Parallel VTK, "A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets"[15], there is another kind of pipeline parallelism available for speed up the time-varying data files. The core of this parallelism is that it divides the visualization pipeline into several parts by the modules, with each module controlled by different processors. The structure of this approach is shown in figure 4.5 below.

![Figure 4.5: The pipeline parallelism from VTK official document](image)

This figure demonstrates how this pipeline parallelism model works for the time-varying data files. In VTK, the visualization pipeline is composed of several modules. We can build up multiple processes run on different processors. These processes are connected with ports through which the data streams can be transferred among multiple modules. As in the figure, there are two processes that are created and named as "consumer" and "producer".[16] The producer takes in charge of reader and filter jobs, and the consumer takes
charge of the remaining, mapper, render and writer. They are connected with ports and controlled by a multiple-process-controller from within multiple methods. In a pipeline, both the consumer and producer start working at the same time, but as the consumer takes charge of second part of pipeline, it needs to wait until the producer outputs its first result. After receiving the data stream successfully, the consumer asks the producer to update and then start applies its modules to process the received data, through mapping and rendering, then a corresponding image will be generated, finishing the pipeline. At the same time the consumer is processing, the producer starts its output update method to read in another data file and process it after receiving its requirement (i.e. which file to process next). Then it outputs its results through output ports and waits for its next requirement. Because what we need to process is a set of time-varying datasets, the consumer needs to update its input ports to receive one more data stream and notify the producer start reading next data file again after the previous pipeline finishes, thus creating another pipeline. The whole process is shown in Table 4.5 below. (In order to make it clearly, here is an example of pipeline divided by two processors)

| CPU 1 | Reading & Filtering 1 | Output | Reading & Filtering 2 | Output | .......... | Waiting | Finish |
|-------|-----------------------|--------|-----------------------|--------|-----------|---------|
| CPU 2 | Waiting               | Input  | Mapping & Render 1    | Input  | .......... | Mapping & Render N |        |

Table 4.5: The pipeline's details of official pipeline parallelism

Based on this approach and reference to the pipeline example of VTK, we can create a code to process our problem. Through its result, we can compare the difference of these two pipeline parallelism models. The Table 4.6 indicates the running results of the second pipeline parallelism model.
<table>
<thead>
<tr>
<th>Numbers of Data Files</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Number</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Running Time (seconds)</td>
<td>6.475732</td>
<td>11.733328</td>
<td>22.910967</td>
<td>44.353120</td>
<td>85.827112</td>
</tr>
</tbody>
</table>

Table 4.6: The running time of official pipeline parallelism when processing different number of data files on two processors

On the other side, we also can provide another table which shows the similar results of the first pipeline parallelism approach.

<table>
<thead>
<tr>
<th>Numbers of Data Files</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Number</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Running Time (seconds)</td>
<td>3.304510</td>
<td>6.701873</td>
<td>13.503549</td>
<td>26.076211</td>
<td>53.871622</td>
</tr>
</tbody>
</table>

Table 4.7: The running time of our pipeline parallelism when processing different number of data files on two processors

Firstly, we need to check the correctness of both these results. Since their productions are a series of image files present same range of datasets. So we can examine the results by checking the images. There are two ways available for images checking which has been discussed above. After checking the correctness, we can compare these two tables. We can easily find that the running time of the first model is faster than the second one when processing the same series of data files. Actually, the reason is obvious from the sketch maps of their pipelines. If we divide the whole pipeline into multiple parts, we can get a table that compares these two parallelisms.
Table 4.8: The pipeline’s details of our pipeline parallelism

Table 4.9: The details of official pipeline parallelism considering communications

These two sketch maps directly show the basic pipeline process of these two approaches. The first one divides the pipelines into stacks as the second one divides the pipeline itself. Although the second one contains more detailed arrangements of multiple modules, the arrangements always lead to more overheads in program. Moreover, the communications between processors are expensive. In the second approach, each pipeline includes a process of transmission, which will waste more and more time as the data files increase in size. Finally, according to the results of Table 4.2, the spending time of each module in the pipeline is almost unchanged even for visualizing different data files on diverse number of processors.

Table 4.10: Spending time of multiple modules of an independent pipeline

<table>
<thead>
<tr>
<th>Modules</th>
<th>Reading</th>
<th>Filtering</th>
<th>Mapping</th>
<th>Rendering</th>
<th>Writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time Percentages</td>
<td>5.51%</td>
<td>13.39%</td>
<td>0.00%</td>
<td>33.28%</td>
<td>47.92%</td>
</tr>
</tbody>
</table>
The results of Table 4.10 indicate that there is almost 0% of running time is spent on reading, filtering and mapping, but 99% on rendering and writing. So the producer usually wastes most of its time on waiting rather than doing something useful. Including the time of consumer in first step for waiting data stream and the time of producer on last step for waiting finishing, this approach is not producing in good performance.
Chapter 5

Data parallelism

High qualitative simulations always indicate visualization system with high requirements on both hardware and software performance. Mathematical models usually produce extremely large data sets (or files) to describe their simulation with high quality and precision. Extremely large data sets often cause difficult I/O, run out of memory and expensive computation of visualization. Based on these features, we can apply some approach to settle the high requirements. Data parallelism is designed for solving this problem. Firstly, extremely large data files are hard for both input and output in file systems. In computer science, Parallel I/O is the most promising way to solve this problem.

However, in visualization systems, unfortunately, this problem can not be solved perfectly. In VTK, there are several modules for parallel IO, such as vtkPDataWriter, vtkPXMLStructuredGridReader... All these modules support parallel reading, but these parallel data files are just a hierarchy. They do not contain any concrete (or user) data. According to an official document about VTK file format [17], some descriptions of "VTK-Parallel-XML" file formats, there are several files represent different pieces of whole data existing in the same file system. Normally, most of scientific data files are generated by scientific programs directly. Modifying a valuable or confidential scientific code to generate multiple extensive data files instead of one large data file may be an overstep of the area of visualization techniques. Therefore, what we are concerned with is the parallel visualization techniques on extremely large dataset rather than the formats of data files.
Figure 5.1: Data Parallelism Approach

Figure 5.1 shows the details of the parallel approach for our problem. As said above, what the advanced scientific program generates is a series of time-varying data files. Each of them contains extremely large dataset. In this problem, the mathematical model we need to visualize contains two hundred thousand points which are interacting with each other in a cube. And there are 20 steps of movements we are required to simulate. Similarly, 20 images will be made. Differing from the last model, each created image presents a figure with large size and high definition. Therefore, a highly qualitative movie will be generated correspondingly. As same as the model above, according to the data format, we use the vtkStructuredGridReader for importing dataset from data files firstly.
However, before mapping, we need to divide dataset into several pieces. Here we use the vtkStructuredGridGeometryFilter to extract various pieces of the large data set and write them down to diverse temporary data files via vtkPolyDataWrtiter. After finishing writing, the master process will send a message to slave processes asking them read different temp files. MPI-send and MPI-receive are required to finish this communication process. After successful delivery, and the corresponding filtering of making sphere model and copying to each location of points got from data presenting particles will be done. Until now, each slave process has created a set of relevant polygonals corresponding to pieces of data.

There are two ways that can be applied to merge these polygonal into a single image. One is merging directly using ports. After processing different part of data, every slave process outputs its filtering results with vtkOutputPort. Correspondingly, the master process applies vtkInputPort to collect the multiple results from slaves. The master input module sends a trigger to multiple output ports asynchronously and asks them to call an update method. The slaves will update their outputs and send it to the master. Each of the transported data is polygonal. The master process needs a module to assemble them together, so a VTK module named vtkAppendPolyData is applied. vtkAppendPolyData is a filter that appends multiple polygonal datasets into one data set. This module only works on the multiple data sets with same polygonal attributes. Since all the appended data sets represent same attributes, they can be assembled completely. After composition on the root process, mapping and rendering modules will be executed and an image will be produced.

The other way of combining the data is mapping and rendering separately at each processor, and then merge together with vtkCompositeManager(in VTK 4.2) or called vtkCompositeRenderManager(in VTK 4.4). vtkCompositeManager operates in each process. And each compositor contains a render window. They use vtkMultiProcessControllers to com-
municate the depth and color buffer to master process’s render window. [18] Using this module in visualization is called parallel rendering. In the parallel rendering approach, the root process distributes pieces of clipped data to slave processes through vtkOutput-Ports. Then the slaves will handle the receiving data using their own method. After filtering and mapping processes, the Composite Manager will be called. Each slave needs this module to assemble their rendering image together. At this time, the master will call an interactor starting method to ask the multiple slaves to start combining their rendering images together and transfer them to the root. Finally, the root process will display the image on screen. According to this composite process, we can find that the interactor starting method is a trigger to communications among various processes. vtkRenderWindowInteractor is a critical module in this approach. It also provides controls for rendering, so it can be used for compositing rendering objects. Based on these features of the parallel rendering module, each piece of divided data is visualized completely by multiple slave processes. The root process just takes in charge of data distribution and composition work. It seems that this approach will be more effective than the first data parallelism method. However, there are two problems influence its performance and usability. The first is that the composition process of rendering objects is much slower than the simple render process. The second is that the interactor starting method does not support off screen rendering method or rendering without X server supports. The composite manager can shut down the rendering window of slave processes, but not the root. There must be a rendering window created in the root process. Even after replacing the render module by the Mesa library and switching on the off-screen rendering option, the visualization program still need the X-server supports and Display arguments. This feature makes this approach difficult to be applied on clusters, especially on the batch or backend of high-performance machines.
Figure 5.2 shows an image of our results. There are two hundred thousand particles contained in a cube. In order to observe the image more clearly, the image should be created with large size. Therefore, a large size with high resolution movie will be made by emerging these large images.

Similarly, before comparing the results, the first thing we need to do is checking their facticity. As it was said before, there are two ways that can be used to examine results, checking the figures of each image and checking size of each image file. In this problem, second one may be more directly and precisely. The created images contain so many particles and in such high resolution. It is impossible to check the differences of two huge pictures by human eyes. Any difference of visualization of any particle will be reflected to image's size. The second method is more appreciate.
5.1 Performance Analysis

In order to explore the performance of Data Parallelism approach, there are several tests to be processed, including calculating its speed up and efficiency. Table 11 indicates the running time of the data parallelism model exploit on large data sets with various numbers of points.

<table>
<thead>
<tr>
<th>Proc. Num.</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data=400k (seconds)</strong></td>
<td>152.274033</td>
<td>157.881744</td>
<td>141.659546</td>
<td>134.980698</td>
<td>132.728317</td>
</tr>
<tr>
<td><strong>Data=200K (seconds)</strong></td>
<td>75.057549</td>
<td>74.983475</td>
<td>68.985092</td>
<td>65.822845</td>
<td>64.569328</td>
</tr>
<tr>
<td><strong>Data=100k (seconds)</strong></td>
<td>36.928329</td>
<td>37.500023</td>
<td>34.644791</td>
<td>33.232735</td>
<td>32.705387</td>
</tr>
<tr>
<td><strong>Data=50k (seconds)</strong></td>
<td>19.076509</td>
<td>19.177038</td>
<td>17.762306</td>
<td>17.158672</td>
<td>17.003225</td>
</tr>
<tr>
<td><strong>Data=10k (seconds)</strong></td>
<td>4.478984</td>
<td>4.650419</td>
<td>4.373367</td>
<td>4.438254</td>
<td>4.537066</td>
</tr>
</tbody>
</table>

Table 5.1: Running time of program when processing different numbers of data objects on various number of processors

From elapsed time above, we find that the running time of the whole simulation is approximately less and less as we increasing the processor numbers. However, this approach can not get a good speed up. The increasing number of process just leads several seconds’ improvement of performance. On the other hand, the program needs more time to visualize larger data files. The running time almost increases linearly as the number of data objects increases. This situation can be shown on figure below.
Running Time Vs. Data Size

Figure 5.3: Running time versus numbers of data objects via multiple processors

Running Time Vs. Processor Number

Figure 5.4: Running time versus numbers of processors processing multiple data objects
From figure 5.3, there are two critical points we can summarize. One is that number of objects which is needed to be processed affect the running time of the program directly. As particle numbers increases, the program needs more time to process its data sets, and the running time of the program correspondingly increases. Moreover, we can find that these curves are all super linear. They show that as more objects needed to be processed, much more works are required and the program will spend much more time to process. That’s because when more objects are loaded, the program needs more memory space to store its data. Complicated loading and writing data from memory pool will increase the overheads of program. Another point is the increasing number of processors does improve performance of the program, but it doesn’t improve well. Especially when it processes small number of objects, the running times of the program almost hasn’t changed when it is run on various numbers of processors. This problem needs further analysis.

![SpeedUp Vs. Processor Number](image)

Figure 5.5: Speed ups versus numbers of processors processing multiple data objects
According to the results we got, we can calculate the speed up and efficiency of this program. The figures of "Speed up Vs. Process Number" (Figure 5.5), "Efficiency Vs. Process Number" (Figure 5.6) are shown above. From the figures, we find that as the number of objects increases, the processors will get more speed up. That's because as more work needed to be done, it is harder for one processor to achieve the visualization successfully. The biggest problem is memory and cache, as they are both the sources of large performance overheads. Dividing large data sets is beneficial for decreasing memory loading and writing problem. Especially on distributed cluster structure, each CPU is using its own memory. Successful solving I/O and memory loading problems is the main intention of data parallelism.

However, according to figure 5.6, the shapes of various efficiency curves are similar. Even though the program processes data files with diverse number of particles, the efficiencies all drop down rapidly. The curves drop from about fifty percent when it is run in
two processors to less than ten percent when it is run in sixteen processors. This fact indi-
cates that this program can not be parallelised well. The serial section of the parallel code
maybe wastes much more running time than the parallel section. Or the overheads of the
parallel code are so huge that the program spends quite a little time. There are several ta-
bles showed below can indicate the problem of the low efficiencies.

<table>
<thead>
<tr>
<th>Process Number</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading</td>
<td>5.23%</td>
<td>4.95%</td>
<td>5.12%</td>
<td>5.10%</td>
</tr>
<tr>
<td>Dividing Data / Writing Temp Files</td>
<td>13.06%</td>
<td>7.32%</td>
<td>5.58%</td>
<td>3.82%</td>
</tr>
<tr>
<td>Filtering</td>
<td>1.30%</td>
<td>2.49%</td>
<td>5.04%</td>
<td>9.92%</td>
</tr>
<tr>
<td>Appending</td>
<td>10.06%</td>
<td>11.31%</td>
<td>14.29%</td>
<td>20.08%</td>
</tr>
<tr>
<td>Mapping</td>
<td>0.03%</td>
<td>0.02%</td>
<td>0.03%</td>
<td>0.03%</td>
</tr>
<tr>
<td>Rendering</td>
<td>43.05%</td>
<td>56.95%</td>
<td>52.54%</td>
<td>43.80%</td>
</tr>
<tr>
<td>Writing Image</td>
<td>27.28%</td>
<td>16.96%</td>
<td>17.40%</td>
<td>17.24%</td>
</tr>
</tbody>
</table>

Table 5.2: The percentages of running time of various modules in whole pipeline when processing 10 thousand data objects on different number of processors.

We can get the running time of various modules by adding time statements on the code
and calling the update method on each module. Table 5.2 shows the percentages of run-
ing time of each module when the program process a data file with ten thousand parti-
cles. Firstly, we should know that the total running times are similar when running on
various numbers of processes (referenced to Table 5.1). Secondly, we also should know
that the parallel sections of pipeline are filtering data. The serial sections are reading,
mapping, rendering and writing image. The dividing data, writing temporary data files
and appending data are overheads of the program. So we can clearly find that as the
number of processes increase, the dividing and writing temp files process spends less and
less time, but the filtering modules and appending process waste more and more time and
the others are similar. On the other hand, in the same pipeline, the rendering modules
spend most time to process. The second critical process is writing the image file.
Table 5.3: The percentages of running time of various modules in whole pipeline when processing 200 thousand data objects on different number of processors.

Table 5.3 displays similar results when the program is processing a data file with two hundred thousand particles. Table 14 displays the results of the program when it handles four hundred thousand particles.

Based on the results from table 5.2 to table 5.4, there are several points that can be found. One is that the running time of serial sections are stable, especially dealing with ex-
tremely large number of objects. The reading, mapping, rendering and writing image processes take similar time to visualize different number of objects. Moreover, the changes of multiple modules’ using time are more and more stable, when the program process large number of objects. According to table 5.4, when the program is executed using more processors, it spends more and more time on the dividing data and writing temp files process, less and less time is spent in parallel sections, filtering and another overhead, appending. The reason is that when more processors are called to run the program, more pieces of data will be divided and more data files will be created, so it needs more time on this procedure. As more pieces of data are divided from large data set, the number of objects that each processor needs to handle will be less, and then it takes less time on the filtering module. After finishing the filtering process, the slave processes are needed to transfer their results to master via ports. As we knew, communications among processors are expensive, so small sizes of data sets will save more running time on the whole transport processes. Therefore, less time is used by appending module when more processors are running.

Based on the analysis above, we realize that when more processors are required for running by the program, just one module of pipeline spend a little more time, but both filtering and appending section are saving time. While the serial parts of the pipeline are almost unaffected, the total running time is less and less. On the other hand, according to these three tables, the changing direction of each module running on diverse numbers of processors are more and more stable when larger data files are visualized. So multiple processes take more time on real work, and their overheads are relatively smaller. The program will get more speed ups.

5.2 Conclusion

Now we can say that the data parallelism does work, but it can not get good speed up
when processing small data files or simple simulated objects. Small data file contains a few objects, which leads to more overheads. Processing greater amounts of simple simulated objects, the program wastes the majority of its running time on serial parts, such as mapping and rendering, so it can not be parallelised well.
Chapter 6

Task parallelism

This is the last parallel model of parallel visualization we are considering. The difference between this approach with others is the independence of each pipeline. Task parallelism is beneficial for visualizing complicated objects which contain multiple independent data sets. So the visualization can be paralleled by multiple processors which execute diverse pipelines. After branches' finish, the master process will merge them together and render on the screen or show in an image. As the development of scientific technology, more and more details and features of scientific models are needed to be observed. So nowadays, many scientific models are becoming more complicated than before. They often consist of dozens of components which present different features of the scientific models. On the other hand, different features are computed with diverse scientific algorithms. Their visualizations can be achieved by using different modules and pipelines.

According to the problem we outlined above, the model contains thousands of points. The visualization objects are simple time-varying data files with extremely large data sets. The target of the visualization is observing the movements of these points. So the task of this problem is so simple that is not appropriated for the application of task parallelism. In order to test task parallelism, we need to choose another scientific model which can be applied. The simplest way is building a number of scientific models directly by VTK modules with various features. In VTK, there are several methods available for creating various polygons, such as vtkImageSource, vtkSphereSource, vtkEarthSource and so on.
Building multiple polygonal objects and processing with different filters is a way to simulate a visualization model which consists of multiple independent pipelines. Applying this model on parallel approach is a way for observing features of task parallelism.

Figure 6.1 shows details of the different pipelines of this created problem. In this problem, we create several scientific polygonal models. Each of these models are made by a simple algorithm then processed with different filters. However, we can modify some arguments of the polygon and make them created more realistically. The visualization program needs more time in simulation using these settings to make the model more similar to real problems and the real world.
6.1 Performance Analysis

There were several tests done to explore the performance of Task Parallelism approach. We needed to compare this model with serial one to find out whether there was any speed up and the merits and defects of the parallel one. The Table 6.1 indicates the running time of the data parallelism model exploit on an extremely large data set with different COMPLEXITY levels of the visualized objects. "COMPLEXITY" is a variable in objects’ creation processes. It indicates amounts of polygons of visualization objects. The larger values of complexity level means that more polygons are contained in objects. And, visualization pipelines will spend more time on the objects’ visualization processes.
<table>
<thead>
<tr>
<th>CPU Number</th>
<th>Complexity = 10</th>
<th>Complexity = 20</th>
<th>Complexity = 30</th>
<th>Complexity = 50</th>
<th>Complexity = 80</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.787386 seconds</td>
<td>7.344162 seconds</td>
<td>19.772768 seconds</td>
<td>76.966476 seconds</td>
<td>279.341095 seconds</td>
</tr>
<tr>
<td>9</td>
<td>1.858008 seconds</td>
<td>3.275298 seconds</td>
<td>9.574819 seconds</td>
<td>28.110550 seconds</td>
<td>57.538181 seconds</td>
</tr>
</tbody>
</table>

Table 6.1: running time of program when processing objects with different complexity level on one and nine processors

According to the table above, the simulation time is longer as the complexity level increases. And we can find that when Complexity equals 10, parallel code spends more time than serial one. That’s because multiple processes waste most of time on communications. However, in task parallelism, each processor only process one object, so this approach can not be applied on multiple number of processors. From the values above, we still can get some useful pointers to analysis this approach. Based on the running time, we still can calculate the speed up and efficiency of the processors with static amount.

![Efficiency Vs. Complexity Level](image)

Figure 6.3: Efficiency versus different complexity levels when running on nine processors
Based on the notion of task parallelism, each process takes charge of different visualizable objects of the data set. The number of processes which are required by the program depends on the number of objects contained in data set. Therefore, according to this specified model, the program just can be paralleled on a specified number of processes. What we can test just records the running time of program when it process different level of complicated objects. This level is controlled by the complexity value of program. According to figure 6.3, more complicated objects lead to more effective use of multiple processors. That’s because more complicated work requires more running time on each process. Multiple processes need more time to setup various models or read data files. They also need more time to process filtering. So before they output their results, more works are required. On the other hand, since the number of objects is limited, the root process just spends a constant time on mapping, rendering and writing image.

6.2 Conclusion

Therefore, if the simulated objects are complicated enough, the program will be parallelised quite well, and multiple processes will be effectively used. Otherwise, parallel code will spend more time than the serial one, since multiple slave processes waste more time on communications with master.

Here we created a test for the task parallelism. Obviously, this approach is not appropriate for our original problem which contains thousands of objects (particles) in a data set. Therefore, deeper performance analysis for a real problem needs to be investigated.
Chapter 7

Conclusions and future work

7.1 Conclusions

In this project, we have visualized a series of time-varying data set which are created by a serial code from a previous project. The mathematical model presents changing situations of a molecular dynamics (MD) system. We have built up a visualization model based on the Visualization Toolkit software system (VTK) to simulate this problem. However, according to this problem, there are thousands of generated data files which represent time-varying movements of particles we need to process. Also, there are several thousands of particles contained in each data file. Normal visualization programs usually take hours to process this problem. In order to speed up the visualization process, we have successfully built up several Parallel Visualization models applied for diverse situations. There are three parallel visualization approaches we have built and applied to our problem. They are Pipeline Parallelism, Data Parallelism and Task Parallelism.

Pipeline parallelism is the most appropriated approach for this problem. It is designed for the visualization of the multiple time-varying data files. According to the real problem, diverse data files represent different situation of same mathematic model. The pipeline of this visualization process is similar to a loop that does the same work several thousands of times. Therefore, pipeline can be divided completely and executed synchronously by multiple processes. From the analysis of results, we can find that this approach has got really good performance. The visualization process has been speeded up with high efficiency. Lastly, we also compared this approach to that of the traditional pipeline parallelism mentioned in VTK official documents. From both theoretic explanations and multiple test results’ analysis, we clearly found that the independence of visualization pipeline makes our approach achieve high speed ups.
Data parallelism is applied on extremely large data sets. The real problem sometimes will generate a set of data files which contain hundreds of thousands or even millions of particles. A simple visualization pipeline needs to load a huge amount of data into memory. This process always adds huge overheads to the program. With the purpose of processing large data set effectively, we divided the data set into several pieces and wrote them to temporary files on the file system. So each temp data file was processed independently by multiple processors. Finally, the root process collected all the outputs returned from each slave and renders them together. According to the results, we found that this approach didn’t get good speed up for our problem. The increasing number of processors led to a few seconds' improvements. Through analysis, we knew that the simulation objects are too simple to be applied well using this type of parallelism, because the pipeline wasted much more time on rendering than the other modules. Parallel rendering is appropriate to solve this problem. However, the required X-server supports and display arguments make it hard to implement. We also found that as the number of data objects increased, this approach got better and better performance. Really extremely large data sets are suitable for data parallelism so it is not so suitable for our problem.

Task parallelism is the parallel approach based on multiple tasks, where the visualization problem is decomposed into several tasks. Each task is finished by one process. This parallelism is beneficial for complicated mathematic model visualization which contains several unique branches, so multiple processes can parallelize it effectively. Obviously, this parallelism does not useful for our real problem. However, it is also hard to find a real scientific problem that contains such a variety or complication of simulation objects. So we built some complicated models to test its performance, and we got good speed ups. Moreover, we found that the higher the complicated level of objects the better the performance, but more tests of this parallelism are needed.

### 7.2 Future Work

Now multiple parallelism approaches have been set up, and the real problem has been visualized with high performance. However, these approaches are just applied on dozens of processors. In order to get deeper information of these approaches, more tests on clus-
ter systems are needed. Furthermore, these approaches will attain better performance running on higher level clusters, such as HPCx.

According to data parallelism, there are two critical things needed to be finished. Based on our results, we already found that larger data set makes multiple processors work more effectively. More complicated problems can make the program spend less time on serial parts of pipeline. Therefore, really extremely large data sets and more complicated models are needed to be applied for further testing. Another point is that parallel rendering is required for higher performance. Our visualization program is building on VTK 4.2 version. But, VTK4.4 (still interim version now) contains more useful parallel modules, such as Parallel-Render-Manager. They can make parallel rendering come true and the program will run more effectively.

According to task parallelism, the most crucial thing is that more complicated visualization models are needed for further test of its performance. This parallelism is the most compatible one amongst the three. It can be applied to the most complex visualization problem. In scientific areas, it is hard to find a real problem which contains so many various objects needing different styles' of visualisation processes. However, this approach is very useful for customer applications or commercial applications, such as large scale design software or some advanced interactive simulation software. After all, building up a parallel visualization system by these approaches and applying it on various problems is the final purpose of this project.
Bibliography

[1] Definitions, History, and Goals of Visualization  
   B.McCormick, T.DeFanti, and M.Brown, 1987

[2] Colliding disk galaxies  
   http://www.mpa-garching.mpg.de/gadget/

[3] Parallel computing (From Wikipedia)  
   http://en.wikipedia.org/wiki/Parallel_computing

[4] VTK Official Download Webpage:  
   http://www.vtk.org/get-software.php

   Sascha Schneider, Thorsten May, Michael Schmidt

   Haber, Robert B. & McNabb, David A.

[7] VTK Official Download Webpage:  
   http://www.vtk.org

[8] An Application Architecture for Large Data Visualization: A Case Study  
   C. Charles Law, Amy Henderson James Ahrens

[9] Introduction to the University of Edinburgh HPC Service (Version 3.0)

[10] Mesa Official Website:  
    http://www.mesa3d.org

    Adrian Jackson

[12] Gromada official download website:  
    http://www.gromada.com/download.html

[13] MPEG Software Simulation Group (MSSG)  
    http://www.mpeg.org/MSSG/

[14] Project progress Log File:  
    http://epcc.ed.ac.uk/~jianqia/vtktutor/Progress.html

James Ahrens, Charles Law, Will Schroeder, Ken Martin, Michael Papka

[16] Referenced to the VTK pipeline example

http://public.kitware.com/cgi-bin/cvsweb.cgi/~checkout~/VTK/Examples/ParallelProcessing/Generic/Cxx/PipelineParallelism.cxx

[17] File Formats for VTK Version 4.2 (Taken from The VTK User’s Guide)


[18] vtkCompositeManager Class Reference

http://www.vtk.org/doc/release/4.0/html/classvtkCompositeManager.html