Parallel Implementation of Hierarchical Tetrahedral - Octahedral (HTO) Subdivision for 3-D Finite Element Mesh Refinement

by

Chulhoon Park

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Engineering.

Computational Analysis and Design Laboratory

Department of Electrical and Computer Engineering

McGill University

Montréal, Canada

June 2006

Your file Votre référence
ISBN: 978-0-494-28613-5
Our file Notre référence
ISBN: 978-0-494-28613-5

ABSTRACT

Parallel computing is being used more and more frequently in 3-D finite element (FE) mesh generation in electromagnetics, due to its improvements in efficiency. When applying parallel computing, the computational problem usually needs to be broken into discrete pieces, so that it can be solved simultaneously with multiple compute resources. Less time is then required than with a single compute resource. In this thesis, an algorithm for hierarchical tetrahedral – octahedral (HTO) subdivision was studied and implemented with a parallel message passing interface (MPI). The data structure was designed in such a way as to store the geometric data during the mesh computation. Also, broadcasting and data gathering was used to build up the final geometric file. The experimental results and the enhancement of system performance are presented, comparing sequential computing with parallel computing. The program was implemented in C language/MPI, and the results obtained have made use of the CLUMEQ [1] supercomputer Centre facilities at McGill University.

[1] CLUMEQ stands for **C**onsortium **L**aval **U**QAM **M**cGill and **E**astern **Q**uebec for high performance computing.

RÉSUMÉ

La computation parallèle est de plus en plus fréquemment utilisé dans la méthode de production d'un maillage d'éléments 3-D finis en électromagnétique à cause des améliorations de son efficacité. Pour appliquer la computation parallèle, la tâche informatique doit généralement être divisé en sous tâches discrètes qui peuvent être accomplies simultanément à l'aide d'ordinateurs multiples, réduisant ainsi le temps qui aurait été requis par un ordinateur unique. Dans cette thèse, nous étudierons un algorithme pour la subdivision tetrahédrale – octahédrale hiérarchisée et nous l'exécuterons utilisant une interface de transmission de message parallèle. La structure des données sera conçue pour stocker les données géométriques dans le processus de production de mailles. Nous procéderons aussi à la transmission et la collection des donnés pour compiler le fichier géométrique final. Les résultats et les améliorations à la performance du système seront présentés, comparant la computation séquentielle avec la computation parallèle. Le programme sera exécuté à l'aide du langage C/MPI, et les résultats seront compilés utilisant les super-ordinateurs du centre CLUMEQ à l'Université McGill.

ACKNOWLEDGMENTS

Firstly, I give thanks to my Lord, Christ Jesus. My Lord gave me strength and wisdom to fulfill the mission of this research work. Also, I would like to thank all the people who have supported me to carry out this work to the end. The greatest thanks belong to my supervisor Dr. Dennis D. Giannacopoulos for his guidance in this research work, and for many helpful discussions. I am also very thankful to Da Qi Ren and Baruyr Mirican, my colleagues at the CAD Laboratory, for providing advice and giving insights that greatly helped me in the work of this thesis. Last, but not the least, I would like to thank my wife, Joanna Park for always being encouraging and supportive.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In the introduction, a basic background will be provided to understand this research work. In discussion will be the finite element method, tetrahedral mesh generation, and tetrahedral mesh refinement. The motivation and objectives, and the overview of this thesis are presented.

1.1 The Finite Element Method

The finite element method (FEM) is a numerical tool used in the study and evaluation of engineering problems for determining approximate solutions [1, 2]. It is widely used for the analysis of problems governed by partial differential and integral equations. The principle of the finite element method is to replace an entire continuous domain by a number of sub-domains. It is assumed that the behaviour of the complex structure, whose solution may be difficult, can be described by simple functions of the coordinates within the element. These functions are known as shape functions, and they describe the relationship between the unknowns at the nodes and the unknowns within an element [1, 3].

The steps to solve a problem are shown in figure 1-1. The first step is that a physical problem should be transformed into a mathematical model followed by an approximate numerical solution of the mathematical model. For a given mathematical

model the FEM is an efficient method of obtaining a numerical approximate solution. Then the second step is to divide the region into a number of smaller regions. It is necessary to reduce the number of degrees of freedom to a finite number. Thus, in using the finite element method the essence of solving the problem is the discretization of the continuous problem by users. The users should properly discretize the problem to lead to a solution. The finite element method is a popular discretization technique for representing a physical system [4].

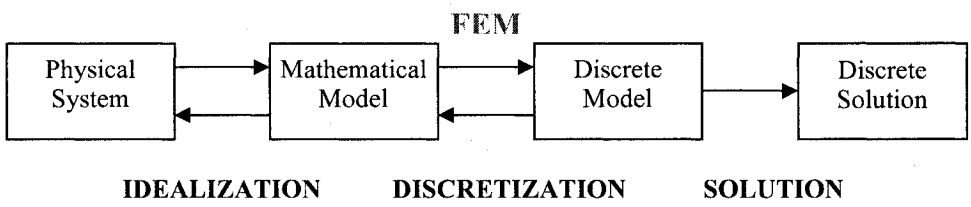

Figure 1-1: A view of the physical simulation process [4].

1.2 Tetrahedral Mesh Generation

A mesh is a list of elements, nodes, faces, edges, and other data that describes the computational domain. The geometry in a finite element analysis is represented by the collection of finite elements used, known as a mesh. Creating the mesh is often a difficult part of 3-D finite element modeling [4, 5]. With the capabilities of modern computers software developers have taken advantage of CAD-like interfaces for finite element programs. For the purpose of graphical observation, mesh generation can be performed graphically, manipulating lines on a computer screen to form elements. Thus, these techniques were developed for the creation of complex geometric models.

Geometries that already existed in some format can now be used directly for simulation, assuming the programs being used have the necessary interface capabilities. Once the geometry is entered in the system, most finite element programs allow for some type of automatic meshing. This is a process by which a finite element mesh can be created automatically.

A tetrahedron is the most flexible element in three dimensions [6]. It is defined by four vertices and a function sampled at these vertices leads to a unique function which has useful properties for reconstruction and interpolation [6].

1.3 Tetrahedral Mesh Refinement

Mesh refinement is the substituting of elements with modified elements, creating points inside domains or on the boundaries of the domains and inserting those points in the initial mesh [6-8]. The increase of the accuracy of the solution in that area of the mesh is the purpose of the refinement.

Successive tetrahedral mesh refinement is crucial in the finite element methods where the elements are based on tetrahedral meshes [9]. Zhang [9] proposed two methods for subdivision of a regular tetrahedron: labelled-edge subdivision and short – edge subdivision. These methods are depicted in figure 1-2.

When refining a tetrahedron into half-sized tetrahedra, new vertices are added at the middle of the six edges. Thus, the four corner tetrahedra and a central octahedron are

produced. The octahedron can be further subdivided into four tetrahedra by cutting in three ways (by adding the edges aa', bb' or cc') [9]. Figure 1-2 shows that an octahedron is subdivided into 4 tetrahedra by adding the edge aa'. The labelled-edge subdivision scheme employs a direct numbering scheme for vertices being generated and subdivides the octahedron in accordance with this numbering [9]. In the short-edge subdivision scheme the shortest of the three interior edges is chosen [9]. In this thesis, the splitting shown in figure 1-2 is used for the hierarchical tetrahedral – octahedral (HTO) subdivision algorithm in chapter 2 to get rid of the octahedra at the finest level and to get a mesh of all tetrahedra.

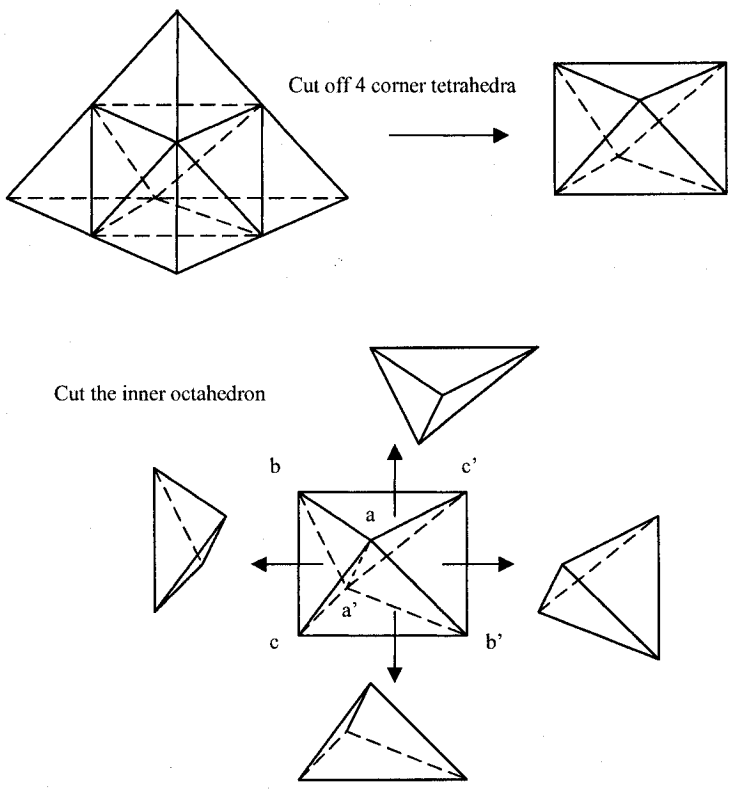

Figure 1-2: Subdivision of a tetrahedron into 8 subtetrahedra by adding the edge aa' [9].

1.4 Motivation and Objectives

Tetrahedral finite elements are amongst the simplest shapes into which 3-D regions can be broken down, and they are well-established in mesh generation. For the purpose of achieving the geometric discretization of the problem domain in 3-D electromagnetics, they are extensively used to analyze and design with the FEM [10-12].

The hierarchical tetrahedral – octahedral (HTO) subdivision algorithm [13] generates a hierarchy for 3-D finite element (FE) meshes. As subdivisions increase, however, the elements in the shape of tetrahedra or octahedra have an enormous number of vertices (nodes) and faces. Due to the size of many FE problems, a direct sequential solution is a time consuming task. Thus, other techniques are useful to solve this problem. Computing with different processing elements (PEs) of a FE problem at the same time (i.e. parallel computing) should greatly reduce computing time.

This thesis is focused on analyzing a real parallel algorithm with message passing interface (MPI). We first design a sequential program for HTO subdivision, and then implement it using parallel computing. Through MPI implementation designed with a master – slave parallel computing structure, we figure out the efficiency of parallelization, the balance of the workload, and the time cost distribution.

1.5 Thesis Overview

In chapter 2, we investigate a method for hierarchical tetrahedral – octahedral (HTO) subdivision, and develop a program to demonstrate sequential computing based on a regular refinement strategy. The data structure was designed to store the geometric data during the mesh computation.

Chapter 3 presents the parallelization of the sequential algorithm using MPI. In this chapter we present an overview of available parallel architecture and general principles for the parallel algorithm design. We also will give an overview of MPI. We designed our parallel structure based on a master - slave structure, considering sub-domain distribution, and load balancing.

The sequential program design in chapter 2 and the parallel program design in chapter 3 are then implemented and tested. The results of experiments are given in chapter 4.

Chapter 5 concludes this research work and investigates the objectives of possible future work related to this research.

Chapter 2

The Sequential Program Design for Hierarchical Tetrahedral – Octahedral (HTO) Subdivision

In this chapter we present the sequential program design for hierarchical tetrahedral – octahedral (HTO) subdivision, so as to simplify later discussion on parallelization of the algorithm in chapter 3. The algorithm follows the regular refinement rule to generate hierarchical subdivision of meshes.

2.1 The Regular Refinement Rule

In the first phase, each tetrahedron marked for refinement is divided into four sub-tetrahedra of equal volume and one octahedron [9, 13, 14]. This is done by adding a new vertex at the midpoint of each edge, building new sub-tetrahedra with the old vertices and the newly inserted vertices. Using this procedure we get four congruent sub-tetrahedra at the corners and one octahedron in the centre of the parent tetrahedral, as shown in figure 2-1.

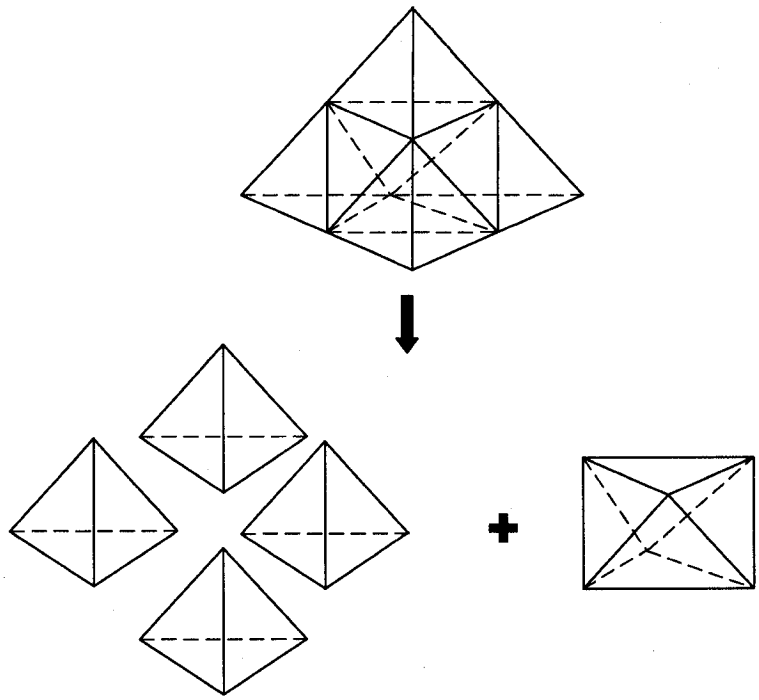

Figure 2-1: Tetrahedral regular refinement [13].

At the centre of the parent tetrahedron there is one octahedron. The regular refinement rule [13, 14] of an octahedron subdivides an element in two steps. First, by connecting all edge midpoints of each face and second, by connecting the triangles at the middle of the faces to the barycentre of the parent element. The result is six octahedra and eight tetrahedra, as shown figure 2-2.

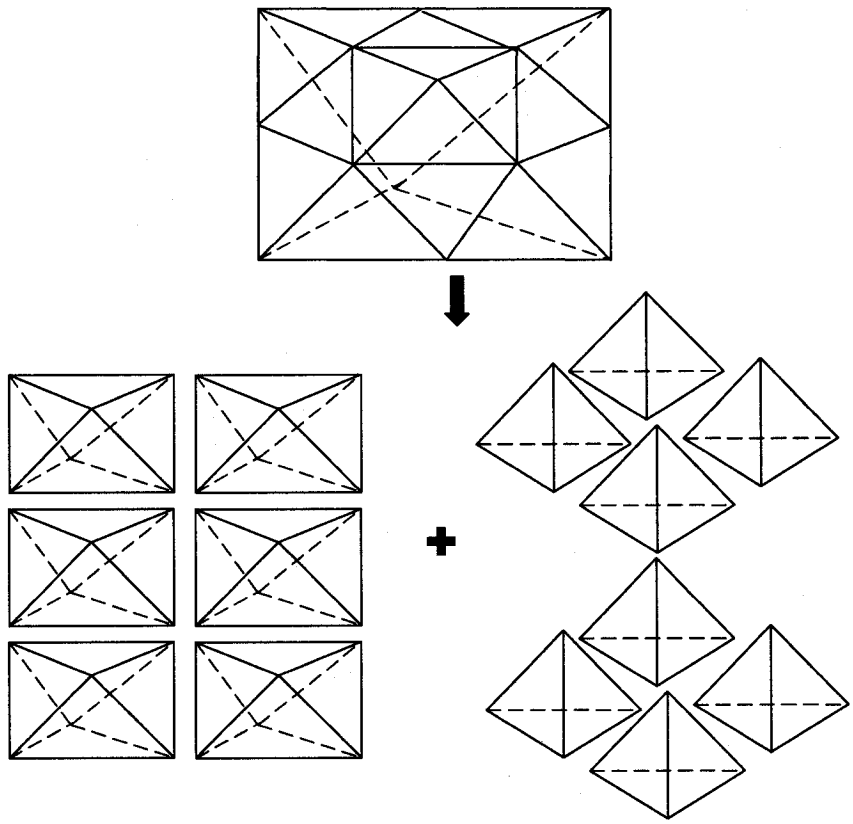

Figure 2-2: Octahedral regular refinement [13].

2.2 The Data Format for Geometric Computing

Computational geometry is a rapidly evolving interconnected field, involving computer science, engineering and mathematics. Geometric computing deals with geometric problems of an algorithm. Computing with geometry has many applications such as computer graphics, computer-aided design visualization, and computer vision

[15]. In general the input to a geometric algorithm is a set of geometric objects, such as the sequence of vertices of a polygon or polyhedron. The output is a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object.

For our implementation we will use Geomview [16] as the computational geometric tool. "Geomview is an interactive program for viewing and manipulating geometric objects. It runs on a wide variety of Unix computers, including Linux, SGI, Sun, and HP [16]." The main purpose of Geomview is to display objects whose geometry is given.

After compiling our program, the outputs data file must be formatted as required by Geomview. Our program is designed to produce formatted outputs for geometric computing so that we can use the Geomview tool for viewing and manipulating the geometric objects. Geomview computes the following simple data type with the information of vertices (nodes) and faces as shown in figure 2-3. The conventional suffix for object file format (OFF) files is '. off' [16].

```
[ST] [C] [N] [4] [N] OFF                                  # Header Keyword
[Ndim]                                                    # Space dimension of vertices
NVertices   NFaces    NEdges                              #NEdges not used or checked
x[0]            y[0]            z[0]                      # 3-D Vertices, possibly with normal
x[1]            y[1]            z[1]
. . . . .
x[NVertices-1]  y[NVertices-1]  z[NVertices-1]


Nv      v[0]    v[1]    v[2]
. . . . .                                                 # Faces
                                                          # Nv = # vertices on this face (i.e. 3)
                                                          # v[0] ... v[2 ]; vertex indices
                                                          # in range 0 .. NVertices-1
```

Figure 2-3: Data format for Geomview [16].

OFF [16] files represent collections of planar polygons with possibly shared vertices, which is a convenient way to describe polyhedra.

An OFF file may begin with the keyword OFF. Ndim is space dimension of vertices and present only if the keyword OFF is presented as NOFF. Three ASCII integers follow: NVertices, NFaces, and NEdges. These are the number of vertices, faces, and edges, respectively. Current software does not use nor check NEdges; it need not be correct but it must be present. The vertex coordinates follow: dimension * NVertices

floating-point values. They are implicitly numbered 0 through NVertices-1. Dimension is either 3 (default) or 4 (specified by the key character 4 directly before OFF in the keyword). Following these are the face descriptions, typically written with one line per face. Each has the form: N Vert1 Vert2 ... VertN [color]

Let N represent the number of vertices on this face, and Vert1 through VertN are indices into the list of vertices (in the range 0 .. NVertices-1).

2.3 The Design of Vertex and Face Refinement

In order to obtain the data file (.off) for the geometric computing, in this section we focus on the design of the vertex and face refinement of tetrahedra and octahedra.

2.3.1 Tetrahedral Vertex and Face Refinement

Firstly, we consider the regular refinement of a tetrahedron. The regular refinement rule [13] refines each edge at the midpoint and each face into four triangles all of which are congruent to the parent face. In the first subdivision, six vertices (m[0], m[1], m[2], m[3], m[4], m[5]) are added at the midpoint of each edge. We regard the initial input data as V_0, V_1, V_2, and V_3. The numbering of the new vertices added is as shown in figure 2-4. As shown in figure 2-4, 20 faces are added in the first subdivision and the information is stored as the output for the output data file.

The functions of add_face ( ), copy_vertex ( ), and find_vertex ( ) (these functions will be discussed in the part of overview of implementation in chapter 4) are used to do this job. After the first subdivision, the parent tetrahedron gives rise to the new faces of the four child tetrahedra and a child octahedron as follows:

- Tetrahedron 1: face (v0, m[0], m[1]), face (v0, m[0], m[2]), face (v0, m[1], m[2]), face (m[0], m[1], m[2])
- Tetrahedron 2: face (v1, m[0], m[3]), face (v1, m[0], m[4]), face (v1, m[3], m[4]), face (m[0], m[3], m[4])
- Tetrahedron 3: face (v2, m[1], m[3]), face (v2, m[1], m[5]), face (v2, m[3], m[5]), face (m[1], m[3], m[5])
- Tetrahedron 4: face (v3, m[2], m[4]), face (v3, m[2], m[5]), face (v3, m[4], m[5]), face (m[2], m[4], m[5])
- Octahedron: face (m[0], m[1], m[3]), face (m[0], m[2], m[4]), face (m[1], m[2], m[5]), face (m[3], m[4], m[5])

As we observe the faces of the octahedron, only 4 faces were generated for the octahedron. These are just the faces not already generated for the surrounding tetrahedra.

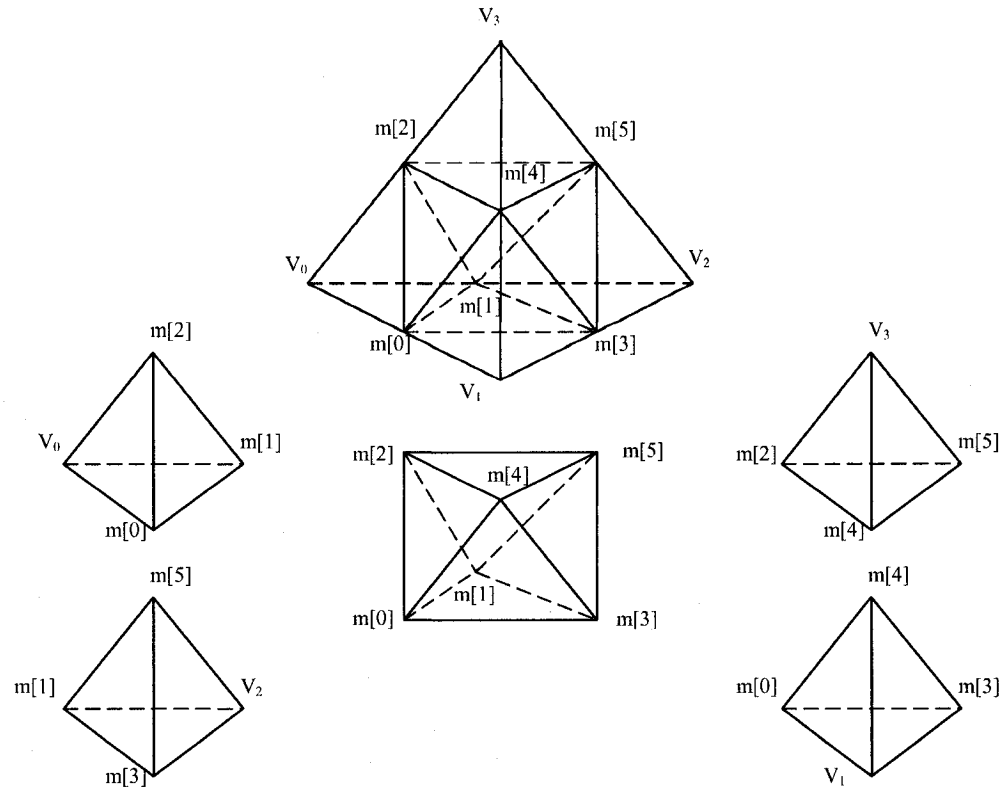

Figure 2-4: Vertices numbering for tetrahedral refinement.

2.3.2 Octahedral Vertex and Face Refinement

We observed that an octahedron, according to the regular refinement rule [13], results in six octahedra and eight tetrahedra. These are obtained by connecting all edge midpoints of each face and by connecting the triangles at the middle of the faces to the barycentre of the parent octahedron. In the first subdivision only one vertex, among 13 new vertices, is added to the data file since the other 12 vertices are duplicated and are already counted in the first subdivision of the parent tetrahedron. However, the 12 vertices are important for continuous subdivision to occur. The numbering of the

octahedra vertices as well as their adding and their next subdivision is as shown in figure 2-5.

As shown in figure 2-5, we see a total of 68 faces; 36 faces from six octahedra and 32 faces form eight tetrahedra. However, most of the faces are duplicated. Finally, only 28 faces among 68 faces excluding duplication are stored for the output data file. The functions of add_face ( ), copy_vertex ( ), and find_vertex ( ) do this job. The newly obtained information of faces is as follows:

- Tetrahedron 1: face (m[1], m[3], m[7]), face (m[1], m[3], m[12]), face (m[1], m[7], m[12]), face (m[3], m[7], m[12])
- Tetrahedron 2: face (m[2], m[3], m[12]), face (m[2], m[9], m[12]), face (m[3], m[9], m[12])
- Tetrahedron 3: face (m[0], m[2], m[5]), face (m[0], m[2], m[12]), face (m[0], m[5], m[12]), face (m[2], m[5], m[12])
- Tetrahedron 4: face (m[0], m[1], m[12]), face (m[0], m[4], m[12]), face (m[1], m[4], m[12])
- Tetrahedron 5: face (m[4], m[6], m[8]), face (m[4], m[6], m[12]), face (m[4], m[8], m[12]), face (m[6], m[8], m[12])
- Tetrahedron 6: face (m[5], m[6], m[12]), face (m[5], m[10], m[12]), face (m[6], m[10], m[12])
- Tetrahedron 7: face (m[9], m[10], m[11]), face (m[9], m[10], m[12]), face (m[9], m[11], m[12]), face (m[10], m[11], m[12])

- Tetrahedron 8: face (m[7], m[8], m[12]), face (m[7], m[11], m[12]), face (m[8], m[11], m[12])
- Octahedron 1: face (v0, m[0], m[2]), face (v0, m[1], m[3])
- Octahedron 2: face (v1, m[0], m[5]), face (v1, m[4], m[6])
- Octahedron 3: face (v2, m[1], m[7]), face (v2, m[4], [8])
- Octahedron 4: face (v3, m[2], m[5]), face (v3, m[9], m[10])
- Octahedron 5: face (v4, m[3], m[7]), face (v4, m[9], m[11])
- Octahedron 6: face (v5, m[6], m[8]), face (v5, m[10], m[11])

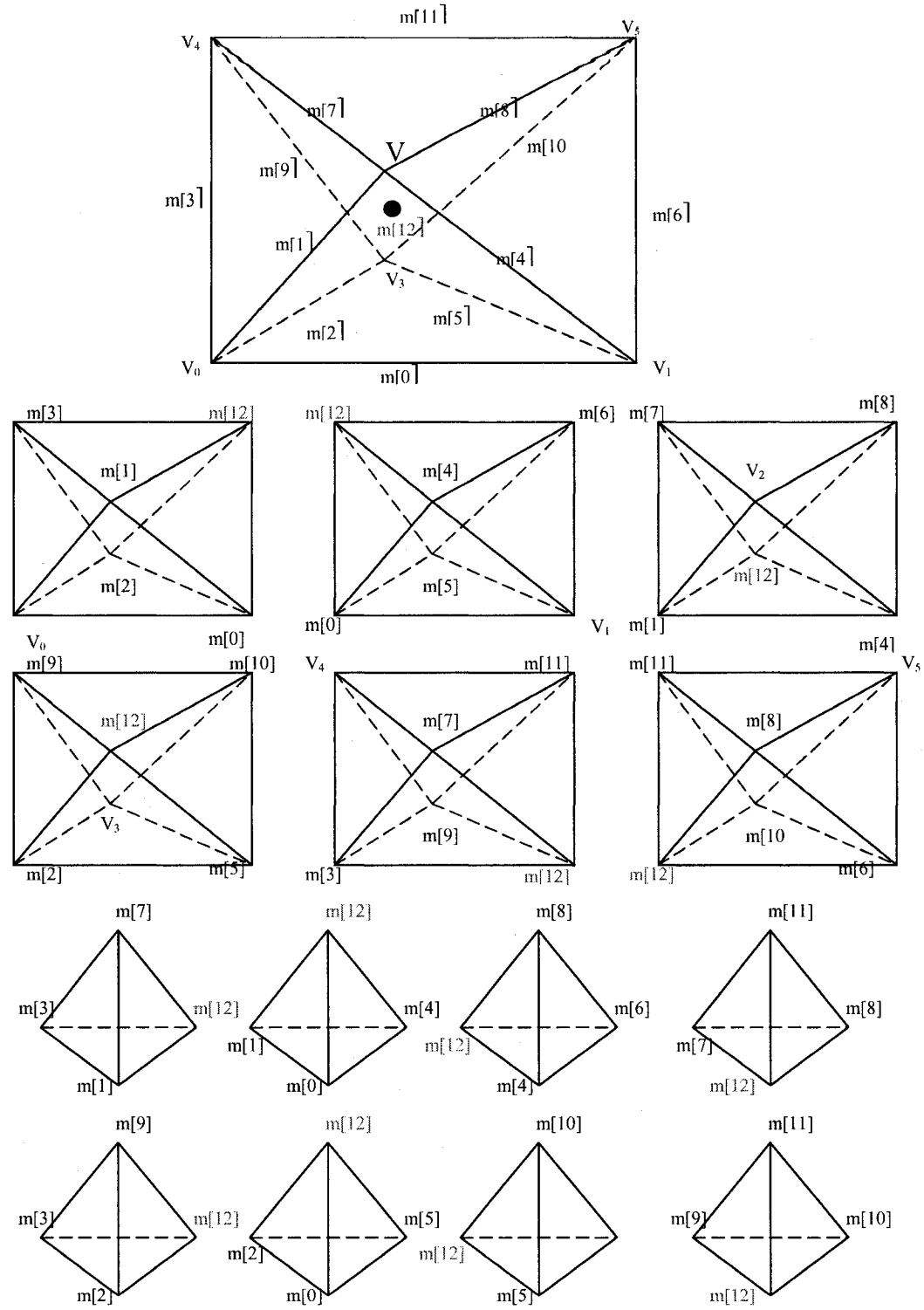

Figure 2-5: Vertices numbering for octahedral refinement: m[12] is barycentre.

2.4 *The Process Functional Design*

The design of the sequential algorithm used is based on a recursive algorithm. A recursive algorithm calls itself with smaller (or simpler) input values [17]. Then it obtains the result for the current input by applying simple operations to the returned values for the smaller (or simpler) input [18]. It may return to its small input values until it reaches the base case. Using recursion, a complex problem can be split into its single simplest case. Recursive functions are important paradigms in recursive programming, for they only know how to solve the simplest case.

As we observed in the sequential program design, the recursive algorithm is necessary for the design of hierarchical tetrahedral-octahedral subdivision. The major advantage of using a recursive algorithm is that it is simpler for the parallelization of the sequential program. Because of this advantage, we have selected a recursive algorithm to design the program.

This sequential algorithm is composed of the main function, the process functions, and other utility functions. In order to reduce the duplication of element information of mesh generation, our design makes use of three different processes of the tetrahedron: processTetra0, processTetra1 and processTetra2. According to the regular tetrahedral refinement rule, the process function of each parent tetrahedron generates four child process functions of the tetrahedra and a child process function of the octahedron as follows:

- The processTetra0 generates five process functions: four processTetra0 functions and one processOcta function.
- The processTetra1 generates five process functions: two processTetra0 functions, two processTetra1 functions, and one processOcta function.
- The processTetra2 generates five process functions: one processTetra2 function, three processTetra1 functions, and one processOcta function.

According to the regular octahedral refinement rule [13], the process function of each parent octahedron generates eight process functions of the tetrahedron and six process functions of the octahedron. In order to reduce the duplication of element information of mesh generation, the eight process functions of the tetrahedron are composed of four processTetra2 functions and four processTetra0 functions.

- The processOcta generates 14 process functions: four processTetra2 function, four processTetra1 functions, and six processOcta function.

Two diagrams of the recursive functional flowchart are shown in figure 2-6 and figure 2-7. Figure 2-7 shows the simplified diagram of the recursive functional flowchart.

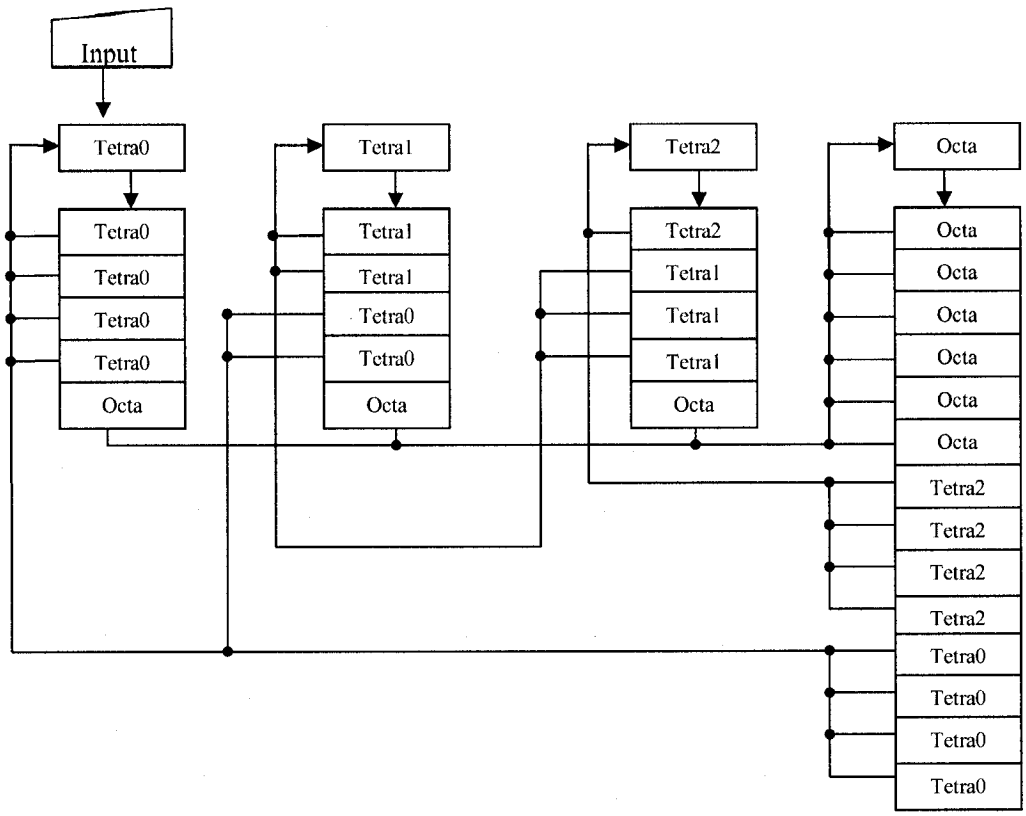

Figure 2-6: Diagram of the recursive functional flowchart: Tetra0 is processTetra0, Tetra1 is processTetra1, Tetra2 is processTetra2, and Octa is processOcta.

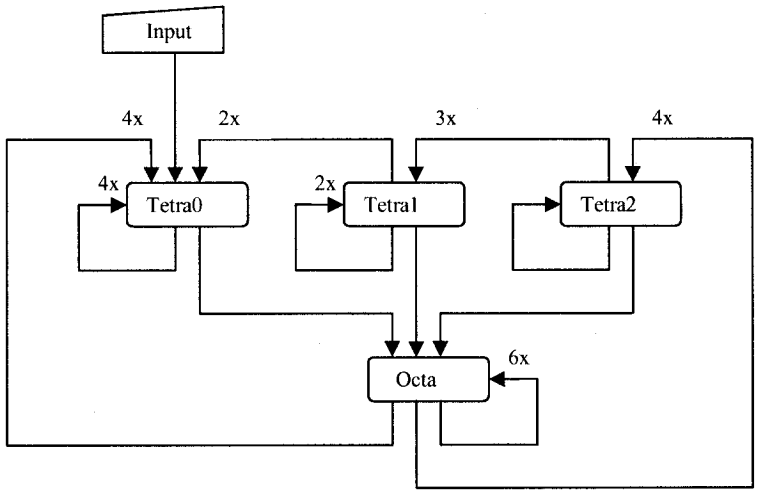

Figure 2-7: Simplified diagram of the recursive functional flowchart: 2x, 3x, 4x, and 6x are total numbers of iterations in each process function.

2.5 *The Design of the Data Flow*

Let us set the initial input data for the ancestor tetrahedral: V_0, V_1, V_2, and V_3. Then let us set the newly added six vertices at the midpoint of each edge: m[0], m[1], m[2], m[3], m[4], and m[5]. The six vertices and the initial vertices produce new data. They become new input data for the next subdivision. After the first subdivision, the input data of each child tetrahedron for the next subdivision is as follows:

- Tetrahedron 1: (V_1, m[0], m[1], m[2])
- Tetrahedron 2: (m[0], V_1, m[3], m[4])
- Tetrahedron 3: (m[1], m[3], V_2, m[5])
- Tetrahedron 4: (m[2], m[4], m[5], V_3)
- Octahedron: (m[0], m[1], m[2], m[3], m[4], m[5])

For octahedral subdivision, let us set 12 vertices at the midpoint of each edge: m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7], m[8], m[9], m[10], m[11]. And let us set the vertex at the barycentre: m[12]. After the first octahedral subdivision, the input data of each child tetrahedron for the next subdivision is as follows:

- Octahedron 1: (v0, m[0], m[1], m[2], m[3], m[12])
- Octahedron 2: (m[0], v1, m[4], m[5], m[12], m[6])
- Octahedron 3: (m[1], m[4], v2, m[12], m[7], m[8])
- Octahedron 4: (m[2], m[5], m[12], v3, m[9], m[10])
- Octahedron 5: (m[11], v4, m[7], m[9], m[12], m[3])

- Octahedron 6: (v5, m[11], m[8], m[10], m[6], m[12])
- Tetrahedron 1: (m[12], m[7], m[3], m[1])
- Tetrahedron 2: (m[3], m[2], m[12], m[9])
- Tetrahedron 3: (m[12], m[0], m[2], m[5])
- Tetrahedron 4: (m[0], m[4], m[12], m[1])
- Tetrahedron 5: (m[12], m[4], m[6], m[8])
- Tetrahedron 6: (m[6], m[10], m[12], m[5])
- Tetrahedron 7: (m[12], m[10], m[9], m[11])
- Tetrahedron 8: (m[11], m[8], m[12], m[7])

The diagram of data flow is shown in figure 2-8. This diagram shows the data flow, such as input data by user, the data generated by each process function, and the stored data.

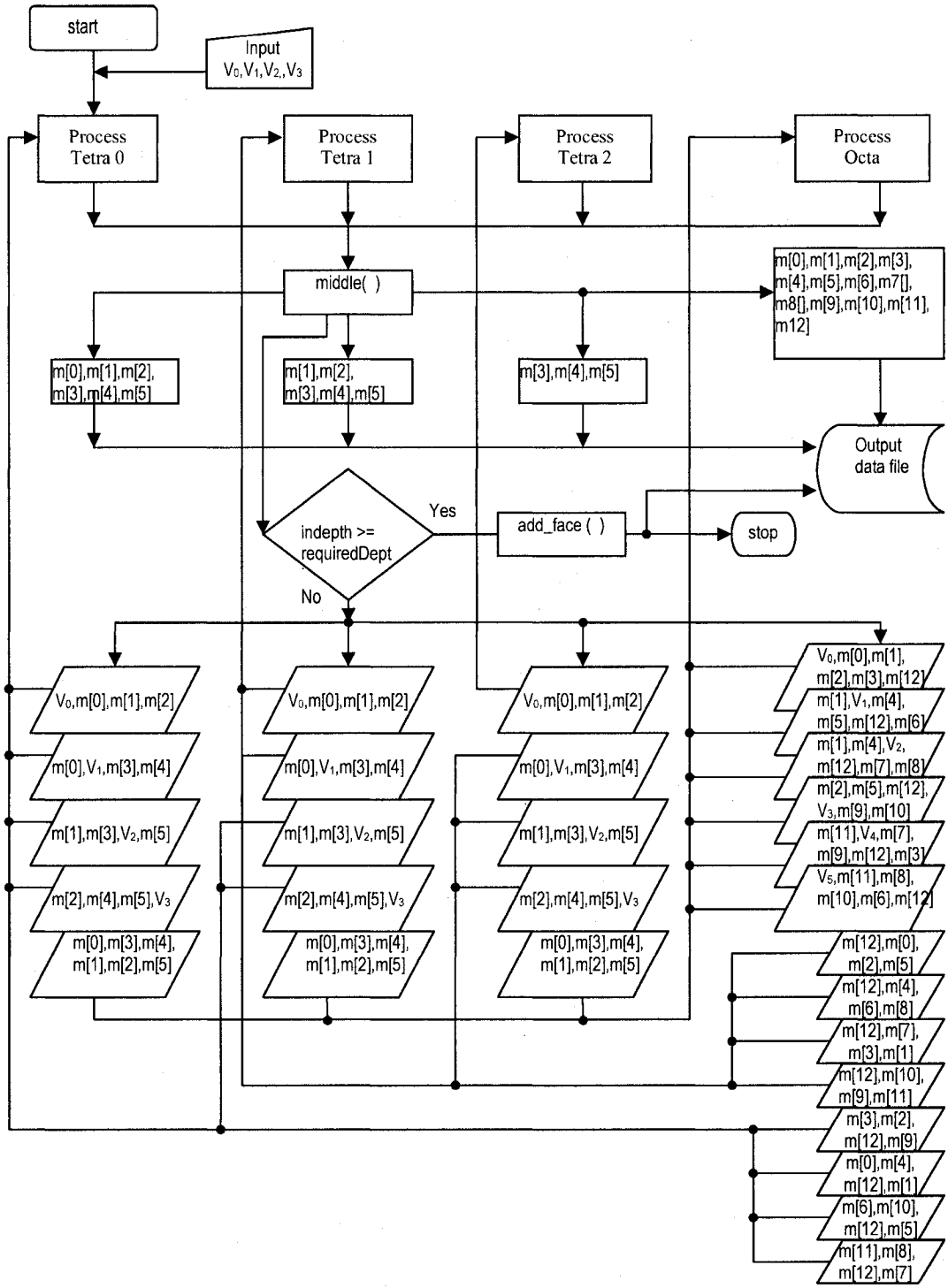

Figure 2-8: Diagram of the data flow.

Chapter 3

Parallelization using MPI

Parallelization is necessary and becomes useful when the processing of a sequential computing problem takes too much time. The main goal of parallelization is to use a parallel computer to reduce the time needed to solve a single computational problem.

In the simplest sense, a parallel computer is a collection of processing elements (PEs) that cooperatively solve the given task. Parallel computing involves taking a problem and dividing it into pieces that are to be solved concurrently [19]. Each concurrent piece is called a process. In order to pass information, communication and synchronization are required between the processes.

When interactive methods are parallelized on a multiprocessor system the distribution of data and the communication scheme between nodes of the system are important for an efficient execution. We discuss our strategy and implement the algorithm for hierarchical tetrahedral – octahedral (HTO) subdivision using the message passing interface (MPI). We will use broadcasting and data gathering to build up the final geometric file.

3.1 Parallel Models

A parallel computer is a collection of processing elements (PEs) used to perform a given task [20, 21]. Parallel computational models can be classified into four different categories: Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD) depending on whether single (S) or multiple (M) streams are used for instructions (I) and data (D) [20-23]. This classification is too rough: *SISD* denotes sequential computers, *MISD* is not very practical, *SIMD* denotes only a small group of architectures but *MIMD* contains tens of architectures [20, 23].

Perkowski [23] classifies parallel architectures into six practical models: SIMD, parallel vector processor, symmetric multiprocessor, massive parallel processor, cluster of workstations and distributed shared memory.

- *The Single Instruction Multiple Data (SIMD)* has only a single control unit so that only one process runs. This model suits such an algorithm where input data can be divided into several groups and processed concurrently [23].
- *Parallel vector processor* contains a small number of powerful vector PEs based on SIMD connected together and to the common shared memory by a crossbar network switch [22, 23].
- *The Symmetric Multiprocessor* is well suited for any algorithm since it contains a small group of common processing elements that are used in sequential computers [22, 23]. Each processing element has equal access to a

common shared memory and I/O devices via a bus. The disadvantage is that the efficiency of the system goes down with the increasing number of PEs, because the speed of data transfer is limited. On the other hand, the cost of this model is low.

- *The Massive Parallel Processor* consists of a large group of common PEs. Each PE has exclusive access to its distributed local memory. The PEs are connected together through the use of a large number of serial lines to obtain high performance [22, 23].

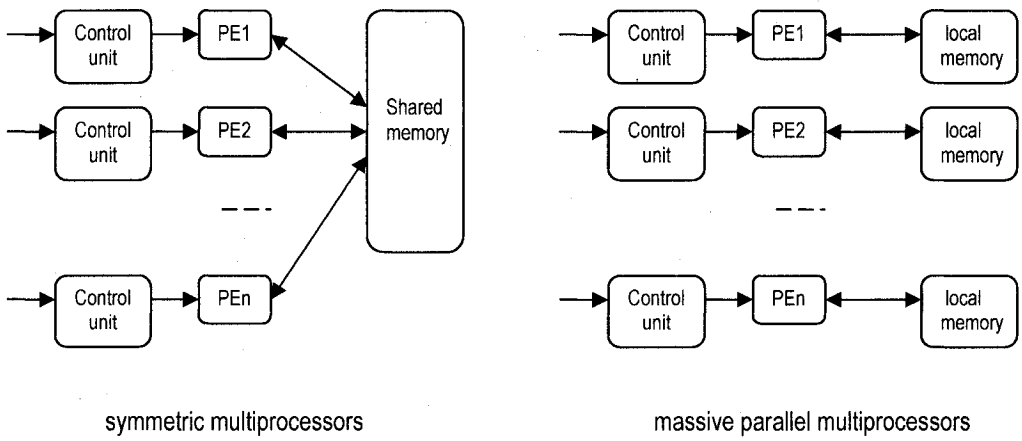

Figure 3-1: Parallel model structures [20].

- *Cluster of workstations* contains a large group of sequential computers. Because these computers are connected together via low cost, this model can be extremely cost effective. Each processing unit is a complete computer having its own operating system, input-output devices, PE, and local memory [22, 23].

- *Distributed Shared Memory (DSM)* combines advantages of the massive parallel processor and the symmetric multiprocessor. The model refers to a wide class of software and hardware implementations allowing a higher number of PEs [22-24].

Each parallel model requires the use of a different programming technique in the design of a parallel algorithm, because it offers different means. An algorithm developed for the architectures with distributed memory can use message passing for the communication between the processes.

3.2 Parallel Programming

Parallel programming is a programming technique that provides the means for executing operations concurrently, either within a single computer, or across a number of systems [22, 25, 26].

We can identify parallel algorithms according to the use of programs and data used in parallel algorithms for the decomposition of computation [27]: Multiple Program Single Data (MPSD) parallelism, Single Program Multiple Data (SPMD) parallelism, and Multiple Program Multiple Data (MPMD) parallelism.

- *Multiple Program Single Data (MPSD)* parallelism subdivides into several distinct functions to be applied in series to individual data items. Each function is exclusively assigned to its PE and a data path is provided from one PE to

another one. Each program task computes a part of the data and the sub-results are combined afterwards.

- *Single Program Multiple Data (SPMD)* parallelism subdivides the data set into streams at the beginning of the computation and this partition remains together for the whole process. The outputs of the processing of streams obtained by the PEs are merged afterwards to get the final result.
- *Multiple Program Multiple Data (MPMD)* parallelism subdivides the input data set into multiple streams [22]. These streams are assigned to processors executing multiple programs. This parallelism is typically designed for architecture with distributed memory. It focuses on problems that cannot be processed on a sequential computer because of technical limitations. Therefore, parallelism based on MPMD algorithms does not enhance speedup and efficiency.

After examining the parallel model structures and the methods of parallelization of algorithms, we have chosen the scheme using the *massive parallel processor* as our parallel platform for our parallel model structure. For this structure, we have considered one built around the C programming language and the message passing interface (MPI) communication library. Also, the parallel programming technique used is the s*ingle program multiple data (SPMD),* which is the most extensively used method for efficient MPI programming.

3.3 *MPI Main Functions in this Work*

MPI defines a library of subprograms for parallel computing that can be called from C and Fortran 77 programs [19, 28-32]. It has been fully designed to allow maximum performance on a wide variety of systems, so it has rapidly received widespread acceptance. It is also based on message passing, one of the most widely used and powerful techniques for programming parallel systems [28]. The basic communication mechanism of MPI is the transmission of data between a pair of processes, one side sending, and the other side receiving.

The fully functional message-passing program in this work is implemented by using the routines: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv, MPI_Gather, MPI_Gatherv, MPI_Bcast, and MPI_Barrier.

- *MPI_Init (&argc, &argv)* initializes MPI. It is required in every MPI program and must be the first MPI call.
- *MPI_Finalize ( )* terminates MPI. All MPI functions must be called between MPI_Init ( ) and MPI_Finalize ( ).
- *MPI_Comm_size (MPI_Comm_world, numprocs, int *size)* determines the number of processes that the user has started for this program. The value numprocs is actually the size of the group associated with the default communicator MPI_COMM_WORLD.
- *MPI_Comm_rank (MPI_Comm comm., int *rank)* determines the label of the calling process.

- *MPI_Send (address, count, datatype, destination, tag, comm)* sends a message. (address, count, datatype) describes count occurrences of items of the form datatype starting at address.
- *MPI_Recv (address, maxcount, datatype, source, tag, comm, status)* receives a message. (address, maxcount, datatype) describe the receive buffer as they do in the case of MPI_Send.
- *MPI_Gather ( )* gathers together values from a group of processes.
- *MPI_Gatherv ( )* gathers into specified locations from all processes in a group.
- *MPI_Bcast ( )* broadcasts a message from the process with rank root to all processes of the group, itself included. It is called by all members of group using the same arguments. On return, the contents of root's communication buffer has been copied to all processes.
- *MPI_Barrier ( )* blocks until all processes have reached this routine. It is used to synchronize all the processes in a communicator.

3.4 The Design of the Parallization

The development of a parallel algorithm that is to be executed concurrently is a major task. In this research work, the parallel design follows a master-slave algorithm structure as shown in figure 3-2.

When the parallel structure is decided, one needs to decide how to assign the structures to the processes. In order to implement a task in parallel, we should divide the computation and the data into pieces. The functional decomposition method is first

dividing the computation into pieces and then determining how to associate data items with the individual computations. In contrast, the domain decomposition method [33] is first dividing the data into pieces and then determining how to associate computation with the data. We decided to use domain decomposition, a standard method in finite element codes, since the whole structure is progressively divided into smaller and smaller pieces.

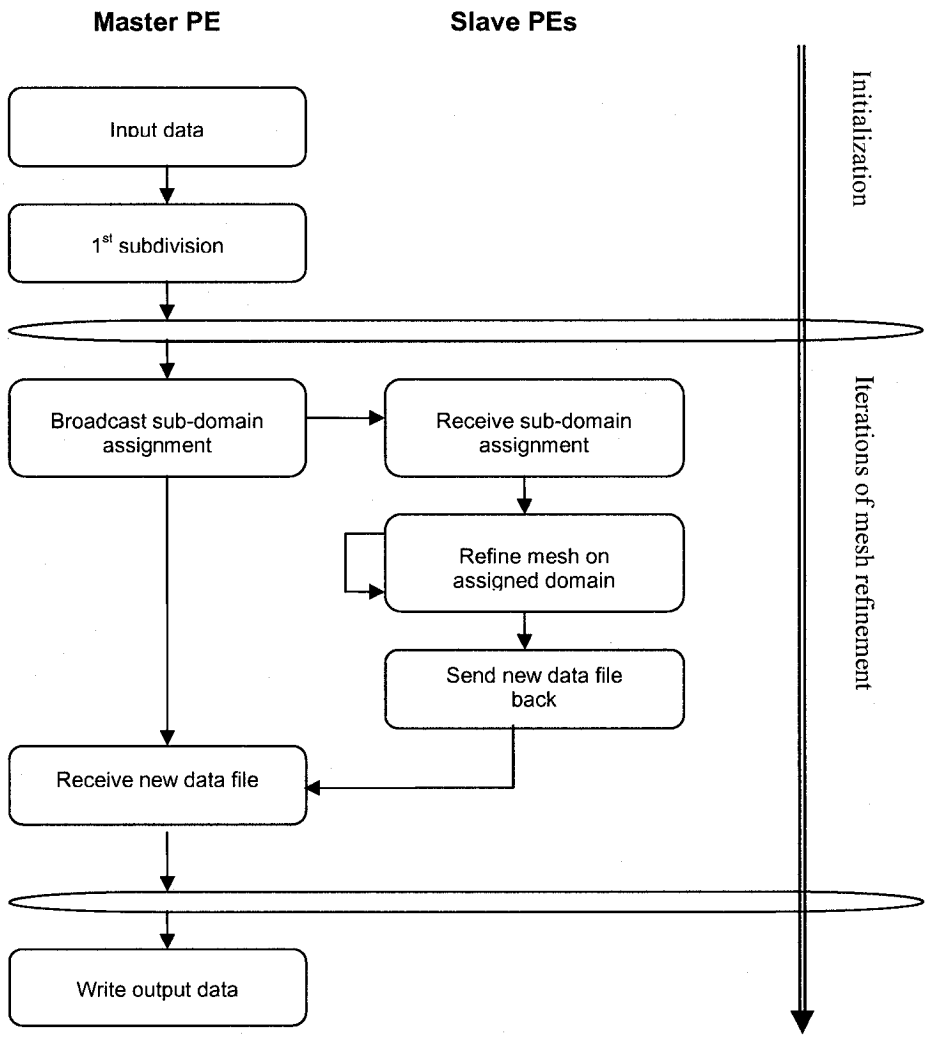

Figure 3-2: The Master-Slave model.

Figure 3-3 below is about the initial domain decomposition for our parallelization.

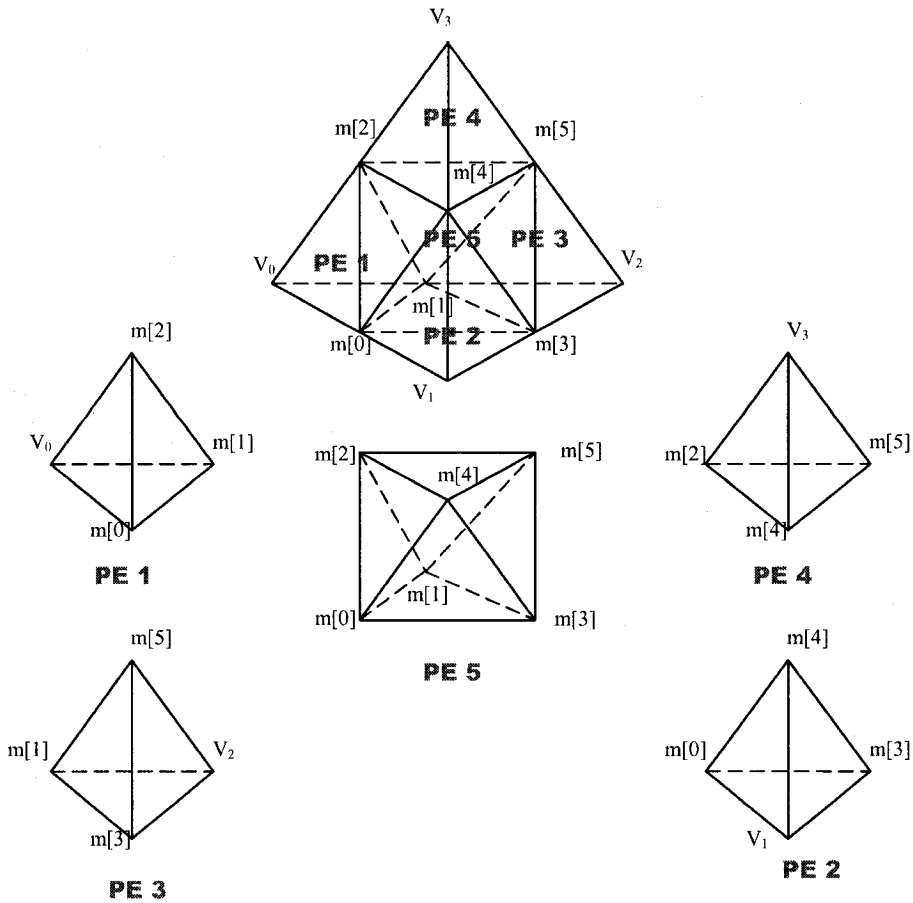

Figure 3-3: Initial domain decomposition.

After we determined a parallel structure model and a method for partitioning, the next step was to determine the communication pattern between processors. We chose MPI because it has been adapted to facilitate inter-processor communications. MPI facilitates master-slave parallel processing, such that all actions performed are broadcast to all the domains when using parallel processing.

As we observed, the master processing element (PE) initiates the program by checking the input data. Then the master PE can assign the initial set into sub-domains. The master PE then broadcasts the complete sub-domain decomposition data and sub-domain assignments to the corresponding slave PEs, which proceed with the mesh refinement of their assigned domains. The master – slave model has the master process receiving from all the slave processes. The master can work out the communication required for all the processes, and then send the required information back to the slave processors, which can act upon it.

If we assume that each element uses up approximately the same amount of calculation-time, then to balance the load we simply need to balance the number of elements per process. In general, a process can be required to send or accept objects from either side, and works out whether to do so or not. Since we have designed the parallelization based on domain decomposition distributing to the slave PEs as shown in figure 3-3, the workload assigned among the slave PE 1 – PE 4 is ideally balanced because the working domain of each PE 1 to 4 is a congruent tetrahedron. The working domain of PE 5 is an octahedron. The table 1 shows the initial load balancing to the slave PEs.

Table 3-1: The initial load distribution.

PE 1	PE 2	PE 3	PE 4	PE 5
V0, m[0], m[1], m[2]	m[0], V1, m[1], m[2]	m[0], m[1], V_2, m[2]	m[0], m[1], m[2], V_3	m[0],m[3],m[4], m[1],m[2],m[5]

Chapter 4

Experiments and Results

This chapter presents experimental setup, the testing methods, and the results of the implementation and then evaluates the design.

4.1 Experimental Setup

The algorithm is implemented using McGill University's CLUMEQ (Consortium Laval UQAM McGill and Eastern Quebec for high performance computing) Supercomputer Centre facilities as shown in figure 4-1 [34]. **Stokes** (compute nodes) is the system where the jobs are submitted [34]. **hn** is the head node that is used to compile programs [34]. See table 4-1 below for additional details:

Table 4-1: Stokes and hn

Stokes	Hn
• Dell PowerEdge 6650	• APPRO-1100
• Dual Xeon 900	• 2AthlonXP 1900+
• 4GB RAM	• 3GB RAM
• Linux RedHat 7.3	• 2*40 GB RAID-0
• NFS server for the nodes	• PBSPro 5.4 server

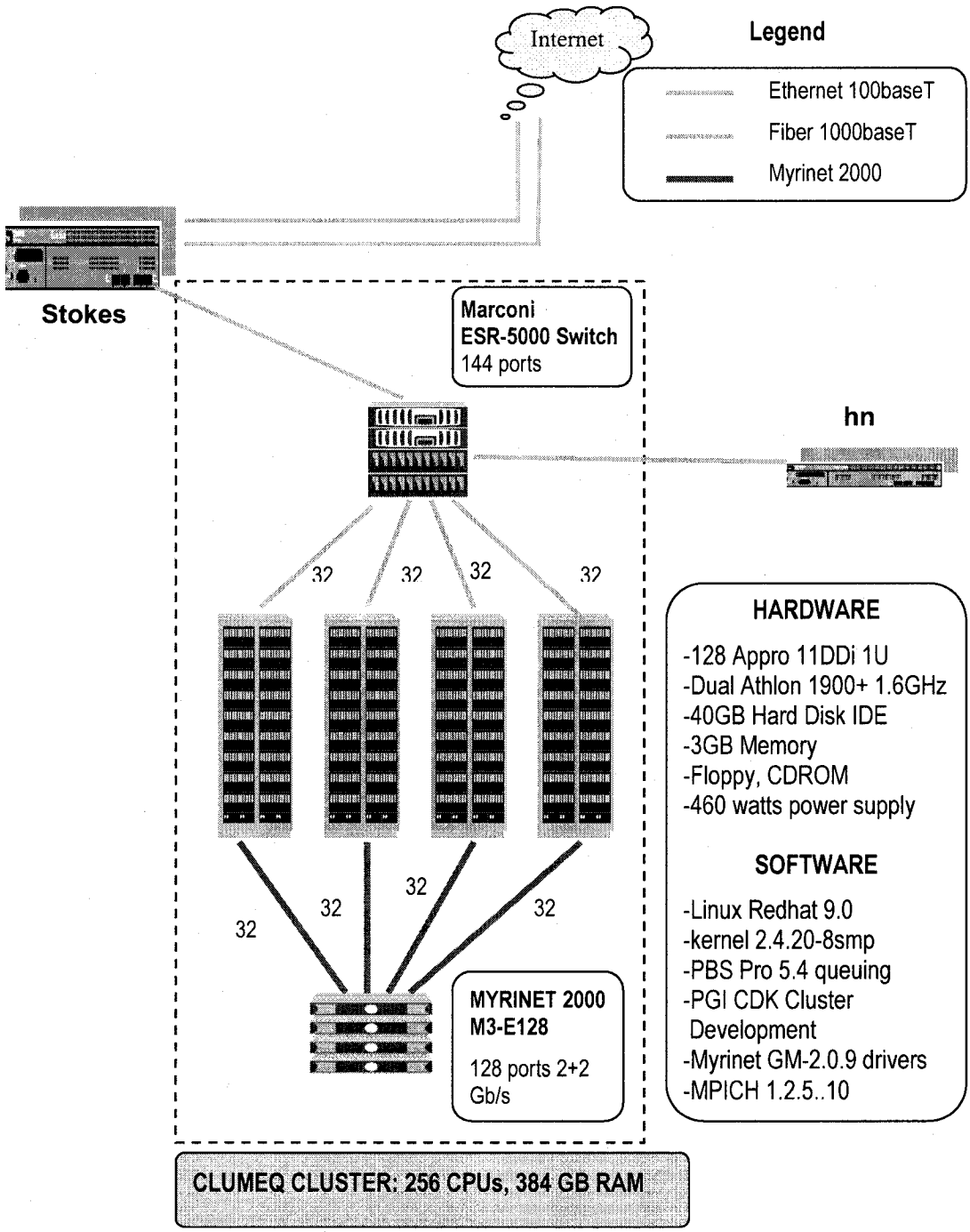

Figure 4-1: CLUMEQ infrastructure [34].

4.2 Measuring the Performance of the Parallel System

The parallel program designed should be properly measured and analyzed, since the performance of a distributed parallel algorithm is influenced by system architecture, system size, and communications delays. Ideally, the performance should increase linearly with the system size. However, in reality performance can degrade with the growth of the system [21, 35, 36].

4.2.1 Speedup Factor

The *speedup factor*, $S(p)$, measures the possible benefits of a parallel performance over a sequential performance, which is defined as [26]:

$$S(p) = \frac{\text{Execution time using single processor system}}{\text{Execution time using a multiprocessor with } p \text{ processors}} \quad (1)$$

If t_s is used as the execution time of the best sequential algorithm running on a single processor and t_p is used as the execution time for solving the same problem on a multiprocessor, then: $S(p) = \frac{t_s}{t_p}$ (2)

4.2.2 Efficiency

Efficiency is measured by calculating how long processors are actually being used for the computation. The efficiency can be defined as follows [26]:

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}} \quad (3)$$

Similarly, if t_s is also used as the execution time of the best sequential algorithm running on a single processor and t_p is used as the execution time for solving the same problem on a multiprocessor, then: $E = \frac{t_s}{t_p \times p}$ (4)

It also can be written as $E = \frac{S(p)}{p} \times 100\%$ when E is given as a percentage. (5)

For instance, if E=50%, the processors are being used for only half of the time during the actual computation, on average. An efficiency of 100% occurs where all the processors are being used on the computation at all times, i.e., the speedup factor, $S(p)$ is p.

4.3 Overview of Implementation

This section describes the implementation of the software described in the previous sections. Prior to the development of the parallel version, a sequential version was written using C code. We used the MPI message-passing library and added it to the sequential version to achieve parallelism. The parallel regular refinement algorithm was designed based on a master – slave structure. The master PE assumes the role to orchestrate the entire set slave PEs. It is responsible for establishing the entire model for analysis and then distributing data among the slave PEs. The master PE initializes the vertices of the initial tetrahedron as shown in figure 4-2 and assigns data to each slave processor. The slave PEs will be waiting for commands until the master PE issues one and then they do the corresponding work on their own copy of the data.

```
if(mype==0)
{
        /* Initializing the vertexes of the initial tetrahedron */
        v[0][0]=0; v[0][1]=0; v[0][2]=0;
        v[1][0]=0; v[1][1]=1; v[1][2]=0;
        v[2][0]=0; v[2][1]=0; v[2][2]=1;
        v[3][0]=1; v[3][1]=0; v[3][2]=0;

        requiredDepth=5;
        middle(v[0], v[1], 0, m[0]);
        middle(v[0], v[2], 0, m[1]);
        middle(v[0], v[3], 0, m[2]);
        middle(v[1], v[2], 0, m[3]);
        middle(v[1], v[3], 0, m[4]);
        middle(v[2], v[3], 0, m[5]);
}
```

Figure 4-2: The master PE initializes the vertices of the initial tetrahedron.

We chose to implement parallelism, using a Single Program, Multiple Data (SPMD) approach. In this approach data is distributed and each slave processor independently processes its corresponding data. The main functions implemented are as follows:

The function **middle( ),** which provides the midpoints of all the edges of the initial tetrahedron, was implemented as shown in figure 4-3 [37-40].

```
int middle (float v0[3], float v1[3], int isCount, float* out)
{
        int i;
        if (isCount)
                vertexNumber++;
        for (i = 0; i < 3; i++)
        {
                out[i] = (v0[i] + v1[i])/2;
```

```
            if (isCount)
                v[vertexNumber][i] = out[i];
        }
        return vertexNumber;
}
```

Figure 4-3: The function middle( ).

The **MPI_Bcast ( )** is used to broadcast a message from the process with rank root to all processes of the group. The **MPI_Barrier ( )** is used to synchronize all the processes in a communicator. Each slave process waits to be assigned [19, 28-30, 41, 42]. See figure 4-4 below for additional details:

```
/* MPI BCAST */
MPI_Bcast (m, 18, MPI_FLOAT, 0, MPI_COMM_WORLD);

/* printf("1st Bcast done\n"); */
MPI_Bcast ((void *) &requiredDepth,1,MPI_INT,0, MPI_COMM_WORLD);

/* printf("2nd Bcast done\n"); */
MPI_Bcast (v, 30, MPI_FLOAT,0,MPI_COMM_WORLD);

/* printf("3rd Bcast done\n"); */

/* need to check to see if the proper values for v's were sent */

MPI_Barrier (MPI_COMM_WORLD);

        if(mype==1)
        {
                v[1][0]=m[0][0], v[1][1]=m[0][1], v[1][2]=m[0][2];
                v[2][0]=m[1][0], v[2][1]=m[1][1], v[2][2]=m[1][2];
                v[3][0]=m[2][0], v[3][1]=m[2][1], v[3][2]=m[2][2];
                processTetra0(v[0], m[0], m[1], m[2], 1, requiredDepth);
        }

        if(mype==2)
        {
                v[0][0]=m[0][0]; v[0][1]=m[0][1]; v[0][2]=m[0][2];
                v[2][0]=m[3][0]; v[2][1]=m[3][1]; v[2][2]=m[3][2];
```

```
        v[3][0]=m[4][0]; v[3][1]=m[4][1]; v[3][2]=m[4][2];
        processTetra0(m[0], v[1], m[3], m[4], 1, requiredDepth);
}

if(mype==3)
{
        v[0][0]=m[1][0]; v[0][1]=m[1][1]; v[0][2]=m[1][2];
        v[1][0]=m[3][0]; v[1][1]=m[3][1]; v[1][2]=m[3][2];
        v[3][0]=m[5][0]; v[3][1]=m[5][1]; v[3][2]=m[5][2];
        processTetra0(m[1], m[3], v[2], m[5], 1, requiredDepth);
}

if(mype==4)
{
        v[0][0]=m[2][0]; v[0][1]=m[2][1]; v[0][2]=m[2][2];
        v[1][0]=m[4][0]; v[1][1]=m[4][1]; v[1][2]=m[4][2];
        v[2][0]=m[5][0]; v[2][1]=m[5][1]; v[2][2]=m[5][2];
        processTetra0(m[2], m[4], m[5], v[3], 1, requiredDepth);
}

if(mype==5)
{
        middle(v[0], v[1], 1, m[0]);
        middle(v[0], v[2], 1, m[1]);
        middle(v[0], v[3], 1, m[2]);
        middle(v[1], v[2], 1, m[3]);
        middle(v[1], v[3], 1, m[4]);
        middle(v[2], v[3], 1, m[5]);
        processOcta(m[0], m[3], m[4], m[1], m[2], m[5], 1, requiredDepth);
}
```

Figure 4-4: The MPI_Bcast ( ) and the MPI_Barrier ( ).

As the slave PEs are assigned, the four tetrahedra and the octahedron are refined regularly in a partition (i.e. mesh refinement). The function **processTetra0 ( ), processTetra1 ( ), processTetra2 ( ),** and **processOcta ( )** were implemented for the mesh refinement. This implies that no communication with neighbouring partitions takes place. See figures 4-5, 4-6, 4-7, and 4-8 below for additional details:

```
void processTetra0 (float v0[3], float v1[3], float v2[3], float v3[3], int depth)
{
        int indepth = depth + 1;
        float m[6][3];

        middle(v0, v1, 1, m[0]);
        middle(v0, v2, 1, m[1]);
        middle(v0, v3, 1, m[2]);
        middle(v1, v2, 1, m[3]);
        middle(v1, v3, 1, m[4]);
        middle(v2, v3, 1, m[5]);

        if (indepth >= requiredDepth)
        {
                add_face(v0, m[0], m[1]);
                add_face(v0, m[0], m[2]);
                add_face(v0, m[1], m[2]);
                add_face(m[0], m[1], m[2]);

                add_face(v1, m[0], m[3]);
                add_face(v1, m[0], m[4]);
                add_face(v1, m[3], m[4]);
                add_face(m[0], m[3], m[4]);

                add_face(v2, m[1], m[3]);
                add_face(v2, m[1], m[5]);
                add_face(v2, m[3], m[5]);
                add_face(m[1], m[3], m[5]);

                add_face(v3, m[2], m[4]);
                add_face(v3, m[2], m[5]);
                add_face(v3, m[4], m[5]);
                add_face(m[2], m[4], m[5]);

                add_face(m[0], m[1], m[3]);
                add_face(m[0], m[2], m[4]);
                add_face(m[1], m[2], m[5]);
                add_face(m[3], m[4], m[5]);

                return;
        }
        processTetra0(v0, m[0], m[1], m[2], indepth);
        processTetra0(m[0], v1, m[3], m[4], indepth);
        processTetra0(m[1], m[3], v2, m[5], indepth);
        processTetra0(m[2], m[4], m[5], v3, indepth);
        processOcta(m[0], m[3], m[4], m[1], m[2], m[5], indepth);
}
```

Figure 4-5: The function processTetra0 ( ).

```
void processTetra1 (float v0[3], float v1[3], float v2[3], float v3[3], int depth)
{
        int indepth = depth + 1;
        float m[6][3];

        middle(v0, v1, 0, m[0]);
        middle(v0, v2, 1, m[1]);
        middle(v0, v3, 1, m[2]);
        middle(v1, v2, 1, m[3]);
        middle(v1, v3, 1, m[4]);
        middle(v2, v3, 1, m[5]);

        if (indepth >= requiredDepth)
        {
                add_face(v0, m[0], m[1]);
                add_face(v0, m[0], m[2]);
                add_face(v0, m[1], m[2]);
                add_face(m[0], m[1], m[2]);

                add_face(v1, m[0], m[3]);
                add_face(v1, m[0], m[4]);
                add_face(v1, m[3], m[4]);
                add_face(m[0], m[3], m[4]);

                add_face(v2, m[1], m[3]);
                add_face(v2, m[1], m[5]);
                add_face(v2, m[3], m[5]);
                add_face(m[1], m[3], m[5]);

                add_face(v3, m[2], m[4]);
                add_face(v3, m[2], m[5]);
                add_face(v3, m[4], m[5]);
                add_face(m[2], m[4], m[5]);

                add_face(m[0], m[1], m[3]);
                add_face(m[0], m[2], m[4]);
                add_face(m[1], m[2], m[5]);
                add_face(m[3], m[4], m[5]);
                return;
        }
        processTetra1(v0, m[0], m[1], m[2], indepth);
        processTetra1(m[0], v1, m[3], m[4], indepth);
        processTetra0(m[1], m[3], v2, m[5], indepth);
        processTetra0(m[2], m[4], m[5], v3, indepth);
        processOcta(m[0], m[3], m[4], m[1], m[2], m[5], indepth);
}
```

Figure 4-6: The function processTetra1 ( ).

```
void processTetra2 (float v0[3], float v1[3], float v2[3], float v3[3], int depth)
{
        int indepth = depth + 1;
        float m[6][3];

        middle(v0, v1, 0, m[0]);
        middle(v0, v2, 0, m[1]);
        middle(v0, v3, 0, m[2]);
        middle(v1, v2, 1, m[3]);
        middle(v1, v3, 1, m[4]);
        middle(v2, v3, 1, m[5]);

        if (indepth >= requiredDepth)
        {
                add_face(v0, m[0], m[1]);
                add_face(v0, m[0], m[2]);
                add_face(v0, m[1], m[2]);
                add_face(m[0], m[1], m[2]);

                add_face(v1, m[0], m[3]);
                add_face(v1, m[0], m[4]);
                add_face(v1, m[3], m[4]);
                add_face(m[0], m[3], m[4]);

                add_face(v2, m[1], m[3]);
                add_face(v2, m[1], m[5]);
                add_face(v2, m[3], m[5]);
                add_face(m[1], m[3], m[5]);

                add_face(v3, m[2], m[4]);
                add_face(v3, m[2], m[5]);
                add_face(v3, m[4], m[5]);
                add_face(m[2], m[4], m[5]);

                add_face(m[0], m[1], m[3]);
                add_face(m[0], m[2], m[4]);
                add_face(m[1], m[2], m[5]);
                add_face(m[3], m[4], m[5]);

                return;
        }
        processTetra2(v0, m[0], m[1], m[2], indepth);
        processTetra1(m[0], v1, m[3], m[4], indepth);
        processTetra1(v2, m[1], m[3], m[5], indepth);
        processTetra1(m[2], v3, m[4], m[5], indepth);
        processOcta(m[0], m[3], m[4], m[1], m[2], m[5], indepth);
}
```

Figure 4-7: The function processTetra2 ( ).

```
void processOcta (float v0[3], float v1[3], float v2[3], float v3[3], float v4[3], float v5[3],
int depth)
{
        int indepth = depth + 1;
        float m[13][3];

        middle(v0, v1, 0, m[0]);
        middle(v0, v2, 0, m[1]);
        middle(v0, v3, 0, m[2]);
        middle(v0, v4, 0, m[3]);
        middle(v1, v2, 0, m[4]);
        middle(v1, v3, 0, m[5]);
        middle(v1, v5, 0, m[6]);
        middle(v2, v4, 0, m[7]);
        middle(v2, v5, 0, m[8]);
        middle(v3, v4, 0, m[9]);
        middle(v3, v5, 0, m[10]);
        middle(v4, v5, 0, m[11]);
        middle(v2, v3, 1, m[12]);

        if (indepth >= requiredDepth)
        {
                add_face(m[1], m[3], m[7]);
                add_face(m[1], m[3], m[12]);
                add_face(m[1], m[7], m[12]);
                add_face(m[3], m[7], m[12]);

                add_face(m[2], m[3], m[12]);
                add_face(m[2], m[9], m[12]);
                add_face(m[3], m[9], m[12]);

                add_face(m[0], m[2], m[5]);
                add_face(m[0], m[2], m[12]);
                add_face(m[0], m[5], m[12]);
                add_face(m[2], m[5], m[12]);

                add_face(m[0], m[1], m[12]);
                add_face(m[0], m[4], m[12]);
                add_face(m[1], m[4], m[12]);

                add_face(m[4], m[6], m[8]);
                add_face(m[4], m[6], m[12]);
                add_face(m[4], m[8], m[12]);
                add_face(m[6], m[8], m[12]);

                add_face(m[5], m[6], m[12]);
                add_face(m[5], m[10], m[12]);
                add_face(m[6], m[10], m[12]);

                add_face(m[9], m[10], m[11]);
                add_face(m[9], m[10], m[12]);
```

```
                add_face(m[9], m[11], m[12]);
                add_face(m[10], m[11], m[12]);

                add_face(m[7], m[8], m[12]);
                add_face(m[7], m[11], m[12]);
                add_face(m[8], m[11], m[12]);

                add_face(v0, m[0], m[2]);
                add_face(v0, m[1], m[3]);
                add_face(v1, m[0], m[5]);
                add_face(v1, m[4], m[6]);
                add_face(v2, m[1], m[7]);
                add_face(v2, m[4], m[8]);
                add_face(v3, m[2], m[5]);
                add_face(v3, m[9], m[10]);
                add_face(v4, m[3], m[7]);
                add_face(v4, m[9], m[11]);
                add_face(v5, m[6], m[8]);
                add_face(v5, m[10], m[11]);

                return;
        }
        processTetra2(m[12], m[7], m[3], m[1], indepth);
        processTetra0(m[3], m[2], m[12], m[9], indepth);
        processTetra2(m[12], m[0], m[2], m[5], indepth);
        processTetra0(m[0], m[4], m[12], m[1], indepth);
        processTetra2(m[12], m[4], m[6], m[8], indepth);
        processTetra0(m[6], m[10], m[12], m[5], indepth);
        processTetra2(m[12], m[10], m[9], m[11], indepth);
        processTetra0(m[11], m[8], m[12], m[7], indepth);
        processOcta(v0, m[0], m[1], m[2], m[3], m[12], indepth);
        processOcta(m[0], v1, m[4], m[5], m[12], m[6], indepth);
        processOcta(m[1], m[4], v2, m[12], m[7], m[8], indepth);
        processOcta(m[2], m[5], m[12], v3, m[9], m[10], indepth);
        processOcta(m[11], v4, m[7], m[9], m[12], m[3], indepth);
        processOcta(v5, m[11], m[8], m[10], m[6], m[12], indepth);
}
```

Figure 4-8: The function processOcta ( ).

The master PE gathers information about locally refined nodes (vertices) and faces. Initially, it retrieves from the partition the number of tetrahedra and octahedra assigned for regular refinement. The function **copy_vertex ( ), add_face ( ),**

find_vertex ( ) were implemented in order to obtain the information for the geometric computing. See figures 4-9, 4-10, and 4-11 below for additional details:

```
void copy_vertex (float v0[3], float* out)
{
        int i;

        for (i = 0; i < 3; i++)
        {
                out[i] = v0[i];
        }
}
```

Figure 4-9: The function copy_vertex ( ).

```
void add_face (float v0[3], float v1[3], float v2[3])
{
        copy_vertex(v0, f[faceNumber][0]);
        copy_vertex(v1, f[faceNumber][1]);
        copy_vertex(v2, f[faceNumber][2]);
        faceNumber++;
}
```

Figure 4-10: The function add_face ( ).

```
long find_vertex (float v0[3])
{
        long i, j = 0;
        for (i = 0; i <= vertexNumber; i++)
        {
                if (v[i][0] == v0[0])
                {
                        if (v[i][1] == v0[1])
                        {
                                if (v[i][2] == v0[2])
                                {
                                        return i;
                                }
                        }
                }
```

```
        }
        return -1;
}
```

Figure 4-11: The function find_vertex ( ).

The **MPI_Gather( )** was also implemented as shown in figure 4-12 for the information gathering together from the group of processes.

```
 MPI_Gather (&vertexNumber, 1, MPI_INT, vtx, 1, MPI_INT, 0,
MPI_COMM_WORLD);
 MPI_Gather (&faceNumber, 1, MPI_INT, face, 1, MPI_INT, 0,
MPI_COMM_WORLD);

        for (i = 0; i < faceNumber; i++)
        {
                facelist[i][0]=find_vertex(f[i][0]);
                facelist[i][1]=find_vertex(f[i][1]);
                facelist[i][2]=find_vertex(f[i][2]);
        }

        /* building the facelist for gathering operation slave nodes */

        if(mype==0)
        {

                for(i=0; i<6; i++)
                {
                        vtx[i]=3*vtx[i];
                                face[i]=3*face[i];
                        }

                sum=0;
        facesum=0;

        for(i=0; i<6; i++)
        {
                sum=sum+vtx[i];
                        facesum=facesum+face[i];
\        }


        recvbuf = (float *) malloc(sizeof(float)*sum);
```

```
face_recvbuf = (int *) malloc(sizeof(int)*facesum);

if(recvbuf==NULL)
        printf("Couldn't allocate memory to recvbuf!!\n");


disp[0]=0;

for(i=1;i<6;i++)
{
        disp[i]=disp[i-1]+vtx[i-1];
}

face_disp[0]=0;

for(i=1;i<6;i++)
{
                face_disp[i]=face_disp[i-1]+face[i-1];
} /* Process 0 */
```

Figure 4-12: The MPI_Gather ( ).

The gathered information must be stored into specified locations from all processes in the group. Finally, The **MPI_Gatherv( )** was implemented for this purpose as shown in figure 4-13.

```
MPI_Gatherv ( v, 3*vertexNumber, MPI_FLOAT, recvbuf, vtx, disp,
                        MPI_FLOAT,0, MPI_COMM_WORLD);
MPI_Gatherv ( facelist, 3*faceNumber, MPI_INT,face_recvbuf, face,
                face_disp, MPI_INT,0, MPI_COMM_WORLD);

        endtime=MPI_Wtime();

        printf("Total Time: Process %d Time:%f\n", mype, endtime-starttime);
        printf("Gather Time: Process %d Time:%f\n", mype, endtime-time2);
        printf("Refinement Time: Process %d Time:%f\n", mype, time1-starttime);
        printf("Information Gathering Time: Process %d Time:%f\n", mype, time2-
time1);


        if(mype==0)
        {
```

```
            sprintf(resultfile, "Final_gathered.off");

            if( (fp=fopen(resultfile,"w"))==NULL)
                    printf("Coudn't open %s proces %d \n", resultfile, mype);

            fprintf(fp, "OFF\n");
                    fprintf(fp, "%d %d %d\n", sum/3, facesum/3, faceNumber);

            for (i = 0; i < sum; i=i+3)
            {
                    fprintf(fp, "%f %f %f\n", recvbuf[i], recvbuf[i+1], recvbuf[i+2]);
            }

            fprintf(fp, "\n");


                    start=0;
                    end=0;
                    offset=0;

                    for(j=1;j<6;j++)
            {
                    start=start+face[j-1];
                    end=start+face[j];
                    offset=offset+(int)vtx[j-1]/3;

              for (i = start; i < end; i=i+3)
            {
                    fprintf(fp, "3 %d %d %d\n", face_recvbuf[i]+offset,
                    face_recvbuf[i+1]+offset, face_recvbuf[i+2]+offset);
            }
    }

    fprintf(fp, "\n");

    fclose(fp);
    free(face_recvbuf);
    free(recvbuf);
```

Figure 4-13: The MPI_Gatherv ( ).

4.4 Results and Evaluation

In our experiments, we first compared the execution time of the sequential computing with that of the parallel computing. For the parallel computing we used 6 PEs. Then we observed the timing cost distribution of each PE. Lastly, the timing cost of mesh refinement was evaluated.

4.4.1 Sequential vs. Parallel execution time

Experimental results and system performance are presented, comparing sequential computing with parallel computing using 6 PEs. The results are summarized in table 4-2 and related graphs are shown in figures 4-14, 4-15, and 4-16.

Table 4-2: Timing results for sequential vs. parallel time cost.

Subdivisions (Iterations)	No. of Elements	Sequential Time (S)	Parallel Time (S)	Speedup Comparison	Efficiency (%)
2	**64**	N/A	0.000632	N/A	N/A
3	**512**	0.006	0.001686	3.559	59.312
4	**4096**	0.17	0.058802	2.891	48.184
5	**32768**	8.95	3.743384	2.390	39.848
6	**262144**	1181.47	569.67	2.074	34.565

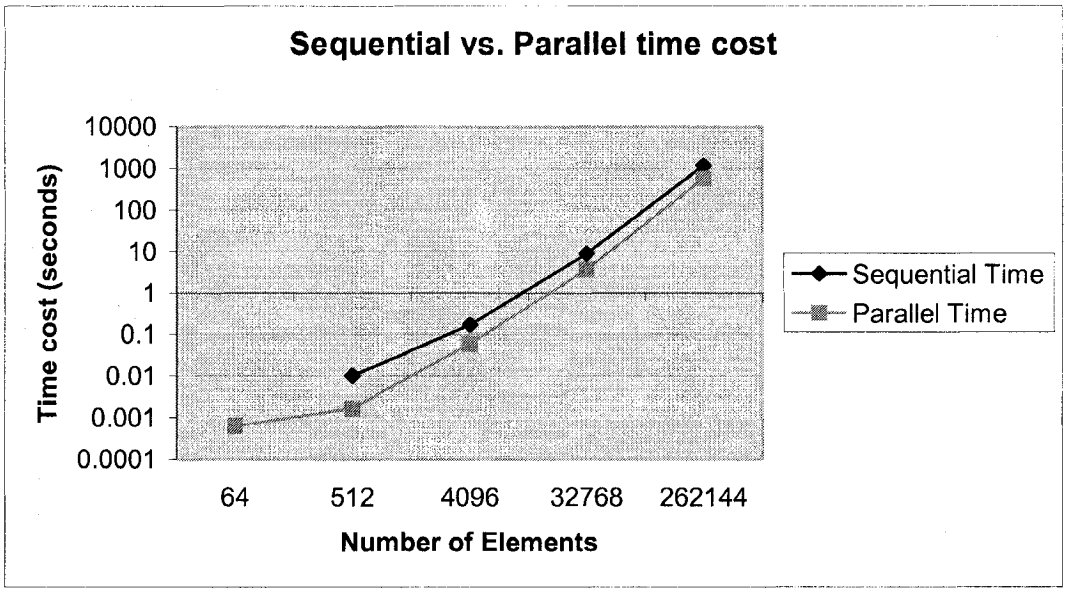

Figure 4-14: Sequential vs. Parallel execution time cost using 6 PEs.

Figure 4-14 shows that in general the parallel execution time is faster then the sequential execution time.

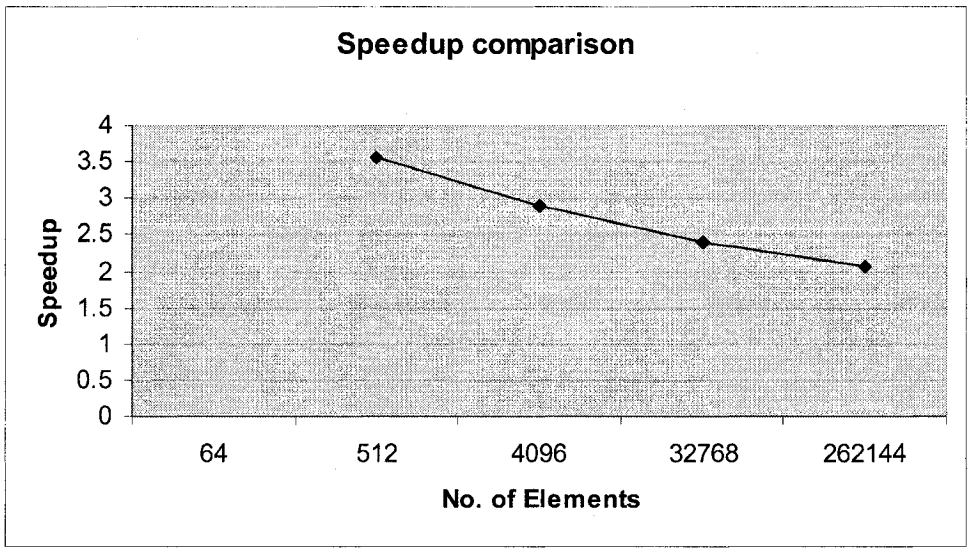

Figure 4-15: Speedup comparison.

From the speedup comparison graph in figure 4-15, it is clear that the speedup factor (S_p) slowly goes down as the number of elements increases.

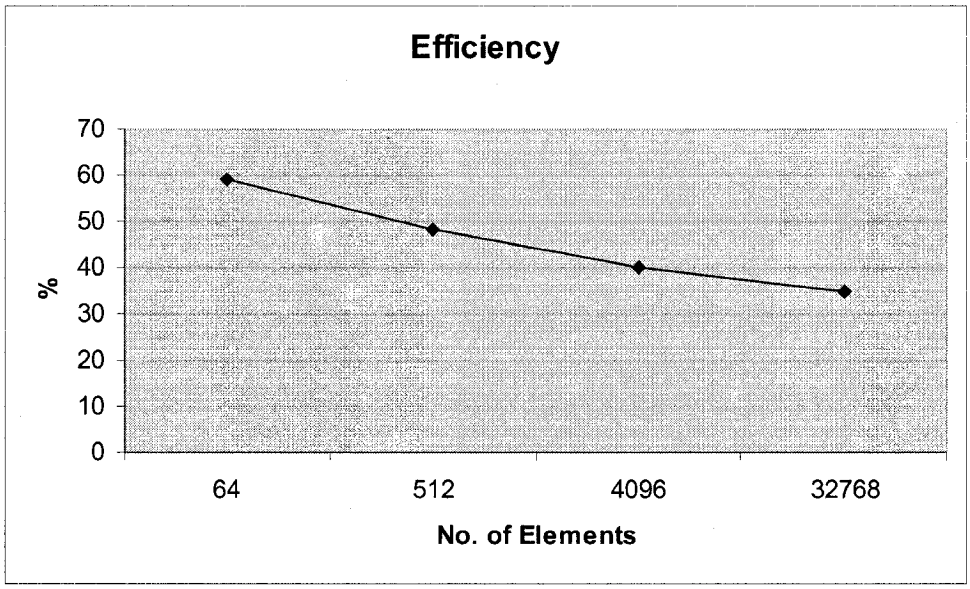

Figure 4-16: Efficiency.

The efficiency graph in figure 4-16 shows clearly that the efficiency decreases as the number of elements increases. The performance above is caused by the significant overload of communication between PEs as the number of elements increases (details are presented in the section 4.4.2).

4.4.2 Time cost distribution of each PE

Experimental results and time cost distribution of each PE are presented. The concept of measuring time cost distribution is shown in figure 4-17.

- **t_1 (Mesh refinement computation time)**: for mesh refinement computation
- **t_2 (Pre-processing time)**: for gathering information to the pre-processor for the MPI_Gather
- **t_3 (Data gathering time)**: for data gathering into specified locations from all processes in a group
- **Overall time cost** = $t_1 + t_2 + t_3$

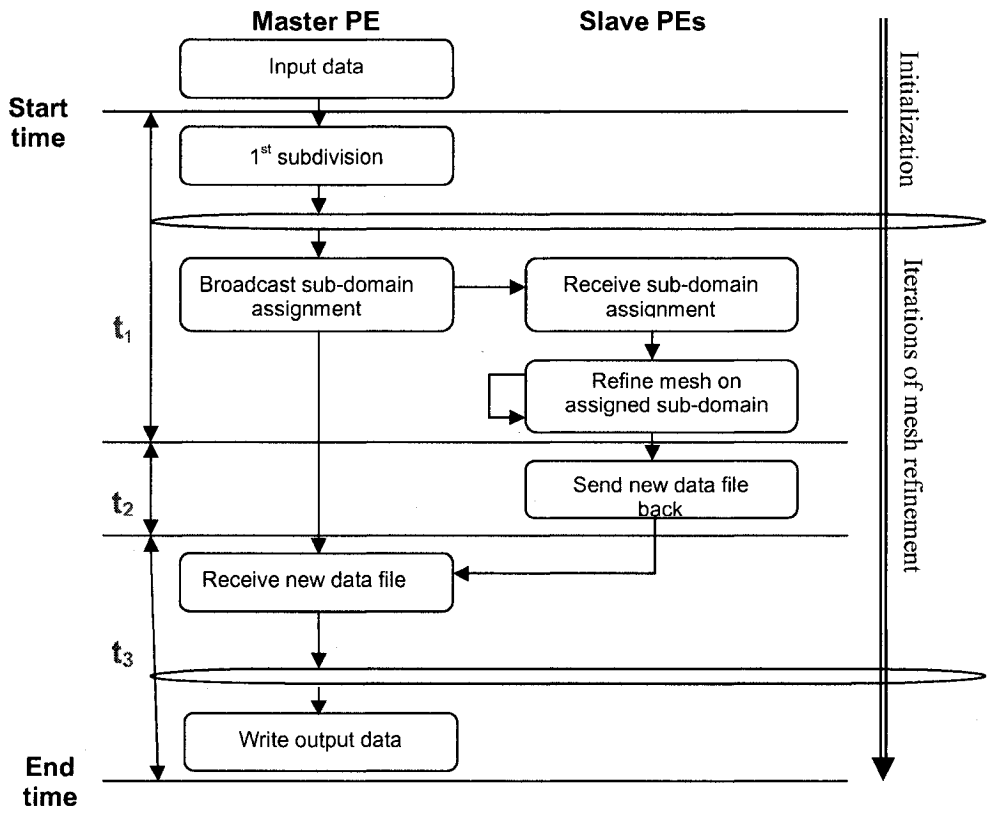

Figure 4-17: Concept of measuring time cost.

Table 4-3 and figure 4-18 below give the distribution time cost results of each PE for the 2nd subdivision.

Table 4-3: Time cost (seconds) results of each PE: 2nd subdivision.

Distribution	ID of Each PE					
	PE 0	**PE 1**	**PE 2**	**PE 3**	**PE 4**	**PE 5**
Mesh refinement	0.000261	0.000375	0.000297	0.000385	0.000174	0.000180
Pre-processing	0.000238	0.000244	0.000237	0.000235	0.000239	0.000231
Data gathering	0.000108	0.000006	0.000006	0.000007	0.000007	0.000008
Overall	0.000607	0.000625	0.000540	0.000627	0.000420	0.000419

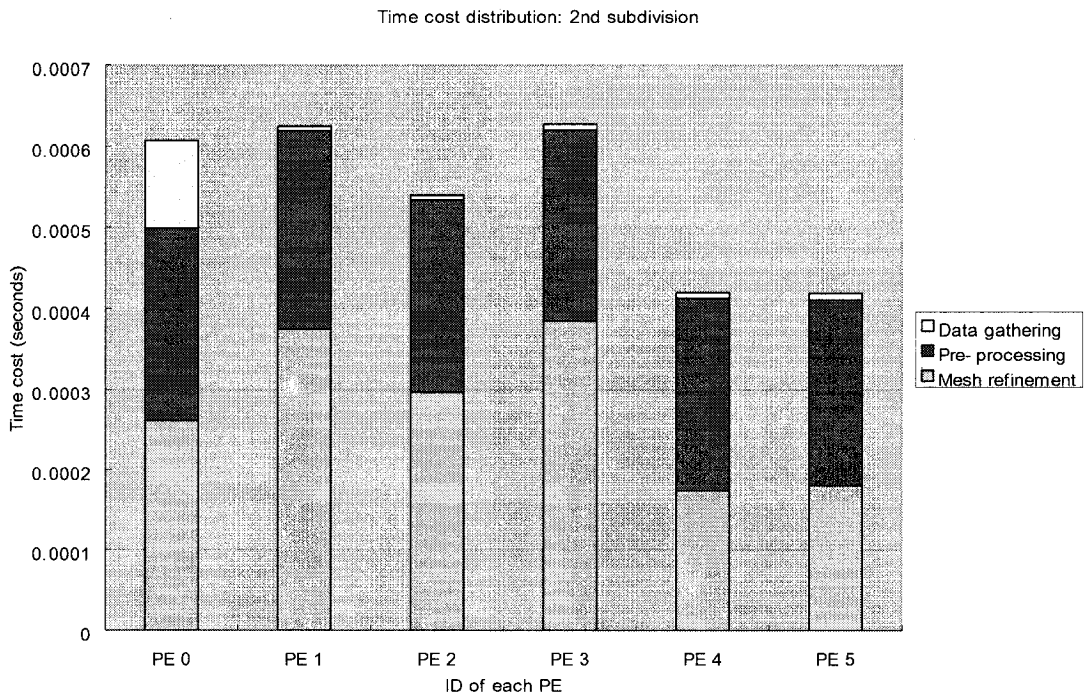

Figure 4-18: Time cost distribution: 2nd subdivision.

Table 4-4 and figure 4-19 below give the distribution time cost results of each PE for the 3rd subdivision.

Table 4-4: Time cost (seconds) results of each PE: 3rd subdivision.

Distribution	ID of Each PE					
	PE 0	**PE 1**	**PE 2**	**PE 3**	**PE 4**	**PE 5**
Mesh refinement	0.000283	0.000303	0.000313	0.000313	0.000196	0.000256
Pre-processing	0.00121	0.001176	0.001178	0.001167	0.001174	0.001118
Data gathering	0.000193	0.000013	0.000012	0.000015	0.000011	0.000023
Overall	0.001686	0.001492	0.001503	0.001495	0.001381	0.001397

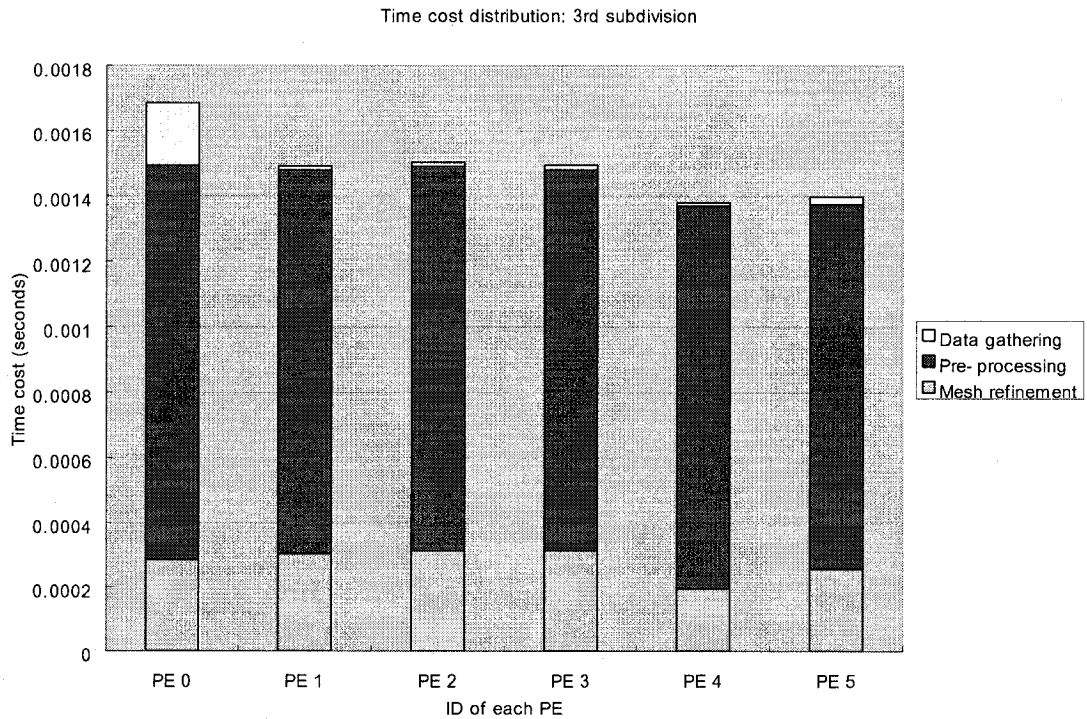

Figure 4-19: Time cost distribution: 3rd subdivision.

Table 4-5 and figure 4-20 below give the distribution time cost results of each PE for the 4th subdivision.

Table 4-5: Time cost (seconds) results of each PE: 4th subdivision.

Distribution	ID of Each PE					
	PE 0	**PE 1**	**PE 2**	**PE 3**	**PE 4**	**PE 5**
Mesh refinement	0.000265	0.000668	0.000641	0.000652	0.000384	0.001029
Pre-processing	0.056878	0.056661	0.056661	0.056688	0.056687	0.056138
Data gathering	0.001659	0.00004	0.000041	0.000057	0.000049	0.001628
Overall	0.058802	0.057369	0.057316	0.057397	0.05712	0.058795

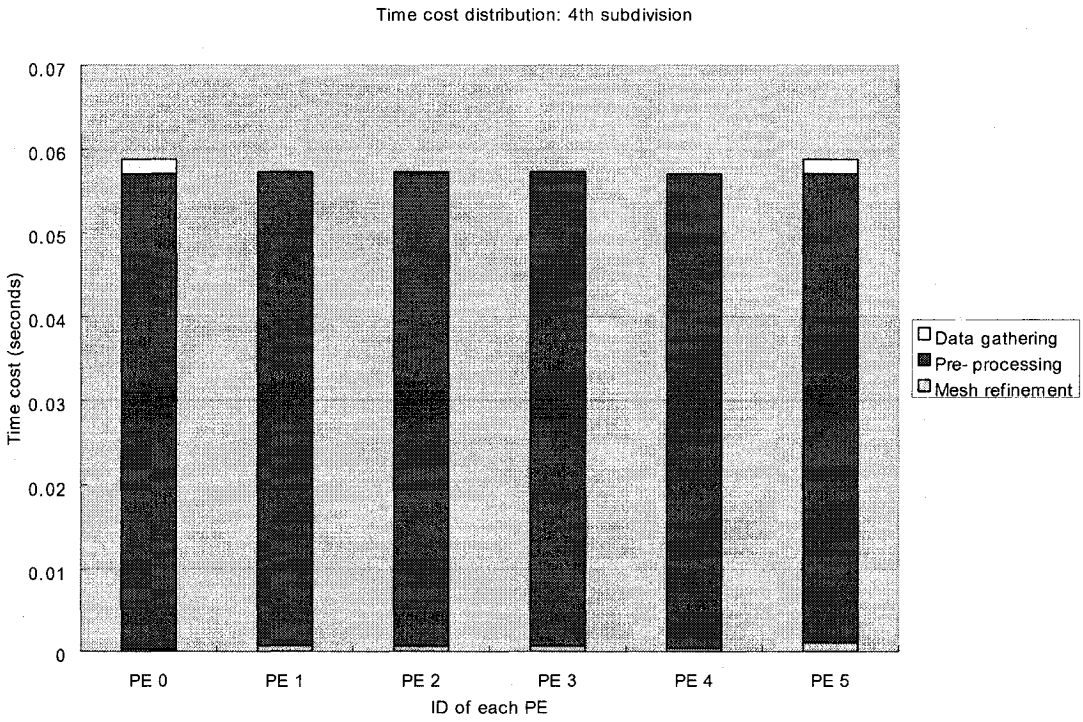

Figure 4-20: Time cost distribution: 4th subdivision.

Table 4-6 and figure 4-21 below give the distribution time cost results of each PE for the 5th subdivision.

Table 4-6: Time cost (seconds) results of each PE: 5th subdivision.

Distribution	ID of Each PE					
	PE 0	**PE 1**	**PE 2**	**PE 3**	**PE 4**	**PE 5**
Mesh refinement	0.000356	0.001915	0.002078	0.001791	0.00195	0.006032
Pre-processing	3.731448	3.729679	3.729704	3.72996	3.72972	3.72546
Data gathering	0.001158	0.003769	0.005299	0.00586	0.007111	0.011593
Overall	3.743384	3.735363	3.737081	3.737611	3.738781	3.743085

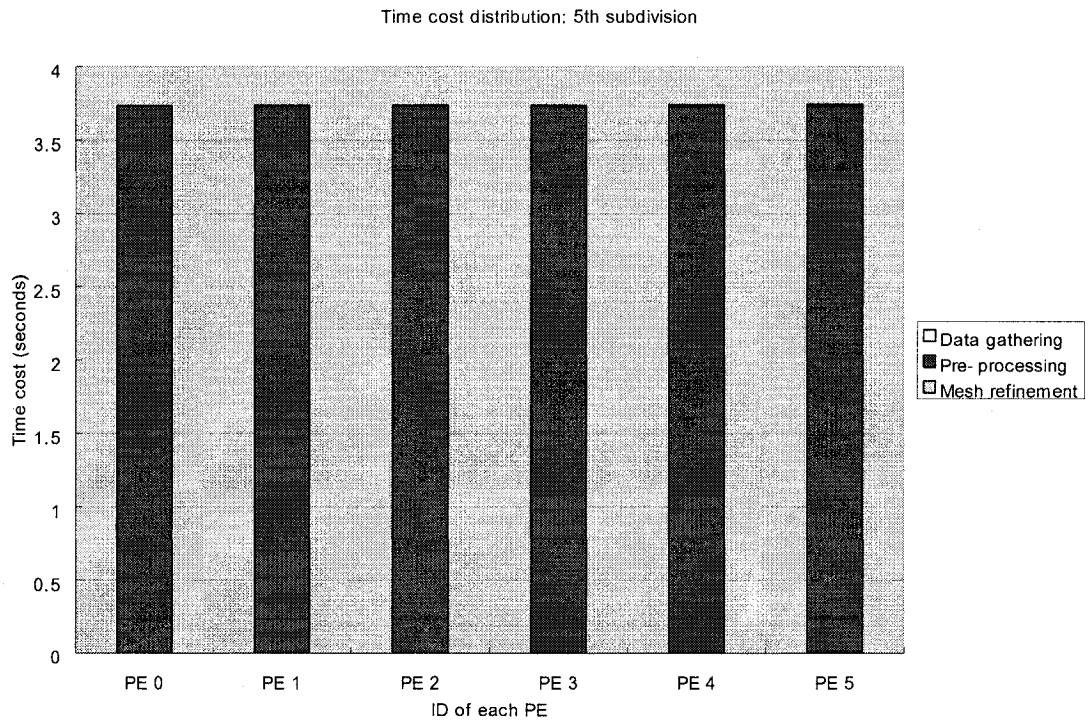

Figure 4-21: Time cost distribution: 5th subdivision.

Table 4-7 and figure 4-22 below give the distribution time cost results of each PE for the 6^{th} subdivision.

Table 4-7: Time cost (seconds) results of each PE: 6^{th} subdivision.

Distribution	ID of Each PE					
	PE 0	**PE 1**	**PE 2**	**PE 3**	**PE 4**	**PE 5**
Mesh refinement	0.000281	0.014	0.014	0.012	0.014	0.048
Pre-processing	569.569	569.603	569.603	569.605	569.603	569.569
Data gathering	0.053	0.02	0.026	0.029	0.036	0.0533
Overall	569.67	569.638	569.643	569.647	569.653	569.67

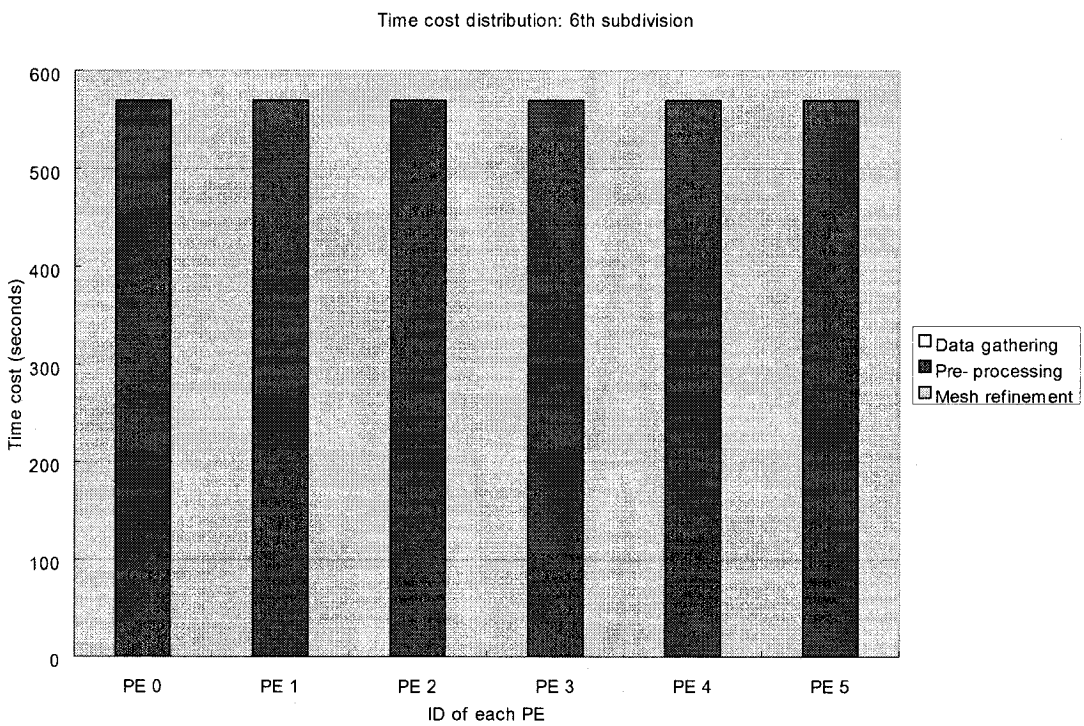

Figure 4-22: Time cost distribution: 6^{th} subdivision.

Figure 4-18 to 4-22 show the time cost distribution comparison of each PE. These figures show that overall the workload is well balanced. These also clearly show that the more the elements increase, the more the time ratio of pre-processing (information gathering; communication cost) increases. The communication cost is significant due to overload for communication between PEs as the number of elements increases. It is clear that the delay between times, when the first and the last contacted processor is able to start, grows with the number of elements. This negatively influences the workload balance. Therefore, the performance of the parallel algorithm drops. In order to reduce the communication cost, a pipelined communication strategy must be designed for our mesh refinement scheme. However, the design seems to be quite efficient in general, even though the mesh refinement algorithm does require a lot of communication.

4.4.3 Time Cost of Mesh Refinement

Mesh refinement is the essential part of hierarchical tetrahedral – octahedral (HTO) subdivision. From table 4-3 to table 4-7, we compare timing results of mesh refinement for each PE and observe the balance of the workload assigned to each PE. Then, we compare it with the sequential computation time cost. Experimental results and time cost of mesh refinement are presented. The table and corresponding graphs are given below. Table 4-8 gives the timing results for MPI for 5 mesh refinement iterations.

Table 4-8: Time cost (seconds) of mesh refinement.

Mesh Refinement Iterations	ID of Each PE					
	PE 0	PE 1	PE 2	PE 3	PE 4	PE 5
2	0.000261	0.000375	0.000297	0.000385	0.000174	0.00018
3	0.000283	0.000303	0.000313	0.000313	0.000196	0.000256
4	0.000265	0.000668	0.000641	0.000652	0.000384	0.001029
5	0.000356	0.001915	0.002078	0.001791	0.00195	0.006032
6	0.000281	0.014	0.014	0.012	0.014	0.048

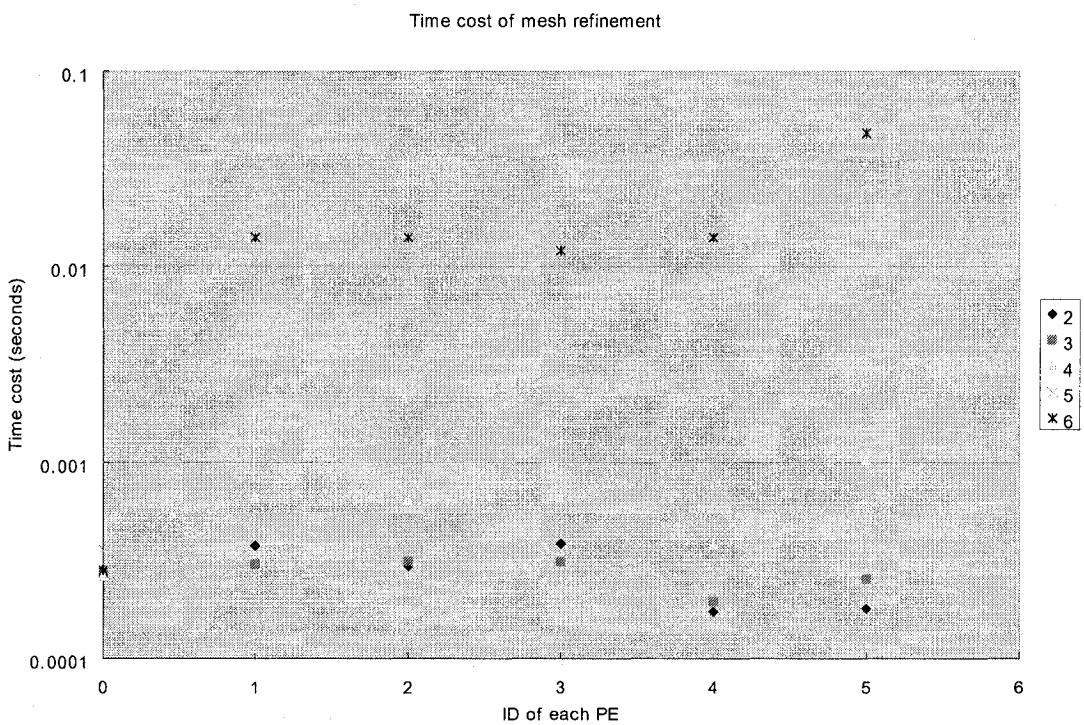

Figure 4-23: Time cost of mesh refinement.

The figure 4-23 shows that PEs 1 to 4 have similar computation time for a given iteration, for the workload assigned to them are ideally balanced. PE 5 has a little longer computation time as the iterations increase, because its working domain is an octahedron. The timing results for mesh refinement comparing sequential computing with each PE are presented in figure 4-24 and 4-25.

Table 4-9: Timing results for sequential computing vs. MPI PEs.

Mesh Refinement Iterations	Seque -ntial	ID of Each PE					
		PE 0	PE 1	PE 2	PE 3	PE 4	PE 5
5	0.015	0.000356	0.001915	0.002078	0.001791	0.00195	0.006032
6	0.085	0.000281	0.014	0.014	0.012	0.014	0.048

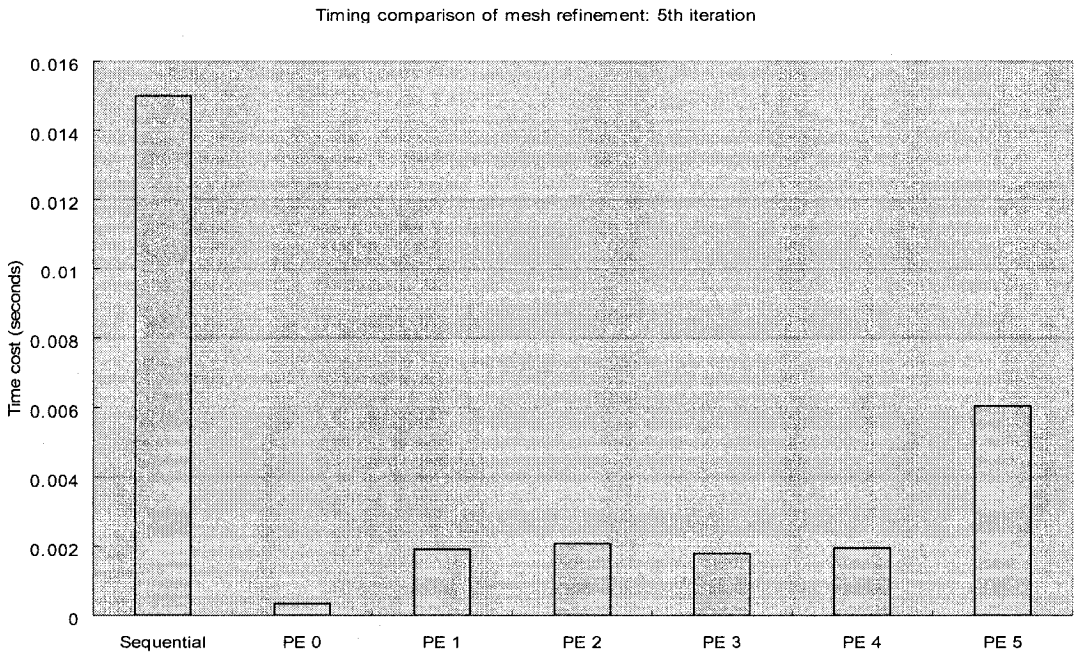

Figure 4-24: Timing comparison of mesh refinement:

Sequential computing vs. MPI PEs at 5^{th} iteration.

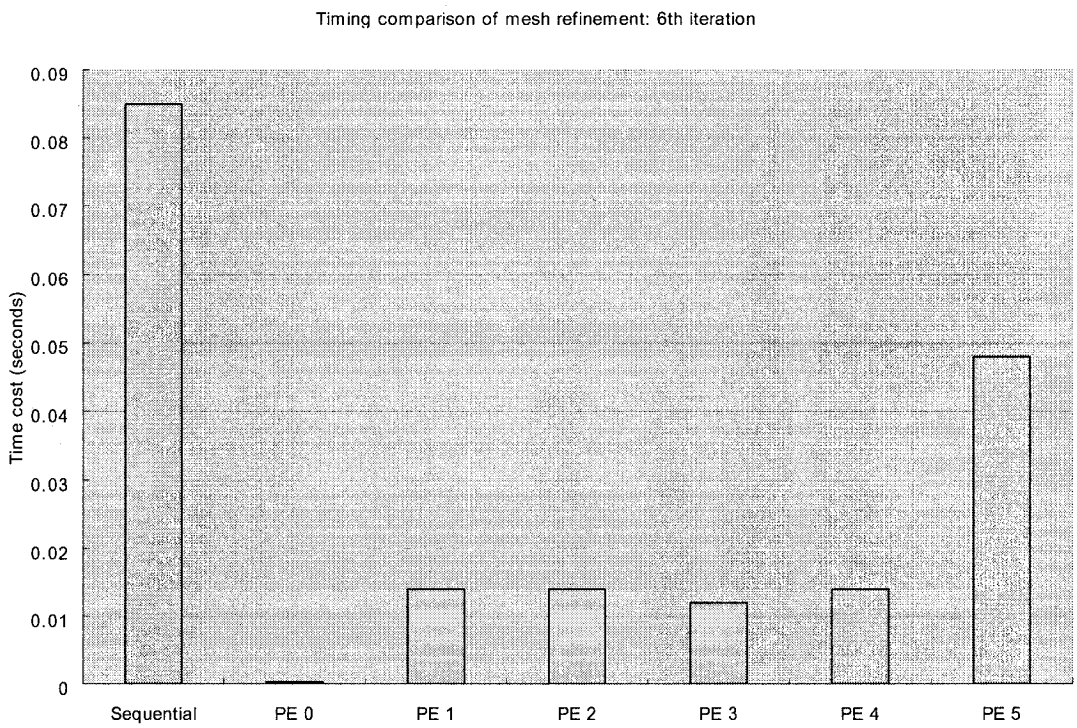

Figure 4-25: Timing comparison of mesh refinement:

Sequential computing vs. MPI PEs at 6^{th} iteration.

The above figures clearly show that the time cost for each mesh refinement iteration is indeed reduced using MPI.

Chapter 5

Conclusions

The parallel mesh refinement algorithm for hierarchical tetrahedral – octahedral (HTO) subdivision was successfully implemented using MPI. The efficiency of parallel computing using speedup factor was observed. The distribution time cost for mesh refinement computation, pre-processing for data gathering, and data gathering were also presented.

From the performance analysis results presented in the previous chapter, we have shown the efficiency of parallel computing over sequential computing. Parallel execution time is much faster then the sequential time, although the speedup factor slowly goes down as the number of elements increase. Most of all, we have clearly shown that the time cost for the mesh refinement is significantly reduced using MPI. The parallel method for hierarchical tetrahedral-octahedral subdivision for 3-D finite element mesh refinement indeed enhanced the performance.

The results of the distribution time cost shows that the communication delay is significant as the number of elements increases due to the build up of a geometric file. In order to reduce the communication delay, a pipelined communication strategy for the mesh refinement algorithm is necessary.

5.1 *The Original Contribution Appearing in a Publication*

The research and experimental results related to parallel hierarchical tetrahedral-octahedral subdivision: modeling, simulation and validation was accepted for publication in conference proceedings at CEFC 2006 (IEEE Conference on Electromagnetic Field Computation). CEFC 2006 was held in Miami (USA), April 30 - May 3, 2006 [43]. This contribution will also appear in *IEEE Transactions on Magnetics* [44] .

5.2 *Future Work*

There is possible future work in continuing this project or in this area. We have worked with a fixed number of processing elements (PEs) – 1 master PE and 5 slave PEs because we considered 4 tetrahedra and 1 octahedron after the first subdivision of an initial tetrahedron and allocated each element to each PE slave. However, it would be necessary to investigate the parallel algorithm using varying numbers of salve PEs with various sub-domain distributions.

Since the communication delay is significant due to the build up of a geometric file, it would be beneficial to modifying the algorithm by the addition of a pipelined communication strategy for the mesh refinement algorithm.

LIST OF REFERENCES

1. Lewis, R.W., P. Nithiarasu, and K.N. Seetharamu, *Fundamentals of the finite element method for heat and fluid flow*. 2004, Chichester, England ; Hoboken, NJ: Wiley.

2. G.Castanos, J. and J.E. Savage. *Parallel Refinement of Unstructured Meshes*. in *Proceedings of the IASTED International Conference*. 1999. MIT, Boston, USA.

3. Jin, J.-M., *The finite element method in electromagnetics*. 2002, New York: Wiley.

4. Felippa, C.A., *Introduction to Finite Element Method*. http://caswww.colorado.edu/course.d/IFEM.d/Home.html.

5. Dorica, M., *Novel Mesh Quality Improvement Systems for Enhanced Accuracy and Efficiency of Adaptive Finite Element Electromagnetics with Tetrahedra*, in *Computational Analysis and Design Laboratory Department of Electrical and Computer Engineering*. 2004, McGill University: Montreal, Canada. p. 79.

6. Albertelli, G. and R.A. Crawfis. *Efficient Subdivision of Finite-Element dataset into Consistent Tetrahedra*. in *'97., Proceedings*. 1997.

7. Yuan, J., L. Zhang, and Z. Li, *A step-by-step Approach for Three-dimensional Finite Element Mesh Generation*. IEEE Transactions on Magnetics, 1998. 34(5): p. 3375-3378.

8. Tsukerman, I. and Alexander Plaks, *Refinement Strategies and Approximation Errors for Tetrahedral Elements*. IEEE Transactions on Magnetics, 1999. 35 No. 3: p. 1342-1345.

9. Zhang, S.Y., *Successive Subdivisions of Tetrahedra and Multigrid Methods on Tetrahedral Meshes*, in *Houston J. of Math*. 1995. p. 541-556.

10. Ren, D.R. and D.D. Giannacopoulous. *A Preliminary Approach to Simulate Parallel Mesh Refinement with Petri Nets for 3-D Finite Element Electromagnetics*. in *Proceedings of ANTEM 2004*. p. 127-130. 2004.

11. Giannacopoulos, D.D. and D.Q. Ren. *Analysis and Design of Parallel 3-D Mesh Refinement Dynamic Load Balancing Algorithms for Finite Element Electromagnetics with Tetrahedra*. in *COMPUMAG 2005*. June 2005. China.

12. Ren, D.Q. and D.D. Giannacopoulos. *Parallel Mesh Refinement for 3-D Finite Element Electromagnetics with Tetrahedra: Strategies for Optimizing System Communication.* in *COMPUMAG 2005.* June 2005. vol. I, p. 120-121. China.

13. Greiner, G. and R. Grosso, *Hierarchical tetrahedral-octahedral subdivision for volume visualization*, in *The Visual Computer.* 2000. p. 357-369.

14. Schaefer, S., J. Hakenberg, and J. Warren. *Smooth Subdivision of Tetrahedral Meshes.* in *Eurographics Symposium on Geometry Processing.* 2004. Germany.

15. Vegter, G., *http://www.cs.rug.nl/~gert/compgeom.html.*

16. Philip, M., *Geomview Manual.* 1998.

17. Burgin, M.S., *Super-recursive algorithms.* Monographs in computer science. 2005, New York, NY: Springer.

18. Wilkinson, B. and C.M. Allen, *Parallel programming : techniques and applications using networked workstations and parallel computers.* 2005, Upper Saddle River, NJ: Pearson/Prentice Hall.

19. Quinn, M.J., *Parallel programming in C with MPI and openMP.* 2004, Boston ; London: McGraw-Hill Higher Education.

20. Kohout, J., *Delaunay Triangulation in Parallel and Distributed Environment*, in *Department of Computer Science and Engineering.* 2005, University of West Bohemia: Pilsen, Czech Republic. p. 127.

21. D'Ambra, P., et al. *Advanced Environments for Parallel and distributed computing.* in *Parallel Computing 28.* 2002: Elsevier Science B.V.

22. Duncan, R., *A Survey of Parallel Computer Architectures*, in *Survey & Tutorial Series.* February 1990. p. 5-15.

23. MA., Perkowski, *Digital systems design using VHDL and Verilog. Lecture Notes 1999.* 1999, Portland State University: Portland.

24. Blazy, S. and O. Marquardt. *Parallel Refinement of Tetrahedral Meshes on Distributed-Memory Machines.* in *Proceedings of the 23rd IASTED International Multi-Conference.* 2005. Innsbruck, Austria: Parallel and Distributed Computing and Networks.

25. Lakshmivarahan, S. and S.K. Dhall, *Analysis and design of parallel algorithms : arithmetic and matrix problems.* McGraw-Hill series in supercomputing and parallel processing. 1990, New York: McGraw-Hill.

26. Gupta, A., A. Grama, and V. Kumar, *Isoefficiency: Measuring the Saclability of Parallel Algorithms and Architectures.* IEEE Parallel and Distributed Technology, August 1993: p. 12-20.

27. TW., C. *An introduction to parallel rending.* in *Parallel Computing 1997.* 1997.

28. Pacheco, P.S., *Parallel Programming with MPI.* 1997: Morgan Kaufmann Publishers, Inc.

29. Gropp, W., E. Lusk, and A. Skjellum, *Using MPI : portable parallel programming with the message-passing interface.* Scientific and engineering computation. 1999, Cambridge, Mass. ; London: MIT Press.

30. Gropp, W., E. Lusk, and R. Thakur, *Using MPI-2 : advanced features of the message-passing interface.* Scientific and engineering computation. 1999, Cambridge, Mass. ; London: MIT Press.

31. Snir, M., *MPI -- the complete reference.* Scientific and engineering computation. 1998, Cambridge, Mass. ; London: MIT Press.

32. Geist, G.A., J.A. Kohl, and P.M. Papadoulos, *PVM and MPI: a Comparison of Features.* 1996. p. 1-16.

33. Takubo, H. and S. Yoshimura, *Parallel decomposition of 100-million DOF meshes into hierachical subdomains.* 1999, University of Tokyo: Tokyo. p. 13.

34. *CLUMEQ Infrastructure*, http://www.clumeq.mcgill.ca/.

35. Hsu, J.-M. and P. Banerjee, *Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer*, in *17th Int'l Symp. on Computer Architecture.* 1990.

36. Andrews, G.R. and F.B. Schneider, *Concepts and Notations for Concurrent Programming*, in *Computing Surveys.* 1983. p. 1-39.

37. Kutti, N.S., *C and Unix programming : a comprehensive guide incorporating the ANSI and POSIX standards.* 2002, Mt. Pleasant, SC: Lightspeed Books.

38. Microsystems, S., *Sun HPC Cluster Tools 5 Sofeware User's Guide.* 2003: Sun Microsystems Inc.

39. Microsystems, S., *Sun MPI 6.0 Software Programming and Reference Manual.* 2003: Sun Microsystems Inc.

40. Schildt, H., *C Programming Language : Explained to Learn Easier*. 1998, Seoul: E-Han Publishers, Inc.

41. Chandra, R., *Parallel programming in OpenMP*. 2001, San Francisco, CA: Morgan Kaufmann Publishers.

42. Buyya, R., *High performance cluster computing*. 1999, Upper Saddle River, N.J. : Prentice Hall PTR.

43. Ren, D.Q., C. Park, B. Mirican, D.D. Giannacopoulos and S. McFee, *Parallel hierarchical tetrahedral-octahedral subdivision: modeling, simulation and validation* in *Invited paper in the 12th Biennial IEEE Conference on Electromagnetic Field Computation (CEFC 2006)*. 2006. Miami, Florida USA.

44. Ren, D.Q., C. Park, B. Mirican, D.D. Giannacopoulos and S. McFee, *Parallel hierarchical tetrahedral-octahedral subdivision: modeling, simulation and validation* IEEE Transactions on Magnetics, (submitted), 2006.