View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

PARALLEL SEISMIC RAY TRACING

A Thesis

by

TARUN KUMAR JAIN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Nancy M. Amato
Co-Chair of Committee,	Lawrence Rauchwerger
Committee Member,	Richard L. Gibson Jr.
Head of Department,	Nancy M. Amato

December 2013

Major Subject: Computer Science

Copyright 2013 Tarun Kumar Jain

ABSTRACT

Seismic ray tracing is a common method for understanding and modeling seismic wave propagation. The wavefront construction (WFC) method handles wavefronts instead of individual rays, thereby providing a mechanism to control ray density on the wavefront.

In this thesis we present the design and implementation of a parallel wavefront construction algorithm (pWFC) for seismic ray tracing. The proposed parallel algorithm is developed using the STAPL library for parallel C++ code.We present the idea of modeling ray tubes with an additional ray in the center to facilitate parallelism. The parallel wavefront construction algorithm is applied to wide range of models such as simple synthetic models that enable us to study various aspects of the method while others are intended to be representative of basic geological features such as salt domes. We also present a theoretical model to understand the performance of the pWFC algorithm.

We evaluate the performance of the proposed parallel wavefront construction algorithm on an IBM Power 5 cluster. We study the effect of using different mesh types, varying the position of source and their number etc. The method is shown to provide good scalable performance for different models.

Load balancing is also shown to be the major factor hindering the performance of the algorithm. We provide two load balancing algorithms to solve the load imbalance problem. These algorithms will be developed as an extension of the current work.

ACKNOWLEDGEMENTS

I want to thank my advisor, Dr. Nancy M. Amato, for her guidance and support over the last two years. When I joined the Parasol group in December 2008, Dr. Nancy M. Amato suggested that I should work on developing a parallel wavefront construction algorithm for seismic ray tracing computation. I found the problem challenging and with continual encouragement form Dr. Amato, I was able to find my way through various problems and design the parallel algorithm.

Dr. Lawrence Rauchwerger provided valuable knowledge and insight that helped me in understanding the various aspects of parallel computing. His suggestions helped me to improve the algorithm. Dr. Richard L. Gibson Jr. provided a lot of helpful suggestions on the thesis.

I would also like to thank Roger Pearce, senior student in our group for sharing his knowledge on the wavefront construction algorithm. I also want to thank every member in the STAPL group who helped me. I could not have finished this work without their help.

I would like to thank my parents for their support and understanding. Finally, I want to thank Ashita Srivastava for supporting me in my endeavors.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Numerous applications in the field of geophysics rely on seismic data for estimation of the structure and the properties of subsurface geologic formations. Recent advances in 3D seismic data acquisition have increased the input data volume by several fold. In addition, processing methods have changed for high resolution processing which results in an increase in the computational cost. With this change in requirements, there is an increased need for parallel algorithms to provide high resolution processing for large data volumes.

Seismic waves are waves of energy that travel through the earth. These waves can be generated as a result of an earthquake, explosion, or some other process. Seismic waves can be distinguished by a number of properties including the speed the waves travel, the direction that the waves move particles as they pass by, etc. P waves (primary waves) are compressional waves that are longitudinal in nature. S waves (secondary waves) are shear waves that are transverse in nature. Shear waves typically follow compressional waves during an earthquake and displace the ground perpendicular to the direction of propagation. Depending on the direction of propagation, the wave can take on different surface characteristics; for example, in the case of horizontally polarized S waves, the ground moves alternately to one side and then the other.

Seismic data is used both before and during the production of hydrocarbons from a reservoir. Figure 1.1 illustrates a process by which seismic data is collected. Given the location of a source of seismic waves, the seismometers (receivers) measure the physical properties of rays such as their arrival time, amplitude and direction of propagation etc.

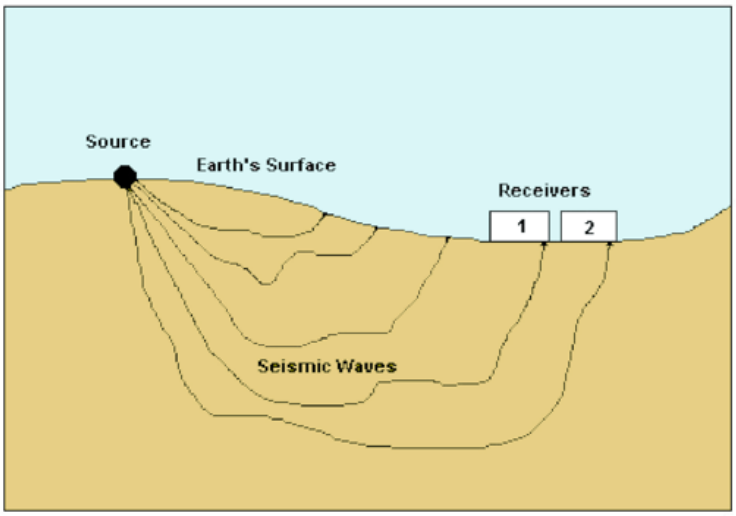


Figure 1.1 A simple earth structure with source and the receivers [1].

Forward modeling, the comparison of synthetic seismograms with recorded seismic data, is one very important application. A hypothetical earth model (material properties and features such as salt domes) is then developed on which seismic ray tracing algorithms are used to simulate the propagation of rays. Seismic data collected in field using seismometers is compared with the data produced by synthetic seismograms (simulation results). If the results match, the hypothetical earth model would match the experimental model. Otherwise, the hypothetical earth model is iteratively refined to match the experimental model.

Potentially even more significant than forward modeling are the seismic inversion and imaging methods that produce images of the subsurface from seismic data acquired to study the Earth's crust [2, 3, 4, 5, 6, 7]. These applications are very important because they produce images of subsurface geologic structure or directly estimate the properties such as seismic wave velocity. These subsurface images are some of the most important data used to guide hydrocarbon exploration efforts. Tomographic reconstruction is routinely used to develop two- and three-dimensional

models of the variations of rock properties [8].

Several types of computational methods are used for seismic modeling such as finite-difference methods and ray tracing methods. The finite-difference method (FDM) [9] attempts to solve exact elastic or acoustic wave equations by discretizing the model and applying finite-differences techniques to solve partial differential equations. This method can obtain the complete solution (total wavefield) [10, 11]. However, the wavefield simulated by finite-difference methods can be very complicated and difficult to interpret [12]. Also, the finite difference method is comparatively slow and requires significant memory resources.

In contrast to the finite-difference method, ray tracing methods solve simpler equations resulting from a high-frequency approximation. Using a ray tracing method [13, 14], we can calculate the travel-times and the amplitudes for different wave types independently. Ray tracing methods are faster compared to finite-difference methods, and are widely used in earthquake seismology as well as seismic imaging. Although ray tracing is relatively faster and less expensive than finite difference methods, conventional ray tracing has some limits in practical application. One of the well known difficulties is the two-point problem, which involves finding an exact ray path between source and receiver points [12]. Also, it is difficult to determine an initial number of and distribution of rays such that they can appropriately cover the areas where the rays diverge rapidly. An initial distribution of rays may not be adequate to provide material information for a portion of the model leading to the artificial shadow problem.

The wavefront construction method (WFC) [15, 16, 17, 18, 19, 20, 21] is an extension of the conventional ray tracing technique. Wavefront construction based ray tracing considers an entire wavefront instead of tracing individual rays and adaptively controls the ray density on the wavefront. The method was developed to improve the

computational efficiency and to overcome conventional ray tracing inherent problems such as the two-point problem and possible artificial shadow problems [22]. By using wavefront based ray tracing, it is possible to avoid artificial shadows typical of conventional ray methods. Earlier WFC algorithms were designed for isotropic (having identical values of a property in all directions) media [16, 17, 19] and subsequently several authors have introduced anisotropic (having different values of a property in different directions) WFC algorithms [20, 23, 24, 25].

The wavefront construction method interpolates the rays on the wavefront which may introduce errors. Very similar to the wavefront construction method, wavefront oriented ray tracing interpolates new rays from the source point when the ray density is less than the preset threshold at a wavefront [26, 25]. An advantage of wavefront oriented ray tracing over the wavefront construction method is the higher accuracy due to a reduction of possible errors from ray interpolation. Both methods are based on similar strategies, so if the wavefronts are constructed with enough rays, the difference between the two methods should be minimal.

1.1 Contributions

The objective of this work is to design and implement a parallel version of the wavefront construction method (pWFC). One modification introduced in pWFC to facilitate parallelism is to add an additional ray central to each adjacent set of four rays. As will be discussed in detail later, this was done to increase locality and reduce synchronization requirements.

The pWFC algorithm dynamically adds or removes rays from the wavefront as needed to maintain the desired density. The dynamic nature of the algorithm may lead to processor load imbalance resulting in loss of efficiency. To prevent performance degradation, we propose two dynamic load balancing strategies, one central-

ized and one decentralized. These algorithms will be further developed as future extensions of the pWFC algorithm.

The pWFC algorithm is developed using the STAPL library for parallel C++ code [27, 28, 29, 30, 31, 32, 33, 34, 35]. STAPL provides a wide range of generic parallel data structures, called `pContainers`, suitable for use in shared memory or distributed memory architectures. The pWFC algorithm uses the STAPL `pGraph` to represent the wavefront. The load balancing strategies are implemented using the element migration capabilities provided by the STAPL `pContainers`.

We study the performance of the pWFC algorithm on an IBM cluster with p575 SMP nodes (16 processor cores per node) available at Texas A&M University (P5-CLUSTER). We study the effect various factors such as the initialization method, the number of sources, material type, etc. We study the performance of the pWFC algorithm on a range of models including simple synthetic models that enable us to study various aspects of the method and on models intended to be representative of basic geological features such as salt domes. The pWFC algorithm is shown to provide scalable parallel performance. Load imbalance is shown to be a major factor for the performance loss, making the future development of load balancing algorithms imperative.

1.2 Outline of Thesis

Section 2 describes the basics of the sequential wavefront construction algorithm and provides a summary of the related work. It also provides an overview of STAPL and its components. Section 3 and Section 4 describe the design and implementation of the parallel wavefront construction algorithm, respectively. Section 5 shows the performance of the proposed algorithms. Section 6 concludes this work.

2. PRELIMINARIES AND RELATED WORK

This chapter presents preliminaries and related work. First, the sequential wavefront construction algorithm is sketched. Next, related work on parallel algorithms for a variety of seismic computations is summarized. Then we briefly describe the STAPL parallel C++ library which is used to implement the parallel wavefront construction algorithm.

2.1 The Sequential Wavefront Construction Method

Rays are used to model the propagation of seismic waves. A *ray* is a collection of *ray segments*, where a segment can be of the P or S wave type. Ray segments are traced in discrete time steps of size dt_{ray}. When a ray segment hits a region boundary, based on the surface properties and the algorithm input parameters, reflected or transmitted segments can be created. Figure 2.1 shows a ray with different ray segments.

A *wavefront* is defined as a surface connecting points with the same travel-time (ratio of distance traveled to the seismic wave speed) on the rays. The wavefront is composed of elementary geometric subdivisions of adjacent rays. Elementary geometric subdivisions are called *wavefront mesh cells* or *wavefront mesh elements*. We use quadrilateral shapes to define the wavefront mesh cell (Figure 2.2).

The wavefront construction method (WFC) [22, 12] is one implementation of the ray tracing method. The basic difference between the WFC method and conventional ray tracing is how individual rays are implemented. Instead of tracing a specified set of rays from a source to the boundary of a model, the WFC method begins with a few rays at the source and then adaptively adds/removes rays as the wavefront propagates away from the source. The major steps in the wavefront construction

algorithm are:

1. **Initialization**: The initial mesh in the wavefront construction method describes the geometric distribution of initial rays. The initial mesh can be regarded as a set of initial conditions for each of the ray directions. In this work, we use two geometries to generate the initial mesh: sphere and cube [12].

2. **Propagation of the Wavefront**: Figure 2.2 shows the wavefront composed of quadrilateral cells propagating through the model space. New rays are interpolated (Figure 2.3) or removed as needed to maintain a certain level of ray density. Thus, it adaptively controls the density of the rays by adding rays when the accuracy is below a threshold and removing rays when the accuracy is high.

The sequential wavefront construction algorithm in sketched in Algorithm 1.

Algorithm 1 Sequential WFC algorithm [22, 12]

Input: Earth model, mesh description, number of sources and their position
1: Initialize the rays using the specified geometry
2: **for** i = 0 to maximum wavefront index **do**
3: Trace the rays by one wavefront timestep
4: If the ray segments intersect a surface in the earth model, then create new ray segments if needed
5: Propagate the wavefront and add/remove rays as needed
6: **end for**

2.2 Parallel Seismic Computation

Parallel algorithms have been developed for a variety of problems in the field of geophysics. However, to the best of our knowledge, there is no publication discussing

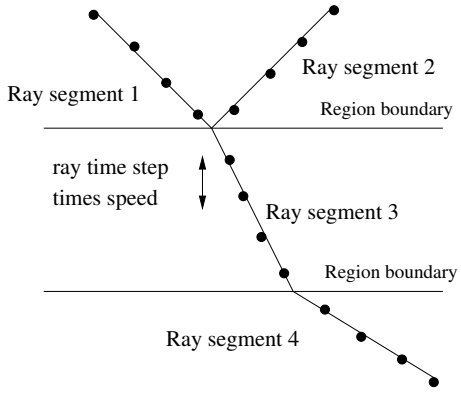


Figure 2.1 Ray organized as a collection of ray segments. Horizontal lines are the region boundaries that separate the regions. Ray segment 1 creates ray segment 2 (reflected ray segment) and Ray segment 3 (transmitted ray segment). Ray segment 4 (transmitted ray segment) is then created when ray segment 2 intersects the second region boundary.

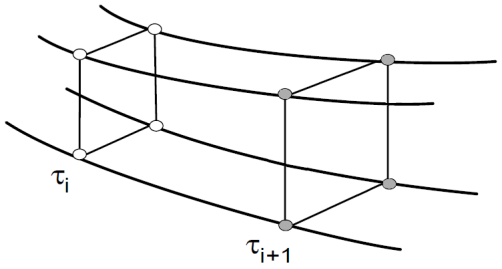


Figure 2.2 Wavefront mesh elements at two different time step τ_i (white point) and τ_{i+1} (gray point).

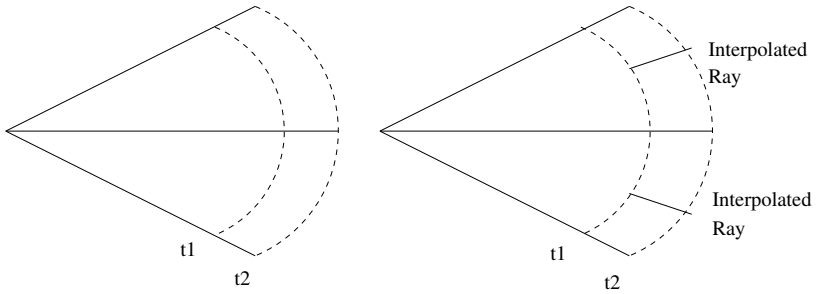


Figure 2.3 Rays are diverging with wave propagation and new rays are inserted at the interpolated points on the wavefront to satisfy accuracy criteria and constant density of rays.

a parallel algorithm for the wavefront construction method. In the following, a brief summary of the work done in the area of parallel computing for seismic applications is given.

Seismic tomography is an inverse problem which aims to build a global seismic wave velocity model of the Earth's interior. Grunberg et al. [36, 37] presented a parallel seismic tomography algorithm that implemented a fast seismic ray tracing in an Earth mesh [36]. Their ray-tracing algorithm is based on the Snell-Descartes law in spherical geometry. They use a master slave model for their parallel algorithm. Each process receives a description of the mesh and a set of rays to trace from the master process. Each process then starts to compute its rays. Subsequent phases such as data exchange (all-to-all communication) and concatenation (all-to-one communication) combine and send the result of computation to the master process.

Migration [38, 39] of seismic data involves repositioning the measured data to determine accurately the topology of the subsurface reflectors. Migration is an inverse process in which the recorded waves are propagated back to their source by systematically solving the wave equation for each successive layer. In [40] a coarse grained paradigm for computation on parallel computers is presented. Their work focused on approximate algorithms for 2D frequency domain migrations on hypercube interconnect networks. They partition rows/columns of the matrices and distribute them among the processors. Balanced load across the processors further improves the efficiency of their algorithm.

The Hierarchical Object Oriented Parallel Environment (HOOPE) [41] is a parallel application framework for pre-stack time migration (PSTM) [38, 39] and pre-stack depth migration (PSDM) [38, 39]. They use a parallel abstraction layer, based on MPI [42], which provides a set of parallel arrays with a global indexing mechanism. Components in this layer also provide necessary functionality for data decomposition

and inter-processor communication.

Some of the recent work on three-dimensional pre-stack time/depth migration [43] focuses on using common azimuth migration (CAM) for the treatment of marine data acquisition. In [43], the frequencies are distributed across MPI tasks and use shared memory parallelism for the low-level loops using OpenMP [44].

Recently, General Purpose Processing on Graphics Processing Units (GPGPU) have also been studied [45, 46, 47] for solving some of these problems. In [46], an implementation of a Reverse Time Migration algorithm [48] on a GPU cluster is presented. They consider a finite difference approach on a regular mesh, in both 2D and 3D.

2.3 STAPL

With the increasing availability of multiprocessor and multicore architectures, there is a growing need to more complex and larger problems which make parallel programming crucial for application development. The Standard Template Adaptive Parallel Library (STAPL) [27, 28, 29, 30, 31, 32, 33, 34, 35] is being developed to help programmers address the difficulties of parallel programming. STAPL (Figure 2.4) is a parallel C++ library with functionality similar to STL, the ISO adopted C++ Standard Template Library [49]. STL is a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications. Similar to STL, STAPL provides a collection of parallel algorithms (`pAlgorithms`), parallel and distributed containers (`pContainers`), and `pViews` to abstract the data access in `pContainers`.

`pContainers`, are the thread-safe distributed equivalent of STL containers. These containers are concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. `pContainers`can be composed and extended by

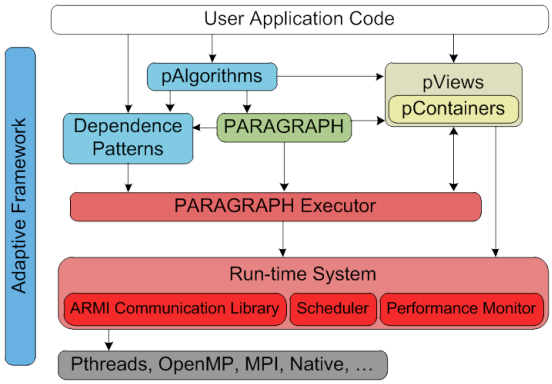


Figure 2.4 STAPL overview.

users via inheritance. Currently, STAPL provides distributed counterparts of all STL containers (e.g., `pArray`, `pVector`, `pList`, `pMap`, etc.), and `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`) [35]. While the `pContainer` data may be distributed, the programmers are provided a shared object view, i.e., a shared data structures with a global address space. Object translation method internally handle this and are able to transparently locate both local and remote elements. STAPL can automatically choose the physical distribution of a `pContainer` data or user can specify it. Individual `pContainer` elements can be redistributed as needed to provide optimized performance. The pWFC algorithm uses a `pMap` to represent the collection of rays and a `pGraph` to represent the wavefront.

It is a common practice in generic programming to decouple the data structures and the algorithms. This abstraction is achieved by STL, the C++ Standard Template Library, using *iterators*. These provide a generic interface for algorithms to access data which is stored in containers. This allows the same algorithm to work on different containers. The STAPL `pView` generalizes the iterator and corresponds to a collection of elements. The STAPL `pView` have *reference semantics*, meaning that a

`pView` simply *references* to the data and does not own it. `pViews` enable parallelism by providing random access in the partitioned data space which is essential for the scalability of STAPL programs. The pWFC algorithm uses `pViews` provided for `pMap` and `pGraph` parallel container.

A `pAlgorithm` is the parallel counterparts of an STL algorithm. STAPL includes a large set of parallel algorithms, including parallel equivalent of STL algorithms. Similar to STL algorithms that use iterators, STAPL `pAlgorithms` use `pViews`. `pContainer` can present multiple interfaces to its users using `pViews`, e.g., enabling the same `pMatrix` to be viewed (or used) as a row-major or column-major matrix or even as a linearized vector [35]. The pWFC algorithm uses the two parallel algorithms, namely `map` and `map_reduce`.

3. PARALLEL WAVEFRONT CONSTRUCTION METHOD

In this chapter, we present the parallel wavefront construction method (pWFC). While the overall algorithm is patterned after the sequential algorithm, it includes some modification to help facilitate parallelism. A theoretical model for the pWFC algorithm is also presented.

3.1 Algorithm Overview

In this section, we provide a high level description of the pWFC algorithm. Each individual step of the algorithm is described in more depth in the following sections. Like numerous other scientific computations, the parallel wavefront construction algorithm uses a iterative algorithm. The first step of the pWFC algorithm (refer to 2) is to initialize a coarse level grid of rays starting from a point in space (*source*). There may be multiple source locations each generating a different set of rays and ray tubes (line 1-3). *Take-off angle mesh* and *Cubed sphere mesh* are the two most widely used algorithms to construct the initial course level grid. Rays start from the source and go through the discretized points on the course grid.

If only the initial set of rays were to be used for computation, they may not be enough to validate that the earth model is correct or not. We may need a larger or a smaller set of rays to achieve the desired accuracy. Wavefront construction algorithm is an adaptive algorithm that can add/remove rays to keep the desired accuracy. Wavefront construction algorithm uses a set of four adjacent rays (forming a quadrilateral) to compute the accuracy of the wavefront. These four points when viewed at different times, give it an appearance of a tube which is referred as *ray tube*. To compute the accuracy, the rays are traced for $\frac{dt_{ray\ tube}}{dt_{ray}}$ time steps (line 6-12), i.e., the ratio of the time step size of the ray tube (also the time step size of the

wavefront) to the time step size of the ray. This is the minimum number of times a ray should be stepped so that information is generated for stepping the ray tubes by one time step. Next, all the ray tubes are stepped by one time step. This may result in interpolation/coarsening of the ray tubes (line 13-16). Load balancing algorithm may used to balance the load among the processors when imbalance is detected (line 17-19). This process is repeated until there are no more ray tubes left (line 4) or until we have reached the maximum simulation time.

Algorithm 2 Parallel WFC algorithm

Input: Earth model, mesh description, number of sources S and their position

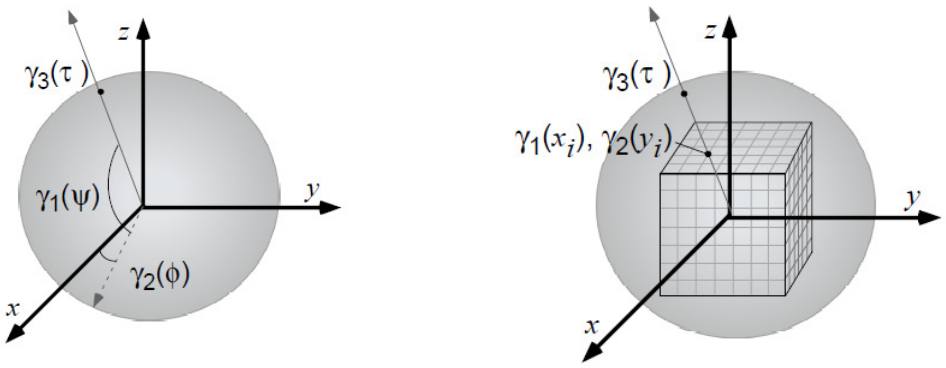
```
 1: parallel for each source in S
 2:     Initialize the rays in R and the ray tubes in RT using the specified geometry
 3: end parallel for
 4: while(RT != ∅)
 5:     for i = 0 to dt_ray_tube / dt_ray
 6:         parallel for each ray ∈ R
 7:             Trace the rays by one time step
 8:             if Ray segments intersect a surface in the earth model
 9:                 Create new ray segments if needed
10:             end if
11:         end parallel for
12:     end for
13:     parallel for each ray tube ∈ RT
14:         Step the ray tube by one time step
15:         Interpolating/coarsening the ray tube if necessary
16:     end parallel for
17:    if Load balancing needed
18:         Use load balancing algorithm
19:    end if
20: end while
```

3.2 Mesh Initialization

In this work, we have considered two methods for the initialization of the rays and the ray tubes: (i) Take-off angle mesh [14] and (ii) Cubed sphere mesh [12]. In the sphere geometry, the initial conditions or the take-off directions of each ray are specified with two ray parameters, γ_1 and γ_2. The third ray parameter can be the travel-time, $\gamma_3 = \tau$, or arclength, $\gamma_3 = s$, along the ray path [14]. The initial mesh is then constructed by connecting the points sharing the same travel-time along the ray paths ($\gamma_1 = \psi$, $\gamma_2 = \phi$, and $\gamma_3 = \tau_0$) (Figure 3.1(a)). Constructing the initial mesh with take-off angles is natural and straightforward to visualize and implement. However, this mesh coordinate system always has geometric poles at the top and the bottom (declination angle, $\psi = \pm 90$), where the population of rays is very dense and it is computationally inefficient [12]. Furthermore, when we compute ray derivatives near or at the pole, it is numerically unstable [12].

As we can use any physical quantity for the ray parameters as long as it can specify the ray uniquely, we can design different types of mesh generation schemes by choosing another set of ray parameters. An alternative choice of ray parameters is the cubed sphere mesh (Figure 3.1(b)), which uses an imaginary cube (focal cube) centered at the source point. Initial rays are projected from the source point for a unit travel-time, passing through the discretized points on each face of the cube. The cube is constructed with a unit length and discretized by NxN points for each face where N is the number of discretized points along an edge.

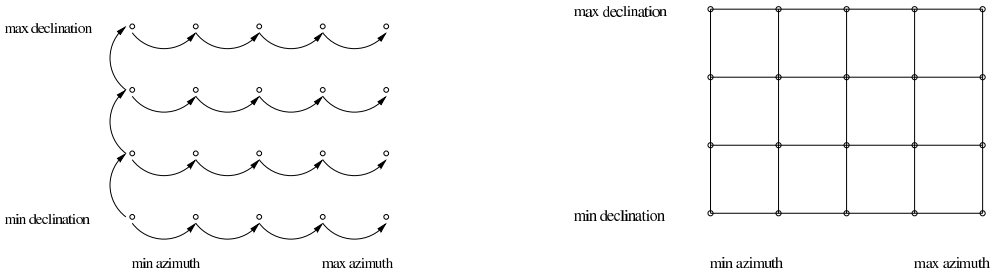
The initialization method is also dependent on the nature of the wave that is used for simulation. qP waves are longitudinal waves and the particles in the solid have vibrations along or parallel to the travel direction of the wave. qS waves are transverse waves and their motion is perpendicular to the direction of wave propagation. qS



(a) Take-off angle mesh (b) Cubed sphere mesh

Figure 3.1 Different mesh initialization geometries (a) Take-off angle mesh coordinate, the ray parameters are defined as $\gamma_1 = \psi$(declination), $\gamma_2 = \phi$(azimuth), and $\gamma_3 = \tau$(travel-time). (b) Suppose we have a unit cube centered at the source point, then a ray can be traced from the source point through an evenly discretized point on the face of the focal cube face. The coordinates of discretized points on the face of focal cube are new ray parameters defined as $\gamma_1 = x_i$ (x_1 component of a face), $\gamma_2 = x_j$ (x_2 component of a face), and $\gamma_3 = \tau$ (travel-time).

waves form two different wavefronts, namely qSA and qSB. The initialization step is fairly straightforward for qP wave simulation. For qS wave simulation, there are some dependencies that have to be respected to initialize the mesh correctly. The dependencies for the qS mesh creation are shown in Figure 3.2(a). First, we create two rays corresponding to the minimum azimuth and minimum declination angle and assign them to either the qSA or the qSB wavefront arbitrarily. Using these rays as the reference, the rays belonging to the minimum declination (leftmost column) can be assigned a wavefront type. Finally, all the rays in the row can be assigned a type using the previous ray in the same row as the reference. Figure 3.2(b) shows the mesh after the ray tubes have been created.



(a) Dependencies (b) Mesh after the initialization.

Figure 3.2 qS mesh creation algorithm (a) Dependencies for the qS mesh creation algorithm and (b) Mesh after the creation of rays and ray tubes. Each vertex is a ray and each square is a ray tube.

3.3 Wavefront Propagation

The wavefront created in the initialization phase is then propagated through the model. As described in the overview (Section A), various steps are involved in stepping the wavefront by one time step. These include tracing the rays, following the ray tube while maintaining numerical accuracy and load balancing as needed. These are described in the next few sections.

3.3.1 Tracing Rays

Rays are used to model the propagation of the wavefront. Each ray is identified using a pair of integers (r, c). Rays are traced in discrete time steps of size dt_{ray}. Ray path of the individual rays can be obtained by solving a set of ordinary differential equations. The current implementation uses the 5th order Runge-Kutta [12] method to numerically solve the ordinary differential equations; other numerical solvers such as Hammings predictor-corrector method or Adams-Moulton predictor formula can be used [12]. Note that ray tracing is not limited to any particular numerical solver as long as we can compute physically valid rays. Also, ray tracing can be performed with different approaches such as the graphical ray method or simple geometric ray

tracing for a multi-layered model with isotropic homogeneous layers.

Algorithm 3 presents the steps involved for tracing the ray by one time step. The first step of the algorithm is to step all the rays by one time step (dt_{ray} time). Note that each ray can be stepped independently (in parallel) of the other rays. Next, we need to make sure that the ray segment does not intersect with any surface as it is traced by one time step (ray behavior depends on material properties). This can be done by checking if the line segment formed by joining the position of the ray segment at time τ and $\tau + dt_{ray}$ intersects with any surface (region boundary). Since ray segments are traced based on the material properties of the region in which they are present, we need to determine the first surface with which a ray segment intersects. This is the position where the ray segment may enter into a region with different material properties. Finally, the result of the intersection query is used by the ray segment. If the ray segment was found to hit a surface, new segments are created based on the properties of the surface.

Algorithm 3 Ray trace algorithm

```
1: parallel for each ray ∈ R
2:     for each ray segment ∈ collection ray segments for the ray
3:         Step the ray segment of the ray by dt_ray time
4:         Find intersecting surface for the line segment (ray_[τ] and ray_[τ+dt_ray])
5:         if Intersection is found
6:             Create ray segments based on the properties of the surface
7:         end if
8:     end for
9: end parallel for
```

If there were n rays distributed uniformly among p processors and c_{ray} is the time needed to step one ray by one time step, then the time to trace all the rays by one

time step is O($\frac{n*c_{ray}}{p}$).

3.3.1.1 Line Segment Intersection

Given a ray segment (start and end point), at each time step we need to query if it intersects any surface (region boundary). This problem is known as line segment intersection query. Since these queries are performed whenever a ray segment steps by one time step, it is necessary to have an efficient method to solve these queries. A brute force solution would be to check if the line segment intersects with any object in the model. There are two general strategies of intersection culling which do this more efficiently by restricting the number of intersection tests: hierarchical bounding volumes [50, 51] and space partitioning [52, 53, 54]. The hierarchical bounding volumes approach is based on the idea of enveloping complicated objects that take a long time to intersect with simpler bounding volumes that are much easier to intersect, such as spheres [50] or axis aligned bounding boxes [51]. Before intersecting the complicated object, the bounding volume is first tested for intersection. If there is no intersection with the bounding volume, then there is no need to intersect the complicated object, thus saving time. The space partitioning approach for reducing intersection tests is to partition the space itself into voxels. Each voxel has a list of objects that are in that voxel. If an object spans several voxels it is in more than one list. In this approach, it is first determined which voxel contains the object of interest and then it is intersected with all objects contained in the voxel. Well known method for space partitioning include KD-tree [55], quad tree (2D models) and octree (3D models).

We use the octree [56] data structure to represent the model information. The octree is one of the best known space partitioning method used for speeding-up of intersection tests with a set of objects in an environment. The algorithm for

the line intersection test is straightforward (see Algorithm 4). The first step is to determine the octant which contains the starting point of the line segment. The line segment is then tested for the intersection with the objects in the octant. If the line segment did not intersect with any object in the octant its end point lies inside in the octant, it implies the line segment did not intersect with any object. Otherwise, the neighboring octant containing the line segment is determined and the line segment is intersected with the objects in that octant. This process is repeated until we either find an intersection or we are certain line segment does not intersection with any object. If multiple intersections are found in an octant, the intersection point closest to the starting point is determined as the intersection point.

Figure 3.3 provides an example of how the KD data structure works. For simplicity of illustration, the example uses a quad tree data structure. First, the start point of the line segment is located in partition 3 of the model. The line segment is intersected with objects in the quadrant (octant in 3D) but no intersection is found. The line segment passes through quadrant 5 (neighbor of quadrant 3). There also no intersection is found with any object. The line segment passes through quadrant 6 and an intersection is found with an object.

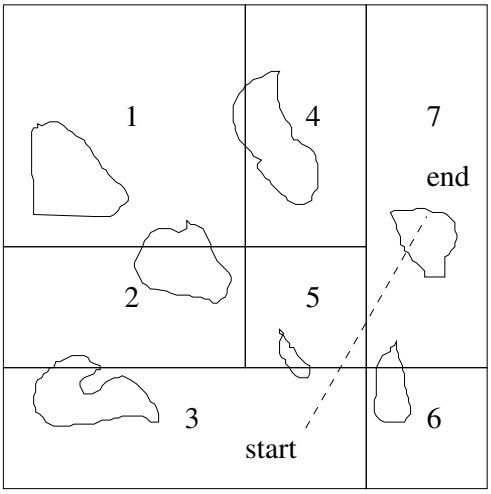


Figure 3.3 Intersection of line segment with objects partitioned in 2D space. Partitions are numbered 1 through 7 and line segment is represented by dotted lines.

Algorithm 4 Intersection algorithm(start point, end point)

```
1: start = start point
2: octant = Find bounding box(start)
3: while line segment(start, end point) does not intersect any object contained in
   octant do
4:    if end point lies in the octant then
5:       return  no intersection
6:    else
7:       octant = neighboring octant containing the line segment
8:    end if
9: end while
10: return  intersection point
```

3.3.2 Following Ray Tubes

In the pWFC algorithm, the initial wavefront propagates through the model space by controlling the ray density of a wavefront to ensure numerical accuracy. The density of the *rays* is monitored using the *ray tube* construct. Like a *ray*, a *ray tube* is a collection of *ray tube segments*. A ray tube is formed by four corner rays. The quadrilateral formed by joining points on the rays of a ray tube at time t create a wavefront cell. Collection of all the wavefronts at time t create the wavefront at time t. Ray tube segments *follow* the ray segments and check for errors (*patch test*) at time step $dt_{ray\ tube}$. The error computation determines if new rays need to be added or if they can be removed. We can use several quantities to determine the error values such as the distance between adjacent rays, the area defined by adjacent four rays (a cell on a wavefront), or other quantities. Figure 2.2 shows the ray tube segment propagation and interpolation/insertion of new rays.

In order to compute the error, ray tubes access some property of the rays at the different time steps. $\frac{dt_{ray\ tube}}{dt_{ray}}$ is the minimum number of times a ray should be stepped so that the ray tubes advance by one time step. The ray tube segment then determines the error value for the wavefront element. Based on the result of the patch test, the ray tube segment may be interpolated or coarsened. These two processes are described in the following sections. The interpolated rays and ray tubes start at the previous ray tube time step because the error was within the acceptable bounds at the previous time step. Hence, ray tubes that did not interpolate and the newly interpolated ray tubes are at different time steps. In our computation, we do not bring the interpolated rays and interpolated ray tubes to the same wavefront time (no synchronization) for the following reasons. These rays and ray tubes can proceed independently of each other. This is very beneficially for the parallel algorithm as it

allows the algorithm to work asynchronously. Also it may happen that the ray tube interpolated at time step t, may need further interpolation to reach acceptable error values. If the algorithm were to be of synchronous nature, a lot of processors would waste idle time waiting (load imbalance) for a few ray tubes to interpolate.

$$
\begin{aligned}
k = \quad & l_{max} - l \\
ray\ ids = \quad & \{(r * 2^k,\ c * 2^k), \\
& (r * 2^k,\ (c+1) * 2^k), \\
& ((r+1) * 2^k,\ c * 2^k), \\
& ((r+1) * 2^k,\ (c+1) * 2^k)\}
\end{aligned}
$$

Each ray is identified using a pair of integers (r, c). The identifiers of the rays are assigned in a manner such that the identifier of an interpolated ray between two rays is the mean of the identifier of the two rays. This way, we can assign the identifiers to rays once and they do not need to change as the ray tubes interpolate or coarsen. Each ray tube has an identifier triplet (l, r, c) where $0 \leq l \leq l_{max}$. There exists a closed form solution from the ray tube identifier to the identifier of the rays with which the ray tube works. The identifiers of the rays that the ray tube works with are

3.3.2.1 Interpolation

As the rays start propagating through the earth model, ray tubes are used to control and ensure the numerical accuracy of the results. When interpolating a ray tube segment, five new ray segments and four new children ray tube segments are created. The interpolation process for the ray tubes with four rays is shown in

Figure 3.4. Once the segments have been created, they can be added to the rays. The algorithm for interpolation of the ray tube segment working with four rays is presented in Algorithm 5. Note that once the segments have been created, the work of adding the rays and the ray tubes can be done independently and completely asynchronously.

Algorithm 5 Interpolation of ray tube

```
Create interpolated ray segments and ray tube segments
for i = 1 to 5 do
    Add-Ray-Segment(interpolated ray id_i, interpolated ray segment_i)
end for
if Ray tube has not been previously interpolated then
    Create children ray tube
end if
for i = 1 to 4 do
    Add the interpolated ray tube segment_i to the child ray tube_i
end for
```

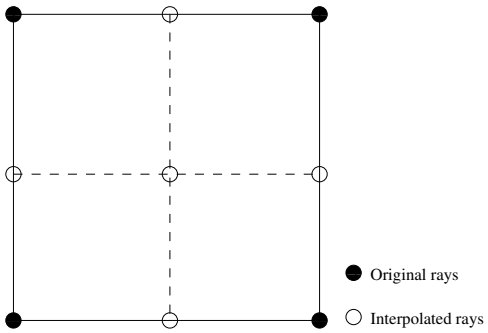


Figure 3.4 Interpolation of the ray tube working with four rays.

When, ray tube $RT = (l,\ r,\ c)$ (where l_{max} is the maximum interpolation level)

is interpolated, the identifiers of the children ray tubes are provided by the following equation.

$$
\begin{aligned}
\textit{ray tube ids} = \quad & \{(l+1,\ 2*r,\ 2*c), \\
& (l+1,\ 2*r,\ 2*c+1), \\
& (l+1,\ 2*r+1,\ 2*c), \\
& (l+1,\ 2*r+1,\ 2*c+1)\}
\end{aligned}
$$

3.3.2.2 Coarsening

Coarsening of the mesh is the inverse of interpolation. While interpolation is a necessary operation to ensure numerical accuracy, coarsening of the ray tubes reduces the work and space requirements. In some regions, the rays diverge rapidly and many rays and ray tubes are interpolated creating more work. It may happen that these interpolated ray tubes cross into a region where fewer rays and ray tubes are needed. If we stop tracing the extra rays and following the unnecessary ray tubes, we can reduce the extra work and improve the execution time of the application.

The pWFC algorithm removes the rays segments and ray tubes segments if they ray tube segment has relatively low numerical error. In the following section, we describe an algorithm to determine if wavefront coarsening can be done. We do this by tracking the state of the ray tube segments. Ray tube segments can be in one of the three states, a) *can* b) *wait* and c) *try*. The state transitions are shown in Figure 3.5. When a ray tube segment is created, it starts in the *can* state. If the error value is less than the threshold value while the segment is in the *can* state, it informs the parent ray tube segment that it wants to coarsen (*coarsen request*) and itself makes

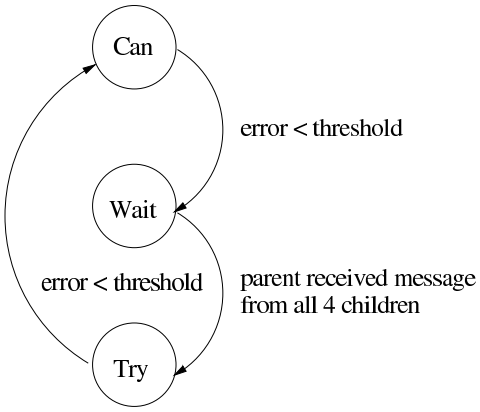


Figure 3.5 State transition diagram for ray tubes.

a transition to the *wait* state. When the parent ray tube segment receives a *coarsen request* from all of its children ray tubes, it informs its children ray tube segments to transit to the *try* state. In the *try* state, if the ray tube segment finds that the error value is less than the threshold value, it provides the information necessary to re-create (*coarsen re-create*) the parent ray tube segment and itself makes a transition to the *can* state. Once the information (*coarsen re-create*) from all four child ray tube segments has been received, the parent ray tube checks if the four children ray tube segments want to coarsen at the same wavefront time and are located in the same region of the earth model. If yes, the parent ray tube deletes the children ray tube segments. It also stops the tracing of ray segments that lie inside the parent ray tube.

Figure 3.6 shows an example of coarsening of ray tubes. Child tubes 1, 2, 3 and 4 are removed and their parent ray tube (bigger square) is recreated. There are a lot of different cases when the ray tube segment cannot coarsen back into the parent ray tube. For example (a) if all the four children ray tube segment inform the parent ray tube that they want to coarsen but they are in different regions and (b) if at least one of the children ray tube segment does not inform the parent ray tube at

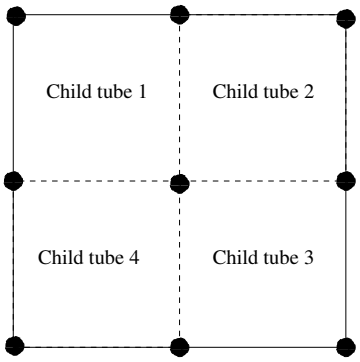


Figure 3.6 Coarsening of ray tubes.

wavefront time τ_1.

3.4 Ray Tube with Additional Ray

One important method for controlling the numerical accuracy of a ray tube is the central finite difference method. The method requires information from the neighboring rays of the mesh element (Figure 3.7). Book-keeping of the neighbor ray information is costly in storage and may require communication with other processors. Instead, of maintaining information related to neighbors, we propose a ray tube with an additional ray in the center. This allows us to use the central finite difference method without the neighbor information. Figure 3.8 shows a ray tube segment being patch tested at the various time steps. Ray tube segment failed the patch test at time step τ_4 and is interpolated at the previous time step τ_3.

Even though the additional ray in the center provides the necessary number rays for the central finite difference method, the method cannot be applied directly. The method requires that (a) central ray's parameters differ with each of the four corner rays in only one ray parameter (b) ray parameters of the diagonally opposite corner rays should also differ in one parameter. In the next two sections, we provide the modifications for the take-off angle mesh and cube sphere mesh generation

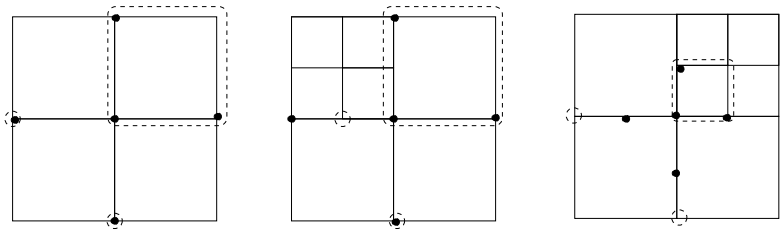


Figure 3.7 Ray tubes are shown in 2D. Dotted rectangle encloses the ray tube that is to be patch tested using the central finite difference method. The dotted circle represents the neighbor ray and black dots represent the ray which should be used for patch test. (a) All the neighboring rays exist that were needed for the patch test. (b) The required left adjacent neighbor of the central ray (black dot) does not exist (c) Both the horizontal and the vertical neighboring rays are not the required rays. Note that the required rays have not been interpolated.

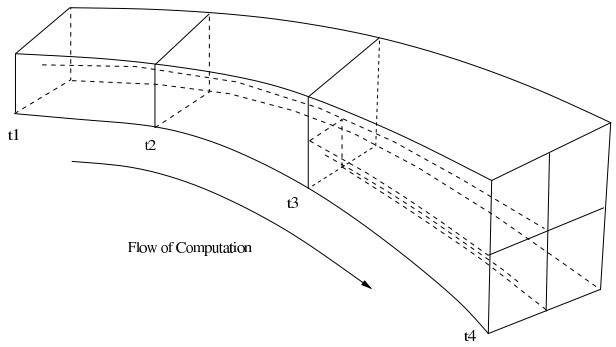


Figure 3.8 Wavefront element with an additional ray in the center. Wavefront mesh element failed the patch test at τ_4 time step and gets interpolated at time step τ_3.

algorithms.

3.4.1 Rotated Take-Off Angle Mesh

First, we describe the *rotated take-off angle mesh* which is a modification of the *take-off angle mesh* (as shown in Figure 3.9). Let ψ_i be the azimuth angle for rays $1 \leq i \leq n_\psi$ where n_ψ is the number of rays in azimuth direction. Let ϕ_j be the declination angle for rays $1 \leq j \leq n_\phi$ where n_ϕ is the number of rays in declination direction. Let $\gamma_1 = \psi$ (declination) and $\gamma_2 = \phi$ (azimuth) be the ray parameters for

a ray in the original ray parameter space.

To achieve the desired ray parameter properties, we rotate the *ray parameter space* such that an axis goes through a diagonal of the rectangle (equations 3.1 to 3.4, Figure 3.9(b)). Mesh generation algorithms may generate ray tubes such that the difference of the ray parameters $\delta\psi \neq \delta\phi$ on the two axis is not the same. For this reason, the mesh cell corresponding to a particular ray tube will be rectangular (Figure 3.9(a)). In redefining the ray coordinates, we therefore apply a shear factor that rescales one coordinate so that the final result is a mesh with cells of equal length in both directions. This simplifies finite difference calculations on the mesh (equations 3.5 to 3.6, Figure 3.9(c)). Equation 3.6 provides the ray parameters γ_1'' and γ_2'' for the ray in the modified ray parameter space.

$$\delta\psi = \frac{max\ (\psi_i) - min\ (\psi_i)}{n_\psi - 1} \tag{3.1}$$

$$\delta\phi = \frac{max\ (\phi_i) - min\ (\phi_i)}{n_\phi - 1} \tag{3.2}$$

$$\theta = \arctan \frac{\delta\psi}{\delta\phi} \tag{3.3}$$

$$\gamma_1' = \gamma_1 * cos(\theta) - \gamma_2 * sin(\theta),\ \gamma_2' = \gamma_1 * sin(\theta) + \gamma_2 * cos(\theta) \tag{3.4}$$

$$sf = \frac{\delta\phi^2 - \delta\psi^2}{\delta\phi^2 + \delta\psi^2} \tag{3.5}$$

$$\gamma_1'' = \gamma_1' + sf * \gamma_2',\ \gamma_2'' = \gamma_2' \tag{3.6}$$

3.4.2 Rotated Cubed Sphere Mesh

The *rotated cubed sphere mesh* is a modification of the cubed sphere mesh. Since the face of the cube is uniformly discretized on both the axis, we only have to rotate the ray parameters space to align the diagonals of the ray tube with the axis of the ray parameter space (Figure 3.10). Note that for the edge of the faces, we add

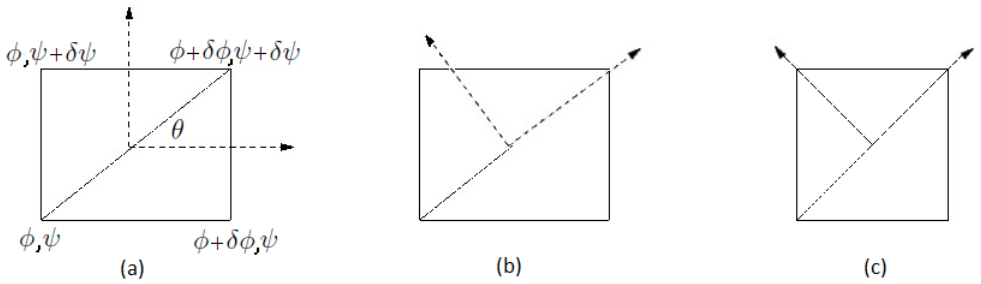


Figure 3.9 Transformation for the rotated take-off angle mesh. (a) ray tube with ray parameters in original space (b) ray tube with ray parameters space rotated (c) ray tube with ray parameter space rotated and then sheared.

duplicate rays (one for each face) as allows us to have a closed formed solution from the ray tube id to the ray ids (each ray belongs to a unique face).

Let $P = (x, y, z)$ be any discretized point on the surface of the cube. Equations 3.8, 3.9 and 3.10 provide the modified ray parameters for point P on the face perpendicular to the x, y, z axis respectively.

$$\theta = \frac{\pi}{4} \tag{3.7}$$

$$\gamma_1 = y * \cos(\theta) - z * \sin(\theta),\ \gamma_2 = y * \sin(\theta) + z * \cos(\theta) \tag{3.8}$$

$$\gamma_1 = z * \cos(\theta) - x * \sin(\theta),\ \gamma_2 = z * \sin(\theta) + x * \cos(\theta) \tag{3.9}$$

$$\gamma_1 = x * \cos(\theta) - y * \sin(\theta),\ \gamma_2 = x * \sin(\theta) + y * \cos(\theta) \tag{3.10}$$

3.4.3 Ray Tube Interpolation

If the ray tube works with five rays, then the identifiers of the rays for a given ray tube $RT = (l,\ r,\ c)$ (where l_{max} is the maximum interpolation level) are provided by the following equations.

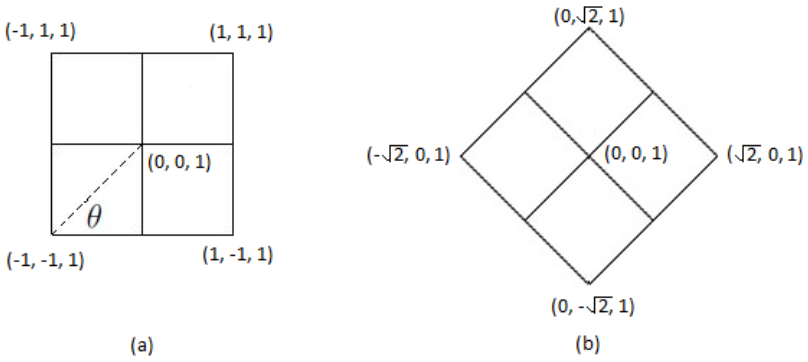


Figure 3.10 Transformation for the cubed sphere mesh. (a) face of the cube with z = 1 (b) ray parameters for the rays in the rotated ray parameters space.

$$
\begin{aligned}
k = \quad & l_{max} - l + 1 \\
ray\ ids = \quad & \{(r * 2^k,\ c * 2^k), \\
& (r * 2^k,\ (c+1) * 2^k), \\
& ((r+1) * 2^k,\ c * 2^k), \\
& ((r+1) * 2^k,\ (c+1) * 2^k), \\
& ((2 * r + 1) * 2^{k-1},\ (2 * c + 1) * 2^{k-1})\}
\end{aligned}
$$

When interpolating a ray tube working with five rays, eight new ray segments are created and four new children ray tube segments. The interpolation process for a ray tube with five rays is shown in Figures 3.11. The identifiers for the children ray tubes are not affected by the addition of a ray to the ray tube.

3.5 Theoretical Model

In this section, we present a theoretical model for the pWFC algorithm. Since we do not have any other implementation to compare our results, we develop this model

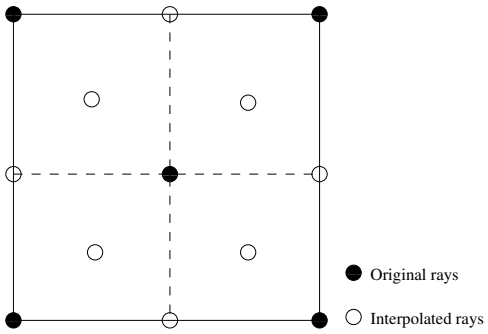


Figure 3.11 Interpolation of the ray tube working with five rays.

to understand the effectiveness of the pWFC algorithm. We work with a simplified model in which we do not have to worry about the dynamic nature of the algorithm. The model assumes that the mesh is uniformly interpolated l_{ex} times at source where l_{ex} is the maximum level of interpolation (determined from experiments). Note that this is not the same as initial mesh refinement, we do this to avoid any interpolation or collapsing of ray tubes and to have balanced load in the theoretical model.

Let n, m be the number of rows and column in the mesh respectively. Also, the processors are laid out in a 2D grid of p_n x p_m size. Suppose we had 8 ray tubes (2 x 4 mesh) created during the initialization phase. If one ray tube is refined three times while the others are refined only one time, the theoretical model assumes all the ray tubes were refined 3 times at the start of the computation. Thus, we assume we had a constant grid of size (32 x 16) propagating during the computation. Figure 3.12 presents a partitioning of the 2D mesh among 8 processors (2 x 4 processor grid). Each processor works on its sub-grid (area of the partition) and communication happens at the perimeter of sub-grid.

Let w_{ij} be the number of the ray tubes processed by the j^{th} processor in the i^{th} iteration of a computation (experimental data). The processors communicate only across the boundary (perimeter) of their partition (equation 3.12). We com-

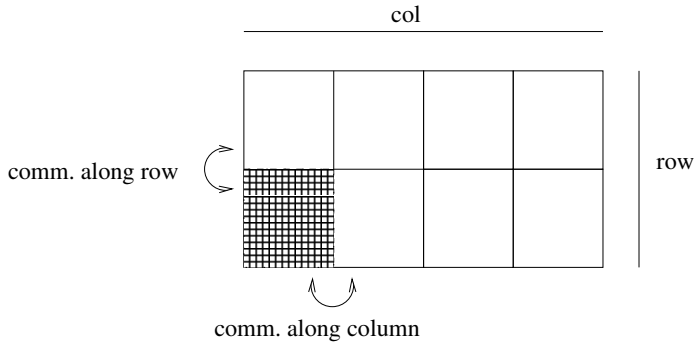


Figure 3.12 Mesh of size n x m partitioned among 8 processors (2 x 4 processor grid). Shaded portion represents the work assigned to a processor. Processors communicate with their neighbors.

pute theoretical scalability (S_p) that can be achieved for a problem instance based on the model of communication and computation (equation 3.13). This is difficult to achieve because of the load imbalance in the application. We then determine the *load imbalance ratio* based on the data collected from actual experiments. *Load imbalance ratio* is defined as the ratio between the sum of maximum work done in all the iterations of the algorithm to the average work done by all the processors in all iterations of the algorithm (equation 3.14). If all processors have equal amount of work in all iterations, the *load imbalance ratio* = 1. If there is a high degree of imbalance, the maximum work done by a processor would much greater than average work and hence the *load imbalance ratio* would be greater than 1. Next, we scale down the theoretical scalability (S_p) by the load imbalance ratio. This gives us the scalability that can be achieved with the given load imbalance (equation 3.14).

$$comm_p = \frac{n}{p_n} + \frac{m}{p_m} \tag{3.11}$$

$$S_p = \frac{comp_1}{comp_p + comm_p} \tag{3.12}$$

$$load\ imbalance\ ratio = \frac{\sum_{i=0}^{\#iter} \max_{0 \le j < p} w_{ij}}{\frac{1}{p} \sum_{i=0}^{\#iter} \sum_{j=0}^{p} w_{ij}} \quad (3.13)$$

$$S\prime_p = \frac{S_p}{load\ imbalance\ ratio} \quad (3.14)$$

To help understand the model better, let us consider an example. Let us have a 4 x 2 mesh grid partitioned among four processors (2 x 2). The maximum level of interpolation for interpolation $l_{ex} = 3$. Also, let us assume that observed load on four processors for three iterations were {2, 4, 8}, {2, 4, 8}, {2, 4, 8} and {2, 8, 16}.

Based on the initial mesh size and observed maximum interpolation level, we assume we had a mesh of size 32 x 16 which was partitioned among four processors. Based on the mesh size and processor grid, we first determine the communication cost (equation 3.16). Maximum theoretical scalability (equation 3.18) is then computed considering the communication that happens across the perimeter of a mesh partition. The load imbalance ratio (equation 3.20) is computed based on the load profiles collected during computation. The maximum scalability (equation 3.22) with load imbalance is computed by scaling down the theoretical scalability by the load imbalance ratio.

$$comm_4 = \frac{32}{2} + \frac{16}{2} \quad (3.15)$$

$$= 24 \quad (3.16)$$

$$S_4 = \frac{32\ X\ 16}{\frac{32\ X\ 16}{4} + 24} \quad (3.17)$$

$$= 3.37 \quad (3.18)$$

$$load\ imbalance\ ratio = \frac{\max\{2,2,2,2\} + \max\{4,4,4,8\} + \max\{8,8,8,16\}}{\frac{\sum 2,2,2,2 + \sum 4,4,4,8 + \sum 8,8,8,16}{4}} \quad (3.19)$$

$$= 1.53 \tag{3.20}$$

$$SI_4 = \frac{3.37}{1.53} \tag{3.21}$$

$$= 2.20 \tag{3.22}$$

4. IMPLEMENTATION IN STAPL

In this section we discuss the implementation of the pWFC algorithm in STAPL. First, we describe the parallel data structures used in parallel algorithm. Then, we discuss the implementation of the parallel algorithm described in the previous section.

4.1 Ray Collection

The ray collection data structure is one of the most important parallel data structures. The main operations the data structure provides are

- adding rays in the container, and
- supporting interpolation of a new ray between two existing rays.

To support these operations, we need a parallel data structure that provides random access and mapping from identifiers to the elements. We use the parallel container `pMap` provided by the STAPL library, for storing the rays. The data structure is convenient as it allows the user to access the elements based on identifiers. To achieve this, we use inheritance to derive from `pMap` and provide the required interfaces. Since the ray collection data structure is a type of mapping container with no relationship between elements, we choose inheritance to preserve the "type of" relationship.

```
1 template <class Key, class Value, class Compare,
2           class Partition=Default, class Traits=Default>
3 class p_map;
```

The pMap is an associative container that provides a mapping from *key* to *value* where *compare* is a used to compare the *keys*. Similar to other STAPL pContainers, users can specify an optional partition and traits.

```
1 template <class Ray, class Interpolation_Manager>
2 class ray_collection : public p_map<Ray_Id,
3                                     Ray,
4                                     less<Ray_Id> >
```

The ray_collection takes two template parameters, *ray* and *interpolation_manager*. The *ray* data structure represents a seismic ray and *interpolation_manager* provides methods to create an interpolated ray using the properties of the existing rays. *Less* is a class that enables lexicographic comparison of the identifiers of the rays.

We first look at the interface used to add the ray in the pContainer. All STL equivalent methods for map require a return type, which in general translates into a synchronous (blocking) method. For this reason, the STAPL pMap provides a set of non-blocking methods, e.g., insert_async and erase_async. These asynchronous methods enable the STAPL RTS to reduce the communication overhead by aggregating messages and allow better allow for better communication/computation overlap. We use the insert_async method which allows addition of an element to the pContainer in a non blocking manner. The complexity of the method is based on the complexity of the pMap insert_async method $O(\log(N))$ where N is the number of rays in the pContainer. This method to add the rays in the pContainer is used when the initial mesh is constructed.

```
1 // FUNCTION TO ADD THE RAY ASYNCHRONOUSLY
```

```
2 void add_ray_async(ray_descriptor_type const& desc,
3                    ray_type const& new_ray) {
4
5   insert_async(make_pair(desc, new_ray));
6 }
```

The interpolation of rays is a complex operation is parallel as neighboring ray tube may attempt to add the same ray at the same time. Also different ray tube segments may add different ray segments to the same ray. These operations must be handled carefully in a parallel algorithm to ensure the integrity of the wavefront. Ray_collection provides an `insert_async` method inherited from `pMap`. This function can be used to interpolate the rays (see following code snippet, line 13-16). The method requires a pair of an identifier (key) and a ray (value) along with a *functor*. If the ray associated with the unique key is not present, then the ray is added to the `pMap`. If the ray is present, then the *functor* is invoked (line 1-12). The *functor* receives the old ray and the new ray. The functor calls the *combine* method of the ray class (line 10). This function checks if the old ray contains the ray segments that are contained in the new ray. If the segments are already present, then the old ray is not modified as already has the required ray segments. Otherwise, the ray segments of the new ray are added to the old ray.

The complexity of the method is based on the complexity of the `pMap` `insert_async` method $O(\log(N))$ where N is the `pContainer` size. Stopping a ray segment from being traced is a much simpler operation than interpolation. This requires modifying the state of a ray segment contained in ray which can be uniquely identified. Since we only need to operate on an element (ray) and we do not need any return value, we use asynchronous methods provided by `pContainers`.

```
1  // FUNCTOR
2  class add_functor {
3
4    public:
5
6    template <class T>
7    result_type operator()(T& old_ray,
8                           T const& new_ray) const {
9
10     old_ray.second.combine(new_ray.second);
11   }
12 };
13
14 // ADD THE INTERPOLATED RAY
15 insert_async(make_pair(interpolate_ray_id, interpolate_ray),
16              add_functor());
```

Tracing of rays from one wavefront to next is another important operation that is done on the collection of the rays objects. This can be achieved by a simple `pAlgorithm` is STAPL (see following code snippet, line 17.) For this, we specify a view over the elements of the ray_collection (line 14) and the specify the work (using a *work function*) that should be applied to each ray object (line 1-11.) Since we want to advance all the rays by one time step, we create a view over all the ray objects (entire domain) in the container. The *work function* receives a view over a ray (line 7) and in turn calls the ray object's trace function (line 9) to advance it by one time step. Since the ray is a container that holds ray segments, it calls the trace method for the ray segments it contains. STAPL provides a `p_for_each` algorithm that applies

a *work function* to each element in a container, our ray_collection. The complexity of the operation is $O(\frac{N}{P})$ where N is the `pContainer` size and P is the number of locations.

```
1  // WORK FUNCTION
2  class step_ray_wf {
3
4    public:
5
6    template <typename View>
7    void operator() (View const& element) const {
8
9      element.step_ray(dt_ray);
10   }
11 };
12
13 // VIEW ON THE pCONTANER
14 map_view view(ray_collection, dom_type(ray_collection));
15
16 // CALL TO PALGORITHM
17 p_for_each(view, step_ray_wf(dt_ray));
```

4.2 Ray Tube Collection

The ray tube collection is another important parallel data structure. The main operations that the ray tube collection data structure provides are

- stepping the ray tubes by one time step, and

- refining and coarsening the wavefront mesh by interpolating or removing the ray tubes.

For managing the collection of the ray tubes, we use the parallel container `pGraph` provided by the STAPL library. Each initial mesh element is the root of a tree and the `pGraph` is used as a forest. This way we can model the parent child relationship of the ray tubes.

The STAPL `pGraph` is a generic data structure that consists of a collection of vertices and relations between vertices called edges. The `pGraph` associates a vertex property with each vertex and an edge property with each edge. These are template arguments that are passed by the user when instantiating a `pGraph`. Additionally, using template arguments, users can indicate if the graph is directed (*directedness*) and if the graph allows multiple edges between the same source and destination (*multiplicity*). Similar to other STAPL `pContainers`, users can specify an optional partition and traits. The `pGraph` declaration is the following:

```
1 p_graph<Directnedss,
2           Multiplicity,
3           vertex_property=no_property, //optional
4           edge_property=no_property,   //optional
5           partition=default,           //optional
6           traits=default_traits        //optional
7         >
```

Similar to how the ray_collection parallel data structure inherits from `pMap`, the raytube_collection inherits from the `pGraph`. We use ray tubes as the vertex property. The parent and the child vertices are connected by directed edges. Edges from

a parent ray tube to the four children ray tubes have unique identifiers 0, 1, 2, 3 to distinguish among them. The ray tube data structure is also passed as a template argument.

```
1 template <class Raytube_Type>
2 class raytube_collection : public p_graph <DIRECTED,
3                                            MULTIEDGES,
4                                            Raytube_Type,
5                                            int> {
```

The pGraph provides an add_vertex method to add a vertex. The add_raytube function uses this interface and provides an intuitive interface to the user. The complexity of the method is based on the complexity of the pGraph method, and can be $O(1)$ amortized time if a hash table used as pGraph's base container.

```
1 // Function to add a raytube with an id
2 descriptor_type add_raytube (raytube_type const& new_tube) {
3
4   return add_vertex(new_tube);
5 }
```

Next, we show how all the ray tubes are stepped by one time step. Ray tubes read the properties of the rays at different time steps and use this information to determine if they need to interpolate or coarsen. The rays which are accessed may not always be local, which may result in a remote access. To avoid blocking on the remote access, we use a common strategy of breaking the operation in two parts. In the first step, the ray tube (asynchronously) requests the information from the rays (see following code snippet, line 5). The rays asynchronously write the requested

information to memory allocated for the ray tube. Later, this information is directly used by the ray tube for computation (line 12). The complexity of the operation is $O(\frac{N}{P})$ where N is the `pContainer` size and P is the number of processors.

```
1  // CREATE A VIEW OVER THE RAY TUBES
2  raytube_collection_view view(raytube_collection);
3
4  // WORKFUNCTION TO READ THE RAY PROPERTY AND WRITE IT
5  raytube_read_wf read_wf(max_time, ray_data);
6
7  // CHECK IF NO RAY TUBES IS LEFT TO PROCESS
8  if(map_reduce(read_wf, logical_or<bool>(), view) == false)
9    break;
10
11 // WORKFUNCTION TO USE THE DATA
12 raytube_consume_wf consume_wf(max_time, ray_data, view);
13
14 // CALL THE MAP FUNCTION
15 map_func(raytube_consume_wf, view);
```

When operating on the ray tubes, some of the operations require working on the parent or child of the ray tube. Here, we show how the *coarsen req* information is provided to the parent of a ray tube asynchronously. Other methods are similar to this. The method iterates over the edges for the ray tube. (see following code snippet, line 9-23). It searches for the edge to the parent ray tube (line 13). Once the edge is found, it synchronously applies an functor representing *coarsen req* action to the parent ray tube (line 21).

```
1  void add_coarsen_req_info(raytube_descriptor& raytube_id,
2                            coarsen_req_info& info) {
3
4    // GET THE ITERATOR OVER THE RAY TUBE (vertex)
5    iterator raytube_it = find_vertex(raytube_id);
6    adj_edge_iterator eit_cur = (*raytube_it).begin();
7    adj_edge_iterator eit_end = (*raytube_it).end();
8
9    // CHECK THE EDGE LIST FOR THE RAY TUBE
10   for(; eit_cur != eit_end ; ++eit_cur) {
11
12     // GET THE EDGE TO THE PARENT
13     if((*eit_cur).property() == EDGE_TO_PARENT) {
14
15       // GET THE PARENT RAY TUBE DESCRIPTOR
16       raytube_descriptor parent_desc = (*eit_cur).target();
17
18       add_coarsen_req_info_functor_type functor(info);
19
20       // ASYNCHRONOUSLY ADD INFORMATION TO PARENT RAY TUBE
21       vp_apply_async(parent_desc, functor);
22     }
23   }
24 }
```

5. PERFORMANCE EVALUATION

This chapter examines the performance of the pWFC algorithm on an IBM cluster available at Texas A&M University. The pWFC algorithm has two separate phases (1) mesh initialization and (2) wavefront propagation. We discuss the performance of each of these phases in the following sections.

5.1 Machine Specification

We conduct our experimental study on an IBM cluster with p575 SMP nodes available at Texas A&M University (P5-CLUSTER). The machine has 52 compute nodes with 16 cores and 25 GB of memory available for applications per node. Table 5.1 shows the details of the configuration of the machine.

Table 5.1 Machine specifications about P5-CLUSTER.

Configuration	P5-CLUSTER
Number of compute nodes	52
Processor cores per node	16
Number of compute processor cores	832
Processor Core type	1.9GHz Power5+ processor
System theoretical peak (compute nodes only)	6.3 TFlop/sec
Physical memory per compute node	32 GB
Memory usable by applications per node	25 GB
Switch Interconnect	2-plane HPS (IBM's High Performance Switch)
Operating System	AIX 5.3

5.2 Experimental Measurement

For the purpose of measuring time taken by various phases of the pWFC algorithm, we use *GetTimeOfDay* (accuracy of $1 * 10^{-6}$ sec) function. Since the time taken in experiments is larger by a few order of magnitudes, the function provides relatively low error in measurements. We also repeat the experiments 32 times to gather enough data points and compute 95% confidence interval.

5.3 Earth Models and Input Parameters

In this section, we provide the description of the models that were used to study the performance of the pWFC algorithm. We also provide a description of the various input parameters that may influence the performance of pWFC algorithm.

5.3.1 Earth Models

The models include simple synthetic models that enable us to study various aspects of the method and others that are intended to be representative of basic geological features such as salt domes.

The homogeneous model has only one region and no surfaces (Figure 5.1). We use three different material properties for this model to study the effect of the material properties (Table 5.2) on the performance of the pWFC algorithm. The layered model (Figure 5.2 and Table 5.3) and the ledge model (Figure 5.3 and Table 5.4) are synthetic models designed to study the geometrical properties of wavefront propagating through isotropic-isotropic medium, isotropic-anisotropic medium and anisotropic-anisotropic medium. The salt canopy model (Figure 5.4 and Table 5.5) [57] and the salt dome model (Figure 5.5 and Table 5.6) [58] provide approximation of representative salt structures. They also show how seismic velocities typically depend on depth in areas such as the Gulf of Mexico (GOM).

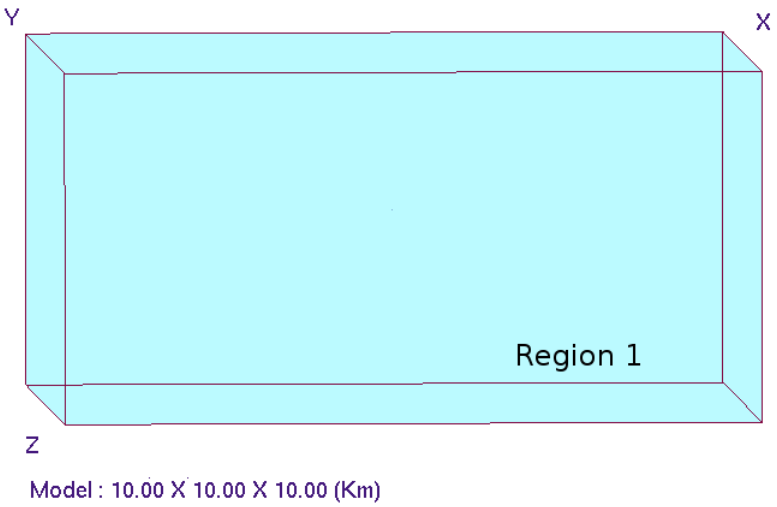


Figure 5.1 Model 1: Homogeneous model

Table 5.2 Model 1: Homogeneous model

Isotropic medium	V_p=3 km/s, V_s=1.73 km/s, ρ=2.67
Weak anisotropic medium	$\begin{pmatrix} 25.9 & 6.825 & 7.075 & 0. & 0. & 0. \\ 6.825 & 25.9 & 7.075 & 0. & 0. & 0. \\ 7.075 & 7.075 & 23.775 & 0. & 0. & 0. \\ 0. & 0. & 0. & 7.325 & 0. & 0. \\ 0. & 0. & 0. & 0. & 7.325 & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.525 \end{pmatrix}$ ρ=2.5
Strong anisotropic medium	$\begin{pmatrix} 59.6738 & 25.8196 & 25.8196 & 0. & 0. & 0. \\ 25.8196 & 93.6941 & 36.0941 & 0. & 0. & 0. \\ 25.8196 & 36.0941 & 93.6941 & 0. & 0. & 0. \\ 0. & 0. & 0. & 28.8001 & 0. & 0. \\ 0. & 0. & 0. & 0. & 23.1444 & 0. \\ 0. & 0. & 0. & 0. & 0. & 23.1444 \end{pmatrix}$ ρ=2.7

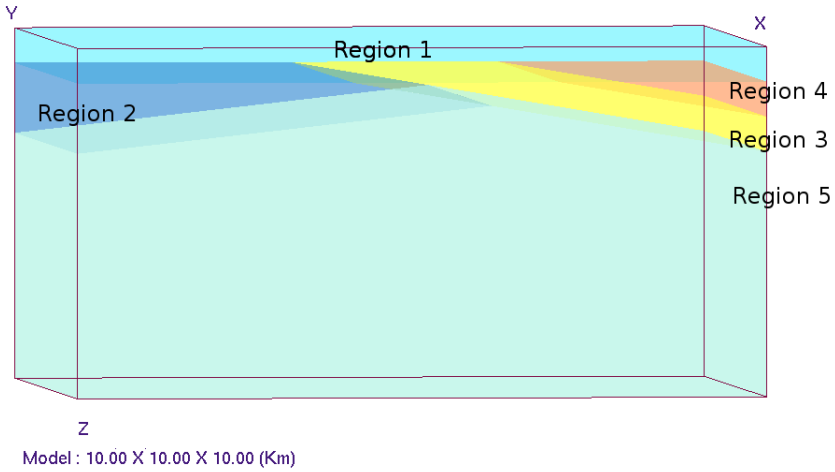


Figure 5.2 Model 2: Layered model

Table 5.3 Model 2: Layered model

Region 1	V_p=2.5 km/s, V_s=1.44, ρ=2.2
Region 2	V_p=3 km/s, V_s=1.73, ρ=2.5
Region 3	V_p=3.2 km/s, V_s=1.8, ρ=2.5
Region 4	V_p=3.6 km/s, V_s=1.8, ρ=2.67
Region 5	V_p=4 km/s, V_s=2.2, ρ=2.7

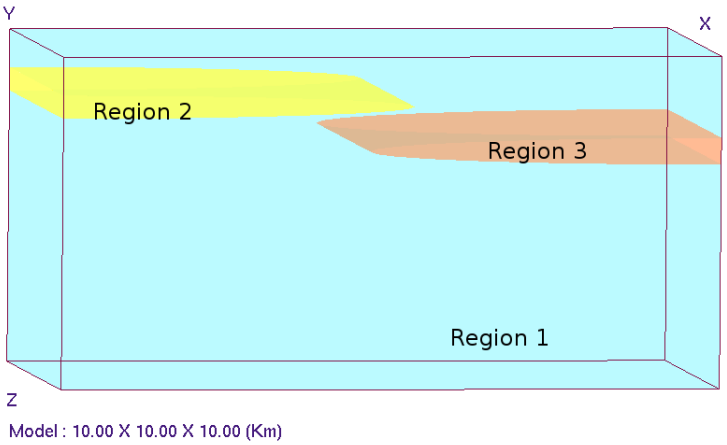


Figure 5.3 Model 3: Ledge model

Table 5.4 Model 3: Ledge model

Region 1	$\begin{pmatrix} 25.9 & 6.825 & 7.075 & 0. & 0. & 0. \\ 6.825 & 25.9 & 7.075 & 0. & 0. & 0. \\ 7.075 & 7.075 & 23.775 & 0. & 0. & 0. \\ 0. & 0. & 0. & 7.325 & 0. & 0. \\ 0. & 0. & 0. & 0. & 7.325 & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.525 \end{pmatrix}$ ρ=2.5
Region 2	V_p=2.8 km/s, V_s=1.5, ρ=2.6
Region 3	V_p=2.8 km/s, V_s=1.5, ρ=2.6

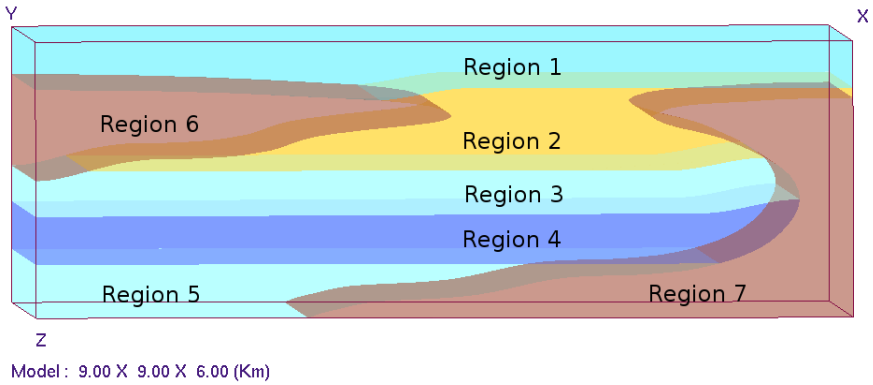


Figure 5.4 Model 4: Salt canopy model [57]

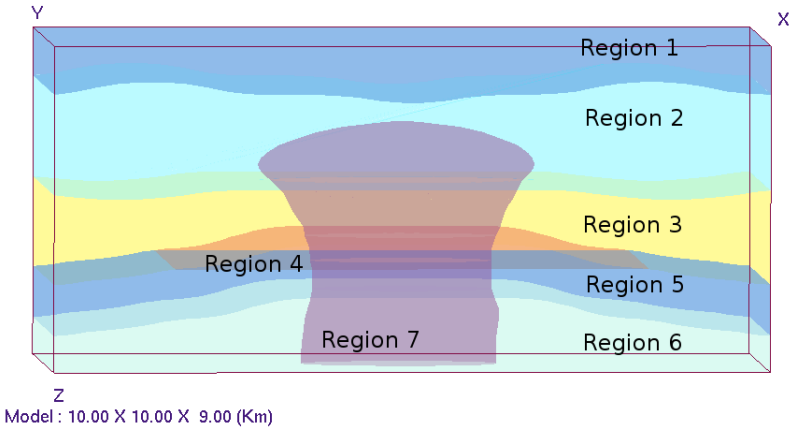


Figure 5.5 Model 5: Salt dome model [58]

Table 5.5 Model 4: Salt canopy model [57]

Region 1	V_p=3 km/s, V_s=1.73 km/s, ρ=2.5
Region 2	$\begin{pmatrix} 20.28 & 13.104 & 15.028 & 0. & 0. & 0. \\ 13.104 & 20.28 & 15.028 & 0. & 0. & 0. \\ 15.028 & 15.028 & 22.542 & 0. & 0. & 0. \\ 0. & 0. & 0. & 4.498 & 0. & 0. \\ 0. & 0. & 0. & 0. & 4.498 & 0. \\ 0. & 0. & 0. & 0. & 0. & 3.588 \end{pmatrix}$ ρ=2.4
Region 3	$\begin{pmatrix} 25.9 & 6.825 & 7.075 & 0. & 0. & 0. \\ 6.825 & 25.9 & 7.075 & 0. & 0. & 0. \\ 7.075 & 7.075 & 23.775 & 0. & 0. & 0. \\ 0. & 0. & 0. & 7.325 & 0. & 0. \\ 0. & 0. & 0. & 0. & 7.325 & 0. \\ 0. & 0. & 0. & 0. & 0. & 9.525 \end{pmatrix}$ ρ=2.5
Region 4	V_p=3.4 km/s, V_s=1.83 km/s, ρ=2.67
Region 5	V_p=3.8 km/s, V_s=1.9 km/s, ρ=2.7
Region 6 & 7	V_p=4.78 km/s, V_s=2.7 km/s, ρ=2.2

Table 5.6 Model 5: Salt dome model [58]

Region 1	V_p=2.7 km/s, V_s=1.5 km/s, ρ=2.55
Region 2	V_p=3 km/s, V_s=1.73 km/s, ρ=2.5
Region 3	V_p=3.2 km/s, V_s=1.8 km/s, ρ=2.55
Region 4	V_p=3.3 km/s, V_s=1.9 km/s, ρ=2.7
Region 5	V_p=3.4 km/s, V_s=1.9 km/s, ρ=2.67
Region 6	V_p=3.6 km/s, V_s=2.1 km/s, ρ=2.7
Salt dome	V_p=4.78 km/s, V_s=2.7 km/s, ρ=2.2

5.3.2 Input Parameters

The two main steps of the pWFC algorithm are (a) initialization of the mesh, and (b) propagation of the wavefront. Initialization of the mesh is dependent on the initial mesh size. Bigger mesh sizes would require more time as more rays and ray tubes have to be initialized.

Many factors may influence the performance of the propagation phase. For example, tracing a ray segment by one time step in an isotropic medium needs much less time than tracing the segment in an anisotropic medium as it requires much simpler equations to propagate in isotropic medium. Furthermore, ray segments may intersect the surfaces at different time steps. Ray tubes are also influenced in a similar way based on the material properties. Ray tube segments may additionally interpolate or coarsen in different iterations. To complete these operations, additional time is needed by the ray tubes. Furthermore, interpolation or coarsening changes the number of rays and ray tubes a partition of the mesh may have. This may lead to some processors being overloaded with work and some underloaded. Another major factor that can effect the performance of the parallel algorithm is the position of the source in the model. Some rays may reach the boundary of the model in fewer time steps while others may take many more time steps. As shown in Figure 5.6, the source is located at the center of the model. After a certain number of iterations ($\frac{1}{\sqrt{3}} * n_i, where\ n_i = total\ \#\ iterations$) some rays reach the model boundary. Thus some processors (locations) may become idle leading to load imbalance.

5.4 Results

This section provides performance results for the pWFC algorithm. First, we investigate the parameters that may affect the performance of the initialization phase. Next we look at the factors that influence the wavefront propagation phase.

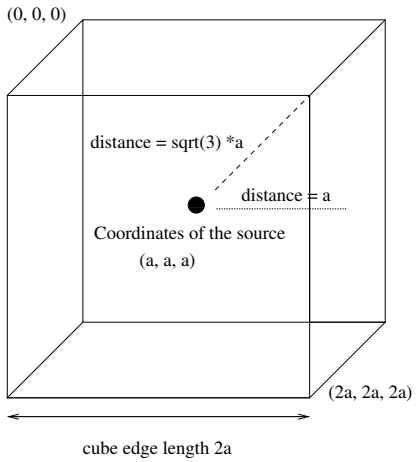


Figure 5.6 Distance from the source point to different points on the model boundary.

5.4.1 Mesh Initialization

The first step of the pWFC algorithm is to create a mesh. The time spent in this phase is dependent only on the initial mesh size and the number of processors. For this reason, we present the running time independent of the earth model. We run experiments with different mesh sizes 320 x 64 and 920 x 192 to see the performance characteristics with the varying problem size. Figure 5.8 presents the time spent to initialize different mesh sizes. Figure 5.7 provides strong scaling results for the initialization phase. From the plots, we can observe that we achieve better scalability for a mesh size of 960 x 192 mesh size than for 320 x 64 mesh. As the problem size is increased, the overhead of the parallelism becomes less significant and leads to improved scalability.

5.4.2 Wavefront Propagation

In this section, we study the performance of the wavefront propagation phase of the pWFC algorithm. Wavefront propagation is dependent on a wide range of factors. In this section, we present effect of these factors in isolation.

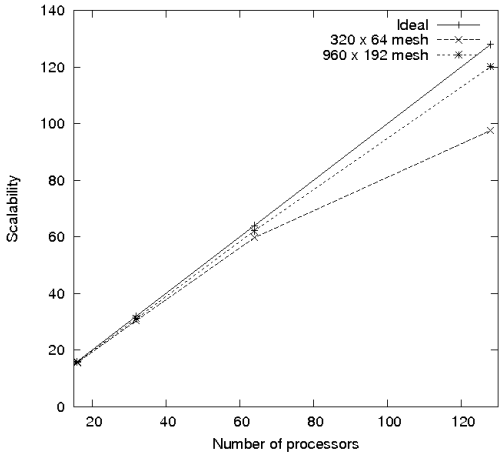
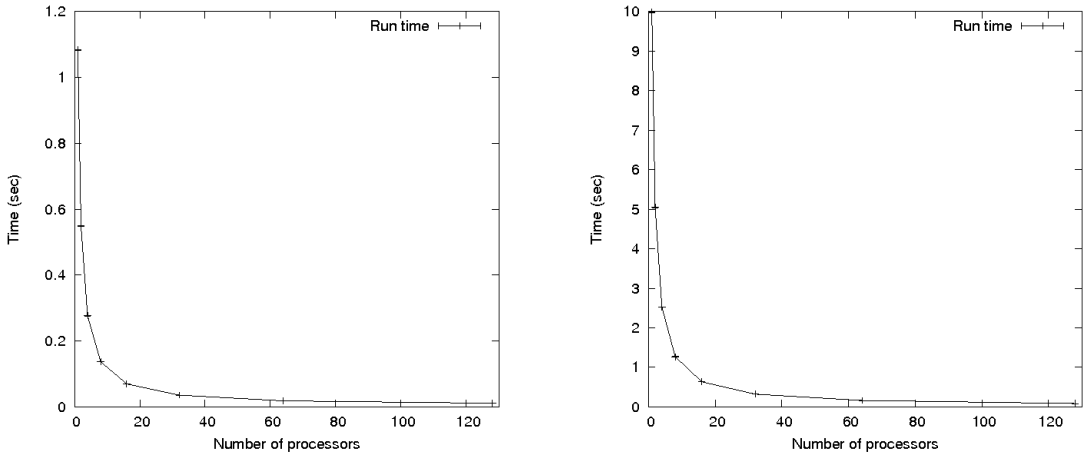


Figure 5.7 Strong scaling of initialization phase.



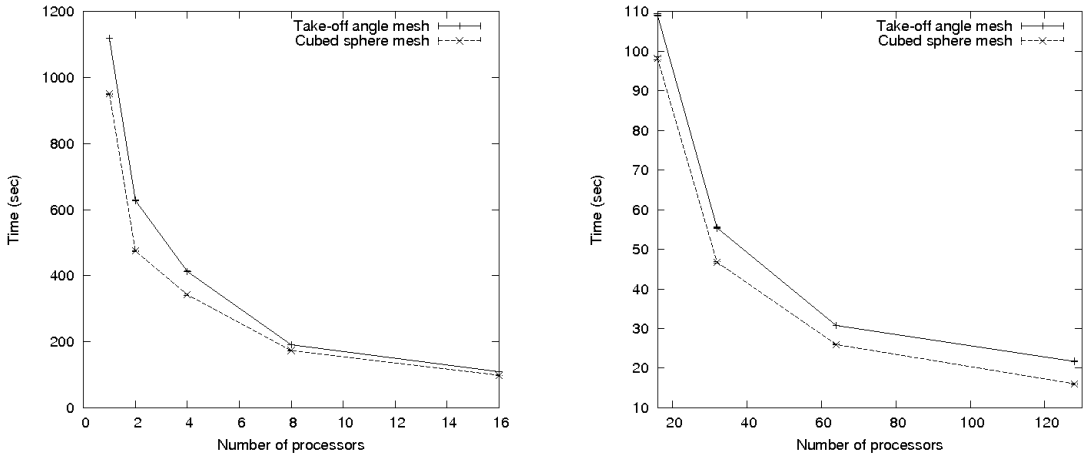
(a) 320 x 64 mesh size (b) 960 x 192 mesh size

Figure 5.8 Execution time for the initialization phase.

5.4.2.1 Mesh Type

First, we study how different initial mesh types may affect the performance of the wavefront propagation. The mesh initialization method describes the geometric distribution of the rays. A better distribution should result in lower execution time

while ensuring the desired numerical accuracy. Here, we compare the performance of a take-off angle mesh and a cubed sphere mesh. We start with same number of rays in homogeneous model with anisotropic medium. For the take-off angle mesh, we generate 256 rays in the azimuth direction from [-180°, 180°] and 96 rays in the declination direction from [-89.9°, 89.9°]. For the cubed sphere mesh, we generate a 64 x 64 mesh for all six faces of the cube. As shown in Figure 5.9, the wavefront initialized with a cubed sphere mesh takes less time to propagate in the model than the take-off angle mesh. The rays are concentrated near the poles of the sphere for the take-off angle mesh which makes the rays farther apart near the horizontal. Thus more interpolation is needed near the horizontal for take-off angle mesh, making it more load unbalanced. Hence, for the rest of our experiments, we use a cubed sphere mesh for initializing the mesh.

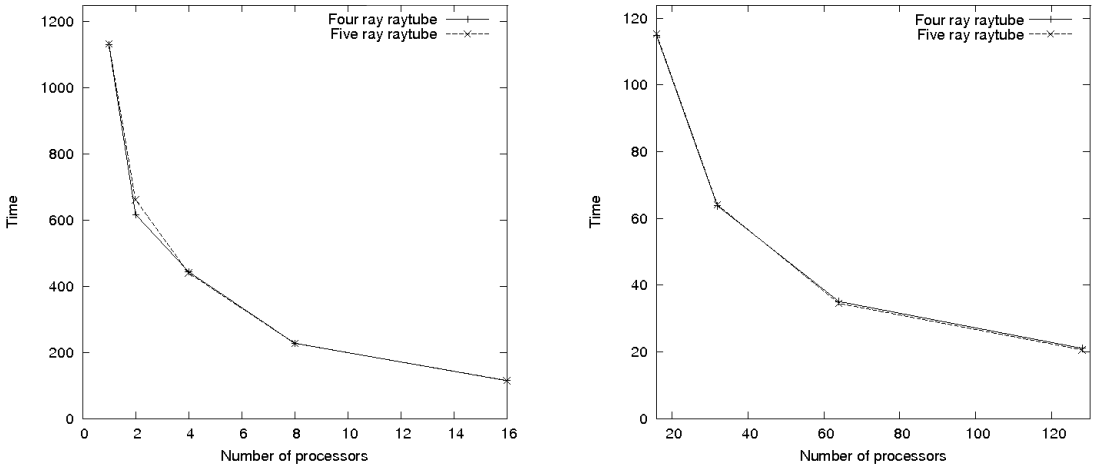


(a) p = 1 to 16 (b) p = 16 to 128

Figure 5.9 Execution time varying with the mesh type.

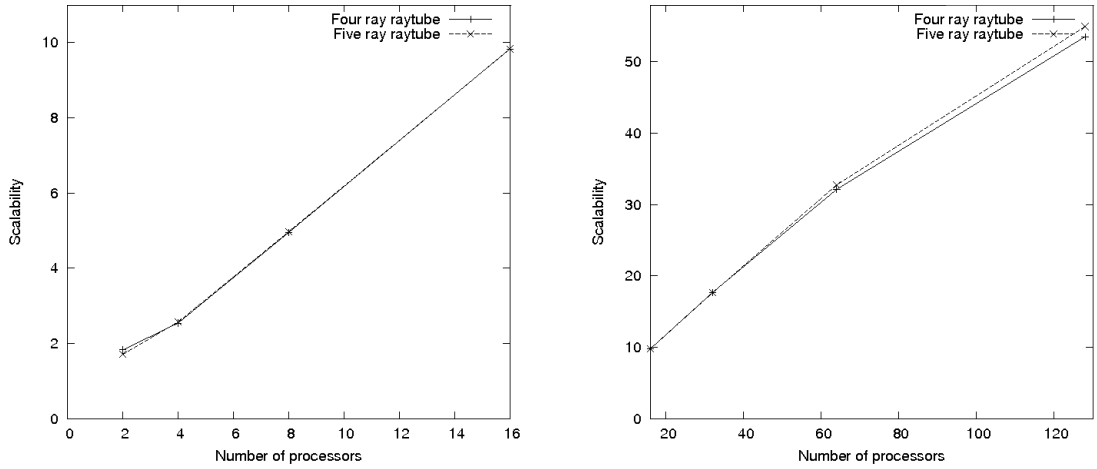
5.4.2.2 Ray Tube Type

Second, we study the effect of using ray tubes with five rays and four rays. For this we use homogeneous model with anisotropic medium. We generate one cubed sphere mesh (320 x 64) positioned at (5, 4.5, 4.5). Figures 5.10 and 5.11 present the execution time and scalability for the propagation phase, respectively. Note that an additional ray is created for each 5-ray ray tube created. This might imply that more time would be required for the mesh to propagate. However, the time taken is almost the same for 4-ray and 5-ray ray tubes (see Figure 5.10). The reason for this is that ray tubes with an additional ray have relatively lower numerical errors than ray tubes with four rays. This reduces the number of ray tube interpolations which reduces the time spent propagating the wavefront.



(a) p = 1 to 16 (b) p = 16 to 128

Figure 5.10 Execution time varying with ray tube type.



(a) p = 1 to 16 (b) p = 16 to 128

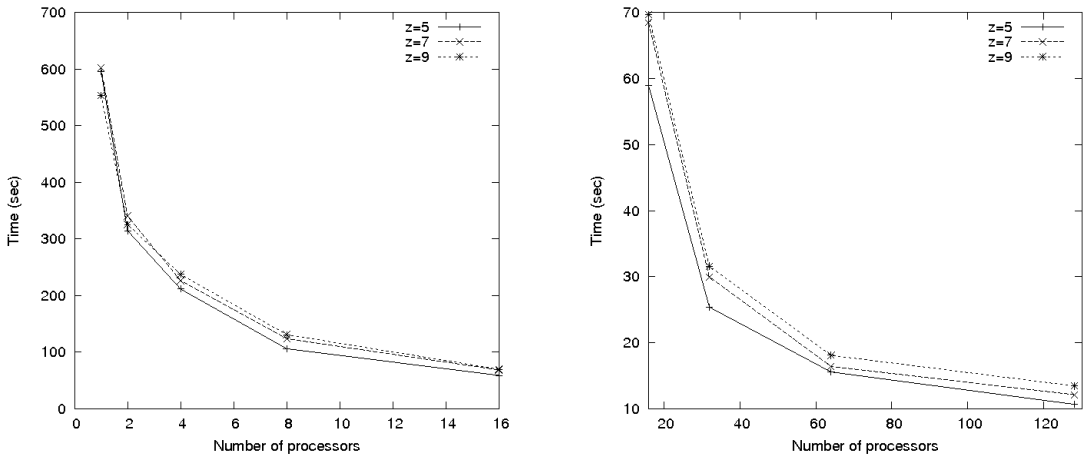
Figure 5.11 Strong scaling for different ray tube type.

5.4.2.3 Position of Source

Next, we study the effect of the position of the source on the performance of the parallel algorithm. We ran experiments on a single region model with no interpolation or coarsening. The position of the source is kept at the center (5, 5, 5) of the model and gradually moved towards a boundary (5, 5, 7) and (5, 5, 9).

Figure 5.12 presents the execution time varying with the position of the source. The execution time is minimum for the source at z=5 and maximum for the source at z=9. The scalability for these different cases is shown in Figure 5.13. As the source is moved away from the center, the scalability reduces. This is based on the fact that load imbalance increases as the source is moved closer to the model boundary.

To study the load imbalance we collected the number of ray tube processed by each processor in each iteration. Load profile shows mean, minimum and the maximum number of ray tubes processed by a processor in an iteration. Processors becomes idle when the minimum number of ray tubes processed by it is equal to



(a) p = 1 to 16 (b) p = 16 to 128

Figure 5.12 Execution time varying with the position of the source.

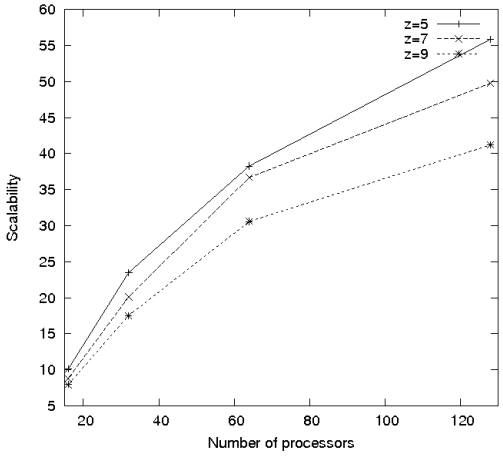
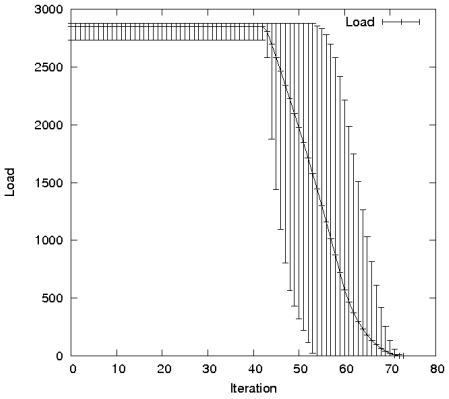


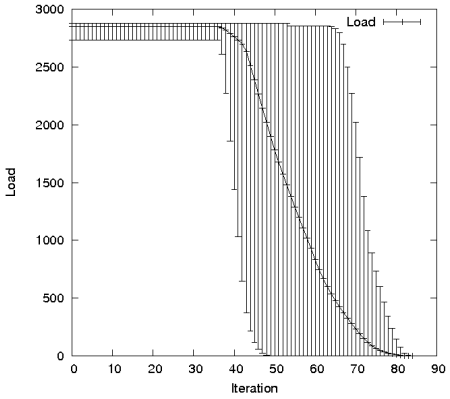
Figure 5.13 Scalability varying with the position of the source.

zero. Figure 5.14 shows the load profiles for various cases. Initially the load is balanced among the processors. The load start to decrease when ray tubes reach the model boundary. Ray tubes reach the model boundary after (a) 41 iterations for source at (5, 5, 5), (b) 36 iterations for source at (5, 5, 7), and (c) 15 iterations for source at (5, 5, 9). Thus, the load is more balanced when the source is at the cen-

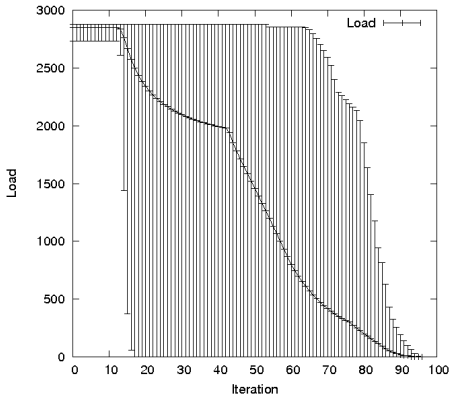
ter of the model. As the imbalance increases, the performance is adversely affected motivating the need of load balancing techniques.



(a) Source position = (5, 5, 5)



(b) Source position = (5, 5, 7)



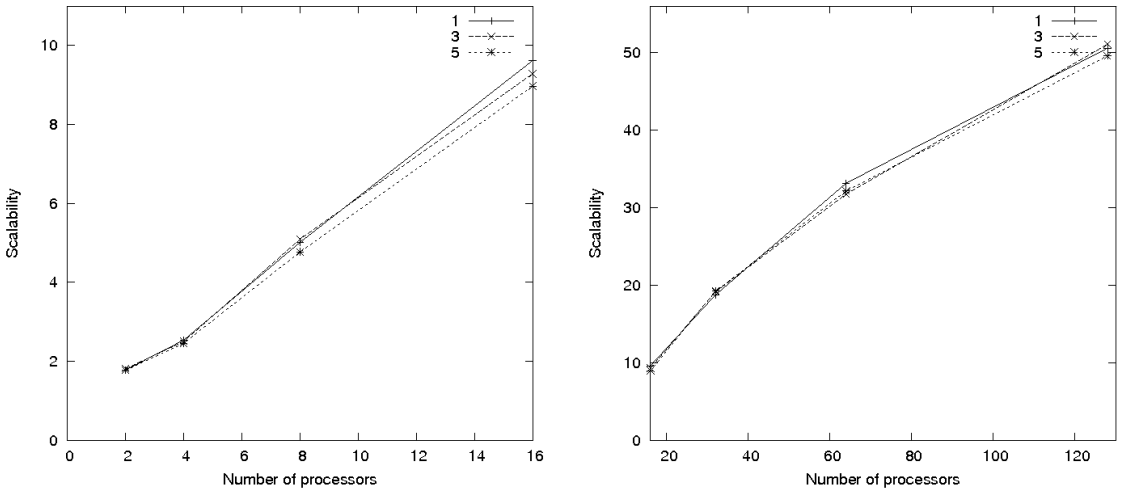
(c) Source position = (5, 5, 9)

Figure 5.14 Load profiles for different positions of the source in the model (number of processor p = 64). Initially the load is balanced among the processors. The load start to decrease when ray tubes reach the model boundary.

5.4.2.4 Number of Sources

The pWFC algorithm also supports using different number of sources. Creating an initial mesh for different sources is independent of each other (completely paral-

lel.) Also working with different number of sources, we initialize earth model and other data structures once. For this experiment we use homogeneous model with anisotropic medium. We generate one, three and five cubed sphere mesh (320 x 64). Note that the number of sources that can be used in an experiment is limited by the memory available for each core. The scalability results for the experiment are presented in Figure 5.15. The graph clearly shows that the pWFC algorithm provides good scalability even when we use different number of sources.



(a) p = 1 to 16 (b) p = 16 to 128

Figure 5.15 Strong scaling for varying number of sources.

5.4.2.5 Material Type

Next, we show the effect of material properties on the pWFC algorithm. For this experiment we use a single region model with one source positioned at (5, 4.5, 4.5). Material properties influence interpolation and coarsening of ray tubes which directly leads to load imbalance. We use a single region model with three different material types namely (a) isotropic medium, (b) weak anisotropic medium, and (c)

strong isotropic medium. Figure 5.16 presents the scalability results for the different material properties.

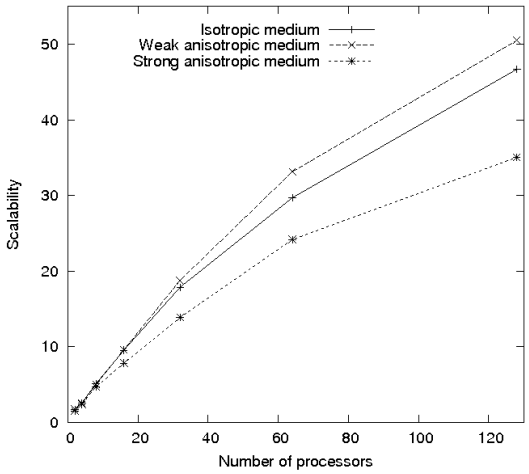
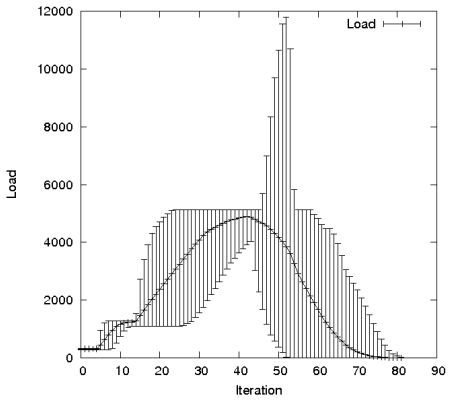


Figure 5.16 Scalability varying with the material type.

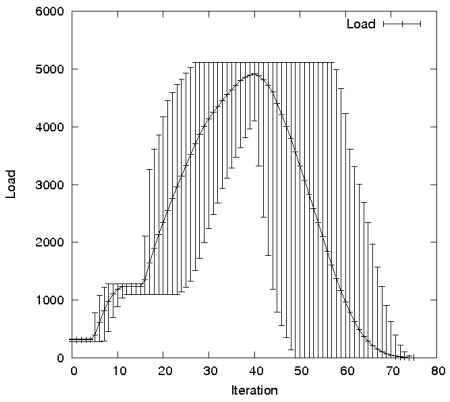
The load profile for the three material types is shown in Figure 5.17. For isotropic medium the wavefront is a sphere and for anisotropic medium the wavefront is more curved (velocity of the ray is dependent on its direction in anisotropic medium). Because of this some rays may be farther apart from their neighboring rays than others. Thus more interpolations may be required for anisotropic medium. Figure 5.17(c), shows that there is severe load imbalance in strong anisotropic medium motivating the future work on using load balancing techniques.

5.4.2.6 Multi-Region Models

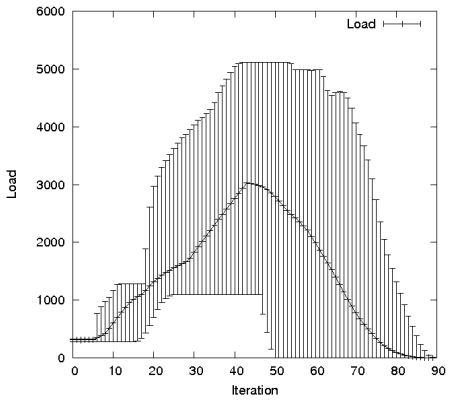
In our previous experiments, we had focused on factors that may affect the performance of the pWFC algorithm. We now focus on using models that have multiple regions separated by surfaces. For the experiment we created one cubed sphere mesh positioned at (6, 5, 5.9) for Model 4, at (6.9, 6.4, 8.6) for Model 5 and at (5, 4.5, 4.5)



(a) Isotropic medium



(b) Weak anisotropic medium



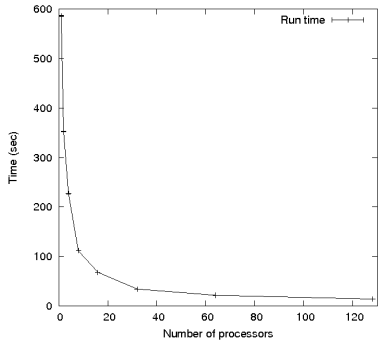
(c) Strong anisotropic medium

Figure 5.17 Load profiles for different material types in a model (p=64).

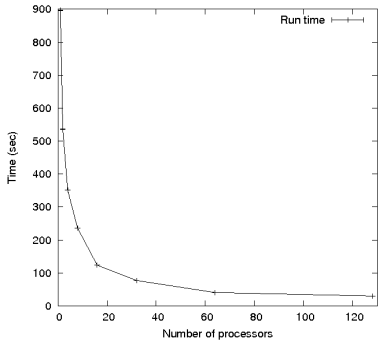
for Models 1-3. Figure 5.18 presents the execution time for the propagation phase for the various earth models.

Strong scaling results for different models and varying initial mesh size are presented in Figure 5.19. To help us explain these results, we also provide the observed load varying with the iterations of the algorithm (the number of ray tube segments processed by a location) for p = 64 (see Figure 5.20).

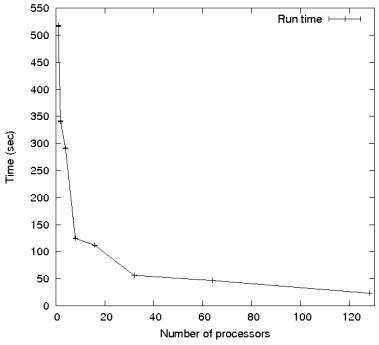
First, let us consider Model 2: Layered model. As shown in Figure 5.20(a), after the first few iterations, the load on the processors starts to increase because of



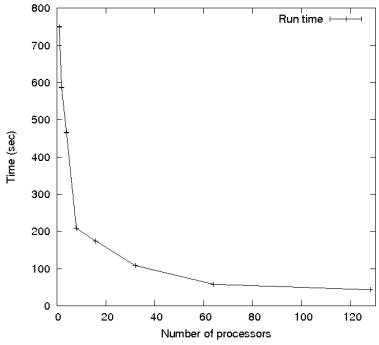
(a) Model 2: Layered model



(b) Model 3: Cylindrical inclusions model

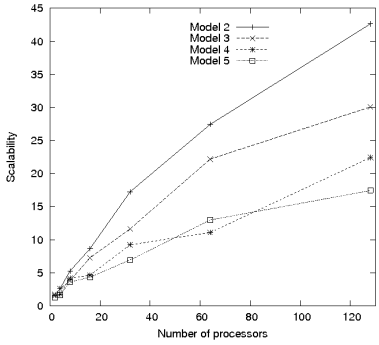


(c) Model 4: Salt canopy model

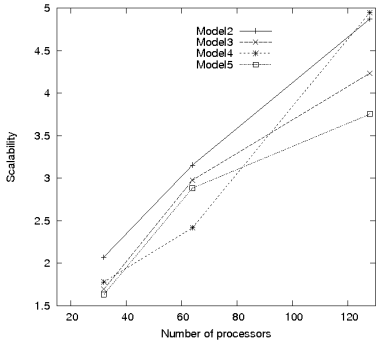


(d) Model 5: Salt dome model

Figure 5.18 Execution time for different models with surfaces.

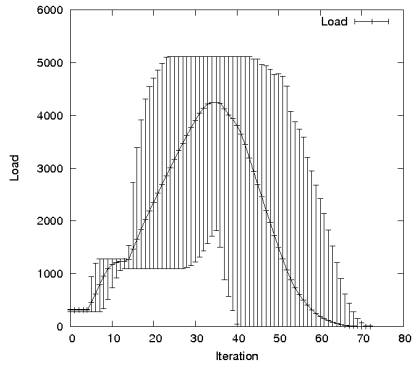


(a) 320 x 64 mesh size

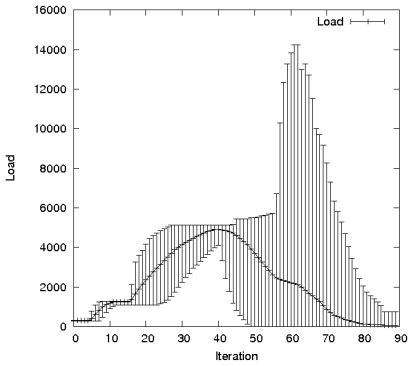


(b) 960 x 192 mesh size

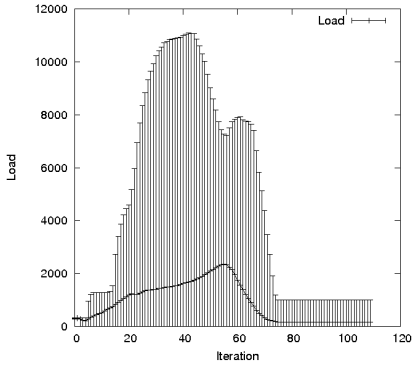
Figure 5.19 Strong scaling for different initial mesh sizes.



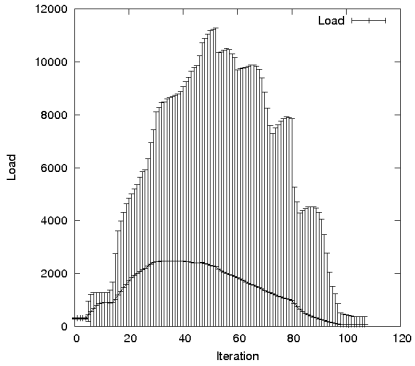
(a) Model 2: Layered model



(b) Model 3: Cylindrical inclusions model



(c) Model 4: Salt canopy model



(d) Model 5: Salt dome model

Figure 5.20 Load profiles for different models with surfaces (p=64).

interpolation of the ray tubes. The load imbalance is present in all further iterations. Around the 40^{th} iteration some of the rays reach the model boundary and locations start to become idle, thus worsening load imbalance. Model 3: Layered model (Figure 5.20(b)) has behavior similar to Model 2. However, there is more load imbalance in this model. Also, after a few ray tubes reach the model domain, some other parts of mesh interpolate, increasing the load imbalance. Model 4: Salt canopy and Model 5: Salt dome have the source near the boundary. For this reason, processors become idle after the first few iterations. Also for these two models, many rays terminate at the region boundaries. This further increases the load imbalance across the processors.

All these factors adversely affect the scalability of the algorithm.

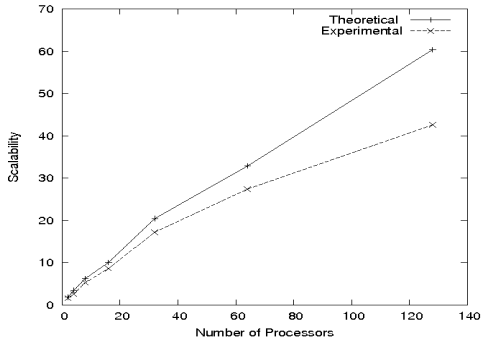
5.5 Comparison with Theoretical Model

In the previous section, we presented the scalability plots for different models and load profiles for (p = 64). These plots indicated that load imbalance is adversely affecting the performance of the parallel algorithm. In this section, we show that load imbalance is a major cause for the performance loss. To prove this, we use the theoretical model presented in section 3.5. We claim that if the nature of the two scalability plots (experimental and theoretical scalability with load imbalance) matches, then load imbalance is the main factor adversely affecting the performance of the proposed parallel algorithm. In our theoretical model, the only factor that can adversely affect the performance is load imbalance. Hence, if the nature of the graphs matches, then our theoretical model is correct.

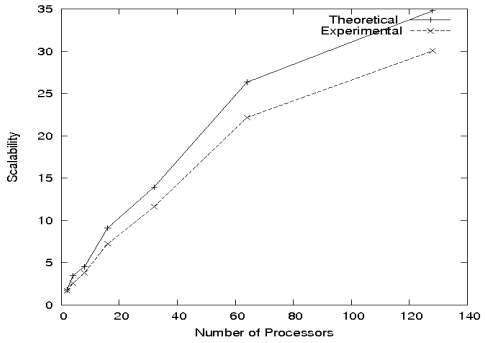
Figure 5.21 provides the graphs comparing the theoretical and the experimental scalability for different earth models. In all the graphs, we can observe that the nature of the two scalability plot matches. For plots (a) and (b), we observe that the theoretical scalability is greater than the observed scalability. In these cases, the mesh was refined gradually over the iterations and only a few processors had refined their mesh to the observed maximum interpolation level (l_{ex}). Also with the refinement of the mesh, the ratio of computation to communication increases (area to perimeter) which increases scalability. Thus, the gap between the observed and theoretical scalability. For the plots (c) and (d) not only the shape but also the scalability values almost match. In these two geometries, the mesh was rapidly refined in the first few iterations and many more processors worked with the mesh at the observed maximum interpolation level.

Thus, based on the plots provided in Figure 5.21 we were able to provide some

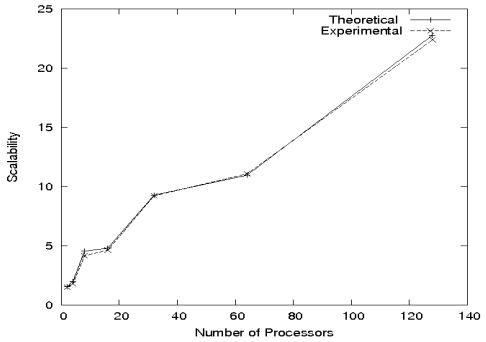
evidence that load imbalance is a major cause for the performance loss.



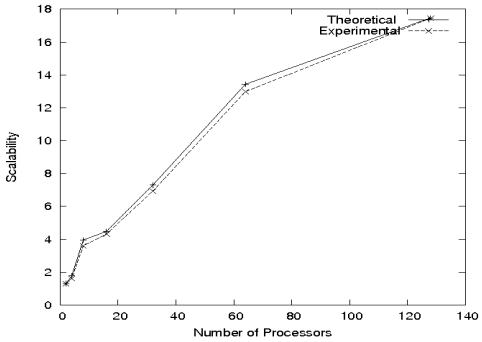
(a) Model 2: Layered model



(b) Model 3: Cylindrical inclusions model



(c) Model 4: Salt canopy model



(d) Model 5: Salt dome model

Figure 5.21 Comparison of experimental scalability and theoretical scalability with load imbalance.

6. CONCLUSION AND FUTURE WORK

In this thesis, we presented a parallel wavefront construction algorithm. We described the design and implementation of the parallel algorithm using STAPL. The parallel algorithm provides dynamic addition and removal of rays and ray tubes. We also introduced the concept of a ray tube with additional ray in the center, which was seen to reduce communication and improve numercial accuracy. Modifications are also provided to initialize the mesh for the two most widely used geometries.

In our experimental study we investigated the performance of the proposed parallel wavefront construction algorithm on a wide range of earth models. We study the effect of various factors such as the position of the source and material properties on the performance of the parallel algorithm. We also investigate the performance of the parallel algorithm on some of the widely used model such as the salt dome model. The various phases are shown to provide good parallel performance. Using our theoretical model, we were able to show that load imbalance is a major reason for the performance loss.

In the future, load balancing should be used to improve the scalability of the parallel algorithm. Load balancing algorithms can be generalized and provided as the part of STAPL to improve the parallel performance. Finally, the proposed parallel algorithm and its implementation solves the seismic ray tracing problem. It would be interesting to investigate if the implementation can be generalized to solve other problems such as seismic tomography etc.

REFERENCES

[1] H. L. Lai, R. Pearce, S. Rodriguez, N. M. Amato, and R. L. Gibson, Algorithms & Applications Group Seismic Ray Tracing., "http://parasol.tamu.edu/dsmft/research/seismic/," Parasol Lab, Texas A&M University, 2009.

[2] W. Beydoun and M. Mendes, "Elastic ray-born l2-migration/inversion," *Geophysics*, vol. 97, pp. 151–160, 1989.

[3] D. Miller, M. Oristaglio, and G. Beylkin, "A new slant on seismic imaging: migration and integral geometry," *Geophysics*, vol. 52, pp. 943–964, 1989.

[4] G. Beylkin and R. Burridge, "Linearized inverse scattering problems in acoustics and elasticity," *Wave Motion*, vol. 12, pp. 15–52, 1990.

[5] D. Lumley, "Angle-dependent reflectivity estimation," *SEG Expanded Abstracts*, vol. 12, pp. 746–749, 1993.

[6] E. Forgues, G. Lambaré, P. de Beukelaar, and V. Richard, "An application of ray + born inversion on real data," *SEG Expanded Abstracts*, vol. 13, pp. 1004–1007, 1994.

[7] H. Jaramillo and N. Bleistein, "The link of kirchhoff migration and demigration to kirchhoff and born modelling," *Geophysics*, vol. 64, pp. 1793–1805, 1999.

[8] J. Zhang and M. Toksoz, "Nonlinear refraction traveltime tomography," *Geophysics*, vol. 63, pp. 1726–1737, 1998.

[9] K. W. Morton and D. F. Mayers, *Numerical Solution of Partial Differential Equations, An Introduction*. New York, NY, USA: Cambridge University Press, 2005.

[10] J. Virieux, "P-sv wave propagation in heterogeneous media: Velocity-stress finite-difference method," *Geophysics*, vol. 51, pp. 889–901, 1986.

[11] A. R. Levander, "Fourth-order finite-difference p-sv seismograms," *Geophysics*, vol. 53, pp. 1425–1436, 1988.

[12] K. J. Lee, *Efficient ray tracing algorithms based on wavefront construction and model based interpolation method*. PhD thesis, Texas A&M University, 2005.

[13] V. Červený, "Seismic rays and ray intensities in inhomogeneous anisotropic media," *Geophysical Journal International*, vol. 29, pp. 1–13, 1972.

[14] V. Červený, *Seismic Ray Theory*. Cambridge, UK: Cambridge University Press, 2001.

[15] V. Vinje, E. Iversen, and H. G. ystdal, "Traveltime and amplitude estimation using wavefront construction," *Geophysics*, vol. 58, pp. 1157–1166, 1993.

[16] G. Lambaré, P. S. Lucio, and A. Hanyga, "Two-dimensional multivalued traveltime and amplitude maps by uniform sampling of a ray field," *Geophysical Journal International*, vol. 125, pp. 584–598, 1996.

[17] P. S. Lucio, G. Lambaré, and A. Hanyga, "3d multidimensional travel time and amplitude maps," *Pure and Applied Geophysics*, vol. 148, pp. 449–479, 1996.

[18] N. Ettrich and D. Gajewski, "Wave front construction in smooth media for prestack depth migration," *Pure and Applied Geophysics*, vol. 148, pp. 481–502, 1996.

[19] V. Vinje, K. Åstebøl, E. Iversen, and H. G. ystdal, "3-d ray modeling by wavefront construction in open models," *Geophysics*, vol. 64, pp. 1912–1919, 1999.

[20] R. L. Gibson, "Ray tracing by wavefront construction in 3-D, anisotropic media," *Eos Transactions, American Geophysical Union*, vol. 80, p. F696, 1999.

[21] H. G. ystdal, E. Iversen, I. Lecomte, V. Vinje, and K. Åstebøl, "Review of ray theory applications in modeling and imaging of seismic data," *Studia Geophysica et Geodaetica*, vol. 46, pp. 113–164, 2002.

[22] R. L. Gibson, V. D. Durussel, and K. J. Lee, "Modeling and velocity analysis with a wavefront construction algorithm for anisotropic media," *Geophysics*, vol. 70, pp. T63–T74, 2005.

[23] J. Mispel and P. Williamson, "3-D wavefront construction for P & SV waves in transvesely isotropic media," in *63rd Conference & Technical Exhibition, European Association of Geoscientists and Engineers (EAGE)*, p. P094, 2001.

[24] K. J. Lee and R. L. Gibson, "Numerical properties of cubed sphere meshes in wavefront construction," in *73rd Internat. Meeting Abstract, Soc. Expl. Geophys.*, pp. 1793–1796, 2003.

[25] T. Kaschwich and D. Gajewski, "Wavefront-oriented ray tracing in 3-D anisotropic media," in *65rd Conference & Technical Exhibition, European Association of Geoscientists and Engineers (EAGE)*, p. P041, 2003.

[26] R. Coman and D. Gajewski, "Estimation of multivalued arrivals in 3-D models using wavefront ray tracing," in *71st Internat. Meeting Abstract, Soc. Expl. Geophys.*, pp. 1265–1268, 2001.

[27] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library (stapl)," *Lecture Notes in Computer Science*, vol. 1511, pp. 402–413, 1998.

[28] P. An, A. Jula, S. Rus, S. Saunders, T. G. Smith, G. Tanase, N. L. Thomas, N. M. Amato, and L. Rauchwerger, "STAPL: A standard template adaptive parallel C++ library," in *Proc. of the International Workshop on Advanced*

Compiler Technology for High Performance and Embedded Processors (IWACT), (Bucharest, Romania), Jul 2001.

[29] P. An, A. Jula, S. Rus, S. Saunders, T. G. Smith, G. Tanase, N. L. Thomas, N. M. Amato, and L. Rauchwerger, "STAPL: An adaptive, generic parallel programming library for C++," in *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, vol. 2624, (Cumberland Falls, KY, USA), Aug 2001.

[30] N. L. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, (Chicago, IL, USA), pp. 277–288, ACM, 2005.

[31] N. L. Thomas, S. Saunders, T. G. Smith, G. Tanase, and L. Rauchwerger, "ARMI: A high level communication library for STAPL," *Parallel Processing Letters*, vol. 16(2), pp. 261–280, 2006.

[32] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger, "Associative parallel containers in STAPL," in *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, vol. 5234, (Urbana-Champaign, IL, USA), pp. 156–171, 2008.

[33] A. A. Buss, T. G. Smith, G. Tanase, N. L. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "Design for interoperability in STAPL: pMatrices and linear algebra algorithms," in *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, vol. 5335, (Edmonton, Alberta, Canada), pp. 304–315, July 2008.

[34] G. Tanase, X. Xu, A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, N. L. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger,

"The STAPL pList," in *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, vol. 5898, (Wilmington, DE, USA), pp. 16–30, October 2009.

[35] A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, G. Tanase, N. L. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library," in *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, (New York, NY, USA), pp. 1–10, ACM, 2010.

[36] M. Grunberg, S. Genaud, and C. Mongenet, "Parallel seismic ray tracing in a global earth model," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 1151–1157, 2002.

[37] M. Grunberg, S. Genaud, and C. Mongenet, "Seismic ray-tracing and earth mesh modeling on various parallel architectures," *The Journal of Supercomputing*, vol. 29, pp. 27–44, 2004.

[38] J. Gazdag and P. Sguazerro, "Migration of seismic data," *Proc. IEEE*, vol. 72, pp. 1302–1315, 1984.

[39] J. Gazdag and P. Sguazerro, "Migration of seismic data by phase shift plus interpolation," *Geophysics*, vol. 49, pp. 124–131, 1984.

[40] V. K. Madisetti and D. G. Messerschmitt, "Seismic migration algorithms on parallel computers," in *Proceedings of the third conference on Hypercube concurrent computers and applications*, pp. 1180–1186, 1988.

[41] L. Jian, L. Yingjun, M. Xiaoxing, C. Min, T. Xianping, Z. Guanqun, and L. Jianzhong, "A hierarchical framework for parallel seismic applications," *Commun. ACM*, vol. 43, pp. 55–59, 2000.

[42] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series, 1994.

[43] H. Calandra, F. Bothorel, and P. Vezolle, "A massively parallel implementation of the common azimuth pre-stack depth migration," *IBM J. Res. Dev.*, vol. 52, pp. 83–91, 2008.

[44] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[45] P. Micikevicius, "3d finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, 2009.

[46] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu, "Fast seismic modeling and reverse time migration on a GPU cluster," in *High Performance Computing & Simulation*, pp. 36–44, 2009.

[47] D. Komatitsch, G. Erlebacher, D. Gddeke, and V. Richard, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, pp. 7692–7714, 2010.

[48] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood, "Reverse time migration," *Geophysics*, vol. 48, pp. 1514–1524, 1983.

[49] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide, Second Edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[50] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," in *International Conference on Computer Graphics and Interactive Techniques*, pp. 110–116, 1980.

[51] H. W. Ghorst, G. Hooper, and D. P. Greenberg, "Improved computational methods for ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 3, no. 1, pp. 52–69, 1984.

[52] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 15–22, 1984.

[53] J. G. Cleary, B. Wyvill, G. M. Birtwistle, and R. Vatti, "Multiprocessor ray tracing," *Computer Graphics Forum*, vol. 5, no. 1, pp. 3–12, 1986.

[54] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated ray-tracing system," *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16–26, 1986.

[55] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[56] C. L. Jackins and S. L. Tanimoto, "Oct-trees and their use in representing three-dimensional objects," *Comp. Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

[57] R. Lu, M. E. Willis, X. H. Campman, and M. N. Toksoz, "Evaluation of elastodynamic interferometric redatuming: a synthetic study on salt dome flank imaging," *Geophysical Journal International*, vol. 176, pp. 889–896, 2009.

[58] J. Arneson, "Endangered earth sinkholes lake peigneur: the swirling vortex of doom. http://www.thelivingmoon.com/45jack_files/03files/endangered_earth_sinkhole_louisiana.html," Blue Knight Production, 2005.

[59] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[60] Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in homogeneous multiple processor systems," *IEEE Transactions on Computers*, vol. c-36, pp. 667–679, 1982.

[61] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *IEEE International Parallel and Distributed Processing Symposium*, 2008.

[62] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE Transactions on Software Engineering*, vol. 13, pp. 32–38, 1987.

[63] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 7, pp. 279–301, 1989.

[64] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," *ACM SIGPLAN Notices*, vol. 28, pp. 91–108, 1993.

APPENDIX A

LOAD BALANCING

In the pWFC algorithm the interpolation and coarsening of the wavefront may lead to load imbalance. Load imbalance exists whenever there is an non-uniform distribution of work among processors. This leads to some processors being overloaded with work while other under loaded processors may remain idle. Load balancing is a generalization of the multiprocessor scheduling problem and is known to be NP-complete [59]. The load balancing problem has been widely studied. Dynamic load balancing methods can be classified into two main categories: centralized [60, 61] and distributed [62, 63]. In centralized schemes, the load information of the various processors is collected at one processor which decides how to distribute (load balance) the load among the processors. The major disadvantage of centralized schemes is that the processor that makes the load distribution decisions becomes a potential bottleneck. Distributed methods are more scalable than centralized methods. For example, in a distributed method, processors might communicate only with the neighbors and diffuses fractions of workload to underloaded neighbors and receive workload from overloaded neighbors. Global balance is achieved by successive migration of workload from overloaded processors to underloaded processors. Distributed schemes employ the neighboring information to redistribute the load between adjacent processors, thereby requiring multiple diffusive steps to achieve global load balance. Distributed schemes generally provide better results for the problems in which imbalance occurs globally throughout the computational domain, while centralized schemes may be advantageous to the problems in which high magnitude imbalance occurs in localized regions.

Figure A.1 presents one possible instance of load imbalance for the pWFC algorithm. The major issues to be addressed are the identification of the overloaded and underloaded processors, the quantity of data to be transferred from overloaded processors to underloaded processor and which transfer to make. Some important issues include how to (i) measure the load on the on a processor and (ii) determine if there is a load imbalance. Once load imbalance has been identified, we need to determine how much, what part of the work load should be moved and to where in order to achieve better load balance. Another important consideration while designing the load balancing algorithm is the granularity of the work load. We chose a coarse grain approach for determining the load with each ray tube segment accounting for a unit of load. This approach has a trade-off that we can update and determine the load on each processor quickly and thus reduce the overhead of the load balancing algorithm. However, the decision to move the load may be sub-optimal. With fine grain accounting (increased memory and time overhead), a better decision could be made.

In this work we study two load balancing strategies: one centralized and one decentralized. The centralized method employs one of the most frequently used strategies for balancing the load known as *GreedyLB* [61] which is also a part of the *Charm++* language [64]. It computes the most overloaded and the most underloaded processor and balances the load between them. For the decentralized scheme, we use *Neighborhood load balancing: NeighborLB* [64], in which the load information is only sent to the neighbors. A processor is overloaded if its load is greater than the average load of its immediate neighbors. This prevents an underloaded processor from transferring its load to its neighbors. The amount of the load that is transferred is directly proportional to the difference of the load between the two processors.

Once the overloaded and underloaded processors have been determined, we need

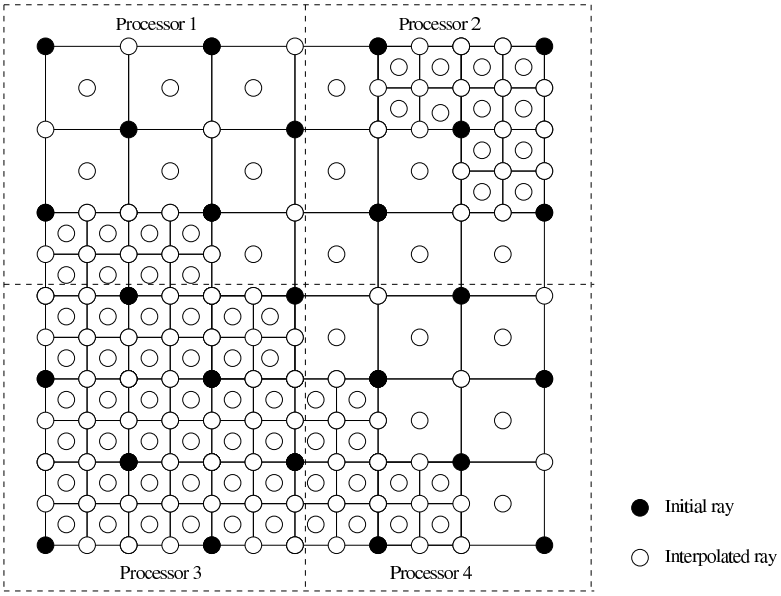


Figure A.1 Possible load imbalance situation in the seismic ray tracing application.

Table A.1 Processor load varying with time for example shown in Figure A.1.

Processor id	**Initial load**	**Current load**
1	6	34
2	5	40
3	7	85
4	6	41

to determine how much data has to be moved and what should be moved. For this we use either of two strategies. Our strategies are based on a commonly used heuristic to preferentially put the migrated load back to the original processor.

1. Strategy 1: If the ratio of the number of the ray tube segments between the overloaded and the underloaded processor $\frac{load(p_{ov})}{load(p_{un})}$ is higher than some preset threshold value t_n, then we need to distribute the load between the processors.

If the currently overloaded processor had received a portion of the wavefront from the currently under loaded processor, then the overloaded processor first transfers ray tubes that were received from the currently underloaded processor. If it cannot find enough load, then it transfers a portion of its load to the currently underloaded processor.

2. Strategy 2: Two different threshold ratios t_p and t_n are used such that $1 < t_p < t_n$. If the load on an overloaded processor p_{ov} is greater than the load on the underloaded processor p_{un} such that $t_p < \frac{load(p_{ov})}{load(p_{un})} < t_n$, then processor p_{ov} only transfers from the load that originally belonged to the current underloaded processor p_{un}. This way, we preferentially send back the load to its home location in case of smaller load imbalance with the aim of reducing the graph cut (reduced communication cost). If load balance imbalance ratio is greater than t_n, the *strategy 1* (in the above paragraph) is used (possibly increasing communication cost for better load balance).

Let us consider two scenarios (a) Two processors 1 and 2 (see Figure A.2 (a)) with current load of 30000 and 35000 respectively with $t_p = 1.1$ and $t_n = 1.3$ ($1.1 < \frac{35000}{30000} < 1.3$) and assume that a part of the mesh was migrated to processor 2 in an attempt to balance load in a previous iteration. (b) two processors 1 and 2 (see Figure A.2 (b)) with current load of 30000 and 35000 respectively with $t_p = 1.1$ and $t_n = 1.3$ ($1.1 < \frac{35000}{30000} < 1.3$).

If we use the *strategy 1* in scenario (a), the load imbalance ratio is less than the threshold and no re-balancing is done. To achieve a balanced distribution, we would consider using a lower threshold value. A lower threshold for *strategy 1* posses a risk of frequent load movement which may lead to increased graph cut (increased communication cost). If we consider *strategy 2* in scenario (a), processor 2 would

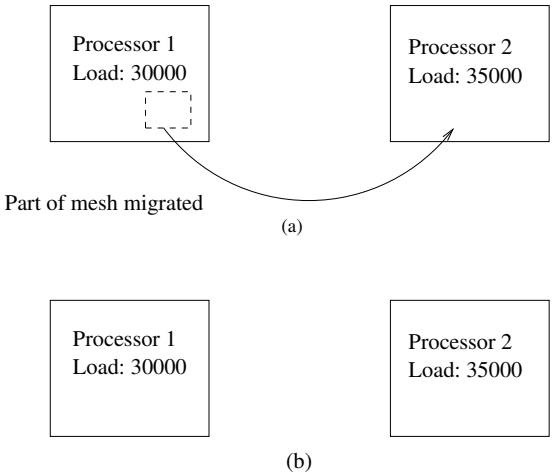


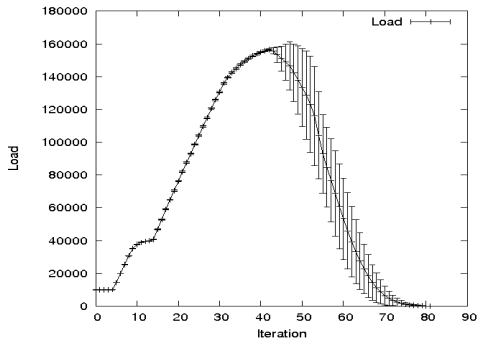
Figure A.2 Load balancing strategy (a) First load imbalance scenario bashed box represents part of mesh migrated to processor 2 in a previous step. The current load on processor 1 and 2 is 30000 and 35000 respectively. (b) Second scenario where the no part of mesh is migrated and current load on processor 1 and 2 is 30000 and 35000 respectively.

return the load (migrated to it from processor 1) back to processor 1 which would improve load balance. Also in scenario (b) the second strategy would not transfer load to processor 1. Thus, based on the two scenarios we can say that *strategy 2* should provide better load balance.

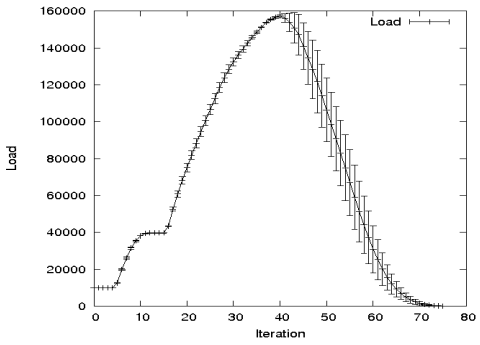
APPENDIX B

LOAD PROFILES

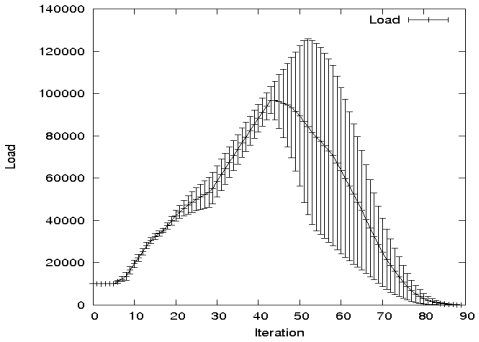
In this section, we present the processor load varying with the iteration of the parallel wavefront construction algorithm. These plots provide the mininum, maximum and average load. Number of processors (p) is provided in the caption of the load profiles.



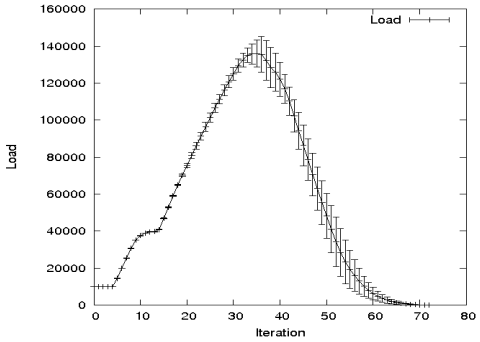
(a) Model 1: Isotropic medium



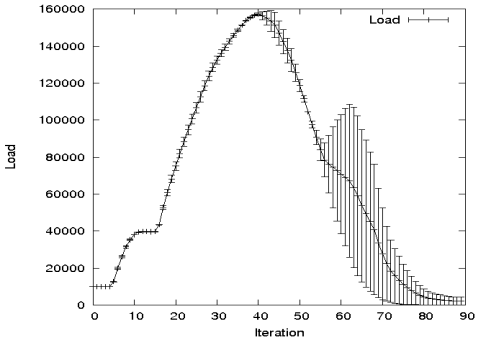
(b) Model 1: Weak Anisotropic medium



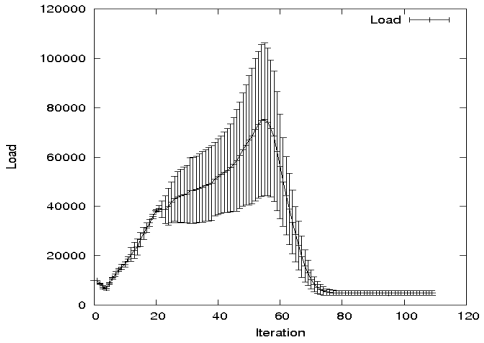
(c) Model 1: Strong Anisotropic medium



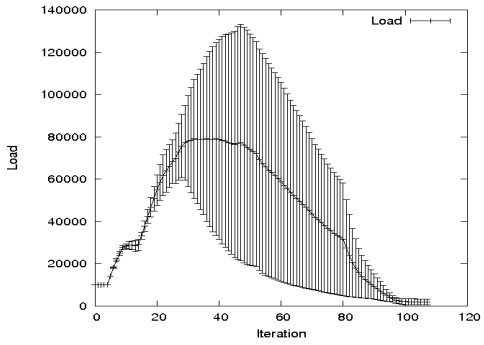
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model

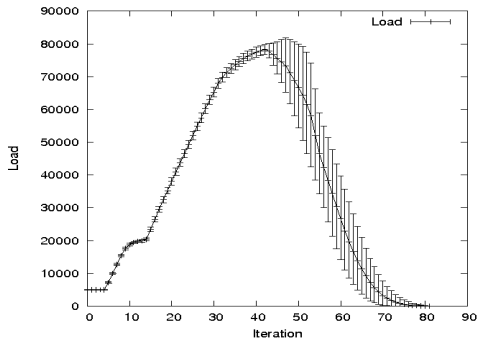


(f) Model 4: Salt canopy model

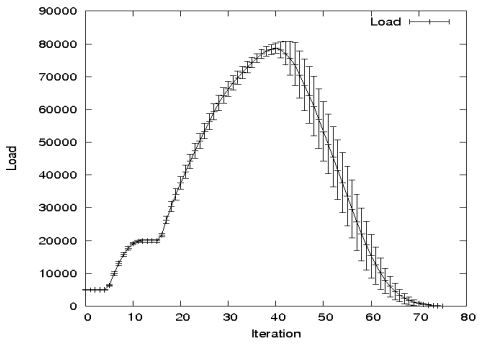


(g) Model 5: Salt dome model

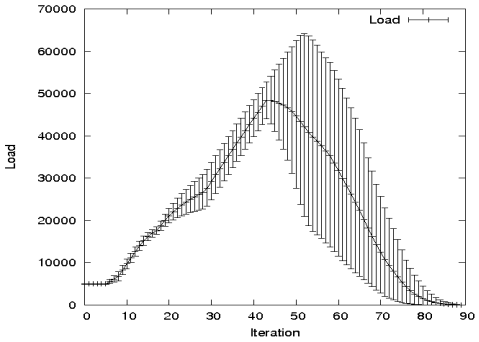
Figure B.1 Processor load varying with the iterations for various earth models (p=2).



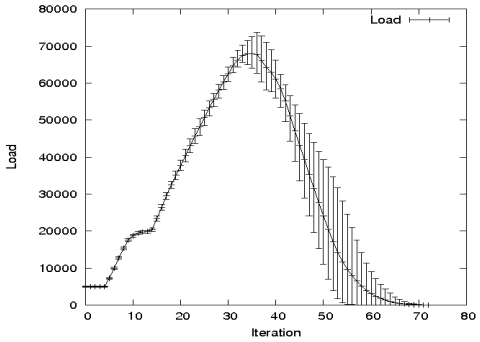
(a) Model 1: Isotropic medium



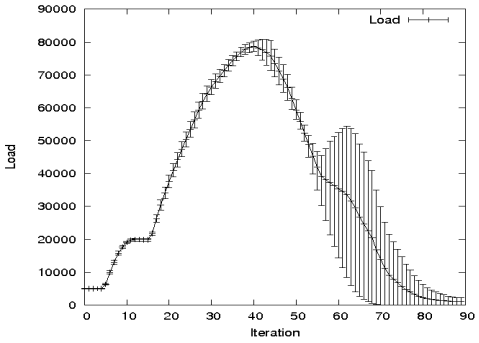
(b) Model 1: Weak Anisotropic medium



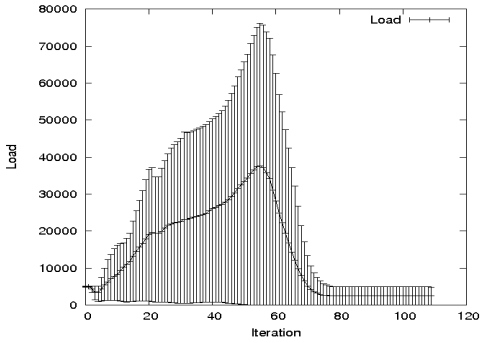
(c) Model 1: Strong Anisotropic medium



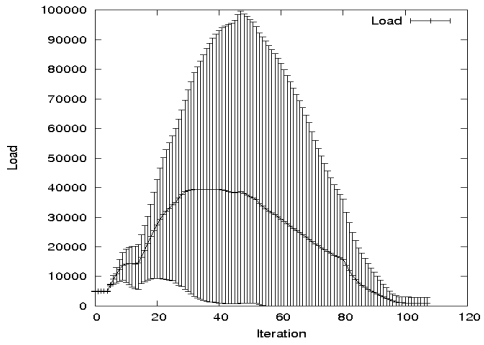
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model

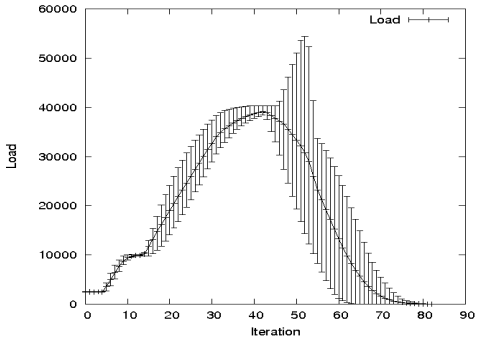


(f) Model 4: Salt canopy model

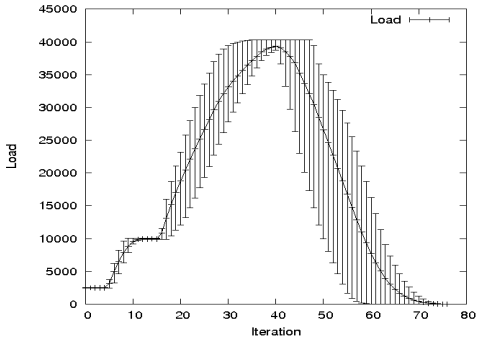


(g) Model 5: Salt dome model

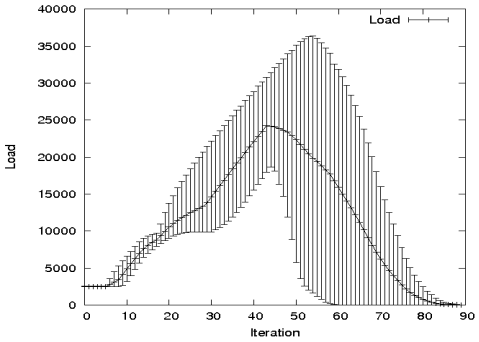
Figure B.2 Processor load varying with the iterations for various earth models (p=4).



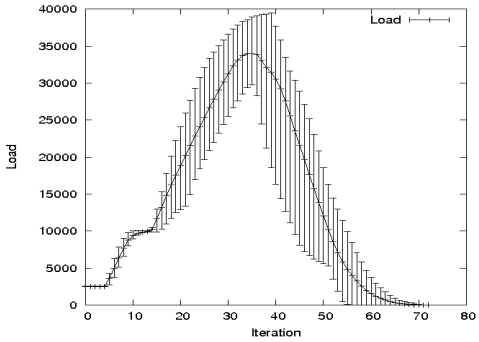
(a) Model 1: Isotropic medium



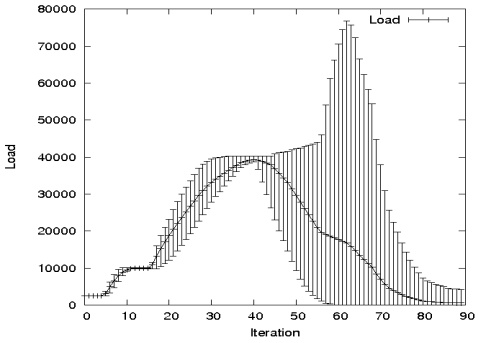
(b) Model 1: Weak Anisotropic medium



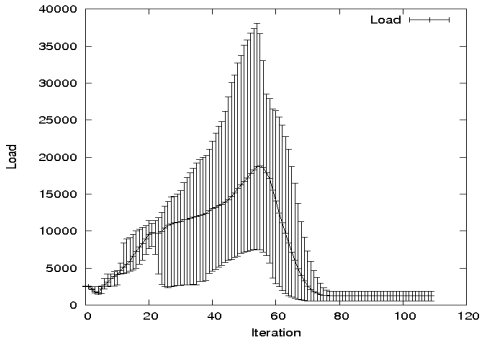
(c) Model 1: Strong Anisotropic medium



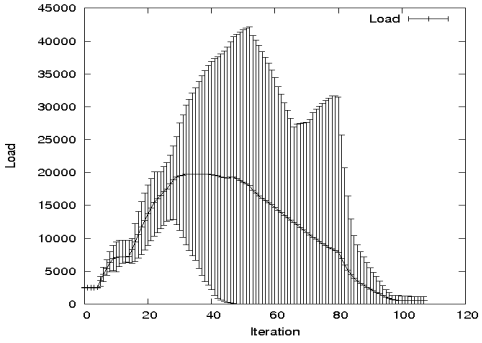
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model

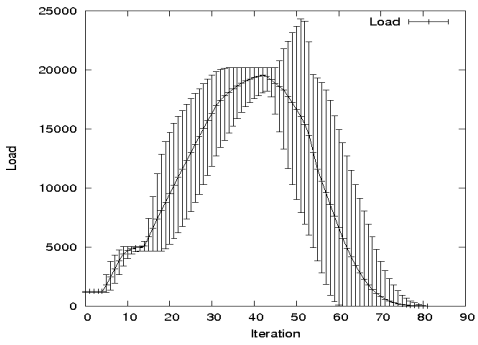


(f) Model 4: Salt canopy model

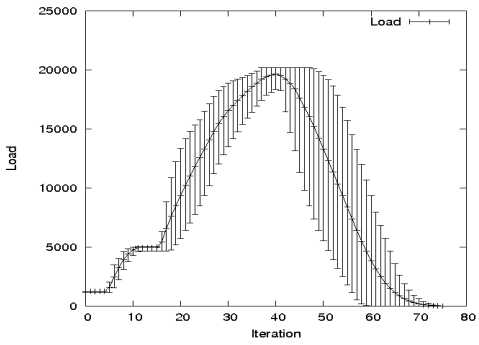


(g) Model 5: Salt dome model

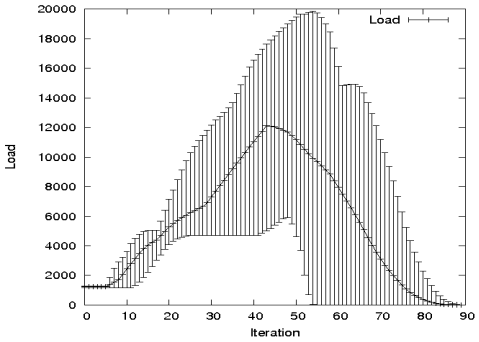
Figure B.3 Processor load varying with the iterations for various earth models (p=8).



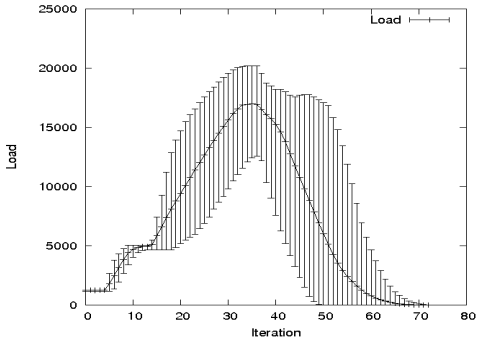
(a) Model 1: Isotropic medium



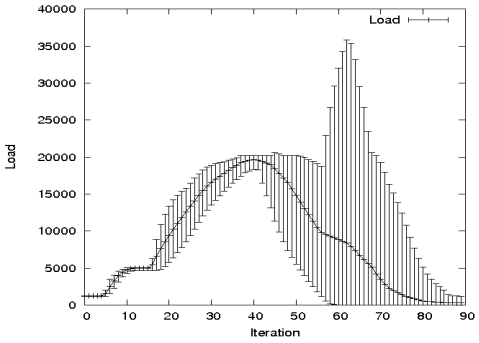
(b) Model 1: Weak Anisotropic medium



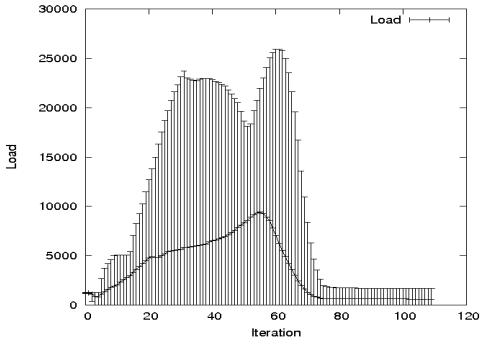
(c) Model 1: Strong Anisotropic medium



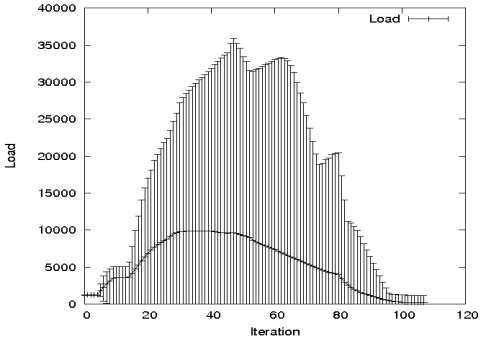
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model

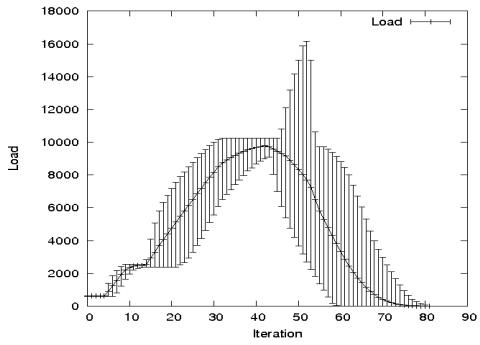


(f) Model 4: Salt canopy model

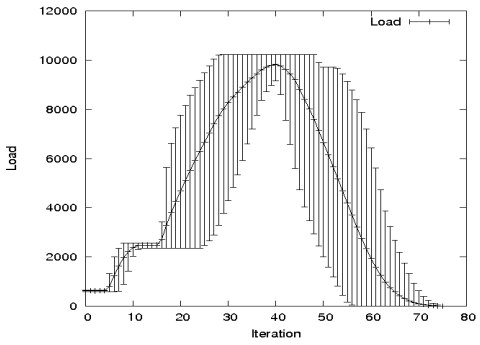


(g) Model 5: Salt dome model

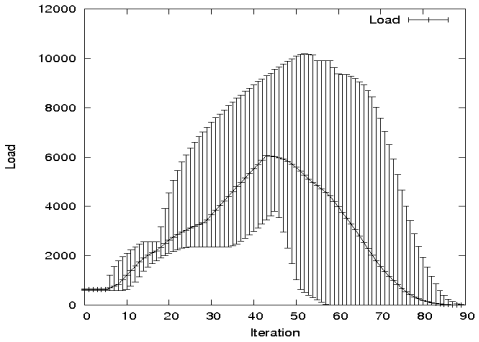
Figure B.4 Processor load varying with the iterations for various earth models (p=16).



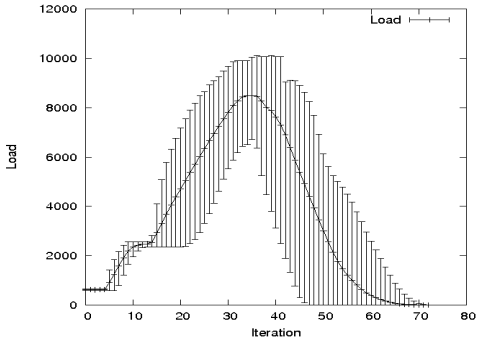
(a) Model 1: Isotropic medium



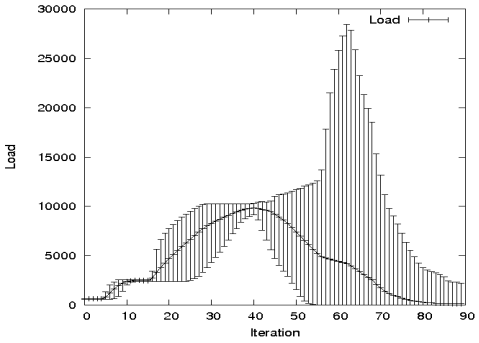
(b) Model 1: Weak Anisotropic medium



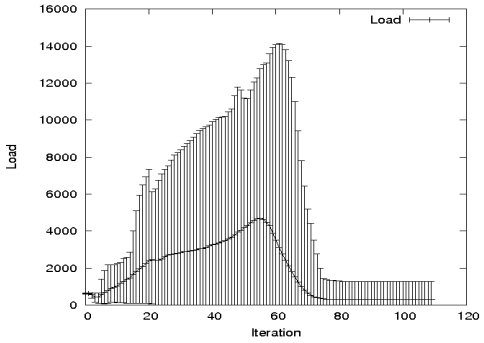
(c) Model 1: Strong Anisotropic medium



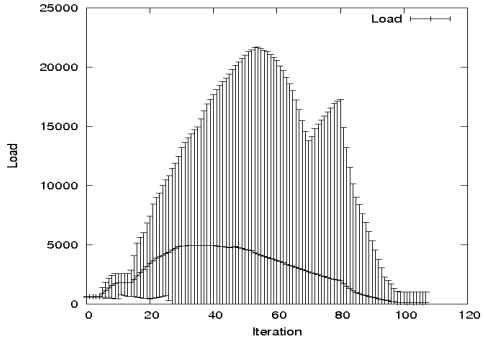
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model

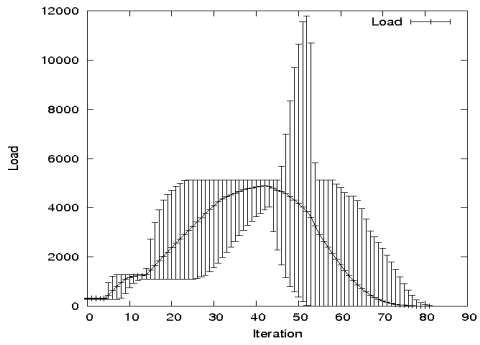


(f) Model 4: Salt canopy model

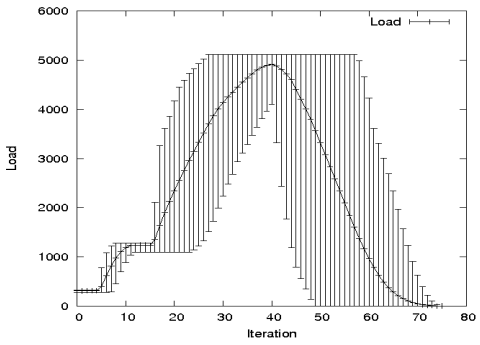


(g) Model 5: Salt dome model

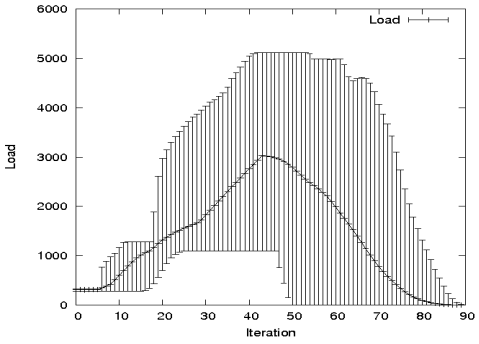
Figure B.5 Processor load varying with the iterations for various earth models (p=32).



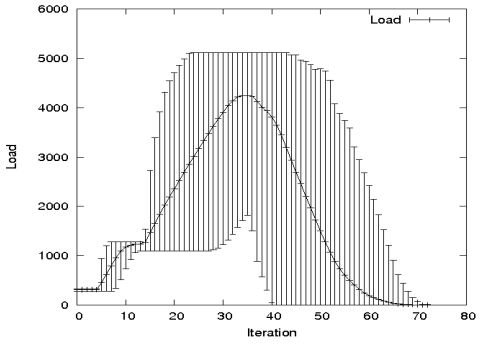
(a) Model 1: Isotropic medium



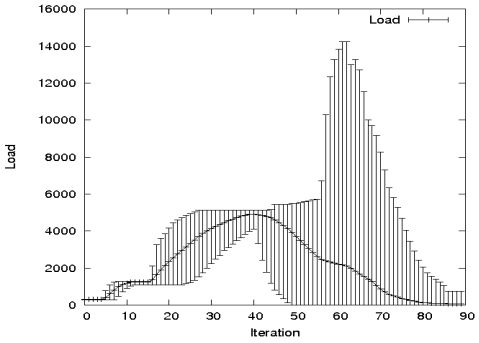
(b) Model 1: Weak Anisotropic medium



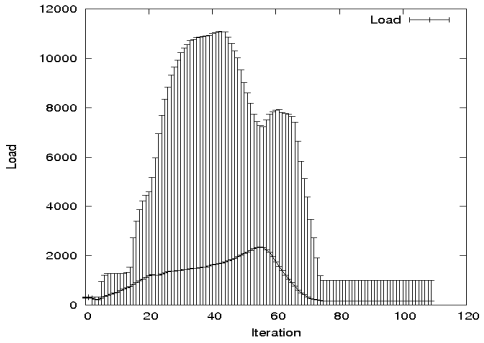
(c) Model 1: Strong Anisotropic medium



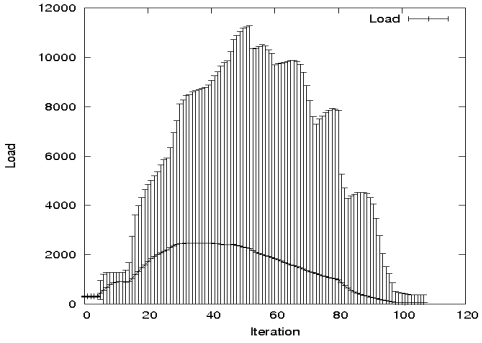
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model

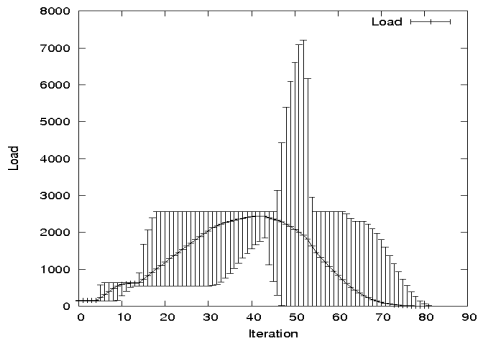


(f) Model 4: Salt canopy model

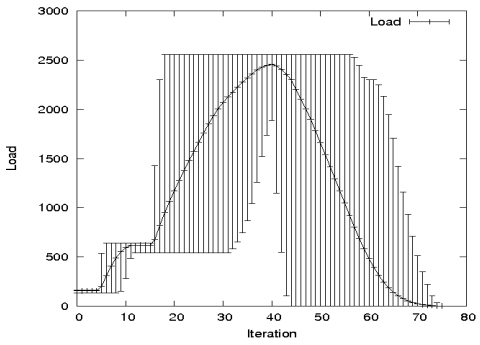


(g) Model 5: Salt dome model

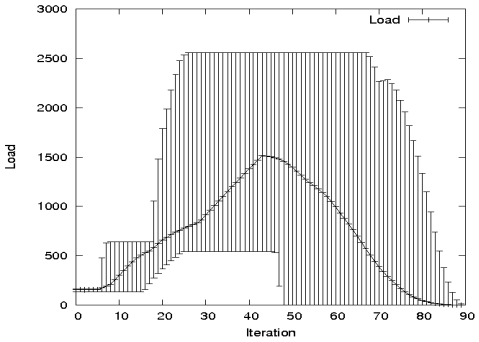
Figure B.6 Processor load varying with the iterations for various earth models (p=64).



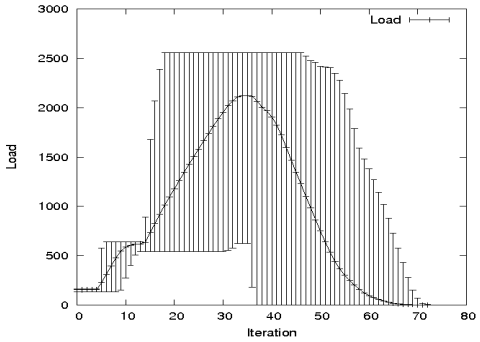
(a) Model 1: Isotropic medium



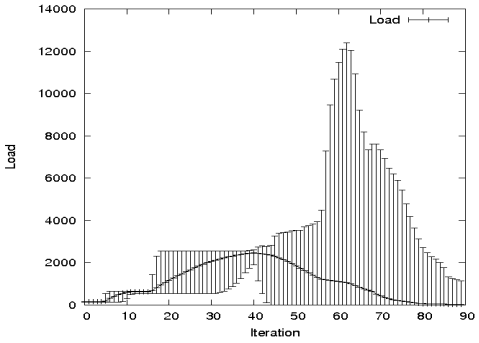
(b) Model 1: Weak Anisotropic medium



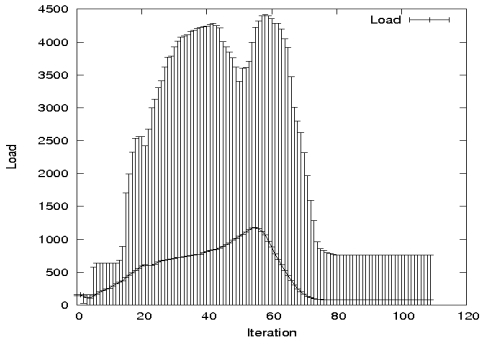
(c) Model 1: Strong Anisotropic medium



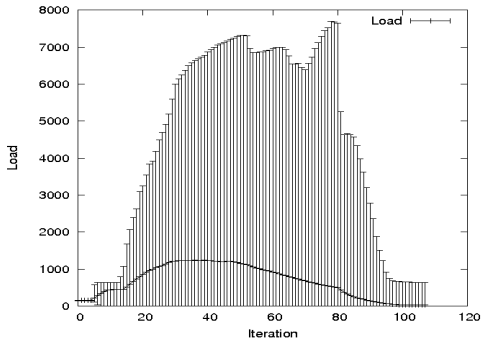
(d) Model 2: Layered model



(e) Model 3: Cylindrical inclusions model



(f) Model 4: Salt canopy model



(g) Model 5: Salt dome model

Figure B.7 Processor load varying with the iterations for various earth models (p=128).