NTNU
Innovation and Creativity

# Seismic processing using Parallel 3D FMM

**Idar Borlaug**

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

## Problem Description

Finding oil includes seismic processing of data obtained from experiments done in the field. The computations are generally very compute intensive and hence lends themselves to parallel computing, since often large datasets spanning storages on several compute nodes is involved.

In this thesis, a seismic application is considered and fast both serial and parallel algorithms studied and compared. This includes evaluating how to best divide the data between different nodes for a given application. In particular, methods for finding salt formations in seismic data using FMM, will be investigated.

Assignment given: 22. January 2007
Supervisor: Anne Cathrine Elster, IDI

# Abstract

This thesis develops and tests 3D Fast Marching Method (FMM) algorithm and apply these to seismic simulations. The FMM is a general method for monotonically advancing fronts, originally developed by Sethian. It calculates the first arrival time for an advancing front or wave. FMM methods are used for a variety of applications including, fatigue cracks in materials, lymph node segmentation in CT images, computing skeletons and centerlines in 3D objects and for finding salt formations in seismic data.

Finding salt formations in seismic data, is important for the oil industry. Oil often flows towards gaps in the soil below a salt formation. It is therefore, important to map the edges of the salt formation, for this the FMM can be used. This FMM creates a first arrival time map, which makes it easier to see the edges of the salt formation.

Herrmann developed a 3D parallel algorithm of the FMM testing waves of constant velocity. We implemented and tested his algorithm, but since seismic data typically causes a large variation of the velocities, optimizations were needed to make this algorithm scale. By optimising the border exchange and eliminating much of the roll backs, we delevoped and implemented a much improved 3D FMM which achieved close to theoretical performance, for up to at least 256 nodes on the current supercomputer at NTNU.

Other methods like, different domain decompositions for better load balancing and running more FMM picks simultaneous, will also be discussed.

# Acknowledgements

I would like to thank Dr. Anne C. Elster for being my advisor on this Master thesis, and for giving me valuable input throughout the project.

I would also like to thank all my friends and co students in room ITV-458 for helping me with various problems underway.

Thanks to Tore Fevang at Schluberger for providing me with the motivation for this assignment, and for helping me get started and understanding the problem.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Finding salt formations in seismic data is very important for the oil industry. Salt formations are mushroom shaped. This creates a cavity below the mushroom head, where oil will flow from the surrounding rock, and be trapped. If one can map the boundaries of the salt formation, it will be made easier for drilling companies to drill in the correct place.

Seismic data are generated by sending low-frequency shock waves into the earth, and reading the wave reflection with sensors. To generate the shock waves, explosives or air guns can be used. The sensors are laid out in a grid, and these sensors listen for waves reflected from the different layers of rock. The amplitude of the reflected waves are then stored. This results in a grid map with wave amplitudes.

Finding salt in these data can be troublesome, because salt is a crystalline form. Therefore, it reflects the waves in many different directions, making it very difficult to read.

The Fast Marching Method (FMM) [11] is a general method for monotonically advancing fronts. It calculates the first arrival time for an advancing front or wave. This can be used to solve a number of problems, follow fatigue cracks in materials [2], lymph node segmentation in CT images [3], computing skeletons and centerlines in 3D objects [4] and for finding salt formations in seismic data.

To solve this problem, a paper was published last year, called Seismic event tracking by global path optimization[5], from Borwn et.al for Amerada Hess Corporation. This paper describes how they tried to map the boundaries of a salt formation. They used the Fast Marching Method for mapping the boundaries, however, they did not use the FMM for the entire domain. They took planes from the seismic 3D data, using Delaunay triangulation. Using these planes or slices from the data, they then tried to map the edges of the salt formations.

In this thesis, the Fast Marching Method will be extended to 3D space. Because solving the FMM for a 3D space will take much longer, a parallel version of the Fast Marching Method will be made.

## 1.1 Contributions

The following highlights the contributions of this master thesis:

- Implementing and testing the Parallel version of the Fast Marching Method developed by Hermann.
- Development of a new parallel Fast Marching Method algorithm, because it will give better results.
- Different domain decompositions have been looked at and tested. To see if they gave a significant difference in speed.
- New computational model for calculating the theoretical execution time for the new parallel Fast Marching Methods have been constructed. Also an speedup equation have been made for PFMM, which roughly indicate the speedup achievable with a given number of nodes.
- Two ideas for improving performance even more have been discussed in Future work.

## 1.2 Outline

- Chapter 1 is this introduction.
- Chapter 2 is background information explaining seismic surveys, super computer, especially Njord, the Fast Marching Method and different variations of the Fast Marching Method.
- Chapter 3 explains my solution to the problem of finding salt formations in seismic data. It goes through the different choices and explains the two different applications that resulted from two parallel variations of the Fast Marching Method
- Chapter 4 describes a model of the runtime of each application
- Chapter 5 shows the different benchmarks and discusses for bout applications
- Chapter 6 is the conclusion
- Bibliography is a list of citations in this thesis, and a description of them. Links are included where available.

# Chapter 2

# Background Material

Finding salt formations in seismic data, explained in Section 2.1, can be difficult. [5] tried to find a solution for this in their article Seismic event tracking by global path optimization discussed in Section 2.2. The Fast Marching Method is a method used in [5] to locate the salt formations. The Fast Marching Method is discussed in Section 2.6. [5] only solved the problem for 2D planes, to solve the whole 3D domain, one needs a lot more computations. To solve that issue, one needs to solve the problem in parallel on many computers. Herrmann made a parallel version of the Fast Marching Method, discussed in Section 2.8. Normal computer could suffice when solving this problem, but it would be faster to run it on a Cluster super computer, discussed in Section 2.3. All the results in this thesis have been created while the program runs at the super computer Njord, discussed in Section 2.4.

## 2.1 Seismic data

One often use seismic data for locating oil reservoirs in the earth. To generate shock waves that traverse the earth they often use explosives or air-guns. Explosives are mostly used on land while air-guns are used on ships. To collect the data from the shock waves a grid with sensors is used. On land, this grid is laid out over the targeted area. On sea, it's pulled after a vessel with several kilometer long cables, carrying the sensors. When the shock waves are reflected back from the earth, the signals are gathered by the sensors and stored. These stored signals are mapped onto a 3D map that models the earth. Out of this 3D map, it's possible to read what lies beneath the surface.

Oil is often located in open spaces created by salt formations. A salt formation looks like a mushroom. On the edges below the top, open space are often formed. In this space, oil can flow from the surrounding rock, making huge areas of easily available oil. Therefore, it's important to find these salt formations.

## 2.2 Seismic event tracking by global path optimization

This is a summary of what is discussed in the article with the same name[5].

Tracking salt boundaries in seismic data is very difficult. It has gaps and other interferences, which makes it difficult for most trackers to function. Most trackers look a few steps ahead (local optimizations), which makes them more prone to go down gaps in the data. An "ideal" auto tracker would:

- track using a global measure of optimality
- intelligently traverse "holes" in the salt events
- naturally track multi-valued salt boundaries.

To make this work, a positive velocity field is needed. To do this with seismic data, a number to the data can be added, so everything becomes positive. To track the shortest distance between two points one will use the FMM and add the two different points. This will highlight the shortest path between those points.

To do this they implemented the FMM in a 2D space by taking 2D slices from the 3D space and running the FMM on them. To make this a 3D solution they used Delaunay triangulation [6]. This made for a quick 3D algorithm when new picks were chosen, because not all triangles needs to be updated with each new pick. A pick is a point chosen by a geologist. He picks two or three picks then run the FMM for each pick and add the results.

## 2.3 Cluster super computers

Cluster super computers are typically many computers or nodes that are connected. These nodes/computers will work on problems, e.g. mathematical problems, and solve them fast. For using such computers, one needs a way to interact with the other computers.

Message passing is a method for passing messages between the same program, running on different nodes in a super computer cluster. This can be messages like "hi I am done", "I got 10 as a result" or "you need this data". Messages can be small or large, they can be a huge array or a flag saying the program is in a specific state.

A well known specification for message passing is MPI, Message Passing Interface [7]. This is a library for message passing, which programs, that run on super computer clusters, can use. It has a variety of different communication styles implemented, that one might need when programming for super computer clusters. This is normal send/receive operations and more complex operations like broadcast- and reduce operations. It has also different forms of send/receive communication, synchronous, blocking and non-blocking, ready mode, etc. All these are useful for different scenarios.

Different mathematical problems are often solved on super computer clusters. Many of these use a grid or array, describing the problem. When solving a problem on a large array, it's normal to distribute a part of this array to each node in the cluster. This way, every node can solve a small part of the problem, resulting in a speed increase. To solve only one part of such a problem one normally needs data from the nodes that have the neighbouring data. This is often solved by using a border around the node's data, containing some of the neighbours data. In Figure 2.1 the border cells are marked by the nodes number. The border cells are often exchanged between the nodes a number of times when a problem is solved. This will involve communication with four nodes in the super computer cluster.

| | | | | | |
|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | |
| 3 | 4 | 4 | 4 | 4 | 5 |
| 3 | 4 | 4 | 4 | 4 | 5 |
| 3 | 4 | 4 | 4 | 4 | 5 |
| 3 | 4 | 4 | 4 | 4 | 5 |
| | 7 | 7 | 7 | 7 | |

Figure 2.1: Border cells

## 2.4 Njord

Njord [8] is the current super computer at the Norwegian University of Science and Technology NTNU, delivered by IBM. It is a computer which uses SMP nodes and distributed memory between the nodes. Each node has 8 CPUs with 2 cores each. Each core can run two threads at the same time (SMT). However the threads does share some vital resources like the floating point unit. Vecause of this its not default to run two threads on each CPU on Njord. Each node can run 16 thread/processes at full speed. There are 62 nodes on Njord, making it possible to run jobs with 992 processes in theory. Some nodes are reserved for special programs. The largest size job that can run is 864 processes. Each node with 16 processes have enough memory for 832 MB for each proses. The CPUs used are IBM Power5+ [9], with 36 MB level 3 cache. The interconnect between the processes on each node is shared memory, between the nodes its a very fast network interconnect.

## 2.5 Manhattan distance

Manhattan distance [10] is a concept coming from Manhattan in New York. Because Manhattan is only square blocks with roads in a grid around them, the distance a cab driver has to take from one point to another in Manhattan is the Manhattan distance. In Figure 2.2 the blue, yellow and red lines describe the Manhattan distance. The green line is the Euclidian distance, which is the shortest distance between two points.

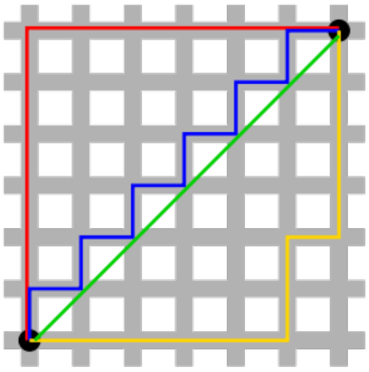


Figure 2.2: Manhattan distance

## 2.6 Fast Marching Method (FMM)

The Fast Marching Method is a solution for the eikonal equation.

$$|\Delta u(x)| = F(x), x \epsilon \Omega \tag{2.1}$$

The eikonal equation gives you a travel time field, with first arrival times at all points within the solution. Sethian[11] has developed a fast marching method that solves this equation in a grid.

Fast Marching Method is a method that solves the travel time field without moving over each point more than once, a one pass algorithm. This is achieved by making a narrowband around the starting point or start structures, and moving this narrowband outwards one point at a time.

In Figure 2.3 you can see the narrowband represented by 0, where the already calculated points being marked as -1. Those which are not modified are marked at 1. For not using numbers later i will mark them as the following. Those outside the narrowband will be marked as OUTSIDE, those on the narrowband as BAND and those inside as KNOWN.

To solve Equation 2.1 correctly, the gradient operator has to be approximated by upwind, entropy-satisfying finite differences [12]. The approximation most often used is from [13].

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | -1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | -1 | -1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | -1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 2.3: Narrowband

$$\begin{aligned}
&[max(D_{ijk}^{-x}G, -D_{ijk}^{+x}G, 0)^2 + \\
&max(D_{ijk}^{-y}G, -D_{ijk}^{+y}G, 0)^2 + \\
&max(D_{ijk}^{-z}G, -D_{ijk}^{+z}G, 0)^2]^{1/2} = 1.
\end{aligned} \tag{2.2}$$

Where

$$\begin{aligned}
D_{ijk}^{-x}G &= \frac{G_{ijk} - G_{i-1jk}}{\triangle x}, D_{ijk}^{+x}G = \frac{G_{i+1jk} - G_{ijk}}{\triangle x} \\
D_{ijk}^{-y}G &= \frac{G_{ijk} - G_{ij-1k}}{\triangle y}, D_{ijk}^{+y}G = \frac{G_{ij+1k} - G_{ijk}}{\triangle y} \\
D_{ijk}^{-z}G &= \frac{G_{ijk} - G_{ijk-1}}{\triangle z}, D_{ijk}^{+z}G = \frac{G_{ijk+1} - G_{ijk}}{\triangle z}
\end{aligned} \tag{2.3}$$

Where G is the arrival time matrix. The simple solution to this problem is to iteratively update all nodes within the array after Equation 2.2 until it finds a stable solution. Because this equation has an upwind property, each point in the array is only dependent on its smaller neighbours, and one can use the faster FMM algorithm.

The fast marching method solves the travel time to different point in a grid. It moves a narrowband outwards from a starting point, which can be a single point or multiple points that are connected. It basically consists of a few simple steps, shown in Figure 2.4.

### 2.6.1 FMM Initial data

Make two arrays, one for storing the band information and one for storing travel times. Set the initial point or points to KNOWN. Add the adjacent points to the narrowband and mark them BAND. The rest must be marked as OUTSIDE.

1. set up initial data
2. loop begin: extract the point from the narrowband with lowest travel time
3. mark the extracted point as KNOWN
4. add neighbours not in narrowband or KNOWN to narrowband.
5. recalculate all adjacent nodes by Equation 2.2
6. loop end

Figure 2.4: Serial FMM algorithm

### 2.6.2 FMM main loop

First, one needs to extract the lowest travel time from the narrowband. The narrowband can be stored efficiently in a min sorted heap[14]. Set this point to KNOWN. This is now removed from the narrowband and will never be added. Add adjacent points to the narrowband and calculate their arrival times according to Equation 2.2. Repeat the loop until all nodes are marked KNOWN.

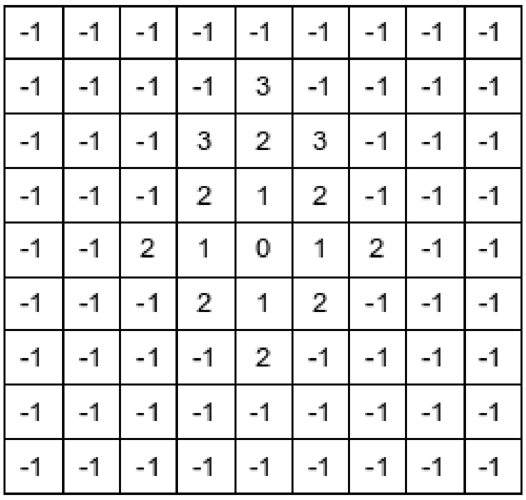
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | 2 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 1 | 2 | -1 | -1 | -1 |
| -1 | -1 | 2 | 1 | 0 | 1 | 2 | -1 | -1 |
| -1 | -1 | -1 | 2 | 1 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | 2 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

Figure 2.5: Traveltime array step n

An ilustration of one loop step can be seen in Figure 2.5 and Figure 2.6. In the first figure all nodes marked 2 are in the narrowband. In the second figure a new step has been made. The top 2 value has been marked KNOWN and removed from the narrowband. Its neighbouring nodes have been added to the narrowband and its values calculated. In the next step the lowest value from the narrowband would be choosen. That would be one of the remaining 2 points.

## 2.7 Spherical vs Cartesian coordinates

To calculate the arrival times, one can use two different coordinate systems, the Cartesian xyz and Spherical $r\theta\phi$. Where $r$ is the length from origin and $\theta$



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | 3 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 3 | 2 | 3 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 1 | 2 | -1 | -1 | -1 |
| -1 | -1 | 2 | 1 | 0 | 1 | 2 | -1 | -1 |
| -1 | -1 | -1 | 2 | 1 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | 2 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

Figure 2.6: Traveltime array step n+1

is the angel between the x axis and the $r$ line and $\phi$ is the angel between z-axis and $r$ along the vertical plane.

To calculate arrival time at a specific point, one can use the 1 order approximation 2.4.

$$
\begin{aligned}
t_{ijk} = t_{i-1jk} + \frac{\triangle x}{v}, t_{ijk} = t_{i+1jk} + \frac{\triangle x}{v} \\
t_{ijk} = t_{ij-1k} + \frac{\triangle y}{v}, t_{ijk} = t_{ij+1k} + \frac{\triangle y}{v} \\
t_{ijk} = t_{ijk-1} + \frac{\triangle z}{v}, t_{ijk} = t_{ijk+1} + \frac{\triangle z}{v}
\end{aligned}
\tag{2.4}
$$

Where v is the velocity at point (i,j,k) and t is the arrival time array.

Using Equation 2.4 will give large errors on sparse grid point configurations, because it has problems with waves propagating at 45 degrees. According to [1] this will result in a 20% error near a point source.

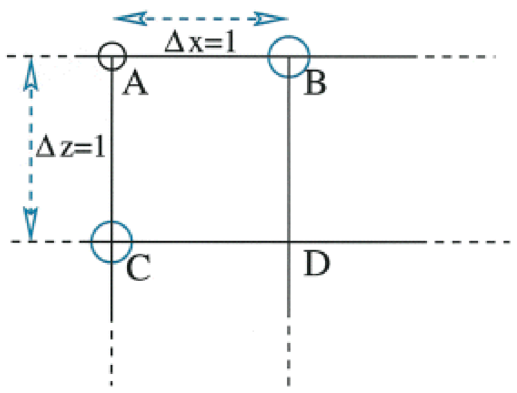


Figure 2.7: Cartesian coordinate system. Figure from [1]

If you have three points A B C where A(0,0) is the center, and B(1,0) is one point away on the x axis and C(0,1) is one point away on the y axis. Both B and C have arrival time 1.See Figure 2.7. When the FMM calculates Bs neighbour D (1,1) it will give it arrival time 2 according to Equation 2.4. But is should have $1 + 1/\sqrt{2}$ which will result in the 20% error. One solution to this is too use higher order approximations, seeing further than one point in the grid. Another is to use Spherical coordinates.

$$\begin{aligned}
&[max(D_{ijk}^{-r}t, -D_{ijk}^{+r}t, 0)^2 + \\
&max(D_{ijk}^{-\theta}t, -D_{ijk}^{+\theta}t, 0)^2 + \\
&max(D_{ijk}^{-\phi}t, -D_{ijk}^{+\phi}t, 0)^2]^{1/2} = 1
\end{aligned} \tag{2.5}$$

where

$$D_{ijk}^{-r}t = \frac{t_{i,j,k} - t_{i-1,j,k}}{\triangle r}, D_{ijk}^{+r}t = \frac{t_{i+1,j,k} - t_{i,j,k}}{\triangle r} \tag{2.6}$$

$$D_{ijk}^{-\theta}t = \frac{t_{i,j,k} - t_{i,j-1,k}}{r\triangle\theta}, D_{ijk}^{+\theta}t = \frac{t_{i,j+1,k} - t_{i,j,k}}{r\triangle\theta} \tag{2.7}$$

$$D_{ijk}^{-\phi}t = \frac{t_{i,j,k} - t_{i,j-1,k}}{rsin\theta\triangle\phi}, D_{ijk}^{+\phi}t = \frac{t_{i,j-1,k} - t_{i,j,k}}{rsin\theta\triangle\phi} \tag{2.8}$$

For using Spherical coordinates Equation 2.2 must be modified. The modified version is Equation 2.5. To use this function, all values for $\theta$ and $\phi$ where $r$ is 0 are set, and added to the narrowband. Unlike the Cartesian version, the heap tends to be stable in Spherical coordinates, because one usually subtracts one and adds one, when in Cartesian coordinates the heap can become pretty large, especially in 3D grids.

[1] concludes that spherical coordinate systems give more accurate results. Especially with point sources where Cartesian coordinates give a high degree of error. The spherical solution is generally as fast as the Cartesian version, but it will not give the best results for head waves.

## 2.8 Parallel Fast Marching Method

M. Herrmann [15] has written an article on domain decomposition parallelization of the Fast Marching Method. This thesis gives a short brief of some of the different methods he investigated.

There are a few problems with the Fast Marching Method when it comes to parallelization. It has a very serial nature. When the narrowband is moved, one needs to find the lowest travel time point, which will be difficult on a parallel version. How the narrowband will move is also difficult to know before hand, therefore it will not be completely straight forward how to divide the matrix between the nodes. It can even move in a spiral shape.

1. Perform step 1 of serial algorithm 2.4
2. locate the local minimum value in the narrowband.
3. Find the global minimum value by exchanging local minimums.
4. Perform step 3-5 of serial algorithm on the node with global minimum value.
5. If global minimum is border value, exchange with neighbouring node.
6. return to step 2 until all values are calculated.

Figure 2.8: Parallel Fast Marching Method 1.

### 2.8.1 Herrmann's Parallel algorithm 1

When making a parallel version of the Fast Marching Method, it's advantageous to split the domain between the computational nodes. Then a problem arise when one needs to find the smallest value in the narrowband. The smallest value can only be on one node (or a few if there are many equal values). The straight forward solution for this is to calculate the local minimum and use an all reduce function to find which node has the global minimum. When one calculates the border nodes, it's also important to send these changes to the other nodes. This solution results in algorithm 2.8.

This algorithm has an inherit serial part, still only one node can calculate at the same time. One can work on much larger datasets compared to the serial algorithm because the dataset is divided between the different nodes. To calculate the global minimum, one can use an allreduce min function, but it will still contain a global communication point for each node in the dataset.

### 2.8.2 Herrmann's Parallel algorithm 2

The next logical step is to get all nodes to work at the same time. This can be achieved if each nodes propagating narrowband will not interfere with the other nodes. The problem comes if a node receives a border value which will give lower arrival time then what's already calculated, then that node needs to roll back to an earlier state. Herrmann's first attempt at solving this problem resulted in algorithm 2.9

There are a few drawbacks for algorithm 2.9. First, one needs to store each state, which will take a huge amount of storage space. This must be done to be able to roll back. Secondly almost all exchanges to border values will result in a rollback. These problems are corrected in the last algorithm.

### 2.8.3 Herrman's Parallel algorithm 3

The last algorithm will try to correct the problems encountered in algorithm 2. Because of the attributes of Equation 2.2, a point that is larger than the new

1. Perform step 1 of serial algorithm 2.4
2. Check if you received a border value with lower value than the highest in your grid. If so, roll back to a state where the new value is the highest. Add this value to narrowband.
3. locate the locally smallest value in narrowband including new border values.
4. Perform step 3-5 of serial algorithm on the node with local minimum value.
5. If local minimum is border value, exchange with neighbouring node.
6. Store current state
7. return to step 2 until all value are calculated.
8. Wait until all nodes are finished or a new border value is received. Go to step 2.

Figure 2.9: Parallel Fast Marching Method 2.

1. Perform step 1 of serial algorithm 2.4
2. Check if you received a border value with lower value than the highest in your grid. If so, roll back to a state where the new value is the highest. By marking all points with higher value as BAND, add those values to narrowband.
3. locate the locally smallest value in narrowband, including new border values.
4. Perform step 3-5 of serial algorithm on the node with local minimum value.
5. If local minimum is border value, exchange with neighbouring node.
6. return to step 2 until all values are calculated.
7. Wait until all nodes are finished or a new border value is received. Go to step 2.

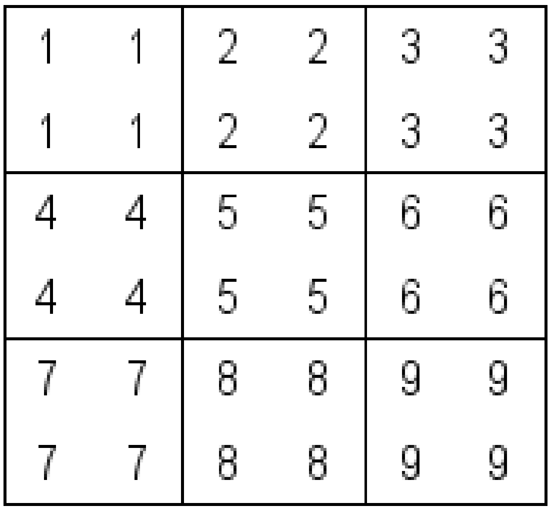
Figure 2.10: Parallel Fast Marching Method 3.

border value can be set back to BAND and retain its value[15]. So there is no reason to save the complete state, one can only rollback to BAND. This resulted in algorithm 2.10

This new algorithm 2.10 makes it possible to do a roll back without storing each state, one only needs to store the original array. The performance of this algorithm will change greatly depending on how many border exchanges must be made, and the number of rollback operations. The performance will vary a lot, depending on the border exchanges and rollbacks. In some situations it can be very fast, but in others it will require a lot of border exchanges and rollbacks, making it slow. It will still be faster than both algorithm 2.8 and algorithm 2.9.

### 2.8.4 Domain decomposition

The domain decomposition will have a great effect on how fast certain problems will be solved. There are a few ways one can divide the domain, the one most often taken is to divide them by minimising borders. Using a rectangular shape 2:1 it will give the least border area. Another approach is to divide the domain so that all nodes touch the center. This approach is taken by Herrmann [15] in his example.



| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 3 | 3 |
| 1 | 1 | 2 | 2 | 3 | 3 |
| 4 | 4 | 5 | 5 | 6 | 6 |
| 4 | 4 | 5 | 5 | 6 | 6 |
| 7 | 7 | 8 | 8 | 9 | 9 |
| 7 | 7 | 8 | 8 | 9 | 9 |

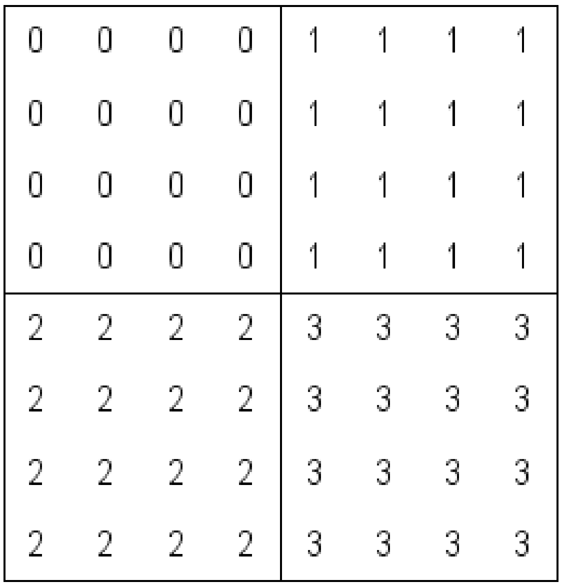
Figure 2.11: Non optimal domain decomposition

Figure 2.11 is a domain decomposition chosen by Herrmann[15] to illustrate that this is not an optimal decomposition. It will not minimise borders and all nodes will not work at the same time. This decomposition can easily be scaled to 27 nodes for 3D space by adding nodes in the z axis.

In Figure 2.12 each node are connected to the middle. If the starting point is in the middle and the velocity field is all 1, then this would result in a circle expanding from the middle with equal work on each node. There would also be no need to do any communication between the nodes. Herrmann used this optimal layout for some of his tests they yielded 0.98 efficiency[15]. The problem is that very few real world cases map to this division. If the velocity field is very varied, one node could end up doing a lot of work. In 2D space this only scales to 4 nodes, and in 3D only 8 nodes. In the optimal case it works very well.

### 2.8.5 Herrmann's results

Hermann showed some graphs, illustrating how each of these domain decompositions would scale. Not surprisingly, the quadratic decomposition worked very well when the start structure was a sphere. This resulted in 0.96 efficiency. The non optimal decomposition didn't do it very well. This is because it will



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |

Figure 2.12: Quadratic grid decomposition

not make all nodes work at the same time, and it is prone to more rollbacks, because of more communication.

# Chapter 3

# Parallel implementation of FMM

Finding salt formations in seismic data, is a challenge. Since this solution is created for solving a 3D domain, it is necessary to use more than one CPU. Therefore, it's important to make a parallel solver. This implicates that one need to change the Fast Marching Method, so it will be parallel. This is not straight forward since the algorithm is inherently serial, only the lowest value in the narrowband can be calculated. In this chapter two solutions are discussed, the second was created because of shortcomings in the first. They are hereafter known as HFMM and PFMM. As HFMM is bassed on Herrmanns algorithm and PFMM is the new algorithm described in this thesis.

## 3.1 Fast Marching Method

The Fast Marching Method is a very fast method for finding arrival times in a velocity field. It is a one pass algorithm and faster than one pass is very hard to make. Unfortunately, it uses a heap for finding the lowest value on the narrowband, giving it O(log N) runtime. This results in a final algorithm of O(N log N), which is a fairly fast one. The problem is that it doesn't easily parallelize. With the seismic datasets there is rarely enough ram to run the algorithm on one node. Not to mention the time it will take an N log N algorithm to pass many gigabytes of data. Therefore, it's important to use a parallel version of the Fast Marching Method.

### 3.1.1 Choosing domain decomposition

Since the seismic datasets are so large, a domain decomposition is essential. There are already two discussed by Herrmann in Section 2.8.4. The quadratic decomposition is the one that will give the best results in the optimal case that Herrmann tried. But there is also another decomposition that Herrmann didn't discuss, rectangular 1:2 scale decomposition. If you divide all your data into 1:2 rectangles you will get the lowest number of border cells.

$$w * C * 2 + h * C * 2 = TC \tag{3.1}$$

Where w = width, h = height, C = cost per unit and TC = Total Cost.

The minim total cost for Equation 3.1 is when $w = 2h$. This will give the least border size for a given 2D domain. Using this as a domain decomposition, one would end up with the rectangular 1:2 scale decomposition.

I will therefore try to see which of these two domain decompositions yields the best results.

### 3.1.2 Rectangular 1:2 scale decomposition

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 |
| 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 |
| 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 |
| 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 |
| 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 |

Figure 3.1: Rectangular 1:2 scale decomposition

In Figure 3.1 each node has a 1:2 rectangular data area, and the node also map to 1:2 rectangular shape. This will minimise the border cells. Such a grid will be very optimal for iterative solutions where one can run one iteration, then exchange borders and run another. Because each node has the smallest possible border size. However, in a FMM this division might not be optimal because one cannot guarantee that all nodes are on the narrowband. The narrowband could only be using 2 or 3 nodes, then the others would just sit there and wait for data. In seismic data it is, however, not that easy to predict how the narrowband will travel. Another advantage with this model is that it will scale, one can easily add nodes and it's easy to create a minimum border layout. The model will also convert to 3D space by just adding another layer of nodes in the z direction.

### 3.1.3 Choosing a parallel FMM

There are basically two parallel algorithms that one can choose from algorithm 2.10, which is the one Herrmann got the best results from, and iterative domain update according to Equation 2.2. The domain update will use many iterations before it will reach a stable solution. It is, however, very easy to parallelize, since it's a iterative numerical approach. But it will not give close to the wall clock time given by algorithm 2.10. Since the algorithm 2.10 is a one pass algorithm. If algorithm 2.10 gives no rollbacks. There is therefore no reason to believe that a iterative approach will beat a one pass with a few rollbacks. The other algorithms presented by Herrmann is worse than algorithm 2.10 so they have not been tested.

I also wanted to test an idea of my own, it's an algorithm that is a cross between Herrmann's 2.10 algorithm and the domain update version. This is discussed later in Section 3.3.

### 3.1.4 Choosing Spherical vs Cartesian coordinates

I did not choose to use spherical or higher order Cartesian equations. The reason for this was that there was not enough time and the important part of this assignment is to find a viable parallel algorithm. If a good parallel FMM is found its very easy to exchange the equation used for finding the new arrival times in each point later. The first order Cartesian approximation is also very well tested and gives good enough results for most purposes, it's also good enough to check that the application calculates the times correctly. I used Equation 2.4 in my application.

## 3.2 HFMM

In this section, I will explain how my implementation of algorithm 2.10 works, and some minor modifications to the algorithm to avoid some issues that kills performance.

### 3.2.1 Parallel Fast Marching Method

The algorithm I used is an implementation of algorithm 2.10 from Herrmann. I made some minor changes, more implementation specific then actually altering the algorithm. The algorithm is explained step by step in Figure 3.2.

1. Setup initial data
2. Add initial point and set that to BAND
3. (a) Extract lowest value from narrowband and set the point to KNOWN
   (b) Calculate new arrival times for all neighbouring nodes that are not KNOWN
   (c) Add neighbouring nodes not in narrowband or KNOWN to narrowband
   (d) If the extracted point is a border point, send it to neighbours
   (e) check for incoming border values, add those to array and roll back if necessary
   (f) go to step (a) until narrowband is empty
4. Check for incoming border changes, add those and roll back if necessary
5. Check if all nodes are finished, then exit or go to step 3

Figure 3.2: Parallel Fast Marching Method, My version

### 3.2.2 Initial data

There is quite a lot of data that need to be allocated for the FMM. You need a velocity array, this is the data you get from seismic readings. The velocity array / seismic data is the one property thats sets your memory requirements.

The velocity aray is floating point numbers, saved as a float array in my application.

The second array you need is arrival times, where your store the result from your application. This is also a float array.

You also need an int array to store which nodes are KNOWN, OUTSIDE and on the narrowband (BAND). I used to store KNOWN as N where n is number of loops. This was used to test a few optimizations. The optimal would be to use two bits to store each point.

The HEAP also needs a pointer array to store each point that belongs to the narrowband. This heap array need to be big enough to store the largest narrowband size in the entire run. It's not that easy to calculate because it can become very large if there are very varying velocity values. I created the array to the same size as velocity array, it is way too big but I didn't want to get into space problems on the heap, for better memory utilization a vector would be preferable.

I also used a last array to store what value was sent. So if my narrowband came across a value that was already sent, because of a rollback, it wouldn't send it again if it had the same value as last time it was sent. This array will take as much place as the inner boarder but I made it as big as the velocity array. Then I didn't have to make a function that translates from a border point to some point in the boarder array. It would have added more complexity, and only given less memory usage which was not a problem on my test machine.

| | My application | Optimal |
|---|---|---|
| Velocity array | ARRAYSIZE*4 | ARRAYSIZE*4 |
| Arrival time array | ARRAYSIZE*4 | ARRAYSIZE*4 |
| Narrow band array | ARRAYSIZE*4 | ARRAYSIZE*1/4 |
| HEAP array | ARRAYSIZE*4+filldata | filldata count*4 + filldata |
| Sent array | ARRAYSIZE*4 | 0 |

Table 3.1: Optimal memory usage and my applications memory usage

This results in the memory usage presented in Table 3.2.2. ARRAYSIZE is number of points $x \cdot y \cdot z$ of the velocity array, which setts the problem size.

The total memory usage will be according to Table 3.2.2 on my application (arraysize*4)*5 + heap filldata and optimal memory usage will be (arraysize*4)*2+1/4 + heap filldata count*4 + filldata. So my application uses around twice as much as the optimal case.

### 3.2.3 Narrowband

Using a min sorted heap [14] we would be able to store the narrowband in a very efficient way. The heap has a few properties that make it ideal for this. It can insert and extract points in O(log N) time, which is very efficient. For my application, we used the heap from Introduction to Algorithms [14] and changed it, so it became a min heap instead of a max heap, which is implemented in the book. To store each point, a struct was used, containing x,y,z coordinates and the value. This worked out very well.

### 3.2.4 Calculation of arrival times

To calculate the new arrival times of a specific point I used equation 2.4. This equation for calculating arrival times is well tested, as its the most straight forward equation.

### 3.2.5 Exchanging border values

The border exchange was synchronous, which gave correct results. However, having a synchronous communication for each border cell made it very slow. Therefore, it was changed to asynchronous communication. The program sends border values once they are done and it checks to see if there are any incoming border values and add those. This mostly eliminates the network delay. The communication for signaling the nodes when you are done were also made asynchronous.

1. Setup initial data
2. Add initial point and set that to BAND
3. (a) Extract lowest value from narrowband and set the point to KNOWN
   (b) Calculate new arrival times for all neighbouring nodes that are not KNOWN
   (c) Add neighbouring nodes not in narrowband or KNOWN to narrowband
   (d) go to step (a) until narrowband is empty
4. Exchange borders, reset band array
5. Calculate new values to the border's neighbours (inner border), if these values are smaller than those already there, add them to narrowband
6. Check if all nodes are finished, then exit or go to step 3

Figure 3.3: Parallel Fast Marching Method revised

To avoid unnecessary rollbacks, a few methods was employed. First store the largest value in the arrival time grid, so its fast to check whether the incoming border value is smaller than the largest value. Receive all the pending border values before checking if a rollback is needed, seemed to reduce rollbacks. If there are three incoming border values and two of them require a rollback, one is executed.

## 3.3 PFMM

Because HFMM didn't scale as well as expected, I altered the algorithm a bit, hoping to get better results. The altered/new algorithm is described in Figure 3.3.

Much of the algorithm is exactly the same. The initial data is mostly the same, however, this algorithm doesn't rely on loop numbering, so one can save some space in the band array. The FMM works just like it does in a serial version, but there are some differences.

Instead of sending border values all the time, the program waits until the FMM is done with all the values in one node. Then send all values to neighbouring nodes. This will make the number of network sends quite a lot less. There are no rollbacks, but the system might work a few iterations before it settles and a stable solution is achieved.

If we use the quadratic decomposition , shown in Figure 2.12, which is used by Herrmann, this system will settle in three iterations. Say, the start point is located on node 0. Then node 0 will work its serial FMM until it's done, then exchange values to node 1 and 2. In the second iteration, nodes 1 and 2 will run their serial FMM, node 1 will check its border and find no need to add any new values. When node 1 and 2 are done, node 3 will start its serial FMM in iteration 3. When node 3 is done, all nodes will check their borders and find no values that need updating and all will exit. This example will work if there are no waves coming back to a node, e.g. in a velocity field with only one value. Given the example Herrmann made by using a sphere located at the center as starting condition, this algorithm will run 1 iteration, it will however need the delay in transferring borders and checking the borders before it can exit so you won't get 100% efficiency compared to the serial version.

If the velocity field is the same value all over, this algorithm will stabilise in the Manhattan distance from the starting point node to the farthest edge iterations. So, if you take the Manhattan distance from the starting node to the farthest edge, you get the number of iterations. However, if you use a velocity field that will give waves that turn back into another node, then you might want to use a few more iterations. Given a 64 node 3D space 4x4x4, the Manhattan distance from the corner to the other will be 8.

# Chapter 4

# Modeling of the FMM algorithm

In this chapter a model for each algorithm is developed. The model will try to give a theoretical equation for runtime.

## 4.1 HFMM

The HFMM algorithm has two important aspects when it comes to calculating the runtime, how far is it from the point to the next node and how many rollbacks will occur. I will now try to explain how one can calculate those aspects and come up with an equation for overall execution time.

$$arraysize = n \cdot m \cdot k \tag{4.1}$$
$$bordersize = n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2 \tag{4.2}$$

Where n is the length in along x axis, m is length along y axis and k is length along z axis.

$\#rollback$ is difficult to estimate. It can be affected by several factors. Since HFMM uses asynchronous communication, one node running faster than another can trigger rollbacks in the neighbouring node. The network delay will also affect how many rollbacks that will occur, late border values might trigger rollbacks. If there are several border values coming in at the same time, and many off them will require a rollback, then only one rollback will be executed. The number off rollbacks will vary depending on the system its ran on. Two important factors affect the number of rollbacks, network delay and variations in execution time from node to node.

The time it takes to execute rollbacks is easier to describe. It consists of two factors, $\#rollback$ and $T_{traverse}$. $T_{traverse}$ is the time it takes to go through the matrix and set the band array back to a previous state. $T_{traverse}$ can be show as

Equation 4.3. *mtime* is the time it takes to modify one point in an array. This results in Equation 4.4 which describes the time it takes to execute all rollbacks on one node. *#rollback* is given by three factors, how high the network delay is, difference in computational speed between the nodes and if the data set require a rollback. This can be estimated through a few runs on the same dataset. But will be affected by borderline and extra border sends, caused by the velocity field.

$$\begin{aligned} T_{traverse} &= arraysize \cdot mtime \\ T_{traverse} &= n \cdot m \cdot k \cdot mtime \end{aligned} \tag{4.3}$$

$$\begin{aligned} T_{rollback} &= \#rollback \cdot T_{traverse} \\ \#rollback &= P(rollback) \cdot (bordersize + extraBorderSends) \end{aligned} \tag{4.4}$$

The time it takes for computing a matrix is explained in Equation 4.5. It is affected by two values how long it takes to execute one loop in the FMM and the array size. $T_{loop}$ describes how long one loop takes. However, how long it takes for each loop is difficult to predict, it consists of a few steps, all neighbours that are OUTSIDE will be calculated and added to narrowband. Calculating the new values are a constant time operation, called *calcn*, but adding the points to the narrowband is a log(N) operation. Which means it will be difficult to predict, because N is not known until runtime. It's very difficult to calculate how long it takes to execute $T_{compute}$, but it will be the same for each time the algorithm is run on the same dataset.

$$\begin{aligned} T_{compute} &= T_{loop} \cdot arraysize \\ T_{loop} &= \#CalcNeighbours \cdot calcn + log(sizeNarrowband) \end{aligned} \tag{4.5}$$

Communication between nodes has a constant time and a not so constant value. The minimum communication time is the time it takes to send all border values and receive them. The difficult part is the extra communication it takes if a rollback occurs. A border value that hasn't changed will not be resent, but if a border value changes because of a rollback, it will be resent. Equation 4.6 show how communication is calculated. $\beta$ is the inverse bandwidth and $\alpha$ is the network delay. For running on njord on one node $\alpha$ is $1.995 \cdot 10^{-6}$ and $\beta$ is $2.143 \cdot 10^{-10}$.

$$\begin{aligned} T_{comm} &= (bordersize + resendCount) \cdot 20 \cdot \beta + \alpha \\ T_{comm} &= ((n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) + resendCount) \cdot 20 \cdot \beta + \alpha \end{aligned} \tag{4.6}$$

To calculate the time it takes for a node to finish, one needs to know when the wave will trigger the node to start. This is not easy to find out, it will not move equally fast in all directions, that is dependant on the velocity array. Where the start point is located is important, if its located close to the border the node besides it will start to work earlier than those on the other side. The time for the wave to hit a node is called $T_{wavetime}$.

The overall time it takes for a node to finish is dependent on a few values, first the $T_{wavetime}$ tells us when it starts. $T_{compute}$ will tell us how long it takes to finish computing that node, $T_{comm}$ the amount of time it takes to send and receive border values, $T_{rollback}$ how long the rollbacks will take. This gives Equation 4.7.

$$T_{node} = T_{wavetime} + T_{compute} + T_{comm} + T_{rollback} \tag{4.7}$$

The overall time it takes to finish the HFMM is given my $T_{overall}$. This is the time it takes for the slowest node to finish. There are a few things that are not accounted for. The time it takes to update all nodes of their running status uses asynchronous communication and is not in $T_{communicate}$. Mostly because it takes too little time to have any affect. $T_{overall}$ is shown in Equation 4.8. P is max number of processors.

$$T_{overall} = MAX(T_{node}0, T_{node}1, ..., T_{node}P - 1) \tag{4.8}$$

Since there are quite a few values that can't be determined before runtime, a few values that vary depending on system and a few values that vary from run to run, this makes it difficult to derive a theoretical model from Equation 4.8.

## 4.2 PFMM

PFMM has one aspect that really affects performance, that's the Manhattan distance from the node which contains the pick to the farthest node.

$T_{compute}$ is the exact same as Equation 4.5 used in HFMM. That means that also $T_{loop}$ is the same. But most of the similarities to HFMMs model ends there.

The border communication for PFMM is given by Equation 4.9. It is only affected by the array size. The border exchange for PFMM is smaller than $T_{comm}$ for HFMM. This is because in PFMM only the arrival time and band array are exchanged.

$$\begin{aligned} T_{comm} &= bordersize \cdot 8 \cdot \beta + \alpha \qquad (4.9) \\ T_{comm} &= (n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) \cdot 8 \cdot \beta + \alpha \end{aligned}$$

After a border communication each node has to check if its necessary with a rollback. This operation check to see if any values on the border needs to be changed. The time for $T_{rollbackcheck}$ is given by Equation 4.10.

$$\begin{aligned} T_{rollbackcheck} &= T_{loop} \cdot bordersize \\ T_{rollbackcheck} &= T_{loop} \cdot (n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) \end{aligned} \tag{4.10}$$

Rollbacks is also an issue with PFMM the penalty for rollbacks are not that severe if the Manhattan distance for the given node is low. Given by Equation 4.11. The rollback will only take as long as $T_{node}$ in worst case, it will stop when nothing else needs to be changed , but if the Manhattan distance is below max one or more rollbacks will be hidden. $M$ is the Manhattan distance to the specific node and $MaxM$ is the max Manhattan distance, given by the distance from the node containing the starting point to the farthest node in the grid.

$$T_{rollback} = T_{node} \cdot (MAX(\#rollbacks0, \#rollbacks1, .., \#rollbacksP - 1) - MaxM + M) \tag{4.11}$$

The time a certain node needs to finish is given by Equation 4.12. See that the starting time is not contained inside this equation as it was for $T_{node}$ in HFMM. The main reason for $T_{node}$ time is the array size.

$$T_{node} = T_{compute} + T_{comm} + T_{rollbackcheck} \tag{4.12}$$

The overall time for PFMM is given by Equation 4.13. The Manhattan distance is the factor that greatly adjusts the $T_{overall}$.

$$T_{overall} = T_{node} \cdot ManhattanDistance + T_{rollback} \tag{4.13}$$

## 4.3 Validation of PFMM

To see if this function works in real life, i tested to see how the Equation corresponds between runs.

As a base for the calclulations the $160x160x160$ matrix was used. Its runtime for the serial code was 8,73 seconds.

$$\begin{aligned}
n = m = k &= 160 \\
T_{overall} &= T_{node} \cdot ManhattanDistance + T_{rollback} \\
T_{overall} &= 8.73 \\
ManhattanDistance &= 1 \\
T_{rollback} &= 0 \\
8.73 &= T_{node} \cdot 1 + 0
\end{aligned}$$

When running only one node, the Manhattan distance willbe 1. $T_{rollback}$ is set to 0 as there is no need for a roll back on the serial version.

$$\begin{aligned}
T_{node} &= T_{compute} + T_{comm} + T_{rollbackcheck} \\
T_{comm} &= 0 \\
T_{rollbackcheck} &= T_{loop} \cdot bordersize \\
T_{rollbackcheck} &= T_{loop} \cdot 160^2 \cdot 6 \\
T_{compute} &= T_{loop} \cdot arraysize \\
T_{compute} &= T_{loop} \cdot 160^3 \\
T_{node} &= T_{loop} \cdot 160^3 + 0 + T_{loop} \cdot 160^2 \cdot 6
\end{aligned}$$

$T_{comm}$ is set to 0 as there are no communication on the serial version. Since i used the parallel version for checking serial performance there is a roll back check at the end. It could be removed but it wasn't, it will give low impact on speed. The result is that $T_{node}$ is only dependent on $T_{loop}$

$$\begin{aligned}
8.73 &= T_{loop} \cdot (160^3 + 160^2 \cdot 6) \\
T_{loop} &= 2.054311 \cdot 10^{-06}
\end{aligned}$$

The result can be seen above. This is the calculated time for $T_{loop}$. The accurate value will vary depending on where in the matrix $T_{loop}$ is executed, but this result is an averge over all points. If the program ran again, $T_{loop}$ should become the same. In a parallel version there will be more $T_{loop}$ at the borders. Those are smaller as there is only one value that needs updating, but it shoudn't affect the model in a significant way.

Now we will use the results in the serial run, to calculate how much time a parallel run will use. This should be close to the measured time. A 8 node configuration will be used, with the same array as the serial run.

This means that each node will have a $80x80x80$ matrix. The $T_{loop}$ value will be used from the previous run. It might be a bit too high because in this run the rollbackcheck will be done more often than on the serial run.

$$\begin{aligned}
T_{loop} &= 2.054311 \cdot 10^{-06} \\
T_{rollbackcheck} &= T_{loop} \cdot (n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) \\
T_{rollbackcheck} &= 2.054311 \cdot 10^{-06} \cdot (80^2 \cdot 6) \\
T_{rollbackcheck} &= 0.078885542
\end{aligned}$$

For a 8 node configuratino $T_{rollbackcheck}$ was 0.078885542 seconds.

$$\begin{aligned}
T_{comm} &= bordersize \cdot 8 \cdot \beta + \alpha \\
T_{comm} &= 80^2 \cdot 6 \cdot 8 \cdot 2.143 \cdot 10^{-10} + 1.995 \cdot 10^{-6} \\
T_{comm} &= 0.001336609
\end{aligned}$$

The communication time for Njord is very low. $T_{comm}$ is very low, only 0.001336609 seconds.

$$\begin{aligned}
T_{compute} &= T_{loop} \cdot arraysize \\
T_{compute} &= 2.054311 \cdot 10^{-06} \cdot 80^3 \\
T_{compute} &= 1.0518072
\end{aligned}$$

$T_{compute}$ is 1.0518072 seconds. The serial runtime was 8.73 seconds which divided by 8 is 1.09125. Close to the calculated performance for each node.

$$\begin{aligned}
T_{node} &= T_{compute} + T_{comm} + T_{rollbackcheck} \\
T_{node} &= 1.0518072 + 0.001336609 + 0.078885542 \\
T_{node} &= 1.1320294
\end{aligned}$$

A node will complete in 1.1320294 seconds.

$$\begin{aligned}
ManhattanDistance &= 4 \\
T_{rollback} &= 0 \\
T_{overall} &= T_{node} \cdot ManhattanDistance + T_{rollback} \\
T_{overall} &= 1.1320294 \cdot 4 \\
T_{overall} &= 4.5281175
\end{aligned}$$

ManhattanDistance is set to 4, this is because that will be the largest distance from the starting point to the fathers edge in a 2x2x2 node grid. $T_{rollback}$ are ignored, we hope there was no rollbacks.

$T_{overall}$ was calculated to 4.52 seconds the measured time was 4.11 seconds. This means that $T_{loop}$ might be a bit incorrect for this run. It is probably a bit high since there are more border checks in a 8 node division than on the serial run. It is still close to the measured performance.

## 4.4 Validation of HFMM

To calculate the performance of the HFMM algorithm, is a challenge. There are many values that only can be estimated. To make this easier I have taked the 8 node run of the HFMM the same setup as in the validation of PFMM.

$$\begin{aligned}
T_{loop} &= 2.054311 \cdot 10^{-06} \\
T_{compute} &= T_{loop} \cdot arraysize \\
T_{compute} &= 2.054311 \cdot 10^{-06} \cdot 80^3 \\
T_{compute} &= 1.0518072
\end{aligned}$$

$T_{compute}$ is 1.0518072 seconds. The same as in PFMM.

$$\begin{aligned}
T_{comm} &= (bordersize + resendCount) \cdot 20 \cdot \beta + \alpha \\
T_{comm} &= (80^2 \cdot 6 + resendCount) \cdot 20 \cdot 2.143 \cdot\ 10^{-10} + 1.995 \cdot 10^{-6} \\
T_{comm} &= 0.001645824 + 4.286 \cdot 10^{-9} \cdot resendCount + 1.995 \cdot 10^{-6}
\end{aligned}$$

$T_{comm}$ is dependent on the resendCount, this value is difficult to predict before the application starts.

$$\begin{aligned}
T_{traverse} &= arraysize \cdot mtime \\
T_{rollback} &= \#rollback \cdot T_{traverse} \\
T_{rollback} &= \#rollback \cdot 80^3 \cdot mtime
\end{aligned}$$

It is difficult to say how many rollbacks a run will have. This depends on differences in computational speed between the nodes, network latency and the velocity field.

$$\begin{aligned}
T_{node} &= T_{wavetime} + T_{compute} + T_{comm} + T_{rollback} \\
T_{node} &= T_{wavetime} + 1.0518072 + 0.001645824 + 4.286 \cdot 10^{-9} \cdot resendCount + 1.995 \cdot 10^{-6} + T \\
T_{node} &= 1.053455 + T_{wavetime} + T_{rollback} + 4.286 \cdot 10^{-9} \cdot resendCount \\
T_{node} &= 1.053455 + T_{wavetime} + T_{rollback}
\end{aligned}$$

This is as close to an answer to $T_{node}$ as one can. The resend count is remove becuase Njord has such good interconnect that it will be negliable. The measured time for this run is 9.9 seconds. Each node uses 1.053455. This means that $T_{wavetime}$ and $T_{rollback}$ accounts for almost 9 seconds. The point start in the middle, that means one of the nodes will have the starting point. It will be exchanged to all nodes within the first few loop runs. So $T_{wavetime}$ is small. Therefore, $T_{rollback}$ accounts for most of the 9 seconds, which is very bad.

# Chapter 5

# Performance analysis

This chapter will describe and explain the different results from the different test runs. All speedup measurements are taken compared to PFMM running on one node. This is because PFMM will give mostly the same results as a serial implementation of algorithm 3.3. The measurements are taken before and after the FMM executes. It doesn't take into account the time it takes to distribute the velocity matrix or initialize all the variables.

All the test have been executed on the njord super computer, unless otherwise specified.

## 5.1 HFMM

HFMM is an implementation of Herrmanns algorithm 2.10. The results from this algorithm didn't give the same timings each time as algorithm 2 did. This will most likely be because of the asynchronous nature of the application. All communication was made asynchronous because the usage of synchronous communication for each loop gave too much overhead, which killed performance, even more than the asynchronous version.

From Figure 5.1, one can see that its not much above 1. This means that the application didn't work faster when it ran on more nodes. This can mostly be credited to the rollback function, as you can see from Figure 5.2 the speedup was drastically better without the rollbacks.

The version without rollbacks show that the rollbacks kill performance, but still there are problems with HFMM. There is also a decrease in performance when we use a non optimal domain decomposition. 9 and 27 nodes, give less performance.

There aren't any speed measurements for HFMM above 64 nodes. This is because it has issues with performance when you go from 16 to 32 and even more when you hit 64 nodes. The best performance for HFMM is when it ran on 8-9 processors.

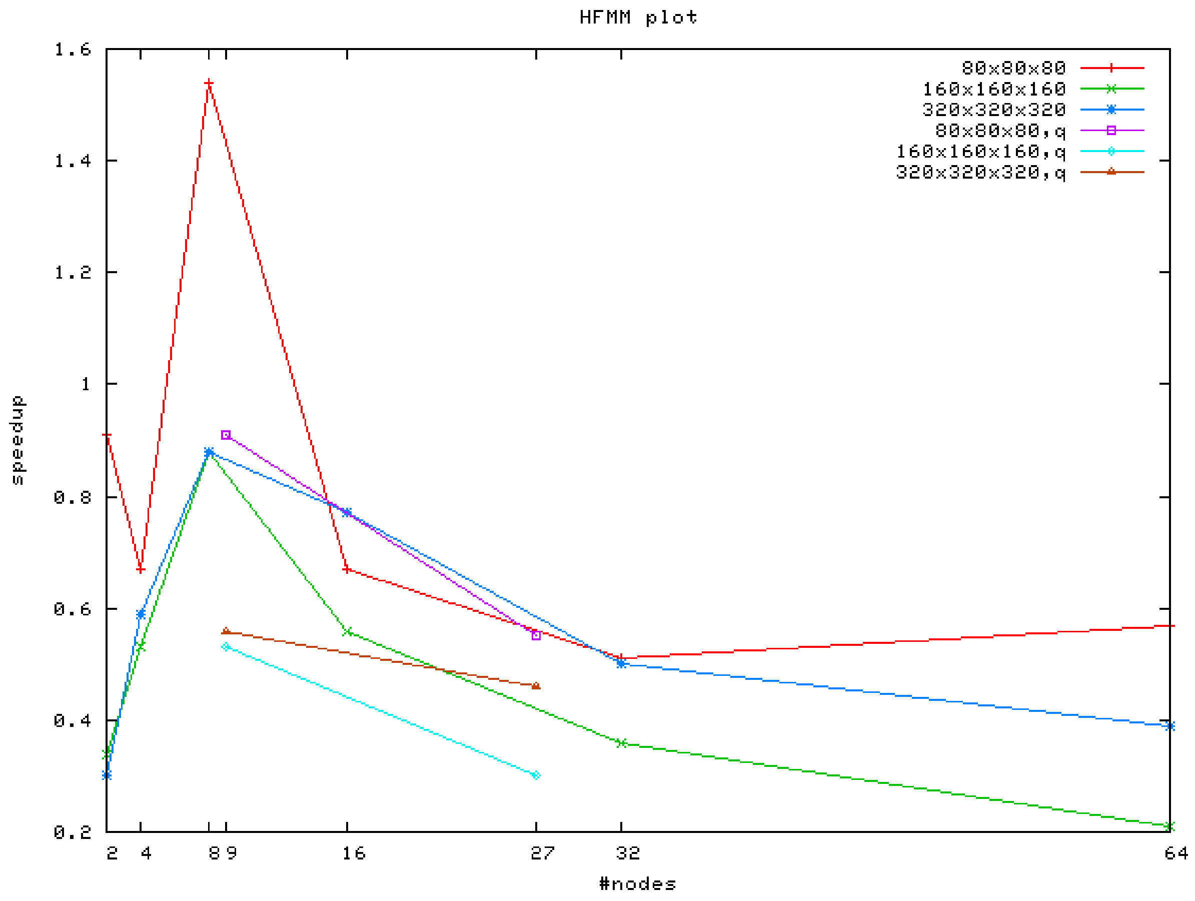


Figure 5.1: Speedup from HFMM

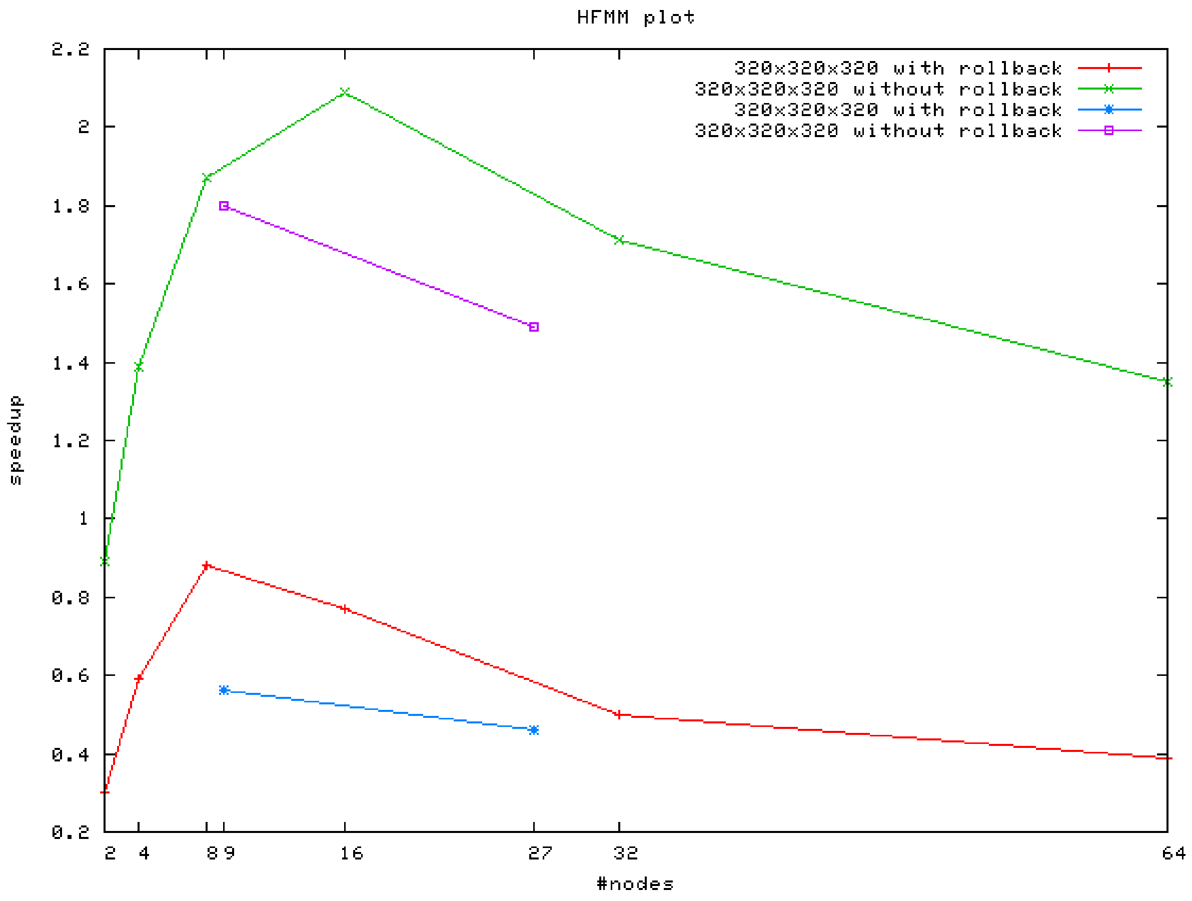


Figure 5.2: Speedup for HFMM with and without rollback

### 5.1.1 Analasis of HFMM

HFMM showed very poor results compared to what one could expect. Herrmann got his algorithm to run with an efficiency of 0.98 with his optimal decomposition of 4 or 8 nodes. He also used a sphere as a starting figure, instead of a point which was used here. His velocity field was all 1. This should, in theory, give an efficiency of 1 without border communication. HFMM uses a lot of communication, it sends each border when it's calculated. It also uses probe functions to check for incoming messages. Since it uses asynchronous communication it should not be hindered as much as it is by communication. It seems the time to send and check for incoming messages uses a lot more resources than one would think.

With some further work it could be possible to remove some of the communication overhead shown in HFMM. That fact doesn't change the fact that it suffered heavily from rollbacks. When it ran with rollback enabled it got so bad that it didn't even beat the serial application.

To avoid rollbacks it would be an idea to use synchronous communication so each node would not get values that were calculated a few loops ago. Then there would be no rollbacks in Herrmanns example and the efficiency described in Herrmann might be achieved, but the overhead of communicating the border nodes when they are discovered makes it impossible for me to achieve these kinds of speedups. It seems theoretically possible to achieve the efficiency Herrmann got, but it's has proven practically difficult. PFMM would most likely get close to that in performance.

## 5.2 PFMM

PFMM worked much better than HFMM.

In Figure 5.3 it is shown that the application increases its speed as the number of nodes increases. As the 80x80x80 matrix reaches 16 nodes, it has a huge performance increase compared to the larger matrices. The reason for this is that mostly all data can be fitted in cache with such a small matrix. However, as it continues form 32 nodes, it drops drastically. This is because the matrix are now so small on each node. Each node will only have a 20x20x20 matrix. Therefore, the time for communicating the borders will be much larger than calculation times. The same can be observed for 160x160x160 matrix. There is also a good increase in performance for 9 and 27 nodes. This can be accredited to the Manhattan distance to all nodes which is 3 on nine nodes, compared to 4 on eight nodes. The same can be said for 27 nodes where the Manhattan distance to the farthest node is 4, but its 5 on 32 nodes. As the matrix size increases, there is work for more nodes.

In Figure 5.4 it is apparent that large matrices gain a lot from more nodes.

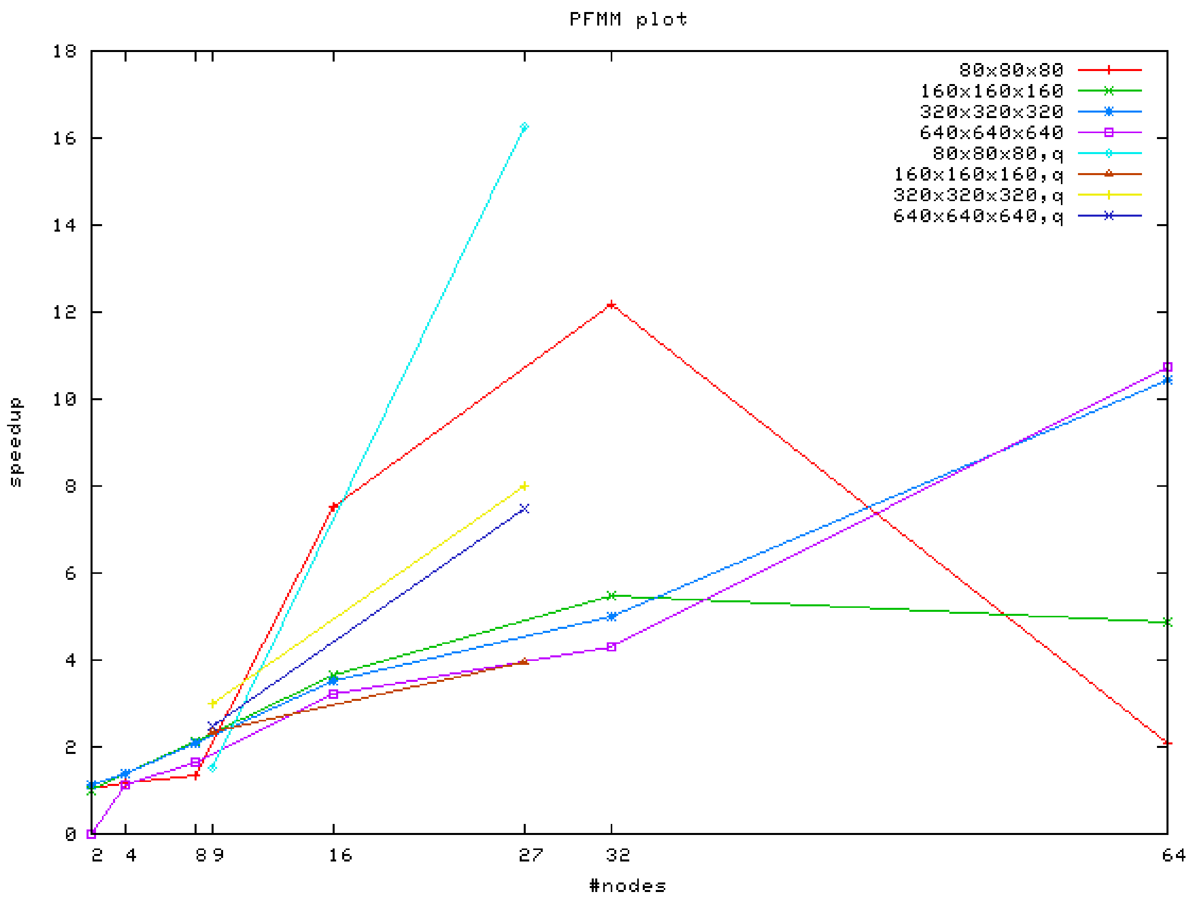


Figure 5.3: Speedup for PFMM

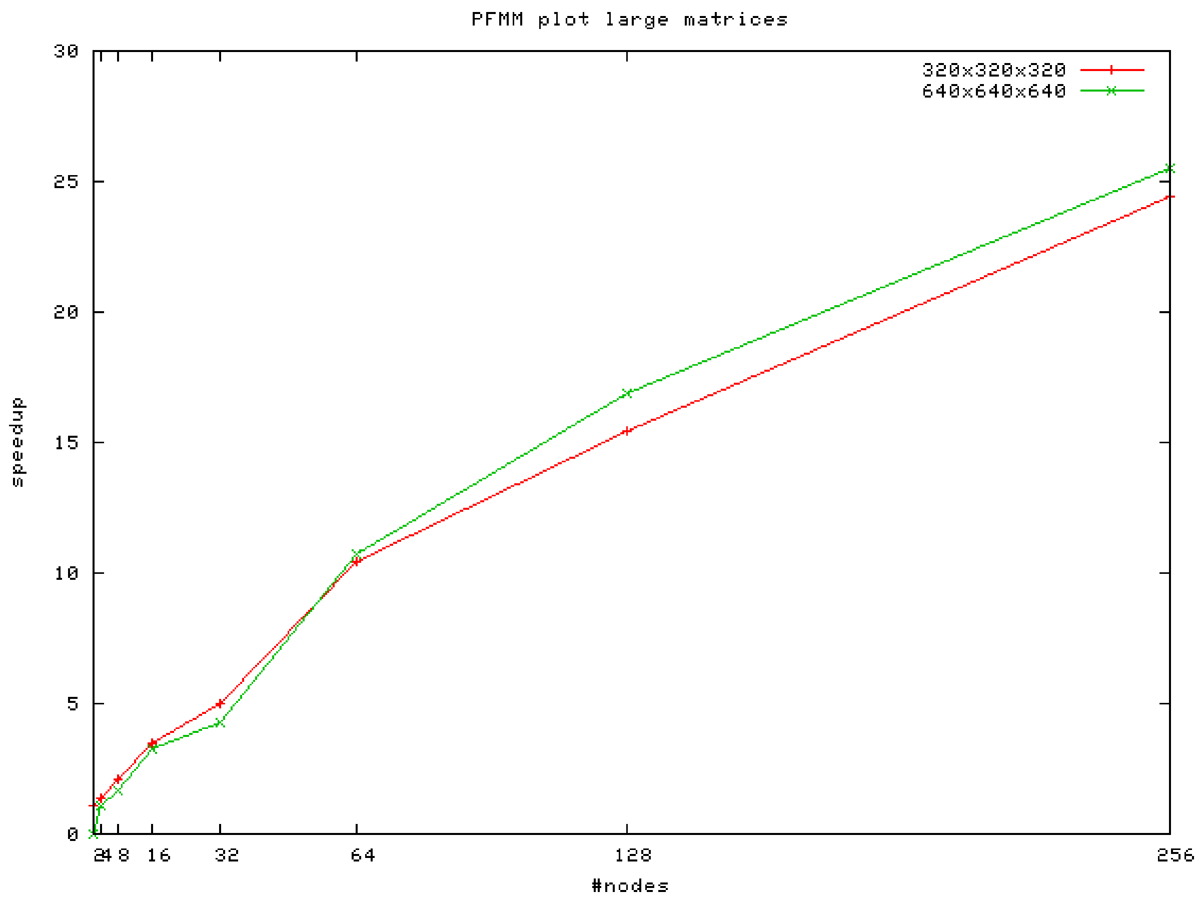


Figure 5.4: Speedup for PFMM, 256 nodes

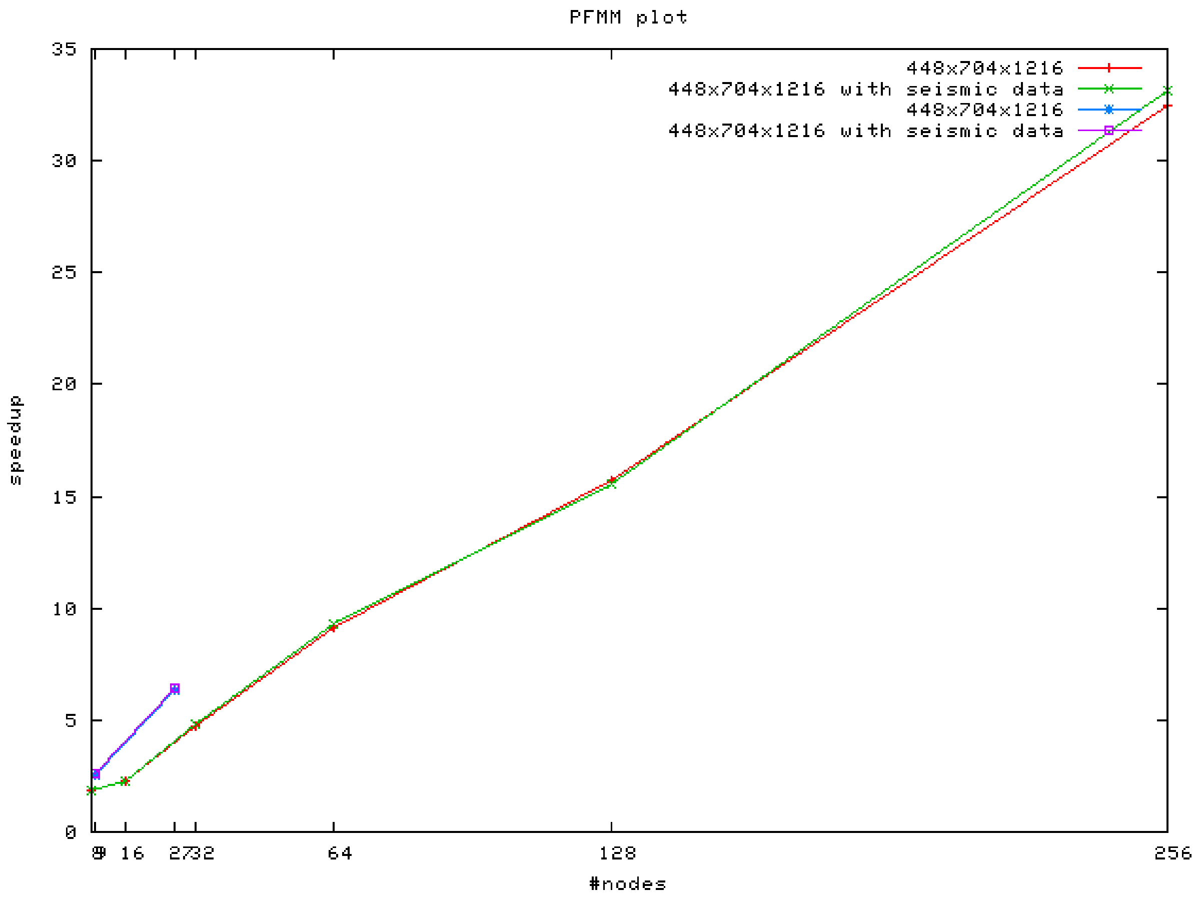


Figure 5.5: Speedup for PFMM, salt data

In Figure 5.6 one can see the difference between using a velocity field of 1 and using real seismic data. The difference is probably because of small differences in the measurements. They are mostly identical, which means that there are no more iterations when using a seismic dataset compared to a velocity field of 1. If there are more iterations, it's carried out by the nodes in the middle.

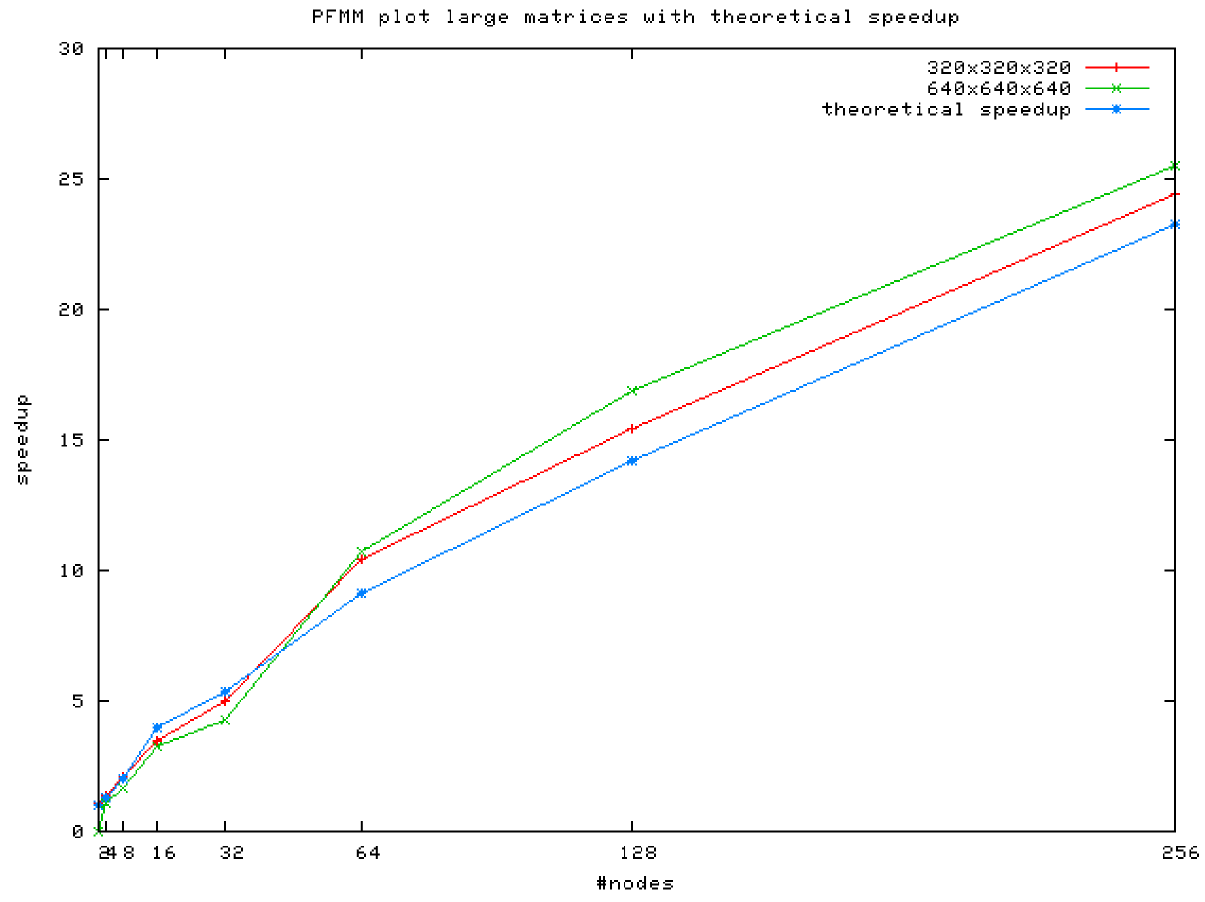


Figure 5.6: Speedup for PFMM compared to theoretical speedup

In Figure 5.6 the speedup is compared to theoretical speedup. The theoretical speedup is calculated from Equation 5.1.

$$M/n = speedup \tag{5.1}$$

Where M is the Manhattan distance and n is the number of nodes used.

This is just to get a perspective on how the theoretical max speedup would be if we remove $T_{rollback}$ and $T_{comm}$.

In the two matrix sizes there is a change between 32 and 64 nodes. When the program uses 64 nodes it runs faster than the theoretical speedup. Even the largest matrix runs past the smaller matrix. The reason they beat the theoretical speedup must be because the problem size fits into level three cache, which is very large on Njord. The reason for the largest matrix becoming faster than

| # nodes | Calculated | Measured |
|---|---|---|
| 2 | 8.7296 | 8.79 |
| 4 | 6.6669 | 6.31 |
| 8 | 4.5232 | 4.11 |
| 16 | 2.8226 | 2.4 |

Table 5.1: Theoretical and measured runtime for PFMM

the smaller one, is because there is less work/transfer time ratio than on the smaller matrix. In the smaller matrix, the transfer time for borders dominates more than on the larger matrix.

Another way to compare the runtime to theoretical, is to use the more advanced model, shown in Section 4.2 PFMM. In Table 5.2 one can see the theoretical time compared to the measured time. This is calculated from Equation 4.13. $T_{comm}$ is set to 0, beacuse its so low that it won't affect the results in a significant way, also we assume no rollbacks.

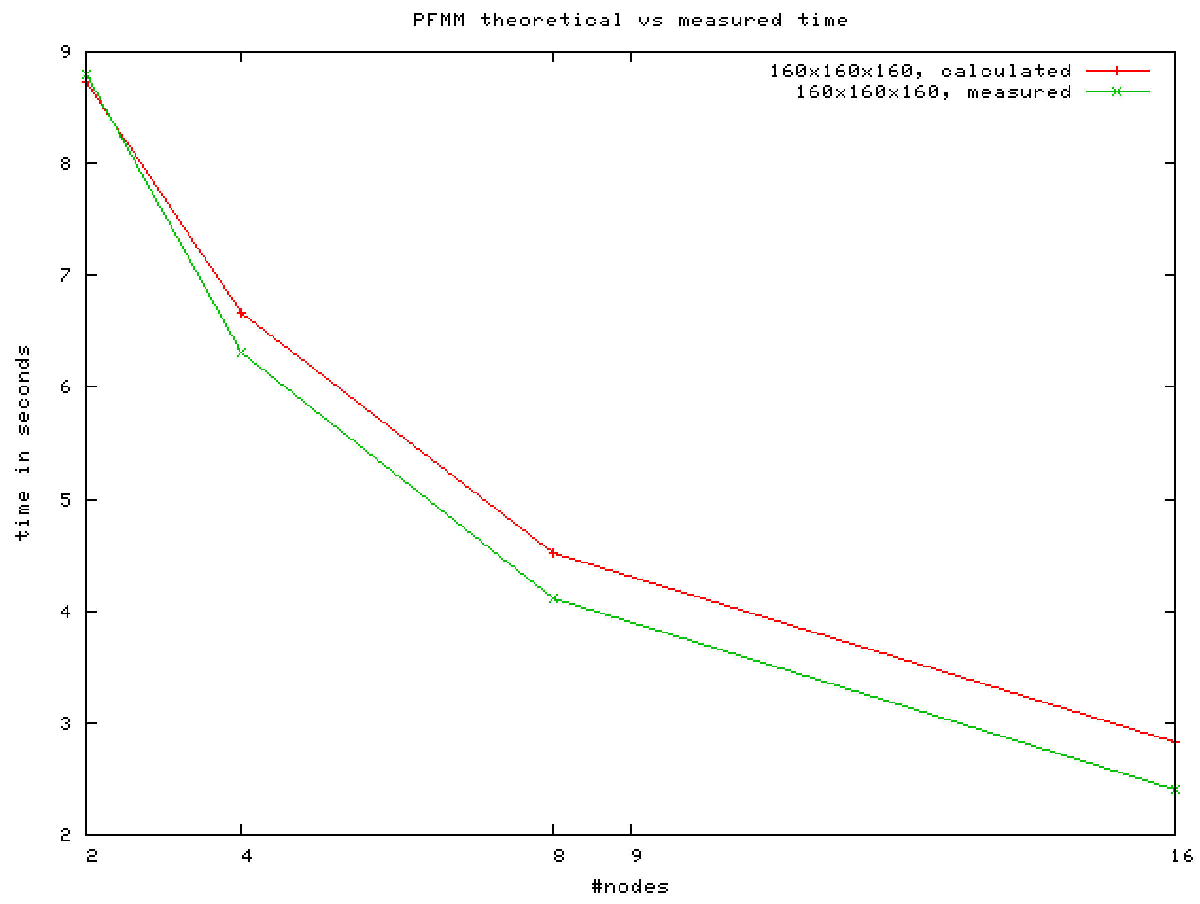


Figure 5.7: PFMM theoretical vs measured time

From Figure 5.7, the difference in theoretical vs measured time can be seen. The reason for its difference can be that $T_{loop}$ are smaller when the matrix size is smaller. Another reason can be better utilization of the cache, and as a result it goes faster.

### 5.2.1 Analasis of PFMM

PFMM gave much better results than HFMM. The reason for this was that it did much less communication and almost avoids rollbacks. In the case presented by Herrmann where we use 4 computing nodes, and start with a sphere in the middle. Then all nodes would have work and no nodes would require a rollback. This would give almost four in speedup compared to the serial version. Though this situation is not interesting for finding salt formations. There the start condition is a single point.

There are a few ways one can calculate arrival times. The normal 1 order approximation. Higher order approximations and using Spherical coordinates. Spherical coordinates gave more correct results but it's hard to follow the head wave. Using Cartesian coordinates approximations doesn't give the correct solution, buts it's not that far from it either. Using a higher order approximation would make it more correct but also take longer to calculate. The applications use 1 order approximation. Mostly because it gives good enough approximations but also because it is faster than the other methods.

When the number of compute nodes increased, the smaller problem sizes had problems maintaining performance. This was because the problem size on each node became so small, that the cost of communicating borders affected the performance. On problem sizes above 320x320x320 could easily run on 256 nodes and still maintain good performance. 160x160x160 had problems above 32 nodes, while 80x80x80 had the same problem. 80x80x80 also gave very good results on 16 to 32 nodes. This would be because the prolbem size then fitted into cache. 160x160x160 might also have such a case between 32 and 64 nodes. But at 64 nodes it gave less performance than at 32 nodes.

The most important factor for speed on PFMM is the Manhattan distance to the farthest node from the starting node. Since all test result was measured by putting the point in the middle of the problem matrix. Distributions which had odd number in each dimension gave the best results. 9 nodes gave much better results than 8 because of the Manhattan distance of 3 vs 4. The same was with 27 and 32, with Manhattan distance of 4 and 5.

Theoretical speedup was calculated by dividing number of nodes by the Manhattan distance. PFMM did beat the theoretical speedup when it reached 64 nodes. This was surprising because $T_{rollback}$ and $T_{comm}$ was ignored in the theoretical speedup. $T_{rollback}$ is probably 0 in all the cases, but $T_{comm}$ is present but probably negligible as Njord has a very fast interconnect. The most probably reason for getting higher than theoretical speedup, is because the problem size must fit in cache. The performance below 64 nodes is still good.

Choosing a decomposition of the problem size that gave the smallest amount or border size, had negligible effect on the speed. It was not possible to distinguish that from normal variations on test runs, which where very small. Normally less than a few percent. The important factor are that the nodes are divided making the Manhattan distance the smallest.

## 5.3 Comparing HFMM and PFMM

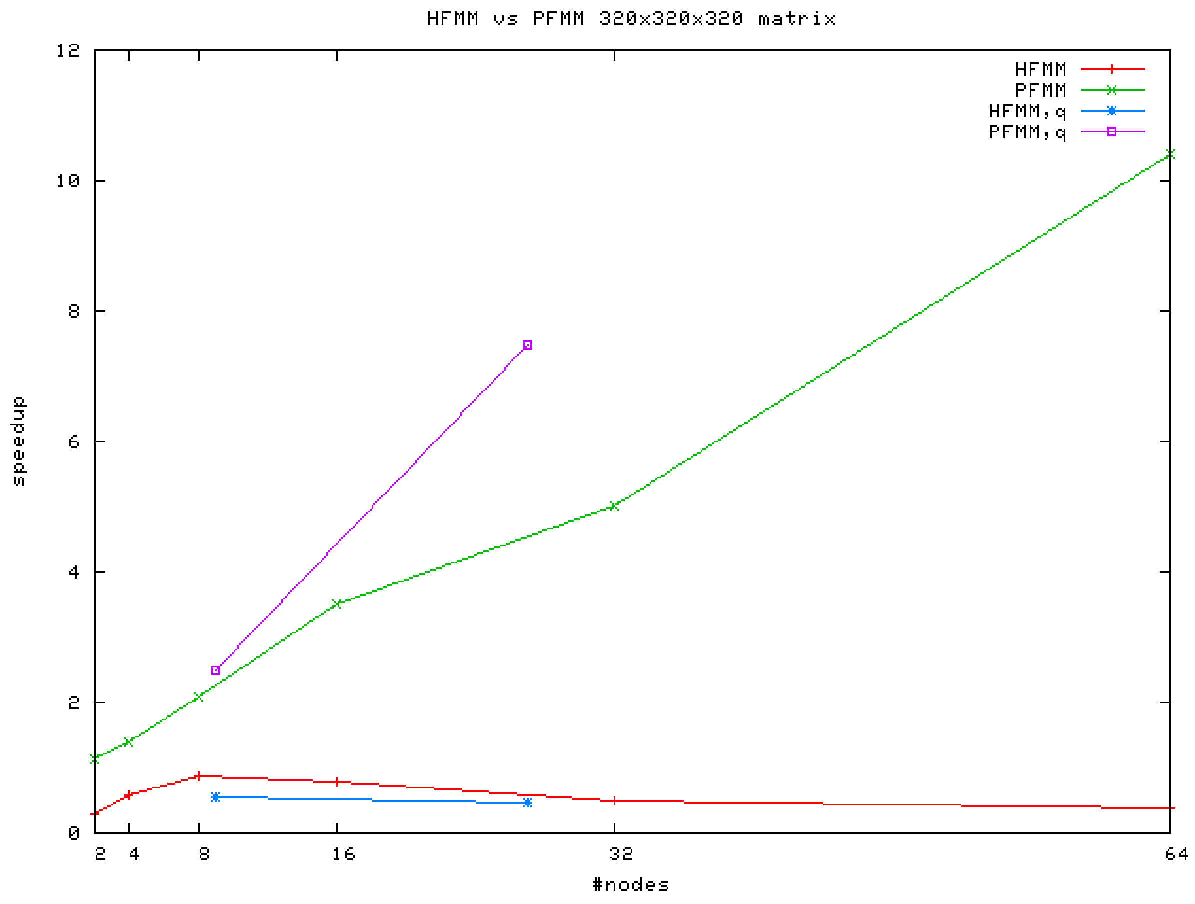


Figure 5.8: 320x320x320 matrix comparing HFMM and PFMM

In Figure 5.8, it is obvious that the PFMM works much better. But in theory the HFMM should work very well. But it kneels under the load of sending incremental border exchanges, and rolling back when receiving a value that requires a rollback. In HFMM the other nodes should start working before they start working in PFMM. But because of the penalty involved in sending and receiving values it doesn't beat the performance of PFMM. When HFMM reaches 8 nodes performance starts to drop, this is because the communication and rollbacks start to take much longer because there are much more communication on 16 and 32 nodes than on 8. PFMM scales better, and is not affected by increasing the amount of nodes.

# Chapter 6

# Conclusion

This thesis has focused on developing a parallel method for the Fast Marhcing Method, used in finding salt formations in seismic data. A algorithm from Herrmann was looked at and used as a reference point. Two different approaches of making a parallel Fast Marching Method was tried and tested.

HFMM used the algorithm described be Herrmann. It didn't perform as fast as expected. This was because of too many rollbacks. Because of the asynchroneous communication and different execution times at each node, rollbacks became plentiful. Rollbacks stand for 80-90% of the execution time for 8 node configuration. At first synchroneous communication was tested, this resulted in too much overhead in communication and performed worse than the asynchroneous version.

PFMM had much better performance. The teoretical speedup of this algorithm number of computational nodes divided by the manhattan distance from the starting point to the farthest node. The application ran almost as fast as the theroretical speedup, but with more than 32 nodes it gave faster than theoretical speedup. Because of the one pass nature of the Fast Marching Method it is impossible to get full cpu utilization of all nodes. This can be improved by running more points at the same time, giving only a small percentage penalty, more info in Future work.

An execution time model was developed, this model performed well for PFMM algorithm. It came very close to the actual execution time. If more work had been laid into estimating $T_{loop}$ it would have been even better. The model for HFMM was much more difficult, asynchroneous communication made it very difficult to predict how many rollbacks would be used. The time used in computation for HFMM is very low compared to what is used for rollbacks and communication.

## 6.1 Future work

There are a few optimizations that would be interesting to try, given that the cpu utilization is so low on each node. One optimization would be running more picks at the same time. Also changing the decomposition so the manhattan distance becomes smaller would increase speed.

The speedup from the application was good compared to what the algorithm could theoretical achieve. However it's not a very good utilization of the computing nodes. The start node would be idle while the other nodes calculate their nodes. There are a way to avoid some of this problem, by running more FMM simultaneous. When a geologist tries to locate Salt they pick two to three points at the same time. If two picks could execute at the same time, it would only give a slight increase in computing time when a node had to process bout waves at the same time. This should only occur if the two picks had a node with the same Manhattan distance from each pick. In the case of a 5x5 node grid. There are two collisions, at distance 2 and 3. Figure 6.1 shows the distance for pick 1, while Figure 6.2 shows the distance for pick 2. In Figure 6.3 the colliding nodes are in bold. Since the max distance is 6 for each pick and bouth distance 2 and 3 has colliding nodes. The distance will become 8. Thats a 33% increase in execution time for running two picks at the same time. Which must be a very good increase in performance. It might be even better for 3 picks but more nodes would probably collide. This was not tested and are left as an optimization for later.

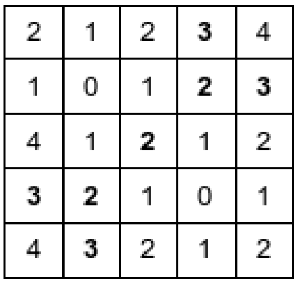
| | | | | |
|---|---|---|---|---|
| 2 | 1 | 2 | 3 | 4 |
| 1 | 0 | 1 | 2 | 3 |
| 2 | 1 | 2 | 3 | 4 |
| 3 | 2 | 3 | 4 | 5 |
| 4 | 3 | 4 | 5 | 6 |

Figure 6.1: Manhattan distance for pick 1

| | | | | |
|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 4 |
| 5 | 4 | 3 | 2 | 3 |
| 4 | 3 | 2 | 1 | 2 |
| 3 | 2 | 1 | 0 | 1 |
| 4 | 3 | 2 | 1 | 2 |

Figure 6.2: Manhattan distance for pick 2

In this thesis only rectangular domain decompositions are tested. It might be an idea to test a domain decomposition that use beams from the pick point. In



| | | | | |
|---|---|---|---|---|
| 2 | 1 | 2 | **3** | 4 |
| 1 | 0 | 1 | **2** | **3** |
| 4 | 1 | **2** | 1 | 2 |
| **3** | **2** | 1 | 0 | 1 |
| 4 | **3** | 2 | 1 | 2 |

Figure 6.3: Manhattan distance for bouth picks, bold where they collide

a 2D case each node could take an equal amount of degrees from the starting point, a cake piece. This will probably give more equal work among the different nodes. The correct result would probably be acheved in 2-3 iterations, beacuse the Manhattan distance would then be 1. The extra iterations are added if the wave moves in and out of different nodes, which is very likely, since the velocity field is not a single value.

Implementing such a division is not straight forward. It will be much more difficult to find out which point in the matrix are on which node. This is because a line at 32 degrees will split many points in the grid. Which one should belong to which node, and how do we add border values. This is solvable but will make the application much more complex. It can also be extended to 3D space by adding another dimension.

# Bibliography

[1] Tariq Alkhalifah and Sergey Fomel. Implementing the fast marching eikonal solver: Spherical versus cartesian coordinates. Available from World Wide Web: citeseer.ist.psu.edu/543803.html.

[2] N. Sukumar, D. L. Chopp, and B. Moran. Extended finite element method and fast marching method for three-dimensional fatigue crack propagation. Available from World Wide Web: citeseer.ist.psu.edu/654120.html.

[3] Yan J Zhuang TG Zhao B Schwartz LH. Lymph node segmentation from ct images using fast marching method. Available from World Wide Web: http://www.ncbi.nlm.nih.gov/sites/entrez?cmd=Retrieve\&db=PubMed\&list_uids=15127747\&dopt=Abstract.

[4] Alexandru Telea and Jarke J. van Wijk. An augmented fast marching method for computing skeletons and centerlines. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 251–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[5] Scott A. Morgan Morgan P.Brown and Amerada Hess Corporation Gary Whittle. Seismic event tracking by global path optimization. *SEG/New Orleans 2006 Annual Meeting*, 2006. Article.

[6] Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, WACG*. LNCS, 1996. Available from World Wide Web: citeseer.ist.psu.edu/126089.html.

[7] MPI Forum. Message passing interface, specification. Available from World Wide Web: http://www.mpi-forum.org/docs/.

[8] Notur. Njord. Available from World Wide Web: http://www.notur.no/hardware/njord/.

[9] IBM. Power5 system microarchitecture. Available from World Wide Web: http://researchweb.watson.ibm.com/journal/rd/494/sinharoy.html.

[10] Eugene F. Krause. Taxicab geometry, 1973.

[11] J. Sethian. A fast marching level set method for monotonically advancing fronts. In *Proc. Nat. Acad. Sci.*, volume 93, pages 1591–1595, 1996. Available from World Wide Web: `citeseer.ist.psu.edu/sethian95fast.html`.

[12] D. Adalsteinsson and J. Sethian. The fast construction of extension velocities in level set methods. *Journal of Computational Physics*, 148:2–22, 1998. Available from World Wide Web: `citeseer.ist.psu.edu/adalsteinsson97fast.html`.

[13] Elisabeth Rouy and Agn&#232;s Tourin. A viscosity solutions approach to shape-from-shading. *SIAM J. Numer. Anal.*, 29(3):867–884, 1992.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001. Available from World Wide Web: `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{\&}path=ASIN/0262531968`.

[15] M. Herrmann. A domain decomposition parallelization of the fast marching method, 2003.

# Appendix A

# Application 1 source code

## A.1 Array

### A.1.1 array_mpi.h

```c
#ifndef ARRAY_H
#define ARRAY_H

#include "mpi.h"
/*
 * Defines the size of the array
 */
#define SIZEX 160//448//8//448
#define SIZEY 160//704//4//704
#define SIZEZ 160//1216//4//1216
/*
 * Calculate the strides for global array
 */
#define stride0 ((SIZEZ+2)*(SIZEY+2))
#define stride1 (SIZEZ+2)
#define stride2 1
/*
 * Gets the index for a position in the global array
 */
#define GETINDEX(i,j,k) stride0*(i+1) + stride1*(j+1) + stride2*(k+1)
/*
 * Get the index for a position in the local array
 */
#define GETLINDEX(i,j,k) ((local_z+2)*(local_y+2)) *(i+1) + ((local_z+2)*(j+1)) + k+1
/*
 * A large float number, should be larger than anything you calculate
 */
#define BIGFLOAT 10000000.0
/*
 * The size of each array, for mallocing memory
 */
```

```
#define ARRAYSIZE (SIZEX+2)*(SIZEY+2)*(SIZEZ+2)
#define LOCALARRAYSIZE (local_x+2)*(local_y+2)*(local_z+2)
/*
 * Rank, MPI rank
 * cartrank, rank in the cartesian grid
 * size, number of nodes used
 */
int rank,cartrank, size;
/*
 * Ranks of nodes that are above,below,west,east,north,south
 */
int above, below, west, east, north ,south;
/*
 * Number of nodes in each dimension
 */
int dims[3];

/*
 * communicator for the cartesian grid
 */
MPI_Comm gridcomm;
/*
 * MY coordinates in the cartesian node grid
 */
int coords[3];
/*
 * size in each dimension of the global array
 */
int x,y,z;
/*
 * Size in each dimension of the local array
 */
int local_x,local_y,local_z;
/*
 * Print the values of a global array to stdout
 */
void printArray(float* array);
/*
 * Print the local array to stdout
 */
void printLocalArray(float* array);
/*
 * print a local int array to stdout
 */
void printLocalIntArray(int* array);
/*
 * Get what node a global position resides in
 */
int getDest(int x, int y, int z);
/*
 * Get global coordinates from local coordinates
 */
int* getGlobalCord(int x, int y, int z);
/*
 * Get local coordinates from global coordinates
 */
int* getLocalCord(int x, int y, int z);
#endif
```

### A.1.2 array_mpi.c

```
#include "array_mpi.h"
#include <stdlib.h>
#include <stdio.h>

/*
 * converts global coord to local coord
 */
int* getLocalCord(int xg, int yg, int zg){
        int *cr;
        cr = malloc(sizeof(int)*3);
        cr[0] = xg;
        cr[1] = yg;
        cr[2] = zg;
    cr[0] -= coords[0] * local_x;
    cr[1] -= coords[1] * local_y;
    cr[2] -= coords[2] * local_z;
        return cr;
}
/*
 * convert local coord to global coord
 */
int* getGlobalCord(int x, int y, int z){
        int *cr;
        cr = malloc(sizeof(int)*3);
        cr[0] = x+ local_x*coords[0];
        cr[1] = y+ local_y*coords[1];
        cr[2] = z+ local_z*coords[2];
        return cr;
        }
/*
 * gets the rank of the node that have input position
 */
int getDest(int x, int y, int z){
        int cr[3];
        int value;
        cr[0] = x/local_x;
        cr[1] = y/local_y;
        cr[2] = z/local_z;
        MPI_Cart_rank(gridcomm,cr,&value);
        return value;
}
/*
 * print the local float array to stdout
 */
void printLocalArray(float* array){
       int i,j,k;
        for(k=-1;k<=local_z;k++){
        printf("%d:array␣z␣=␣%d\n",cartrank,k);
                for(i=-1;i<=local_x;i++){
                        printf("%d:␣",cartrank);
                        for(j=-1;j<=local_y;j++){
                                printf("␣%8f␣",array[GETLINDEX(i,j,k
                                    )]);
                        }
                        printf("\n");
                }
```

```
		printf("\n\n");
		}
}
/*
 * print a local integer array to stdout
 */
void printLocalIntArray(int* array){
		int i,j,k;
		for(k=-1;k<=local_z;k++){
		printf("%d: array␣z␣=␣%d\n",cartrank,k);
				for(i=-1;i<=local_x;i++){
						printf("%d:␣",cartrank);
						for(j=-1;j<=local_y;j++){
								printf("␣%d␣",array[GETLINDEX(i,j,k)
									]);
						}
						printf("\n");
				}
		printf("\n\n");
		}
}
/*
 * print the global array to stdout
 */
void printArray(float* array){
		int i,j,k;

		for(k=-1;k<=z;k++){
		printf("array␣z␣=␣%d\n",k);
				for(i=-1;i<=x;i++){
						for(j=-1;j<=y;j++){
								printf("␣%8f␣",array[GETINDEX(i,j,k)
									]);
						}
				printf("\n");
				}
		printf("\n\n");
		}
}
```

## A.2 Fast Marching Method

### A.2.1 fmm_mpi.h

```
#include "heap.h"

/*
 * a struct to store all variables for a given fmm implementation
 */
typedef
struct f {
  Heap* heap; // the heap
  float* timearray; // the timearray storing arrivaltimes
  int* bandarray; // storing band information
  float* velocityarray; // velocity field
  float* sentarray; // array containting values of sent points, used to avoid sending the same value multiple times
  int x,y,z,posx,posy,posz; // x,y,z is size of local fmm matrix, posx, posy, posz is global coords for the starting point
  } FmmData;

typedef
/*
 * Struct used for sending a border cell to another node. String its value and posistion with band information
 */
struct me {
        float value;
        int x;
        int y;
        int z;
  int n;
} MPI_Element;
/*
 * initilalize the FMM set velocity arrat, position of starting point and size of array
 */
FmmData* initFMM(float* velarray,int posx,int posy, int posz, int x, int y, int z);
/*
 * Free up used variables in FMM
 */
void freeFMM(FmmData* data);
/*
 * Execute the FMM
 */
void executeFMM(FmmData* data);
```

### A.2.2 fmm_mpi.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "fmm_mpi.h"
#include "heap.h"
#include "array_mpi.h"


#define BAND 0
#define OUTSIDE -1
#define KNOWN n

//#define DEBUG
/*
 * number of loops
 */
int n;

/*
 * largest_solution is the largest value in the array
 * rollbacksmallest is the smallest value of received border values,
     which dictates the rollback number
 */
float largest_solution, rollbacksmallest;

/*
 * the n value which one should rollback to
 */
int rollbackn;
/*
 * mpi datatype for sending border points
 */
MPI_Datatype mpi_element_struct;
/*
 * Used for debug output
 */
float valuemax=0;
float valuemin=0;
/*
 * array containing the working status of each node
 */
int* working;

/*
 * Add a point to the heap
 */
void addToHeap(FmmData* data, int px, int py, int pz){
  Element *temp;
  #ifdef DEBUG
  printf("%d: adding %d, %d, %d to heap,heap size is %d maxsize if %
      d\n",cartrank,px,py,pz,data->heap->heapsize,data->heap->maxsize
      );
  #endif
  temp= malloc(sizeof(Element));
  temp->value = data->timearray[GETLINDEX(px,py,pz)];
  temp->x = px;
  temp->y = py;
  temp->z = pz;
  heapInsert(data->heap,temp);
  #ifdef DEBUG
  printf("%d:␣inserted␣%d,␣%d,␣%d␣to␣heap,heap␣size␣is␣%d\n",
     cartrank,px,py,pz,data->heap->heapsize);
  #endif

}

/*
 * initialize the FMM
 * velarray is the velocity field
 * posx,y,z is position of the starting point
 * x,y,z is the size of the array, most are read from array_mpi.h
 */
FmmData* initFMM(float* velarray, int posx,int posy, int posz, int x
   , int y, int z){
  working = malloc(sizeof(int) * size);
  /* init datatypes*/
  MPI_Element e;
  MPI_Datatype type[5] = { MPI_FLOAT, MPI_INT, MPI_INT, MPI_INT,
     MPI_INT };
  int blocklen[5] = { 1, 1, 1, 1 , 1};
  MPI_Aint disp[5];
  disp[0] = 0;
  disp[1] = sizeof(float);
  disp[2] = sizeof(int) + disp[1];
  disp[3] = sizeof(int) + disp[2];
  disp[4] = sizeof(int) + disp[3];
  MPI_Type_create_struct(5, blocklen, disp, type, &
     mpi_element_struct);
  MPI_Type_commit(&mpi_element_struct);
  /* end init datatypes*/
  #ifdef DEBUG
  printf("%d:␣done␣init␣mpi␣datatypes\n",cartrank);
  #endif
  int send = 0;
  FmmData* data;
  data = malloc(sizeof(FmmData));
  if(data == 0){
    printf("%d:␣Failed␣to␣allocate␣memory,␣exiting\n",cartrank);
    exit(1);
  }

  data->bandarray = malloc(sizeof(int)*LOCALARRAYSIZE);
  if(data->bandarray == 0){
    printf("%d:␣Failed␣to␣allocate␣memory,␣exiting\n",cartrank);
    exit(1);
  }
  data->timearray = malloc(sizeof(float)*LOCALARRAYSIZE);
  if(data->timearray == 0){
    printf("%d:␣Failed␣to␣allocate␣memory,␣exiting\n",cartrank);
    exit(1);
  }

  data->velocityarray = velarray;
```

```
#ifdef DEBUG
printf("%d:␣initializing␣heap\n",cartrank);
#endif
data->heap = initHeap(LOCALARRAYSIZE);
data->x = x;
data->y = y;
data->z = z;
data->posx = posx;
data->posy = posy;
data->posz = posz;

#ifdef DEBUG
printf("%d:␣clearing␣memory␣bandarray␣%p␣size␣%d\n",cartrank,data->bandarray,sizeof(int)*LOCALARRAYSIZE);
fflush(stdout);
#endif
/*
 * setting the band array to outside
 */
memset(data->bandarray,OUTSIDE,sizeof(int)*LOCALARRAYSIZE);
#ifdef DEBUG
printf("%d:␣Heap␣ok,␣clearing␣memory␣timearray␣%p␣size␣%d\n",cartrank,data->timearray,sizeof(float)*LOCALARRAYSIZE);
#endif
/*
 * zeroing the arrival time array
 */
int i = 0;
for(i=0;i<LOCALARRAYSIZE;i++)
  data->timearray[i] = 0;
//memset(data->timearray,0,sizeof(float)*LOCALARRAYSIZE);
//bzero(data->timearray,sizeof(float)*LOCALARRAYSIZE);
data->sentarray = malloc(sizeof(float)*LOCALARRAYSIZE);
bzero(data->sentarray,sizeof(float)*LOCALARRAYSIZE);
#ifdef DEBUG
printf("%d:␣done␣allocating␣memory,␣setting␣starting␣point\n",cartrank);
fflush(stdout);
#endif
/*
 * inserting the starting point on the correct node and add it to the heap / narrow band
 */
if(getDest(posx,posy,posz) == cartrank){
  int *cr = getLocalCord(posx,posy,posz);
  // insert starting point
  data->timearray[GETLINDEX(cr[0],cr[1],cr[2])] = 0;
  data->bandarray[GETLINDEX(cr[0],cr[1],cr[2])] = BAND;
  //printf("%d: sat pos %d %d %d, as known\n",cartrank,cr[0],cr[1],cr[2]);
  Element* element;
  element = malloc(sizeof(Element));
  element->x = cr[0];
  element->y = cr[1];
  element->z = cr[2];
  element->value = data->timearray[GETLINDEX(cr[0],cr[1],cr[2])];
  heapInsert(data->heap,element);
  send = 1;
  }
  return data;
}
/*
 * return min
 */
float min(float per, float truls){
  if(per > truls)
    return truls;
  else return per;
}
/*
 * return max
 */
float max(float per, float truls){
  if(per > truls)
    return per;
  else return truls;
}

/*
 * prints xy plane from a float array
 */
void printFloatArray(int sizex, int sizey, int z,float* array){
  int i,j;
  printf("\n␣Printing␣matrix\n");
  for(i=0;i<sizex;i++){
    for(j=0;j<sizey;j++){
      printf("␣%8.2f␣",array[GETLINDEX(i,j,z)]);
    }
    printf("\n");
  }
}

/*
 * calculate the arrival time for a point x,y,z
 */
float calcDistance(FmmData* data, int x, int y, int z){
  float sol;
  sol = BIGFLOAT;
  if(data->bandarray[GETLINDEX(x+1,y,z)] > BAND){
    sol = min(data->timearray[GETLINDEX(x+1,y,z)] +1/data->
        velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣x+1\n",cartrank,data->timearray[
       GETLINDEX(x+1,y,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
       ;
    #endif
  }
  if(data->bandarray[GETLINDEX(x-1,y,z)] > BAND){
    sol = min(data->timearray[GETLINDEX(x-1,y,z)] +1/data->
        velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣x-1\n",cartrank,data->timearray[
       GETLINDEX(x-1,y,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
       ;
    #endif
  }
```

```
    if(data->bandarray[GETLINDEX(x,y+1,z)] > BAND){
      sol = min(data->timearray[GETLINDEX(x,y+1,z)] +1/data->
          velocityarray[GETLINDEX(x,y,z)],sol);
      #ifdef DEBUG
      printf("%d:␣sol␣is␣%f␣for␣y+1\n",cartrank,data->timearray[
         GETLINDEX(x,y+1,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
         ;
      #endif
    }
    if(data->bandarray[GETLINDEX(x,y-1,z)] > BAND){
      sol = min(data->timearray[GETLINDEX(x,y-1,z)] +1/data->
          velocityarray[GETLINDEX(x,y,z)],sol);
      #ifdef DEBUG
      printf("%d:␣sol␣is␣%f␣for␣y-1\n",cartrank,data->timearray[
         GETLINDEX(x,y-1,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
         ;
      #endif
    }
    if(data->bandarray[GETLINDEX(x,y,z+1)] > BAND){
      sol = min(data->timearray[GETLINDEX(x,y,z+1)] +1/data->
          velocityarray[GETLINDEX(x,y,z)],sol);
      #ifdef DEBUG
      printf("%d:␣sol␣is␣%f␣for␣z+1\n",cartrank,data->timearray[
         GETLINDEX(x,y,z+1)] +1/data->velocityarray[GETLINDEX(x,y,z)])
         ;
      #endif
    }
    if(data->bandarray[GETLINDEX(x,y,z-1)] > BAND){
      sol = min(data->timearray[GETLINDEX(x,y,z-1)] +1/data->
          velocityarray[GETLINDEX(x,y,z)],sol);
      #ifdef DEBUG
      printf("%d:␣sol␣is␣%f␣for␣z-1\n",cartrank,data->timearray[
         GETLINDEX(x,y,z-1)] +1/data->velocityarray[GETLINDEX(x,y,z)])
         ;
      #endif
    }

    return sol;
}

/*
 * calculate a new point the the arrival time array if its inside
     the array and not KNOWN, if its outside add it to the heap
 */
void calcElement(FmmData* data, int px, int py, int pz){
        int add = 0;
        float sol;
        Element *temp;
        /*
         * Check if the point is inside the array*/
        if(px >= 0 && px < data->x && py >= 0 && py < data->y  && pz
            >= 0 && pz < data->z){
        /* Make sure the point is not KNOWN */
        if(data->bandarray[GETLINDEX(px,py,pz)] <= BAND){
                sol = calcDistance(data,px,py,pz);
                /* Check if the number is not smaller than the one
                   we calculated , when we exit , should never happend
                    in serial version*/
        if(data->timearray[GETLINDEX(px,py,pz)] != 0 && data->timearray[GETLINDEX(px,py,pz)] <= sol){
            return;
        }
        /* If the point is OUSIDE add it to the heap/narrowband */
        if(data->bandarray[GETLINDEX(px,py,pz)] == OUTSIDE){
            data->bandarray[GETLINDEX(px,py,pz)] = BAND;
            add = 1;
        }

        #ifdef DEBUG
        printf("%d: sol is %f, for %d %d %d\n",cartrank,sol,px,py,pz);
        #endif
        /* store max and min for debug purposes */
        #ifdef DEBUG
        if(sol > valuemax && sol != BIGFLOAT){
            valuemax=sol;
        }
        if(sol < valuemin){
            valuemin = sol;
        }
        #endif
        /* set the new arrivaltime */
        data->timearray[GETLINDEX(px,py,pz)] = sol;
        #ifdef DEBUG
        printf("%d: x %d , y %d, z %d, k[i] %d l[i] %d m[i] %d, value %f \n",cartrank,data->x, data->y, data->z,px,py,pz, data->timearray[GETLINDEX(px,py,pz)]);
        #endif
        /* add it to the narrowband if it should be added */
        if(add){
            addToHeap(data,px,py,pz);
            }
        }
    }
}

//remove ???
void checkforchange(FmmData* data,int px, int py, int pz){
  float sol = BIGFLOAT;
  int i;
  int o[6] = {px-1,px,px+1,px,px,px};
  int l[6] = {py,py-1, py,py+1,py,py};
  int m[6] = {pz,pz,pz,pz,pz-1,pz+1};
  for(i=0;i<6;i++){
    if(o[i] >= 0 && o[i] < data->x && l[i] >= 0 && l[i] < data->y && m[i] >= 0 && m[i] < data->z && data->bandarray[GETLINDEX(o[i],l[i],m[i])] > BAND){

      #ifdef DEBUG
      printf("%d: checking for rollback %d %d %d \n",cartrank,o[i],l[i],m[i]);
      #endif
      sol = calcDistance(data,px,py,pz);

      if(sol < data->timearray[GETLINDEX(o[i],l[i],m[i])] && data->bandarray[GETLINDEX(o[i],l[i],m[i])] > BAND){
        // new value is smaller lets add this to our heap

        #ifdef DEBUG
        printf("%d:␣rolling␣back␣%d␣%d␣%d␣old␣value␣%f␣new␣value␣%f\n",cartrank,o[i],l[i],m[i],data->timearray[GETLINDEX(o[i],l[i],m[i])],sol);
        #endif
        data->timearray[GETLINDEX(o[i],l[i],m[i])] = sol;
        data->bandarray[GETLINDEX(o[i],l[i],m[i])] = BAND;
        addToHeap(data,o[i],l[i],m[i]);
        checkforchange(data,o[i],l[i],m[i]);
      }
    }
  }
}

/*
 * Add a new element to the array
 */
void addElement(FmmData* data,MPI_Element e){
  int* cr;
  int i,j,k;
  cr = getLocalCord(e.x,e.y,e.z);

  #ifdef DEBUG
  printf("%d:␣adding␣element␣%d␣%d␣%d,␣to␣local␣%d␣%d␣%d\n",cartrank,e.x,e.y,e.z,cr[0],cr[1],cr[2]);
  #endif
  /*
   * checking if we have added it before
   * Should be always no since we don't send the same value multiple times
   */
  if(data->timearray[GETLINDEX(cr[0],cr[1],cr[2])] == e.value){
    #ifdef DEBUG
    printf("%d:␣alreaddy␣added␣%d␣%d␣%d\n",cartrank,e.x,e.y,e.z);
    #endif
    return;
  }
  data->timearray[GETLINDEX(cr[0],cr[1],cr[2])] = e.value;
  data->bandarray[GETLINDEX(cr[0],cr[1],cr[2])] = e.n;
  int o[6] = {cr[0]-1,cr[0],cr[0]+1,cr[0],cr[0],cr[0]};
  int l[6] = {cr[1],cr[1]-1, cr[1],cr[1]+1,cr[1],cr[1]};
  int m[6] = {cr[2],cr[2],cr[2],cr[2],cr[2]-1,cr[2]+1};
  /*
   * recalculate all neighbours
   */
  for(i=0;i<6;i++){

    #ifdef DEBUG
    printf("%d:␣calc␣element␣%d␣%d␣%d\n",cartrank,o[i],l[i],m[i]);
    #endif
    calcElement(data, o[i], l[i], m[i]);
  }
  /*
```

```
   * set rollback values, so we can check if a rollback is necesarry
   */
  /*if(largest_solution > e.value && rollbacksmallest > e.value){
  rollbacksmallest = e.value;
  rollbackn = e.n;
  }*/
}

/*
 * rollback all values above input value
 */
void rollback(FmmData* data, float value){
  int i,j,k;
  for(i=0;i<local_x;i++)
    for(j=0;j<local_y;j++)
      for(k=0;k<local_z;k++){
        if(data->bandarray[GETLINDEX(i,j,k)] > BAND && data->
            timearray[GETLINDEX(i,j,k)] > value){

          data->bandarray[GETLINDEX(i,j,k)] = BAND;
          addToHeap(data,i,j,k);
        }
      }
}

/*
 * send new values and check for incoming border values
 */
void sendRecvBorderChanges(FmmData* data,int x, int y, int z,int
    send){
  MPI_Element e;
  int run= 1;
  int *cr;
  int reast,rwest,rnorth,rsouth,rabove,rbelow;
  int seast,swest,snorth,ssouth,sabove,sbelow;
  reast = rwest = rnorth = rsouth = rabove = rbelow = 0;
  seast = swest = snorth = ssouth = sabove= sbelow = 0;
  /*
   * if we are to send a value
   */
  if(send){
    /*
     * see where we have to send the value
     */
    if(x == 0){
      snorth = 1;
    }
    if(x == local_x-1){
      ssouth = 1;
    }
    if(y == 0){
      swest = 1;
      }
    if(y == local_y-1){
      seast = 1;
    }
    if(z == 0){
      sbelow = 1;
```

```
        }
        if(z == local_z -1){
          sabove = 1;
        }
        #ifdef DEBUG
        printf("%d:␣%d␣%d␣%d␣sending␣to␣north␣%d␣%d␣south␣%d␣␣%d␣west␣%d
            ␣%d␣east␣%d␣%d␣below␣%d␣%d␣above␣%d␣%d\n",cartrank,x,y,z,
            snorth,north,ssouth,south,swest,west,seast,east,sbelow,below,
            sabove,above);
        #endif


    if(snorth || ssouth || swest || seast || sabove || sbelow){
        cr = getGlobalCord(x,y,z);
        #ifdef DEBUG
        printf("%d:␣sending␣element␣at␣%d␣%d␣%d␣gave␣global␣coord␣%d␣%d␣
            %d\n",cartrank,x,y,z,cr[0],cr[1],cr[2]);
        #endif
        e.x = cr[0];
        e.y = cr[1];
        e.z = cr[2];
        e.value = data->timearray[GETLINDEX(x,y,z)];
        data->sentarray[GETLINDEX(x,y,z)] = e.value;
        e.n = data->bandarray[GETLINDEX(x,y,z)];

        if(snorth){
          MPI_Send(&e,1,mpi_element_struct,north,1,gridcomm);
        }
        if(ssouth){
          MPI_Send(&e,1,mpi_element_struct,south,1,gridcomm);
        }
        if(swest){
          MPI_Send(&e,1,mpi_element_struct,west,1,gridcomm);
        }
        if(seast){
          MPI_Send(&e,1,mpi_element_struct,east,1,gridcomm);
        }
        if(sabove){
          MPI_Send(&e,1,mpi_element_struct,above,1,gridcomm);
        }
        if(sbelow){
          MPI_Send(&e,1,mpi_element_struct,below,1,gridcomm);
        }
    }
    }
    rollbacksmallest = BIGFLOAT;
    rollbackn = 0;
    /*
     * a loop to receive all incoming border values
     */
    while(run){
    MPI_Iprobe(MPI_ANY_SOURCE,1,gridcomm,&run,MPI_STATUS_IGNORE);
    if(run){
        MPI_Iprobe(north,1,gridcomm,&rnorth,MPI_STATUS_IGNORE);
        MPI_Iprobe(south,1,gridcomm,&rsouth,MPI_STATUS_IGNORE);
        MPI_Iprobe(east,1,gridcomm,&reast,MPI_STATUS_IGNORE);
        MPI_Iprobe(west,1,gridcomm,&rwest,MPI_STATUS_IGNORE);
        MPI_Iprobe(above,1,gridcomm,&rabove,MPI_STATUS_IGNORE);
    MPI_Iprobe(below,1,gridcomm,&rbelow,MPI_STATUS_IGNORE);
    if(rnorth || rsouth || rwest || reast || rabove || rbelow){

      if(rnorth){
        MPI_Recv(&e,1,mpi_element_struct,north,1,gridcomm,MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rsouth){
        MPI_Recv(&e,1,mpi_element_struct,south,1,gridcomm,MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rwest){
        MPI_Recv(&e,1,mpi_element_struct,west,1,gridcomm,MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(reast){
        MPI_Recv(&e,1,mpi_element_struct,east,1,gridcomm,MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rabove){
        MPI_Recv(&e,1,mpi_element_struct,above,1,gridcomm,MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rbelow){
        MPI_Recv(&e,1,mpi_element_struct,below,1,gridcomm,MPI_STATUS_IGNORE);
        addElement(data,e);
      }
    }
  }
  }
  /*
   * rollback if necessary
   */
  if(rollbackn){
    rollback(data,rollbacksmallest);
    n = rollbackn;
    rollbackn = 0;
    rollbacksmallest = BIGFLOAT;
  }
}

/*
 * old synchroneous border exhcange
 */
/*
void sendRecvBorderChanges(FmmData* data,int x, int y, int z,int send){
  MPI_Element e;

  int *cr;
  int reast,rwest,rnorth,rsouth,rabove,rbelow;
  int seast,swest,snorth,ssouth,sabove,sbelow;
```

```
    reast = rwest = rnorth = rsouth = rabove = rbelow = 0;
    seast = swest = snorth = ssouth = sabove= sbelow = 0;
    if(send){
      if(x == 0){
        snorth = 1;
      }
      if(x == local_x -1){
        ssouth = 1;
      }
      if(y == 0){
        swest = 1;
        }
      if(y == local_y -1){
        seast = 1;
      }
      if(z == 0){
        sbelow = 1;
      }
      if(z == local_z -1){
        sabove = 1;
      }
      #ifdef DEBUG
      printf("%d: %d %d %d sending to north %d %d south %d  %d west %d
           %d east %d %d below %d %d above %d %d\n",cartrank,x,y,z,
          snorth,north,ssouth,south,swest,west,seast,east,sbelow,below,
          sabove,above);
      #endif
    }

    MPI_Send(&swest,1,MPI_INT,west,0,gridcomm);
    MPI_Recv(&reast,1,MPI_INT,east,0,gridcomm,MPI_STATUS_IGNORE);
    MPI_Send(&seast,1,MPI_INT,east,0,gridcomm);
    MPI_Recv(&rwest,1,MPI_INT,west,0,gridcomm,MPI_STATUS_IGNORE);

    MPI_Send(&ssouth,1,MPI_INT,south,0,gridcomm);
    MPI_Recv(&rnorth,1,MPI_INT,north,0,gridcomm,MPI_STATUS_IGNORE);
    MPI_Send(&snorth,1,MPI_INT,north,0,gridcomm);
    MPI_Recv(&rsouth,1,MPI_INT,south,0,gridcomm,MPI_STATUS_IGNORE);

    MPI_Send(&sabove,1,MPI_INT,above,0,gridcomm);
    MPI_Recv(&rbelow,1,MPI_INT,below,0,gridcomm,MPI_STATUS_IGNORE);
    MPI_Send(&sbelow,1,MPI_INT,below,0,gridcomm);
    MPI_Recv(&rabove,1,MPI_INT,above,0,gridcomm,MPI_STATUS_IGNORE);

    if(snorth || ssouth || swest || seast || sabove || sbelow){
      cr = getGlobalCord(x,y,z);

      #ifdef DEBUG
      printf("%d: sending element at %d %d %d gave global coord %d %d
          %d\n",cartrank,x,y,z,cr[0],cr[1],cr[2]);
      #endif
      e.x = cr[0];
      e.y = cr[1];
      e.z = cr[2];
      e.value = data->timearray[GETLINDEX(x,y,z)];
      e.n = data->bandarray[GETLINDEX(x,y,z)];

      if(snorth){
        MPI_Send(&e,1,mpi_element_struct,north,1,gridcomm);
      }
      if(ssouth){
        MPI_Send(&e,1,mpi_element_struct,south,1,gridcomm);
      }
      if(swest){
        MPI_Send(&e,1,mpi_element_struct,west,1,gridcomm);
      }
      if(seast){
        MPI_Send(&e,1,mpi_element_struct,east,1,gridcomm);
      }
      if(sabove){
        MPI_Send(&e,1,mpi_element_struct,above,1,gridcomm);
      }
      if(sbelow){
        MPI_Send(&e,1,mpi_element_struct,below,1,gridcomm);
      }
    }

    if(rnorth || rsouth || rwest || reast || rabove || rbelow){

      if(rnorth){
        MPI_Recv(&e,1,mpi_element_struct,north,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rsouth){
        MPI_Recv(&e,1,mpi_element_struct,south,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rwest){
        MPI_Recv(&e,1,mpi_element_struct,west,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(reast){
        MPI_Recv(&e,1,mpi_element_struct,east,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rabove){
        MPI_Recv(&e,1,mpi_element_struct,above,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rbelow){
        MPI_Recv(&e,1,mpi_element_struct,below,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
    }
}
*/

/*
 * check if someone wants to update their working status
 */
```

```
void checkOthers(){
  int i;
  int flag;
  for(i = 0; i<size;i++){
    MPI_Iprobe(i,9,gridcomm,&flag,MPI_STATUS_IGNORE);
    if(flag){
      MPI_Recv(&working[i],1,MPI_INT,i,9,gridcomm,MPI_STATUS_IGNORE)
          ;
    }
  }
}
/*
 * notify others that my working status is changed
 */
void notifyOthers(int value){
  int i;
  for(i = 0; i<size;i++){
    MPI_Send(&value,1,MPI_INT,i,9,gridcomm);
  }
}

/*
 * Execute the FMM
 */
void executeFMM(FmmData* data){
  int add=0;
  int posx,posy,posz;
  int run = 1;
  int sendrun = 1;
  int senddata = 0;
  int end = 0;
  int sum = 0;
  int i;
  #ifdef DEBUG
  printf("data size is %d %d %d\n",data->x, data->y, data->z);
  #endif
  for(i = 0; i<size;i++){
    working[i] = 1;
  }
  largest_solution = 0;
  n = 1;
  /* loop will run until all nodes are done */
  while(!end){
  /* working loop, will run until there are no more work to be done
      */
  while(run){


    if(heapGetMin(data->heap)){

     Element* e,*temp;

     e = heapExtractMin(data->heap);
     #ifdef DEBUG
     printf("%d: setting %d %d %d to known\n",cartrank,e->x,e->y,e->
         z);
     #endif
     data->bandarray[GETLINDEX(e->x,e->y,e->z)] = KNOWN;
     int k[6] = {e->x-1,e->x,e->x+1,e->x,e->x,e->x};
     int l[6] = {e->y, e->y-1, e->y, e->y+1,e->y,e->y};
     int m[6] = {e->z, e->z, e->z, e->z,e->z-1,e->z+1};
     int i;
     posx = e->x;
     posy = e->y;
     posz = e->z;
     /*
      * update largest_solution if this solution is the largest
      */
     if(data->timearray[GETLINDEX(posx,posy,posz)] >
         largest_solution){
      largest_solution = data->timearray[GETLINDEX(posx,posy,posz)];
     }
     /* check if this point has been sent before */
     if(data->sentarray[GETLINDEX(posx,posy,posz)] == 0){
      senddata = 1;
     }else if(data->sentarray[GETLINDEX(posx,posy,posz)] <= data->
         timearray[GETLINDEX(posx,posy,posz)]){
      senddata = 0;
     }

     /*printFloatArray(data->x,data->y,-2,data->timearray);
     printFloatArray(data->x,data->y,-1,data->timearray);
     printFloatArray(data->x,data->y,0,data->timearray);
     printFloatArray(data->x,data->y,1,data->timearray);
     printFloatArray(data->x,data->y,2,data->timearray);*/
     for(i=0;i<6;i++){

      if(k[i] >= 0 && k[i] < data->x && l[i] >= 0 && l[i] < data->y
          && m[i] >= 0 && m[i] < data->z){
        calcElement(data, k[i], l[i], m[i]);
      }
     }

    #ifdef DEBUG

     /*printFloatArray(data->x,data->y,-2,data->timearray);
     printFloatArray(data->x,data->y,-1,data->timearray);
     printFloatArray(data->x,data->y,0,data->timearray);
     printFloatArray(data->x,data->y,1,data->timearray);
     printFloatArray(data->x,data->y,2,data->timearray);*/
    #endif
     free(e);

     n++;
    }
    /* Send changes to border and look for incoming changes to the
        border */
    sendRecvBorderChanges(data,posx,posy,posz,senddata);

    #ifdef DEBUG
    if(cartrank == 0|| cartrank == -2){printLocalArray(data->
        timearray);
    printLocalIntArray(data->bandarray);
    }
    #endif
    senddata = 0;
```

```
      /*Do we end the loop?*/
      if(heapGetMin(data->heap) == 0){
        run = 0;
      }else{
        run = 1;
      }
      checkOthers();
      #ifdef DEBUG
      if(cartrank == 0 && n%1000 == 0){
        printf("%d:␣reached␣n␣%d\n",cartrank,n);
      }
      #endif
    }
      /* see if border changes are coming */
      sendRecvBorderChanges(data,0,0,0,0);
      /* if we have work to do lets notify others and start to work
         again, if not let others know we are done */
      if(heapGetMin(data->heap) != 0){
        run = 1;
        // notify other i am still working
        notifyOthers(1);
      }else if(working[cartrank] == 1){
        // notify that i have stopped working
        notifyOthers(0);
      }
      checkOthers();
      sum = 0;
      for(i = 0; i<size;i++){
        sum += working[i];
      }
      if(sum == 0){
        end = 1;
      }
    }
    #ifdef DEBUG
    printf("valuemax␣%f␣valuemin␣%f\n",valuemax,valuemin);
    #endif
}


/*
 * free the variables used in the FMM
 */
void freeFMM(FmmData* data){
  //free(data->timearray);
  free(data->bandarray);
  free(data);
}
```

## A.3 Application

### A.3.1 mpi_app.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "array_mpi.h"
#include "fileio.h"
#include <mpi.h>
#include <string.h>
#include "fmm_mpi.h"
#include "time.h"

#define MPIDEBUG 0
#define DEBUG 0

/* set that no dimensions sould be cyclic */
int periods[3] = {0,0,0};
/* set the number of dimensions to use in the cartesian node grid*/
int ndims = 3;

int div_x, div_y, div_z;

float* local_array;
float* global_array;
float* file_array;

/*
 * different mpi datatypes for exchanging the borders
 */
MPI_Datatype xz_plane;
MPI_Datatype zy_plane;
MPI_Datatype xy_plane;
MPI_Datatype y_column;
MPI_Datatype y_column_resized;
/*
 * A test function to check if the array dosen't contain a value
 */
int checkArray(int x, int y, int z,float value){
    int i,j,k;
    int rvalue = 1;
    for(i= 0; i<x;i++)
      for(j=0;j<y;j++)
        for(k=0;k<z;k++){
          if(local_array[GETLINDEX(i,j,k)] != value){
            rvalue = 0;
            return rvalue;
          }
        }
    return rvalue;
  }
/*
 * Check if a global array dosen't contain a specific value
 */
int checkGArray(int x, int y, int z,float value){
    int i,j,k;
    int rvalue = 1;
    for(i= 0; i< x;i++)
      for(j=0;j<y;j++)
        for(k=0;k<z;k++){
          if(global_array[GETINDEX(i,j,k)] != value){
            rvalue = 0;
            return rvalue;
          }
        }
    return rvalue;
  }

/*
 * Divide the global matrix into smaller matrixes for each node.
 * This function calculates the local dimensions
 */
void divide_matrix(){

  div_x = x/dims[0];
  div_y = y/dims[1];
  div_z = z/dims[2];
  local_x = div_x;
  local_y = div_y;
  local_z = div_z;
  if(local_x * dims[0] != x){
    if(coords[0] == dims[0]){
      local_x = (x-(local_x*dims[0])) + local_x;
    }
  }
  if(local_y * dims[1] != y){
    if(coords[1] == dims[1]){
      local_y = (y-(local_y*dims[1])) + local_y;
    }
  }
  if(local_z * dims[2] != z){
    if(coords[2] == dims[2]){
      local_z = (z-(local_z*dims[2])) + local_z;
    }
  }
  printf("%d:␣local_x␣%d␣local_y␣%d␣local_z␣%d␣coords␣%d,%d,%d␣\n",
      cartrank, local_x, local_y, local_z,coords[0],coords[1],coords
      [2]);
}

/*
 * initialize datatypes for border exchange
 */
void initMPIDatatypes(){
  MPI_Type_vector(local_y,local_z,local_z+2,MPI_FLOAT,&zy_plane);
  MPI_Type_commit(&zy_plane);

  MPI_Type_vector(local_x,local_z,(local_z+2)*(local_y+2),MPI_FLOAT
      ,&xz_plane);
  MPI_Type_commit(&xz_plane);

  MPI_Type_vector(local_y,1,local_z+2,MPI_FLOAT,&y_column);
  MPI_Type_commit(&y_column);
  MPI_Type_create_resized(y_column,0,(local_z+2)*(local_y+2)*sizeof(
      float),&y_column_resized);
  MPI_Type_vector(1,2,1,y_column_resized,&xy_plane);
  MPI_Type_commit(&xy_plane);
}
/*
 * Initialize the program allocating local matrixes and initializing
     data types
 */
void init(){
  divide_matrix();
  local_array = malloc(sizeof(float)*LOCALARRAYSIZE);
  if(local_array == 0){
    printf("Coudn't␣allocate␣enough␣memory␣for␣local_array\n");
    exit(1);
  }
  bzero(local_array,sizeof(float)*LOCALARRAYSIZE);
  initMPIDatatypes();
}
/*
 * a function for reading a file into each node, where the file
     contains a global array, each node will read its respective part
     into their local matrixes
 */
void scatterdata(char* filename){
  int i,j;
  int* cr;
  int value;
  char* errorstr;
  int reslen;
  FILE* f;
  int offset = 0;
  printf("%d:␣openeing␣file␣%s\n",cartrank,filename);
  f = fopen(filename,"rb");
  if(!f){
    printf("%d:␣unable␣to␣open␣file␣%d␣\n",cartrank,f);
    fflush(stdout);
    return;
  }
  if(DEBUG){
    printf("%d:␣opened␣file␣%d\n",cartrank,f);
  }
  for(i = 0; i<local_x;i++){
    for(j= 0; j< local_y; j++)
    {
      cr = getGlobalCord(i,j,0);
      offset = sizeof(float)*(cr[2]+cr[1]*z+cr[0]*z*y);
      fseek(f,offset,SEEK_SET);
      fread(&local_array[GETLINDEX(i,j,0)],sizeof(float),local_z,f);
      free(cr);
    }
  }
  fclose(f);

}

/*
 * gather all the local matrixes into a global matrix on node 0
 */
float* gatherdata(float* iarray){
```

```
int i,j,k,dest,flag,r,t;
float* farray;
MPI_Request* requests;
MPI_Status* status;
requests = malloc(sizeof(MPI_Request)*local_y*local_x*2);
status = malloc(sizeof(MPI_Status)*local_y*local_x*2);
if(cartrank == 0){
  if(DEBUG){
    printf("%d:started␣gathering\n",cartrank);
  }
  farray = malloc(sizeof(float)*ARRAYSIZE);
  bzero(farray,sizeof(float)*ARRAYSIZE);

}
if(cartrank != 0){
for(i=0;i<local_x;i++)
  for(j= 0; j<local_y;j++){
    MPI_Send(&iarray[GETLINDEX(i,j,0)],local_z,MPI_FLOAT,0,i*local_y+j,gridcomm);//,&requests[(local_y*i)+j]);
  }

if(DEBUG){
  printf("%d:␣Done␣sending␣\n",cartrank);
}
}
if(cartrank==0){
  if(MPIDEBUG){
    printf("%d:␣starting␣setting␣recvs\n",cartrank);
  }
  for(r=0;r<x;r+=local_x)
    for(t=0; t<y;t+=local_y)
      for(k=0;k<dims[2];k++){

        dest = getDest(r,t,k*local_z);
        if(DEBUG){
          printf("%d:␣receiving␣from␣%d\n",cartrank,dest);
        }
        for(i=0;i<local_x;i++)
          for(j=0;j<local_y;j++){
          MPI_Irecv(&farray[GETINDEX(i+r,j+t,k*local_z)],local_z,MPI_FLOAT,dest,i*local_y+j,gridcomm,&requests[(local_y*local_x)+(i*local_y+j)]);
          if(dest == 0){
            MPI_Isend(&iarray[GETLINDEX(i,j,0)],local_z,MPI_FLOAT,0,i*local_y+j,gridcomm,&requests[(local_y*i)+j]);
          }
          }
        if(dest == 0){
          MPI_Waitall(local_y*local_x*2,requests,status);
        }
        if(dest != 0){
        MPI_Waitall(local_y*local_x,&requests[local_y*local_x],status);
        }
      }
}

printf("%d:␣Done␣gathering␣\n",cartrank);
```

```
  return farray;
}


/*
 * check if the global array is the same as the array inside a file
 */
void checkData(char* filename){

  if(cartrank == 0){
  int i,j,k;
    printf("%d: Reading file\n",cartrank);
    file_array = malloc(ARRAYSIZE*sizeof(float));
    bzero(file_array,ARRAYSIZE*sizeof(float));
    readfile(file_array,filename,x,y,z);

    printArray(file_array);
    printf("%d: checking data consistency %f\n",cartrank,file_array[
        GETINDEX(0,0,0)]);
    for(i=0;i<x;i++)
      for(j=0;j<y;j++)
        for(k=0;k<z;k++){
          if(file_array[GETINDEX(i,j,k)] != global_array[GETINDEX(i,
              j,k)]){
            printf("%d: error at %d %d %d file %lf global %lf\n",
                cartrank,i,j,k,file_array[GETINDEX(i,j,k)],
                global_array[GETINDEX(i,j,k)]);
          }
        }
  }
}

/*
 * Exchange borders
 */
void exchangeBorders(){


  // sending /recving north south
  MPI_Send(&local_array[GETLINDEX(0,0,0)],1,zy_plane,north,0,
      gridcomm);
  MPI_Recv(&local_array[GETLINDEX(local_x,0,0)],1,zy_plane,south,0,
      gridcomm,MPI_STATUS_IGNORE);
  MPI_Send(&local_array[GETLINDEX(local_x-1,0,0)],1,zy_plane,south
      ,1,gridcomm);
  MPI_Recv(&local_array[GETLINDEX(-1,0,0)],1,zy_plane,north,1,
      gridcomm,MPI_STATUS_IGNORE);


  //sending/recving east, west
  MPI_Send(&local_array[GETLINDEX(0,0,0)],1,xz_plane,west,2,gridcomm
      );
  MPI_Recv(&local_array[GETLINDEX(0,local_y,0)],1,xz_plane,east,2,
      gridcomm,MPI_STATUS_IGNORE);
  MPI_Send(&local_array[GETLINDEX(0,local_y-1,0)],1,xz_plane,east,3,
      gridcomm);
  MPI_Recv(&local_array[GETLINDEX(0,-1,0)],1,xz_plane,west,3,
      gridcomm,MPI_STATUS_IGNORE);

  //sending/receving above, below
  MPI_Send(&local_array[GETLINDEX(0,0,local_z -1)],1,xy_plane,above
       ,4,gridcomm);
  MPI_Recv(&local_array[GETLINDEX(0,0,-1)],1,xy_plane,below,4,
       gridcomm,MPI_STATUS_IGNORE);
  MPI_Send(&local_array[GETLINDEX(0,0,0)],1,xy_plane,below,5,
       gridcomm);
  MPI_Recv(&local_array[GETLINDEX(0,0,local_z)],1,xy_plane,above,5,
       gridcomm,MPI_STATUS_IGNORE);
}

/*
 * a test function for writing a file with values
 */
void writeafile(){
  FILE* f;
  f = fopen("/work/idarbo/per.conv","w");
  int i,j,k;
  float value;
  for(i=0;i<x;i++)
    for(j=0;j<y;j++)
      for(k=0;k<z;k++){
      //value = i+j+k;
      value = 1;
      if(j < 8 && k < 8){
        value = 9;
      }

      fwrite(&value,sizeof(float),1,f);
      }
  fclose(f);
}

int main(int argc, char** argv){
  float* array,*time1,*time2;
  int timeusec,timesec, timeusec2, timesec2,rtimesec, rtimeusec;
  FmmData* data;
        x = SIZEX;
        y = SIZEY;
        z = SIZEZ;
        int i, j, k;
        /* initialize MPI*/
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Dims_create(size,ndims,dims);

  MPI_Cart_create(MPI_COMM_WORLD,ndims, dims, periods, 0, &gridcomm)
       ;

  MPI_Cart_shift(gridcomm, 0, 1, &north, &south);

  MPI_Cart_shift(gridcomm, 1, 1, &west, &east);
  MPI_Cart_shift(gridcomm, 2, 1, &below, &above);

  MPI_Comm_rank(gridcomm, &cartrank);

```

```
MPI_Cart_coords(gridcomm, cartrank, 3, coords);
//if(cartrank ==0)
//writeafile();
/* init the program */
init();

if(MPIDEBUG){
  printf("%d:␣west␣%d␣east␣%d␣south␣%d␣north␣%d␣below␣%d␣above␣%d␣cartrank␣%d\n",rank,west,east,south,north,below,above,cartrank);
  fflush(stdout);
}



if(cartrank == 0){
  printf("%d:␣Init␣complete␣reading␣data␣from␣file\n",cartrank);
  fflush(stdout);
}
/* read data from file */
//scatterdata(argv[1]);

/* instead of reading set the array to 1.0 */
for(i=-1;i<=local_x;i++)
  for(j=-1;j<=local_y;j++)
    for(k=-1;k<=local_z;k++)
      local_array[GETLINDEX(i,j,k)] = 1.0;
//printLocalArray(local_array);
//MPI_Barrier(gridcomm);
if(cartrank == 0){
  printf("%d:␣Read␣data␣exchanging␣borders\n",cartrank);
  fflush(stdout);
}
/* exchange borders so border values will be correct in the velocity array */
exchangeBorders();

if(cartrank == 0){
  printf("%d:␣Initializing␣FMM\n",cartrank);
  fflush(stdout);
}
/* initialize the FMM */
data = initFMM(local_array,x/2,y/2,z/2,local_x,local_y,local_z);

if(cartrank == 0){
  printf("%d:␣Executing␣FMM\n",cartrank);
  fflush(stdout);
}
/* take timing and execute the FMM*/
timeusec = getTimeInMicroseconds();
timesec = getTimeInSeconds();
executeFMM(data);
timeusec2 = getTimeInMicroseconds();
timesec2 = getTimeInSeconds();
fixTime(timesec,timeusec,timesec2,timeusec2,&rtimesec,&rtimeusec);
if(cartrank == 0){
  printf("%d:␣execute␣FMM␣took␣%d␣seconds␣and␣%d␣microseconds\n",
      cartrank,rtimesec,rtimeusec);
    }
    if (DEBUG) {
      printLocalArray(data->timearray);
    }
    MPI_Barrier(gridcomm);

    if(cartrank == 0){
      printf("%d:␣FMM␣done␣gathering␣data\n",cartrank);
    }
    /*gather if you can place all data on one node in memory, else
       write to file*/
    //global_array = gatherdata(data->timearray);

    //global_array = gatherdata(local_array);




    if(cartrank == 0){
      if (DEBUG) {
                //printArray(global_array);
      }
    }
    //checkData(argv[1]);

    printf("%d␣is␣done␣ending␣gracefully.\n",cartrank);
    MPI_Finalize();

}
```

# Appendix B

# Application 2 source code

## B.1 Array

### B.1.1 array_mpi.h

```
#ifndef ARRAY_H
#define ARRAY_H

#include "mpi.h"
/*
 * Defines the size of the array
 */
#define SIZEX 448//448//8//448
#define SIZEY 704//704//4//704
#define SIZEZ 1216//1216//4//1216
/*
 * Calculate the strides for global array
 */
#define stride0 ((SIZEZ+2)*(SIZEY+2))
#define stride1 (SIZEZ+2)
#define stride2 1
/*
 * Gets the index for a position in the global array
 */
#define GETINDEX(i,j,k) stride0*(i+1) + stride1*(j+1) + stride2*(k
    +1)
/*
 * Get the index for a position in the local array
 */
#define GETLINDEX(i,j,k) ((local_z+2)*(local_y+2)) *(i+1) + ((
    local_z+2)*(j+1)) + k+1
/*
 * A large float number, should be larger than anything you
     calculate
 */
#define BIGFLOAT 10000000.0

/*
 * The size of each array, for mallocing memory
 */
#define ARRAYSIZE (SIZEX+2)*(SIZEY+2)*(SIZEZ+2)
#define LOCALARRAYSIZE (local_x+2)*(local_y+2)*(local_z+2)

/*
 * Rank, MPI rank
 * cartrank, rank in the cartesian grid
 * size, number of nodes used
 */
int rank,cartrank, size;
/*
 * Ranks of nodes that are above,below,west,east,north,south
 */
int above, below, west, east, north ,south;
/*
 * Number of nodes in each dimension
 */
int dims[3];
/*
 * Different datatypes for local arrays
 */
MPI_Datatype xz_plane;
MPI_Datatype zy_plane;
MPI_Datatype xy_plane;
MPI_Datatype y_column;
MPI_Datatype y_column_resized;

MPI_Datatype int_xz_plane;
MPI_Datatype int_zy_plane;
MPI_Datatype int_xy_plane;
MPI_Datatype int_y_column;
MPI_Datatype int_y_column_resized;
/*
 * communicator for the cartesian grid
 */
MPI_Comm gridcomm;
/*
 * MY coordinates in the cartesian node grid
 */
int coords[3];
/*
 * size in each dimension of the global array
 */
int x,y,z;
/*
 * Size in each dimension of the local array
 */
int local_x,local_y,local_z;

/*
 * Print the values of a global array to stdout
 */
void printArray(float* array);
/*
 * Print the local array to stdout
 */
void printLocalArray(float* array);
/*
```

```
 * print a local int array to stdout
 */
void printLocalIntArray(int* array);
/*
 * Get what node a global position resides in
 */
int getDest(int x, int y, int z);
/*
 * Get global coordinates from local coordinates
 */
int* getGlobalCord(int x, int y, int z);
/*
 * Get local coordinates from global coordinates
 */
int* getLocalCord(int x, int y, int z);
/*
 * Start exchanging of borders
 */
void exchangeBorders(float* array);
/*
 * Initializa mpi data types
 */
void initMPIDatatypes();
/*
 * Start exchanging integer borders
 */
void int_exchangeBorders(int* array);
/*
 * Initialize integer datatypes
 */
void int_initMPIDatatypes();
/*
 * Wait for all int exchange borders
 */
void wait_exchange_int();
/*
 * wait for all float exchange borders
 */
void wait_exchange_float();
/*
 * Wait for all border exchanges
 */
void wait_exchange_all();

#endif
```

### B.1.2 array_mpi.c

```c
#include "array_mpi.h"
#include <stdlib.h>
#include <stdio.h>
/*
 * stores requests and status from non blocking border exchange
 */
MPI_Request request[24];
MPI_Status status[24];
/*
 * converts global coord to local coord
 */
int* getLocalCord(int xg, int yg, int zg){
        int *cr;
        cr = malloc(sizeof(int)*3);
        cr[0] = xg;
        cr[1] = yg;
        cr[2] = zg;
   cr[0] -= coords[0] * local_x;
   cr[1] -= coords[1] * local_y;
   cr[2] -= coords[2] * local_z;
        return cr;
}

/*
 * convert local coord to global coord
 */
int* getGlobalCord(int x, int y, int z){
        int *cr;
        cr = malloc(sizeof(int)*3);
        cr[0] = x+ local_x*coords[0];
        cr[1] = y+ local_y*coords[1];
        cr[2] = z+ local_z*coords[2];
        return cr;
        }

/*
 * gets the rank of the node that have input position
 */
int getDest(int x, int y, int z){
        int cr[3];
        int value;
        cr[0] = x/local_x;
        cr[1] = y/local_y;
        cr[2] = z/local_z;
        MPI_Cart_rank(gridcomm,cr,&value);
        return value;
}

/*
 * print the local float array to stdout
 */
void printLocalArray(float* array){
        int i,j,k;
        for(k=-1;k<=local_z;k++){
        printf("%d:array␣z␣=␣%d\n",cartrank,k);
                for(i=-1;i<=local_x;i++){
                        printf("%d:␣",cartrank);
                        for(j=-1;j<=local_y;j++){
                                printf("␣%8f␣",array[GETLINDEX(i,j,k)]);
                        }
                        printf("\n");
                }
        printf("\n\n");
        }
}
/*
 * print a local integer array to stdout
 */
void printLocalIntArray(int* array){
        int i,j,k;
        for(k=-1;k<=local_z;k++){
        printf("%d:array␣z␣=␣%d\n",cartrank,k);
                for(i=-1;i<=local_x;i++){
                        printf("%d:␣",cartrank);
                        for(j=-1;j<=local_y;j++){
                                printf("␣%d␣",array[GETLINDEX(i,j,k)]);
                        }
                        printf("\n");
                }
        printf("\n\n");
        }
}

/*
 * print the global array to stdout
 */
void printArray(float* array){
        int i,j,k;

        for(k=-1;k<=z;k++){
        printf("array␣z␣=␣%d\n",k);
                for(i=-1;i<=x;i++){
                        for(j=-1;j<=y;j++){
                                printf("␣%8f␣",array[GETINDEX(i,j,k)]);
                        }
                printf("\n");
                }
        printf("\n\n");
        }
}

/*
 * start non blocking exchange border communication
 */
void exchangeBorders(float* array){


        // sending /recving north south
        MPI_Isend(&array[GETLINDEX(0,0,0)],1,zy_plane,north,0,gridcomm,&request[0]);
        MPI_Irecv(&array[GETLINDEX(local_x,0,0)],1,zy_plane,south,0,
                gridcomm,&request[1]); //MPI_STATUS_IGNORE);
        MPI_Isend(&array[GETLINDEX(local_x -1,0,0)],1,zy_plane,south
                ,1,gridcomm,&request[2]);
        MPI_Irecv(&array[GETLINDEX(-1,0,0)],1,zy_plane,north,1,
                gridcomm,&request[3]); //MPI_STATUS_IGNORE);


        //sending/recving east, west
        MPI_Isend(&array[GETLINDEX(0,0,0)],1,xz_plane,west,2,
                gridcomm,&request[4]);
        MPI_Irecv(&array[GETLINDEX(0,local_y,0)],1,xz_plane,east,2,
                gridcomm,&request[5]); //MPI_STATUS_IGNORE);
        MPI_Isend(&array[GETLINDEX(0,local_y -1,0)],1,xz_plane,east
                ,3,gridcomm,&request[6]);
        MPI_Irecv(&array[GETLINDEX(0,-1,0)],1,xz_plane,west,3,
                gridcomm,&request[7]); //MPI_STATUS_IGNORE);

        //sending/receving above, below
        MPI_Isend(&array[GETLINDEX(0,0,local_z -1)],1,xy_plane,above
                ,4,gridcomm,&request[8]);
        MPI_Irecv(&array[GETLINDEX(0,0,-1)],1,xy_plane,below,4,
                gridcomm,&request[9]); //MPI_STATUS_IGNORE);
        MPI_Isend(&array[GETLINDEX(0,0,0)],1,xy_plane,below,5,
                gridcomm,&request[10]);
        MPI_Irecv(&array[GETLINDEX(0,0,local_z)],1,xy_plane,above,5,
                gridcomm,&request[11]); //MPI_STATUS_IGNORE);
}

/*
 * initialize mpi datatypes
 */
void initMPIDatatypes(){
        MPI_Type_vector(local_y,local_z,local_z+2,MPI_FLOAT,&
                zy_plane);
        MPI_Type_commit(&zy_plane);

        MPI_Type_vector(local_x,local_z,(local_z+2)*(local_y+2),
                MPI_FLOAT,&xz_plane);
        MPI_Type_commit(&xz_plane);

        MPI_Type_vector(local_y,1,local_z+2,MPI_FLOAT,&y_column);
        MPI_Type_commit(&y_column);
        MPI_Type_create_resized(y_column,0,(local_z+2)*(local_y+2)*
                sizeof(float),&y_column_resized);
        MPI_Type_vector(1,2,1,y_column_resized,&xy_plane);
        MPI_Type_commit(&xy_plane);
}

/*
 * start exchanging integer border
 */
void int_exchangeBorders(int* array){


        // sending /recving north south
        MPI_Isend(&array[GETLINDEX(0,0,0)],1,int_zy_plane,north,6,
                gridcomm,&request[12]);
        MPI_Irecv(&array[GETLINDEX(local_x,0,0)],1,int_zy_plane,
            south,6,gridcomm,&request[13]); //MPI_STATUS_IGNORE);
        MPI_Isend(&array[GETLINDEX(local_x -1,0,0)],1,int_zy_plane,
            south,7,gridcomm,&request[14]);
        MPI_Irecv(&array[GETLINDEX(-1,0,0)],1,int_zy_plane,north,7,
            gridcomm,&request[15]); //MPI_STATUS_IGNORE);


        //printf("%d: sending to %d reciving from %d\n",cartrank,
            west,east);
        //sending/recving east, west


        MPI_Isend(&array[GETLINDEX(0,0,0)],1,int_xz_plane,west,8,
            gridcomm,&request[16]);
        MPI_Irecv(&array[GETLINDEX(0,local_y,0)],1,int_xz_plane,east
            ,8,gridcomm,&request[17]); //MPI_STATUS_IGNORE);
        MPI_Isend(&array[GETLINDEX(0,local_y -1,0)],1,int_xz_plane,
            east,9,gridcomm,&request[18]);
        MPI_Irecv(&array[GETLINDEX(0,-1,0)],1,int_xz_plane,west,9,
            gridcomm,&request[19]); //MPI_STATUS_IGNORE);

        //sending/receving above, below

        MPI_Isend(&array[GETLINDEX(0,0,local_z -1)],1,int_xy_plane,
            above,10,gridcomm,&request[20]);
        MPI_Irecv(&array[GETLINDEX(0,0,-1)],1,int_xy_plane,below,10,
            gridcomm,&request[21]); //MPI_STATUS_IGNORE);
        MPI_Isend(&array[GETLINDEX(0,0,0)],1,int_xy_plane,below,11,
            gridcomm,&request[22]);
        MPI_Irecv(&array[GETLINDEX(0,0,local_z)],1,int_xy_plane,
            above,11,gridcomm,&request[23]); //MPI_STATUS_IGNORE);
}

/*
 * initialise integer mpi datatypes
 */
void int_initMPIDatatypes(){
        MPI_Type_vector(local_y,local_z,local_z+2,MPI_INT,&
            int_zy_plane);
        MPI_Type_commit(&int_zy_plane);

        MPI_Type_vector(local_x,local_z,(local_z+2)*(local_y+2),
            MPI_INT,&int_xz_plane);
        MPI_Type_commit(&int_xz_plane);

        MPI_Type_vector(local_y,1,local_z+2,MPI_INT,&int_y_column);
        MPI_Type_commit(&int_y_column);
        MPI_Type_create_resized(int_y_column,0,(local_z+2)*(local_y
            +2)*sizeof(int),&int_y_column_resized);
        MPI_Type_vector(1,2,1,int_y_column_resized,&int_xy_plane);
        MPI_Type_commit(&int_xy_plane);
}

/*
 * Wait for int border exchange to finish
 */
void wait_exchange_int(){
   MPI_Waitall(12,&request[12],&status[12]);
```

```
}
/*
 * wait for float border exchange to finish
 */
void wait_exchange_float(){
  MPI_Waitall(12,request,status);
}

/*
 * wait for all border exchanges to finish
 */
void wait_exchange_all(){
  MPI_Waitall(24,request,status);
}
```

## B.2 Fast Marching Method

### B.2.1 fmm_mpi.h

```
#include "heap.h"

typedef
/*
 * a struct to store all variables for a given fmm implementation
 */
struct f {
  Heap* heap; // the heap
  float* timearray; // the timearray storing arrivaltimes
  int* bandarray; // storing band information
  float* velocityarray; // velocity field
  int x,y,z,posx,posy,posz; // x,y,z is size of local fmm matrix,
      posx, posy, posz is global coords for the starting point
  } FmmData;

/*
 * initilalize the FMM set velocity arrat, position of starting
    point and size of array
 */
FmmData* initFMM(float* velarray,int posx,int posy, int posz, int x,
    int y, int z);
/*
 * Free up used variables in FMM
 */
void freeFMM(FmmData* data);
/*
 * Execute the FMM
 */
void executeFMM(FmmData* data);
```

### B.2.2 fmm_mpi.c

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include "fmm_mpi.h"
#include "heap.h"
#include "array_mpi.h"


#define BAND 0
#define OUTSIDE -1
#define KNOWN n

//#define DEBUG
/*
 * number of loops
 */
int n;

/*
 * largest_solution is the largest value in the array
 * rollbacksmallest is the smallest value of received border values,
     which dictates the rollback number
 */
float largest_solution, rollbacksmallest;

/*
 * the n value which one should rollback to
 */
int rollbackn;
/*
 * mpi datatype for sending border points
 */
MPI_Datatype mpi_element_struct;
/*
 * Used for debug output
 */
float valuemax=0;
float valuemin=0;
/*
 * array containing the working status of each node
 */
int* working;

/*
 * Add a point to the heap
 */
void addToHeap(FmmData* data, int px, int py, int pz){
  Element *temp;
  #ifdef DEBUG
  printf("%d: adding %d, %d, %d to heap, heap size is %d maxsize if %
      d\n",cartrank,px,py,pz,data->heap->heapsize,data->heap->maxsize
      );
  #endif
  temp= malloc(sizeof(Element));
  temp->value = data->timearray[GETLINDEX(px,py,pz)];
  temp->x = px;
  temp->y = py;
  temp->z = pz;
  heapInsert(data->heap,temp);
  #ifdef DEBUG
  printf("%d: inserted %d, %d, %d to heap,heap size is %d\n",
      cartrank,px,py,pz,data->heap->heapsize);
  #endif

}

/*
 * initialize the FMM
 * velarray is the velocity field
 * posx,y,z is position of the starting point
 * x,y,z is the size of the array, most are read from array_mpi.h
 */
FmmData* initFMM(float* velarray, int posx,int posy, int posz, int x
    , int y, int z){
  working = malloc(sizeof(int) * size);
  /* init datatypes*/
  MPI_Element e;
  MPI_Datatype type[5] = { MPI_FLOAT, MPI_INT, MPI_INT, MPI_INT,
      MPI_INT };
  int blocklen[5] = { 1, 1, 1, 1 , 1};
  MPI_Aint disp[5];
  disp[0] = 0;
  disp[1] = sizeof(float);
  disp[2] = sizeof(int) + disp[1];
  disp[3] = sizeof(int) + disp[2];
  disp[4] = sizeof(int) + disp[3];
  MPI_Type_create_struct(5, blocklen, disp, type, &
      mpi_element_struct);
  MPI_Type_commit(&mpi_element_struct);
  /* end init datatypes*/
  #ifdef DEBUG
  printf("%d: done init mpi datatypes\n",cartrank);
  #endif
  int send = 0;
  FmmData* data;
  data = malloc(sizeof(FmmData));
  if(data == 0){
    printf("%d: Failed to allocate memory, exiting\n",cartrank);
    exit(1);
  }

  data->bandarray = malloc(sizeof(int)*LOCALARRAYSIZE);
  if(data->bandarray == 0){
    printf("%d: Failed to allocate memory, exiting\n",cartrank);
    exit(1);
  }
  data->timearray = malloc(sizeof(float)*LOCALARRAYSIZE);
  if(data->timearray == 0){
    printf("%d: Failed to allocate memory, exiting\n",cartrank);
    exit(1);
  }

  data->velocityarray = velarray;
  #ifdef DEBUG
  printf("%d: initializing heap\n",cartrank);
  #endif
  data->heap = initHeap(LOCALARRAYSIZE);
  data->x = x;
  data->y = y;
  data->z = z;
  data->posx = posx;
  data->posy = posy;
  data->posz = posz;

  #ifdef DEBUG
  printf("%d: clearing memory bandarray %p size %d\n",cartrank,data->bandarray,sizeof(int)*LOCALARRAYSIZE);
  fflush(stdout);
  #endif
  /*
   * setting the band array to outside
   */
  memset(data->bandarray,OUTSIDE,sizeof(int)*LOCALARRAYSIZE);
  #ifdef DEBUG
  printf("%d: Heap ok, clearing memory timearray %p size %d\n",cartrank,data->timearray,sizeof(float)*LOCALARRAYSIZE);
  #endif
  /*
   * zeroing the arrival time array
   */
  int i = 0;
  for(i=0;i<LOCALARRAYSIZE;i++)
    data->timearray[i] = 0;
  //memset(data->timearray,0,sizeof(float)*LOCALARRAYSIZE);
  //bzero(data->timearray,sizeof(float)*LOCALARRAYSIZE);
  data->sentarray = malloc(sizeof(float)*LOCALARRAYSIZE);
  bzero(data->sentarray,sizeof(float)*LOCALARRAYSIZE);
  #ifdef DEBUG
  printf("%d: done allocating memory, setting starting point\n",cartrank);
  fflush(stdout);
  #endif
  /*
   * inserting the starting point on the correct node and add it to the heap / narrow band
   */
  if(getDest(posx,posy,posz) == cartrank){
    int *cr = getLocalCord(posx,posy,posz);
    // insert starting point
    data->timearray[GETLINDEX(cr[0],cr[1],cr[2])] = 0;
    data->bandarray[GETLINDEX(cr[0],cr[1],cr[2])] = BAND;
    //printf("%d: sat pos %d %d %d, as known\n",cartrank,cr[0],cr[1],cr[2]);
    Element* element;
    element = malloc(sizeof(Element));
    element->x = cr[0];
    element->y = cr[1];
    element->z = cr[2];
    element->value = data->timearray[GETLINDEX(cr[0],cr[1],cr[2])];
    heapInsert(data->heap,element);
    send = 1;
  }
  return data;
}
/*
 * return min
 */
float min(float per, float truls){
  if(per > truls)
    return truls;
  else return per;
}
/*
 * return max
 */
float max(float per, float truls){
  if(per > truls)
    return per;
  else return truls;
}

/*
 * prints xy plane from a float array
 */
void printFloatArray(int sizex, int sizey, int z,float* array){
  int i,j;
  printf("\n␣Printing␣matrix\n");
  for(i=0;i<sizex;i++){
    for(j=0;j<sizey;j++){
      printf("␣%8.2f␣",array[GETLINDEX(i,j,z)]);
    }
    printf("\n");
  }
}

/*
 * calculate the arrival time for a point x,y,z
 */
float calcDistance(FmmData* data, int x, int y, int z){
  float sol;
  sol = BIGFLOAT;
  if(data->bandarray[GETLINDEX(x+1,y,z)] > BAND){
    sol = min(data->timearray[GETLINDEX(x+1,y,z)] +1/data->
         velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣x+1\n",cartrank,data->timearray[
        GETLINDEX(x+1,y,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
        ;
    #endif
  }
  if(data->bandarray[GETLINDEX(x-1,y,z)] > BAND){
    sol = min(data->timearray[GETLINDEX(x-1,y,z)] +1/data->
         velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣x-1\n",cartrank,data->timearray[
        GETLINDEX(x-1,y,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
        ;
    #endif
  }
  if(data->bandarray[GETLINDEX(x,y+1,z)] > BAND){
    sol = min(data->timearray[GETLINDEX(x,y+1,z)] +1/data->
        velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣y+1\n",cartrank,data->timearray[
        GETLINDEX(x,y+1,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
        ;
    #endif
  }
  if(data->bandarray[GETLINDEX(x,y-1,z)] > BAND){
    sol = min(data->timearray[GETLINDEX(x,y-1,z)] +1/data->
        velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣y-1\n",cartrank,data->timearray[
        GETLINDEX(x,y-1,z)] +1/data->velocityarray[GETLINDEX(x,y,z)])
        ;
    #endif
  }
  if(data->bandarray[GETLINDEX(x,y,z+1)] > BAND){
    sol = min(data->timearray[GETLINDEX(x,y,z+1)] +1/data->
        velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣z+1\n",cartrank,data->timearray[
        GETLINDEX(x,y,z+1)] +1/data->velocityarray[GETLINDEX(x,y,z)])
        ;
    #endif
  }
  if(data->bandarray[GETLINDEX(x,y,z-1)] > BAND){
    sol = min(data->timearray[GETLINDEX(x,y,z-1)] +1/data->
        velocityarray[GETLINDEX(x,y,z)],sol);
    #ifdef DEBUG
    printf("%d:␣sol␣is␣%f␣for␣z-1\n",cartrank,data->timearray[
        GETLINDEX(x,y,z-1)] +1/data->velocityarray[GETLINDEX(x,y,z)])
        ;
    #endif
  }

  return sol;
}

/*
 * calculate a new point the the arrival time array if its inside
     the array and not KNOWN, if its outside add it to the heap
 */
void calcElement(FmmData* data, int px, int py, int pz){
        int add = 0;
        float sol;
        Element *temp;
        /*
         * Check if the point is inside the array*/
        if(px >= 0 && px < data->x && py >= 0 && py < data->y  && pz
            >= 0 && pz < data->z){
        /* Make sure the point is not KNOWN */
        if(data->bandarray[GETLINDEX(px,py,pz)] <= BAND){
                sol = calcDistance(data,px,py,pz);
                /* Check if the number is not smaller than the one
                    we calculated, when we exit, should never happend
                     in serial version*/
                if(data->timearray[GETLINDEX(px,py,pz)] != 0 && data->timearray[GETLINDEX(px,py,pz)] <= sol){
                        return;
                }
                /* If the point is OUSIDE add it to the heap/narrowband */
                if(data->bandarray[GETLINDEX(px,py,pz)] == OUTSIDE){
                        data->bandarray[GETLINDEX(px,py,pz)] = BAND;
                        add = 1;
                }

                #ifdef DEBUG
                printf("%d:␣sol␣is␣%f,␣for␣%d␣%d␣%d\n",cartrank,sol,px,py,pz);
                #endif
                /* store max and min for debug purposes */
                #ifdef DEBUG
                if(sol > valuemax && sol != BIGFLOAT){
                        valuemax=sol;
                }
                if(sol < valuemin){
                        valuemin = sol;
                }
                #endif
                /* set the new arrivaltime */
                data->timearray[GETLINDEX(px,py,pz)] = sol;
                #ifdef DEBUG
                printf("%d:␣x␣%d␣,␣y␣%d,␣z␣%d,␣k[i]␣%d␣l[i]␣%d␣m[i]␣%d,␣value␣%f␣\n",cartrank,data->x, data->y, data->z,px,py,pz, data->timearray[GETLINDEX(px,py,pz)]);
                #endif
                /* add it to the narrowband if it should be added */
                if(add){
                        addToHeap(data,px,py,pz);
                        }
                }
        }
}

//remove ???
void checkforchange(FmmData* data,int px, int py, int pz){
  float sol = BIGFLOAT;
  int i;
  int o[6] = {px-1,px,px+1,px,px,px};
  int l[6] = {py,py-1, py,py+1,py,py};
  int m[6] = {pz,pz,pz,pz,pz-1,pz+1};
  for(i=0;i<6;i++){
    if(o[i] >= 0 && o[i] < data->x && l[i] >= 0 && l[i] < data->y && m[i] >= 0 && m[i] < data->z && data->bandarray[GETLINDEX(o[i],l[i],m[i])] > BAND){

      #ifdef DEBUG
      printf("%d:␣checking␣for␣rollback␣%d␣%d␣%d␣\n",cartrank,o[i],l[i],m[i]);
      #endif
      sol = calcDistance(data,px,py,pz);
```

```

        if(sol < data->timearray[GETLINDEX(o[i],l[i],m[i])] && data->
            bandarray[GETLINDEX(o[i],l[i],m[i])] > BAND){
          // new value is smaller lets add this to our heap

          #ifdef DEBUG
          printf("%d:␣rolling␣back␣%d␣%d␣%d␣old␣value␣%f␣new␣value␣%f\
              n",cartrank,o[i],l[i],m[i],data->timearray[GETLINDEX(o[i
              ],l[i],m[i])],sol);
          #endif
          data->timearray[GETLINDEX(o[i],l[i],m[i])] = sol;
          data->bandarray[GETLINDEX(o[i],l[i],m[i])] = BAND;
          addToHeap(data,o[i],l[i],m[i]);
          checkforchange(data,o[i],l[i],m[i]);
        }
      }
    }
}

/*
 * Add a new element to the array
 */
void addElement(FmmData* data,MPI_Element e){
    int* cr;
    int i,j,k;
    cr = getLocalCord(e.x,e.y,e.z);

    #ifdef DEBUG
    printf("%d:␣adding␣element␣%d␣%d␣%d,␣to␣local␣%d␣%d␣%d\n",cartrank
        ,e.x,e.y,e.z,cr[0],cr[1],cr[2]);
    #endif
    /*
     * checking if we have added it before
     * Should be always no since we don't send the same value multiple
          times
     */
    if(data->timearray[GETLINDEX(cr[0],cr[1],cr[2])] == e.value){
      #ifdef DEBUG
      printf("%d:␣alreaddy␣added␣%d␣%d␣%d\n",cartrank,e.x,e.y,e.z);
      #endif
      return;
    }
    data->timearray[GETLINDEX(cr[0],cr[1],cr[2])] = e.value;
    data->bandarray[GETLINDEX(cr[0],cr[1],cr[2])] = e.n;
    int o[6] = {cr[0]-1,cr[0],cr[0]+1,cr[0],cr[0],cr[0]};
    int l[6] = {cr[1],cr[1]-1, cr[1],cr[1]+1,cr[1],cr[1]};
    int m[6] = {cr[2],cr[2],cr[2],cr[2],cr[2]-1,cr[2]+1};
    /*
     * recalculate all neighbours
     */
    for(i=0;i<6;i++){

      #ifdef DEBUG
      printf("%d:␣calc␣element␣%d␣%d␣%d\n",cartrank,o[i],l[i],m[i]);
      #endif
      calcElement(data, o[i], l[i], m[i]);
    }
    /*
    * set rollback values, so we can check if a rollback is necesarry
    */
   /*if(largest_solution > e.value && rollbacksmallest > e.value){
   rollbacksmallest = e.value;
   rollbackn = e.n;
   }*/
}

/*
 * rollback all values above input value
 */
void rollback(FmmData* data, float value){
   int i,j,k;
   for(i=0;i<local_x;i++)
      for(j=0;j<local_y;j++)
         for(k=0;k<local_z;k++){
            if(data->bandarray[GETLINDEX(i,j,k)] > BAND && data->
                timearray[GETLINDEX(i,j,k)] > value){

              data->bandarray[GETLINDEX(i,j,k)] = BAND;
              addToHeap(data,i,j,k);
            }
         }
}

/*
 * send new values and check for incoming border values
 */
void sendRecvBorderChanges(FmmData* data,int x, int y, int z,int
     send){
   MPI_Element e;
   int run= 1;
   int *cr;
   int reast,rwest,rnorth,rsouth,rabove,rbelow;
   int seast,swest,snorth,ssouth,sabove,sbelow;
   reast = rwest = rnorth = rsouth = rabove = rbelow = 0;
   seast = swest = snorth = ssouth = sabove= sbelow = 0;
   /*
    * if we are to send a value
    */
   if(send){
      /*
       * see where we have to send the value
       */
      if(x == 0){
         snorth = 1;
      }
      if(x == local_x-1){
         ssouth = 1;
      }
      if(y == 0){
         swest = 1;
         }
      if(y == local_y-1){
         seast = 1;
      }
      if(z == 0){
         sbelow = 1;
```

```
  }
  if(z == local_z -1){
    sabove = 1;
  }
  #ifdef DEBUG
  printf("%d:␣%d␣%d␣%d␣sending␣to␣north␣%d␣%d␣south␣%d␣␣%d␣west␣%d␣%d␣east␣%d␣%d␣below␣%d␣%d␣above␣%d␣%d\n",cartrank,x,y,z,snorth,north,ssouth,south,swest,west,seast,east,sbelow,below,sabove,above);
  #endif


if(snorth || ssouth || swest || seast || sabove || sbelow){
  cr = getGlobalCord(x,y,z);
  #ifdef DEBUG
  printf("%d:␣sending␣element␣at␣%d␣%d␣%d␣gave␣global␣coord␣%d␣%d␣%d\n",cartrank,x,y,z,cr[0],cr[1],cr[2]);
  #endif
  e.x = cr[0];
  e.y = cr[1];
  e.z = cr[2];
  e.value = data->timearray[GETLINDEX(x,y,z)];
  data->sentarray[GETLINDEX(x,y,z)] = e.value;
  e.n = data->bandarray[GETLINDEX(x,y,z)];

  if(snorth){
    MPI_Send(&e,1,mpi_element_struct,north,1,gridcomm);
  }
  if(ssouth){
    MPI_Send(&e,1,mpi_element_struct,south,1,gridcomm);
  }
  if(swest){
    MPI_Send(&e,1,mpi_element_struct,west,1,gridcomm);
  }
  if(seast){
    MPI_Send(&e,1,mpi_element_struct,east,1,gridcomm);
  }
  if(sabove){
    MPI_Send(&e,1,mpi_element_struct,above,1,gridcomm);
  }
  if(sbelow){
    MPI_Send(&e,1,mpi_element_struct,below,1,gridcomm);
  }
}
}
rollbacksmallest = BIGFLOAT;
rollbackn = 0;
/*
 * a loop to receive all incoming border values
 */
while(run){
MPI_Iprobe(MPI_ANY_SOURCE,1,gridcomm,&run,MPI_STATUS_IGNORE);
if(run){
  MPI_Iprobe(north,1,gridcomm,&rnorth,MPI_STATUS_IGNORE);
  MPI_Iprobe(south,1,gridcomm,&rsouth,MPI_STATUS_IGNORE);
  MPI_Iprobe(east,1,gridcomm,&reast,MPI_STATUS_IGNORE);
  MPI_Iprobe(west,1,gridcomm,&rwest,MPI_STATUS_IGNORE);
  MPI_Iprobe(above,1,gridcomm,&rabove,MPI_STATUS_IGNORE);
```

```
      MPI_Iprobe(below,1,gridcomm,&rbelow,MPI_STATUS_IGNORE);
      if(rnorth || rsouth || rwest || reast || rabove || rbelow){

        if(rnorth){
          MPI_Recv(&e,1,mpi_element_struct,north,1,gridcomm,
              MPI_STATUS_IGNORE);
          addElement(data,e);
        }
        if(rsouth){
          MPI_Recv(&e,1,mpi_element_struct,south,1,gridcomm,
              MPI_STATUS_IGNORE);
          addElement(data,e);
        }
        if(rwest){
          MPI_Recv(&e,1,mpi_element_struct,west,1,gridcomm,
              MPI_STATUS_IGNORE);
          addElement(data,e);
        }
        if(reast){
          MPI_Recv(&e,1,mpi_element_struct,east,1,gridcomm,
              MPI_STATUS_IGNORE);
          addElement(data,e);
        }
        if(rabove){
          MPI_Recv(&e,1,mpi_element_struct,above,1,gridcomm,
              MPI_STATUS_IGNORE);
          addElement(data,e);
        }
        if(rbelow){
          MPI_Recv(&e,1,mpi_element_struct,below,1,gridcomm,
              MPI_STATUS_IGNORE);
          addElement(data,e);
        }
      }
    }
  }
  /*
   * rollback if necessary
   */
  if(rollbackn){
    rollback(data,rollbacksmallest);
    n = rollbackn;
    rollbackn = 0;
    rollbacksmallest = BIGFLOAT;
  }
}

/*
 * old synchroneous border exhcange
 */
/*
void sendRecvBorderChanges(FmmData* data,int x, int y, int z,int
    send){
  MPI_Element e;

  int *cr;
  int reast,rwest,rnorth,rsouth,rabove,rbelow;
  int seast,swest,snorth,ssouth,sabove,sbelow;
  reast = rwest = rnorth = rsouth = rabove = rbelow = 0;
  seast = swest = snorth = ssouth = sabove= sbelow = 0;
  if(send){
    if(x == 0){
      snorth = 1;
    }
    if(x == local_x -1){
      ssouth = 1;
    }
    if(y == 0){
      swest = 1;
      }
    if(y == local_y -1){
      seast = 1;
    }
    if(z == 0){
      sbelow = 1;
    }
    if(z == local_z -1){
      sabove = 1;
    }
    #ifdef DEBUG
    printf("%d: %d %d %d sending to north %d %d south %d  %d west %d
         %d east %d %d below %d %d above %d %d\n",cartrank,x,y,z,
        snorth,north,ssouth,south,swest,west,seast,east,sbelow,below,
        sabove,above);
    #endif
  }

  MPI_Send(&swest,1,MPI_INT,west,0,gridcomm);
  MPI_Recv(&reast,1,MPI_INT,east,0,gridcomm,MPI_STATUS_IGNORE);
  MPI_Send(&seast,1,MPI_INT,east,0,gridcomm);
  MPI_Recv(&rwest,1,MPI_INT,west,0,gridcomm,MPI_STATUS_IGNORE);

  MPI_Send(&ssouth,1,MPI_INT,south,0,gridcomm);
  MPI_Recv(&rnorth,1,MPI_INT,north,0,gridcomm,MPI_STATUS_IGNORE);
  MPI_Send(&snorth,1,MPI_INT,north,0,gridcomm);
  MPI_Recv(&rsouth,1,MPI_INT,south,0,gridcomm,MPI_STATUS_IGNORE);

  MPI_Send(&sabove,1,MPI_INT,above,0,gridcomm);
  MPI_Recv(&rbelow,1,MPI_INT,below,0,gridcomm,MPI_STATUS_IGNORE);
  MPI_Send(&sbelow,1,MPI_INT,below,0,gridcomm);
  MPI_Recv(&rabove,1,MPI_INT,above,0,gridcomm,MPI_STATUS_IGNORE);

  if(snorth || ssouth || swest || seast || sabove || sbelow){
    cr = getGlobalCord(x,y,z);

    #ifdef DEBUG
    printf("%d: sending element at %d %d %d gave global coord %d %d
        %d\n",cartrank,x,y,z,cr[0],cr[1],cr[2]);
    #endif
    e.x = cr[0];
    e.y = cr[1];
    e.z = cr[2];
    e.value = data->timearray[GETLINDEX(x,y,z)];
    e.n = data->bandarray[GETLINDEX(x,y,z)];

    if(snorth){
        MPI_Send(&e,1,mpi_element_struct,north,1,gridcomm);
      }
      if(ssouth){
        MPI_Send(&e,1,mpi_element_struct,south,1,gridcomm);
      }
      if(swest){
        MPI_Send(&e,1,mpi_element_struct,west,1,gridcomm);
      }
      if(seast){
        MPI_Send(&e,1,mpi_element_struct,east,1,gridcomm);
      }
      if(sabove){
        MPI_Send(&e,1,mpi_element_struct,above,1,gridcomm);
      }
      if(sbelow){
        MPI_Send(&e,1,mpi_element_struct,below,1,gridcomm);
      }
    }

    if(rnorth || rsouth || rwest || reast || rabove || rbelow){

      if(rnorth){
        MPI_Recv(&e,1,mpi_element_struct,north,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rsouth){
        MPI_Recv(&e,1,mpi_element_struct,south,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rwest){
        MPI_Recv(&e,1,mpi_element_struct,west,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(reast){
        MPI_Recv(&e,1,mpi_element_struct,east,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rabove){
        MPI_Recv(&e,1,mpi_element_struct,above,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
      if(rbelow){
        MPI_Recv(&e,1,mpi_element_struct,below,1,gridcomm,
            MPI_STATUS_IGNORE);
        addElement(data,e);
      }
    }
}
*/

/*
 * check if someone wants to update their working status
 */
```

```c
void checkOthers(){
  int i;
  int flag;
  for(i = 0; i<size;i++){
    MPI_Iprobe(i,9,gridcomm,&flag,MPI_STATUS_IGNORE);
    if(flag){
      MPI_Recv(&working[i],1,MPI_INT,i,9,gridcomm,MPI_STATUS_IGNORE)
          ;
    }
  }
}
/*
 * notify others that my working status is changed
 */
void notifyOthers(int value){
  int i;
  for(i = 0; i<size;i++){
    MPI_Send(&value,1,MPI_INT,i,9,gridcomm);
  }
}

/*
 * Execute the FMM
 */
void executeFMM(FmmData* data){
  int add=0;
  int posx,posy,posz;
  int run = 1;
  int sendrun = 1;
  int senddata = 0;
  int end = 0;
  int sum = 0;
  int i;
  #ifdef DEBUG
  printf("data size is %d %d %d\n",data->x, data->y, data->z);
  #endif
  for(i = 0; i<size;i++){
    working[i] = 1;
  }
  largest_solution = 0;
  n = 1;
  /* loop will run until all nodes are done */
  while(!end){
  /* working loop, will run until there are no more work to be done
      */
  while(run){


    if(heapGetMin(data->heap)){

     Element* e,*temp;

     e = heapExtractMin(data->heap);
     #ifdef DEBUG
     printf("%d: setting %d %d %d to known\n",cartrank,e->x,e->y,e->
         z);
     #endif
     data->bandarray[GETLINDEX(e->x,e->y,e->z)] = KNOWN;
```

```
      int k[6] = {e->x-1,e->x,e->x+1,e->x,e->x,e->x};
      int l[6] = {e->y, e->y-1, e->y, e->y+1,e->y,e->y};
      int m[6] = {e->z, e->z, e->z, e->z,e->z-1,e->z+1};
      int i;
      posx = e->x;
      posy = e->y;
      posz = e->z;
      /*
       * update largest_solution if this solution is the largest
       */
      if(data->timearray[GETLINDEX(posx,posy,posz)] >
          largest_solution){
       largest_solution = data->timearray[GETLINDEX(posx,posy,posz)];
      }
      /* check if this point has been sent before */
      if(data->sentarray[GETLINDEX(posx,posy,posz)] == 0){
       senddata = 1;
      }else if(data->sentarray[GETLINDEX(posx,posy,posz)] <= data->
          timearray[GETLINDEX(posx,posy,posz)]){
       senddata = 0;
      }

      /*printFloatArray(data->x,data->y,-2,data->timearray);
      printFloatArray(data->x,data->y,-1,data->timearray);
      printFloatArray(data->x,data->y,0,data->timearray);
      printFloatArray(data->x,data->y,1,data->timearray);
      printFloatArray(data->x,data->y,2,data->timearray);*/
      for(i=0;i<6;i++){

       if(k[i] >= 0 && k[i] < data->x && l[i] >= 0 && l[i] < data->y
            && m[i] >= 0 && m[i] < data->z){
         calcElement(data, k[i], l[i], m[i]);
       }
      }

    #ifdef DEBUG

      /*printFloatArray(data->x,data->y,-2,data->timearray);
      printFloatArray(data->x,data->y,-1,data->timearray);
      printFloatArray(data->x,data->y,0,data->timearray);
      printFloatArray(data->x,data->y,1,data->timearray);
      printFloatArray(data->x,data->y,2,data->timearray);*/
    #endif
      free(e);

      n++;
    }
    /* Send changes to border and look for incoming changes to the
        border */
    sendRecvBorderChanges(data,posx,posy,posz,senddata);

    #ifdef DEBUG
    if(cartrank == 0|| cartrank == -2){printLocalArray(data->
        timearray);
    printLocalIntArray(data->bandarray);
    }
    #endif
    senddata = 0;
```

```
      /*Do we end the loop?*/
      if(heapGetMin(data->heap) == 0){
        run = 0;
      }else{
        run = 1;
      }
      checkOthers();
      #ifdef DEBUG
      if(cartrank == 0 && n%1000 == 0){
        printf("%d:␣reached␣n␣%d\n",cartrank,n);
      }
      #endif
    }
      /* see if border changes are coming */
      sendRecvBorderChanges(data,0,0,0,0);
      /* if we have work to do lets notify others and start to work
          again, if not let others know we are done */
      if(heapGetMin(data->heap) != 0){
        run = 1;
        // notify other i am still working
        notifyOthers(1);
      }else if(working[cartrank] == 1){
        // notify that i have stopped working
        notifyOthers(0);
      }
      checkOthers();
      sum = 0;
      for(i = 0; i<size;i++){
        sum += working[i];
      }
      if(sum == 0){
        end = 1;
      }
    }
    #ifdef DEBUG
    printf("valuemax␣%f␣valuemin␣%f\n",valuemax,valuemin);
    #endif
}


/*
 * free the variables used in the FMM
 */
void freeFMM(FmmData* data){
    //free(data->timearray);
    free(data->bandarray);
    free(data);
}
```

## B.3 Application

### B.3.1 mpi_app.c

```
#include <stdio.h>
#include <stdlib.h>
#include "array_mpi.h"
#include "fileio.h"
#include <mpi.h>
#include <string.h>
#include "fmm_mpi.h"
#include "time.h"

#define MPIDEBUG 0
#define DEBUG 0

/* set that no dimensions sould be cyclic */
int periods[3] = {0,0,0};
/* set the number of dimensions to use in the cartesian node grid*/
int ndims = 3;

int div_x, div_y, div_z;

float* local_array;
float* global_array;
float* file_array;

/*
 * change velocity matrix so that all values are positive and use
     sqrt to reduce the difference between the values
 */
void fixMatrix(){
  float min,max;
  float rmin,rmax;
  int i,j,k;
  min = BIGFLOAT;
  max = BIGFLOAT*-1;
  for(i=0;i<local_x;i++)
    for(j=0;j<local_y;j++)
      for(k=0;k<local_z;k++){
        if(local_array[GETLINDEX(i,j,k)] > max)
          max = local_array[GETLINDEX(i,j,k)];
        if(local_array[GETLINDEX(i,j,k)] < min)
          min = local_array[GETLINDEX(i,j,k)];
      }
  MPI_Allreduce(&min,&rmin,1,MPI_FLOAT,MPI_MIN,gridcomm);
  MPI_Allreduce(&max,&rmax,1,MPI_FLOAT,MPI_MAX,gridcomm);
  min = rmin +1;
  max = rmax;
  for(i=0;i<local_x;i++)
    for(j=0;j<local_y;j++)
      for(k=0;k<local_z;k++){
        local_array[GETLINDEX(i,j,k)] = sqrt(local_array[GETLINDEX(i
            ,j,k)] + min);
      }
}
/*
```

```
 * A test function to check if the array dosen't contain a value
 */
int checkArray(int x, int y, int z,float value){
    int i,j,k;
    int rvalue = 1;
    for(i= 0; i<x;i++)
      for(j=0;j<y;j++)
        for(k=0;k<z;k++){
          if(local_array[GETLINDEX(i,j,k)] != value){
            rvalue = 0;
            return rvalue;
          }
        }
    return rvalue;
  }

/*
 * Check if a global array dosen't contain a specific value
 */
int checkGArray(int x, int y, int z,float value){
    int i,j,k;
    int rvalue = 1;
    for(i= 0; i< x;i++)
      for(j=0;j<y;j++)
        for(k=0;k<z;k++){
          if(global_array[GETINDEX(i,j,k)] != value){
            rvalue = 0;
            return rvalue;
          }
        }
    return rvalue;
  }

/*
 * Divide the global matrix into smaller matrixes for each node.
 * This function calculates the local dimensions
 */
void divide_matrix(){

  div_x = x/dims[0];
  div_y = y/dims[1];
  div_z = z/dims[2];
  local_x = div_x;
  local_y = div_y;
  local_z = div_z;
  if(local_x * dims[0] != x){
    if(coords[0] == dims[0]){
      local_x = (x-(local_x*dims[0])) + local_x;
    }
  }
  if(local_y * dims[1] != y){
    if(coords[1] == dims[1]){
      local_y = (y-(local_y*dims[1])) + local_y;
    }
  }
  if(local_z * dims[2] != z){
    if(coords[2] == dims[2]){
      local_z = (z-(local_z*dims[2])) + local_z;
    }
  }
  printf("%d:␣local_x␣%d␣local_y␣%d␣local_z␣%d␣coords␣%d,%d,%d␣\n",cartrank, local_x, local_y, local_z,coords[0],coords[1],coords[2]);
}


/*
 * Initialize the program allocating local matrixes and initializing data types
 */
void init(){
  divide_matrix();
  local_array = malloc(sizeof(float)*LOCALARRAYSIZE);
  if(local_array == 0){
    printf("Coudn't␣allocate␣enough␣memory␣for␣local_array\n");
    exit(1);
  }
  bzero(local_array,sizeof(float)*LOCALARRAYSIZE);
  initMPIDatatypes();
}

/*
 * a function for reading a file into each node, where the file contains a global array, each node will read its respective part into their local matrixes
 */
void scatterdata(char* filename){
  int i,j;
  int* cr;
  int value;
  char* errorstr;
  int reslen;
  FILE* f;
  int offset = 0;
  printf("%d:␣openeing␣file␣%s\n",cartrank,filename);
  f = fopen(filename,"rb");
  if(!f){
    printf("%d:␣unable␣to␣open␣file␣%d␣\n",cartrank,f);
    fflush(stdout);
    return;
  }
  if(DEBUG){
    printf("%d:␣opened␣file␣%d\n",cartrank,f);
  }
  for(i = 0; i<local_x;i++){
    for(j= 0; j< local_y; j++)
    {
      cr = getGlobalCord(i,j,0);
      offset = sizeof(float)*(cr[2]+cr[1]*z+cr[0]*z*y);
      fseek(f,offset,SEEK_SET);
      fread(&local_array[GETLINDEX(i,j,0)],sizeof(float),local_z,f);
      free(cr);
    }
  }
  fclose(f);

```

```
}

/*
 * gather all the local matrixes into a global matrix on node 0
 */
float* gatherdata(float* iarray){
  int i,j,k,dest,flag,r,t;
  float* farray;
  MPI_Request* requests;
  MPI_Status* status;
  requests = malloc(sizeof(MPI_Request)*local_y*local_x*2);
  status = malloc(sizeof(MPI_Status)*local_y*local_x*2);
  if(cartrank == 0){
    if(DEBUG){
      printf("%d:started gathering\n",cartrank);
    }
    farray = malloc(sizeof(float)*ARRAYSIZE);
    bzero(farray,sizeof(float)*ARRAYSIZE);

  }
  if(cartrank != 0){
  for(i=0;i<local_x;i++)
    for(j= 0; j<local_y;j++){
      MPI_Send(&iarray[GETLINDEX(i,j,0)],local_z,MPI_FLOAT,0,i*
          local_y+j,gridcomm);//,&requests[(local_y*i)+j]);
    }
  if(DEBUG){
    printf("%d: Done sending \n",cartrank);
  }
  }
  if(cartrank==0){
    if(MPIDEBUG){
      printf("%d: starting setting recvs\n",cartrank);
    }
    for(r=0;r<x;r+=local_x)
      for(t=0; t<y;t+=local_y)
        for(k=0;k<dims[2];k++){

          dest = getDest(r,t,k*local_z);
          if(DEBUG){
            printf("%d: receiving from %d\n",cartrank,dest);
          }
          for(i=0;i<local_x;i++)
            for(j=0;j<local_y;j++){
            MPI_Irecv(&farray[GETINDEX(i+r,j+t,k*local_z)],local_z,
                MPI_FLOAT,dest,i*local_y+j,gridcomm,&requests[(
                local_y*local_x)+(i*local_y+j)]);
            if(dest == 0){
              MPI_Isend(&iarray[GETLINDEX(i,j,0)],local_z,MPI_FLOAT
                  ,0,i*local_y+j,gridcomm,&requests[(local_y*i)+j]);
            }
            }
          if(dest == 0){
            MPI_Waitall(local_y*local_x*2,requests,status);
          }
          if(dest != 0){
          MPI_Waitall(local_y*local_x,&requests[local_y*local_x],
              status);
            }
         }
   }

   printf("%d:␣Done␣gathering␣\n",cartrank);
   return farray;
}

/*
 * check if the global array is the same as the array inside a file
 */
void checkData(char* filename){

   if(cartrank == 0){
   int i,j,k;
      printf("%d:␣Reading␣file\n",cartrank);
      file_array = malloc(ARRAYSIZE*sizeof(float));
      bzero(file_array,ARRAYSIZE*sizeof(float));
      readfile(file_array,filename,x,y,z);

      printArray(file_array);
      printf("%d:␣checking␣data␣consistency␣%f\n",cartrank,file_array[
          GETINDEX(0,0,0)]);
      for(i=0;i<x;i++)
         for(j=0;j<y;j++)
            for(k=0;k<z;k++){
               if(file_array[GETINDEX(i,j,k)] != global_array[GETINDEX(i,
                   j,k)]){
                  printf("%d:␣error␣at␣%d␣%d␣%d␣file␣%lf␣global␣%lf\n",
                      cartrank,i,j,k,file_array[GETINDEX(i,j,k)],
                      global_array[GETINDEX(i,j,k)]);
               }
            }
   }
}

/*
 * a test function for writing a file with values
 */
void writeafile(){
   FILE* f;
   f = fopen("/work/idarbo/per.conv","w");
   int i,j,k;
   float value;
   for(i=0;i<x;i++)
      for(j=0;j<y;j++)
         for(k=0;k<z;k++){
         //value = i+j+k;
         value = 1;
         if(j < 8 && k < 8){
            value = 9;
         }

         fwrite(&value,sizeof(float),1,f);
         }
   fclose(f);
}

```

```
int main(int argc, char** argv){
  float* array,*time1,*time2;
  int timeusec,timesec, timeusec2, timesec2,rtimesec, rtimeusec;
  FmmData* data;
        x = SIZEX;
        y = SIZEY;
        z = SIZEZ;
        int i, j, k;
        /*initialize MPI*/
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Dims_create(size,ndims,dims);

  MPI_Cart_create(MPI_COMM_WORLD,ndims, dims, periods, 0, &gridcomm)
      ;

  MPI_Cart_shift(gridcomm, 0, 1, &north, &south);

  MPI_Cart_shift(gridcomm, 1, 1, &west, &east);
  MPI_Cart_shift(gridcomm, 2, 1, &below, &above);

  MPI_Comm_rank(gridcomm, &cartrank);

  MPI_Cart_coords(gridcomm, cartrank, 3, coords);
  //if(cartrank ==0)
  //writeafile();

  /* init the program */
  init();

  if(MPIDEBUG){
    printf("%d: west %d east %d south %d north %d below %d above %d 
        cartrank %d\n",rank,west,east,south,north,below,above,
        cartrank);
    fflush(stdout);
  }



  if(cartrank == 0){
    printf("%d: Init complete reading data from file\n",cartrank);
    fflush(stdout);
  }
  /* read data from file */
  scatterdata(argv[1]);
  fixMatrix();
  /* instead of reading set the array to 1.0 */
  /*for(i=-1;i<=local_x;i++)
    for(j=-1;j<=local_y;j++)
      for(k=-1;k<=local_z;k++)
        local_array[GETLINDEX(i,j,k)] = 1.0;
  */
  //printLocalArray(local_array);
  if(cartrank == 0){
    printf("%d: Read data exchanging borders\n",cartrank);
    fflush(stdout);
  }
  /* exchange borders so border values will be correct in the
      velocity array */
  exchangeBorders(local_array);
  wait_exchange_float();
  if(cartrank == 0){
    printf("%d:␣Initializing␣FMM␣point␣is␣%d␣%d␣%d␣\n",cartrank,x/2,
        y/2,z/2);
    fflush(stdout);
  }
  /* initialize the FMM */
  data = initFMM(local_array,x/2,y/2,z/2,local_x,local_y,local_z);

  if(cartrank == 0){
    printf("%d:␣Executing␣FMM\n",cartrank);
    fflush(stdout);
  }
  /* take timing and execute the FMM*/
  MPI_Barrier(gridcomm);
  timeusec = getTimeInMicroseconds();
  timesec = getTimeInSeconds();
  executeFMM(data);
  timeusec2 = getTimeInMicroseconds();
  timesec2 = getTimeInSeconds();
  fixTime(timesec,timeusec,timesec2,timeusec2,&rtimesec,&rtimeusec);
  if(cartrank == 0){
    printf("%d:␣execute␣FMM␣took␣%d␣seconds␣and␣%d␣microseconds\n",
        cartrank,rtimesec,rtimeusec);
  }

  /*
   * Code for checking output, in production this is the part one
       should store the arrival time array
   */
    if(DEBUG){
      printLocalArray(data->timearray);
    }
  MPI_Barrier(gridcomm);

  if(cartrank == 0){
    printf("%d:␣FMM␣done␣gathering␣data\n",cartrank);
  }
  //global_array = gatherdata(data->timearray);

  //global_array = gatherdata(local_array);




  if(cartrank == 0){

        //printArray(global_array);
    if(DEBUG){
          //printArray(global_array);
    }
    /* if(checkGArray(x,y,z,0.0)){
      printf("%d: global_array is all 0.0\n",cartrank);
    } else {
      printf("%d: global_array is good\n",cartrank);
      }*/
  }

  printf("%d is done ending gracefully.\n",cartrank);
  MPI_Finalize();

}
```

# Appendix C

# Common files

## C.1 Heap

### C.1.1 heap.h

```
#ifndef __HEAP_H
#define __HEAP_H
typedef
/*
 * a element in the heap
 */
struct e {
        float value;
        int x;
        int y;
        int z;
} Element;

/*
 * heap storage struct
 */
typedef struct h {
  Element** array;
  int maxsize;
  int heapsize;
} Heap;
/*
 * Initialize the heap to a specific max size.
 * The heap should not exceed the size, it will then fail
 */
Heap* initHeap(int size);
/*
 * Return and remove the smallest element in the heap
 */
Element* heapExtractMin(Heap* heap);
/*
 * Returns the smallest element in the heap without removing it
 */
Element* heapGetMin(Heap* heap);
```

```
/*
 * Insert a new element to the heap
 */
void heapInsert(Heap* heap,Element* key);
#endif
```

### C.1.2 heap.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "heap.h"

//#define DEBUG
/*
 * Initialize the heap with size as max heap size*/
Heap* initHeap(int size){
  Element **array;
  Heap* heap;
  #ifdef DEBUG
  printf("Ready␣to␣allocate␣heap\n");
  #endif
  heap = malloc(sizeof(Heap));
  array = malloc(sizeof(Element*)*size);

  #ifdef DEBUG
  printf("Allocated␣memory␣for␣heap\n");
  #endif
  heap->array = array;
  heap->maxsize = size;
  heap->heapsize=0;

  #ifdef DEBUG
  printf("returning␣heap\n");
  #endif
  return heap;
}

/*
 * A support function that checks that the heap is realy min sorted
 * Only used in testing if the heap works correctly, shoudn't be
     used in production code
 */
float checkHeapConsistensy(Heap* heap){
        int i;
        float min = 10000000000.0;
  if(heap->heapsize > 1){

  #ifdef DEBUG
  printf("checking␣consistency,␣size␣is␣%d\n", heap->heapsize);
  #endif
        for(i=1;i<=heap->heapsize;i++){
                if(min > heap->array[i]->value)
                min = heap->array[i]->value;
        }
  #ifdef DEBUG
  printf("root␣is␣%f,␣smallest␣is␣%f\n", heap->array[1]->value, min)
     ;
  #endif
  }
  return min;
}

/*
 * Returns the parten to a specific position in the heap
 */
int inline getParent(int pos){
  return pos>>1; //pos/2;
}
/*
 * Gets the left child of a specific position in the heap
 */
int inline getLeft(int pos){
  return pos <<1; // pos * 2
}
/*
 * Gets the right child of a specific position in the heap
 */
int inline getRight(int pos){
  return (pos <<1) +1; // pos*2+1
}

/*
 * Move one value to its correct position in the min sorted heap
 * Run once for each new element in the heap
 */
void MinHeapify(Heap* heap,int pos){
  int l,r;
  int smallest = 0;
  l = getLeft(pos);
  r = getRight(pos);
  if(l <= heap->heapsize && heap->array[l]->value < heap->array[pos
     ]->value){
    smallest = l;
  }else{
    smallest = pos;
  }
  if(r<= heap->heapsize && heap->array[r]->value < heap->array[
     smallest]->value)
    smallest = r;
  if(smallest != pos){
    Element* temp;
    temp = heap->array[pos];
    heap->array[pos] = heap->array[smallest];
    heap->array[smallest] = temp;
    MinHeapify(heap,smallest);
  }
}

/*
 * return and remove the smallest element from the heap
 */
Element* heapExtractMin(Heap* heap){
  if(heap->heapsize<1) return NULL;
  Element* temp = heap->array[1];
  heap->array[1] = heap->array[heap->heapsize];
  heap->heapsize--;
  #ifdef DEBUG
  printf("New␣heapsize␣%d\n",heap->heapsize);
  #endif

  MinHeapify(heap,1);
  return temp;
```

```
}

/*
 * Return the smallest element from the heap
 * Does not remove anything from the heap
 */
Element* heapGetMin(Heap* heap){
  if(heap->heapsize<1) return NULL;
  Element* temp = heap->array[1];
  return temp;
}

/*
 * Add a new element to the heap
 * this function adds a element to a position and moves it up until
     its found its place in the heap.
 */
void heapIncreaseKey(Heap* heap,int pos, Element* key){
  Element* temp;
  //if(key->value > array[pos]->value){
  if(heap->array[pos] != 0){
    exit(2);
  }
  #ifdef DEBUG
  printf("Inserting␣into␣%d\n",pos);
  #endif
  heap->array[pos] = key;
  while (pos > 1 && heap->array[getParent(pos)]->value > heap->array
      [pos]->value){
    #ifdef DEBUG
    printf("chaging␣position␣between␣%d␣and␣%d\n",pos,getParent(pos)
        );
    #endif
    temp = heap->array[pos];
    heap->array[pos] = heap->array[getParent(pos)];
    heap->array[getParent(pos)] = temp;
    pos = getParent(pos);
  }
  //checkHeapConsistensy(heap);
}

/*
 * add a new element to the heap
 */
void heapInsert(Heap* heap,Element* key){
  heap->heapsize++;
  if(heap->heapsize > heap->maxsize)
  {
    printf("Exceeded␣heap␣max␣size␣of␣%d\n",heap->maxsize);
    exit(5);
  }
  #ifdef DEBUG
  printf("New␣heapsize␣%d␣array␣at␣%p\n",heap->heapsize,heap->array)
      ;
  #endif
  heap->array[heap->heapsize] = 0;
  heapIncreaseKey(heap,heap->heapsize,key);
}
```

### C.1.3 testheap.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "heap.h"

Heap* heap;

int main(int argc, char** argv){
  int i;
  heap = initHeap(500);
  Element *temp;
  Element en;
  Element to;
  Element tre;
  Element fire;
  Element fem;
  Element seks;
  Element sju;
  Element atte;
  Element ni;
  Element ti;
  en.value = 1.0;
  to.value = 2.0;
  tre.value = 3.0;
  fire.value = 4.0;
  fem.value = 5.0;
  seks.value = 6.0;
  sju.value = 7.0;
  atte.value = 8.0;
  ni.value = 9.0;
  ti.value = 10.0;

  heapInsert(heap,&en);
  printf("Inserted en\n");
  temp = heapExtractMin(heap);
  printf("extracted en with value %f\n",temp->value);

  heapInsert(heap,&fem);
  heapInsert(heap,&fire);
  heapInsert(heap,&tre);
  heapInsert(heap,&to);
  temp = heapExtractMin(heap);
  printf("extracted en with value %f\n",temp->value);

  temp = heapExtractMin(heap);
  printf("extracted en with value %f\n",temp->value);

  temp = heapExtractMin(heap);
  printf("extracted en with value %f\n",temp->value);

  temp = heapExtractMin(heap);
  printf("extracted en with value %f\n",temp->value);

  for(i=0;i<500;i++){
    heapInsert(heap,&ni);
  }
}
```

## C.2 Time

### C.2.1 time.h

```
int getTimeInMicroseconds(void);
int getTimeInSeconds(void);
/*
 * Check time in sec and usec so that they are correct
 */
void fixTime(int sec, int usec, int sec2, int usec2,int *rsec, int *
    rusec);
```

### C.2.2 time.c

```
#include <sys/time.h>

int getTimeInMicroseconds(void)
{
        struct timeval tv;
        struct timezone tz;
        gettimeofday(&tv,&tz);
        return tv.tv_usec;
}

int getTimeInSeconds(void)
{
        struct timeval tv;
        struct timezone tz;
        gettimeofday(&tv,&tz);
        return tv.tv_sec;
}
/*
 * Change sec and usec so they are correct
 */
void fixTime(int sec, int usec, int sec2, int usec2,int *rsec, int *
    rusec){
  *rsec = sec2-sec;
  if(usec2-usec < 0){
    *rsec--;
    *rusec = (usec2-usec) + 1000000;
  }else{
    *rusec = usec2-usec;
  }
}
```

## C.3 Fileio

### C.3.1 fileio.h

```c
/*
 * stores an array in a text file
 */
void printdatafile(float* array, int x, int y, int z);
/*
 * Reads a binary float file into an array
 */
void readfile(float* array,char* file, int x, int y, int z);
/*
 * saves a float array as a png image
 */
void printFloatImage(float* array, char* file, int sizex, int sizey,
    int z);
/*
 * Reads float array from a text file
 */
void readTextFile(float * array, char* file, int x, int y, int z);
```

### C.3.2 fileio.c

```
//#include <gd.h>
#include "/opt/freeware/include/gd.h"
#include <stdio.h>
#include <stdlib.h>
#include "array_mpi.h"
#include <string.h>
#define MAXFLOAT ((float)3.40282347e+38)
#define DEBUG
/*
 * stores an array in a text file
 */

void printdatafile(float* array, int x, int y, int z){
  FILE* f;
  int i, j ,k;
  f = fopen("out.data", "w");
  if(!f){
    printf("Error␣opening␣file␣out.data␣");
    return;
    }

  for(k=0;k<z;k++){
    for(j=0;j<y;j++){
      for(i=0;i<x;i++){
        fprintf(f,"%d.0␣%d.0␣%d.0␣%lf␣\n",i,j,k,array[GETINDEX(i,j,k)]);
      }
      fprintf(f,"\n");
    }
  fprintf(f,"\n");
  }
  fclose(f);
}
/*
 * Reads a binary float file into an array
 */
void readfile(float * array, char* file ,int x, int y, int z){
  FILE* f;
  int read = 0;
  int i,j,k;
  f = fopen(file ,"rb");
  printf("x␣=␣%d,␣y=␣%d,␣z=%d\n",x,y,z);
  fflush(stdout);
  bzero(array,sizeof(float)*x+2*y+2*z+2);
  printf("done␣zeroing␣array,␣starting␣read\n");
  fflush(stdout);
  for(i=0;i<x;i++)
    for(j=0;j<y;j++){
      read = fread(&array[GETINDEX(i,j,0)],sizeof(float),z,f);
      if(read != z){
        printf("Coudn't␣read␣hole␣file␣exiting,␣read␣%d\n",read);
        fclose(f);
        exit(1);
      }
  }
  printf("done␣read,␣closing␣file\n");
  fflush(stdout);
  fclose(f);
}

/*
 * Reads float array from a text file
 */
void readTextFile(float* array, char* file, int x, int y, int z){
  FILE *f;
  float temp;
  char ctemp = 'h';
  int i,j,t;
  f = fopen(file,"r");
  if(!f){
    printf("Error opening file %s ",file);
    return;
  }
  bzero(array,sizeof(float)*ARRAYSIZE);
  memset(array,1,sizeof(float)*ARRAYSIZE);
  fread(&ctemp,sizeof(char),1,f);
  for(i=0;i<2;i++){
    while(ctemp != '\n'){
      printf(".");
      fread(&ctemp,sizeof(char),1,f);
    }
    printf("\n");
  }
//  for(t=0;t<y*x;t++){
  t=0;
  while(fscanf(f,"%d %d %f\n",&i,&j,&temp) != EOF){
  array[GETINDEX(i-1,j-1,z)] = temp;
  t++;
  }
  printf("read %d values\n",t);
  fclose(f);
}

/*
 * saves a float array as a png image
 */
void printFloatImage(float* array, char* file, int sizex, int sizey,
    int z){
  /* Declare the image */
  gdImagePtr im;
  /* Declare output files */
  FILE *pngout;
  #ifdef DEBUG
  printf("printing image to file \n");
  #endif
  int maxt, mint;
  float max = 0,min = MAXFLOAT;
  int i,j,r,g,b,t;
    for(i=0;i<sizex;i++){
            for(j=0;j<sizey;j++){
    if(array[GETINDEX(i,j,z)] < min && array[GETINDEX(i,j,z)] !=
        BIGFLOAT){
      min = array[GETINDEX(i,j,z)];
      printf("new min value at %d %d %d value is %f\n",i,j,z,min);
    } else if(array[GETINDEX(i,j,z)] > max&& array[GETINDEX(i,j,z)] != BIGFLOAT){
      max = array[GETINDEX(i,j,z)];
    }
    }
  }

  for(i=0;i<sizex;i++){
          for(j=0;j<sizey;j++){
      if(array[GETINDEX(i,j,z)] == BIGFLOAT|| array[GETINDEX(i,j,z)] == BIGFLOAT*2){
      array[GETINDEX(i,j,z)] = max;
    }
    }
  }
  #ifdef DEBUG
  printf("Maxvalue is %f, min value is %f\n",max,min);
  #endif
  /* Allocate the image: 64 pixels across by 64 pixels tall */
  im = gdImageCreate(sizex, sizey);

  /* Allocate the color black (red, green and blue all minimum).
    Since this is the first color in a new image, it will
    be the background color. */
    maxt = 0;
    mint = 9999999;
  for(i=0;i<sizex;i++){
  for(j=0;j<sizey;j++){
  //printf("color is %d\n",(int) (((float)array[GETINDEX(i,j,z)] / (float)(max-min)) *255));

  t = (int) ((((float)array[GETINDEX(i,j,z)]-min) / (float)(max-min)) *(255*4));
  if(maxt < t)
    maxt = t;
  if(mint > t)
    mint = t;
  //printf("%d\n",t);
  /* red- yellow -green - cyan - blue*/
  /*if(t<255){
    r= 255;
    g= t;
    b=0;
  } else if(t<255*2){
    r= 255*2 -t;
    g = 255;
    b = 0;
  } else if(t<255*3){
    r = 0;
    g = 255;
    b = t - 255*2;
  } else{
    r = 0;
    g = 255*4 - t;
    b = 255;
  }*//* blue - cyan - green - yeallow - red*/
  t = 255*4 -t;
  if(t<255){
      r= 255;
      g= t;
      b=0;
    } else if(t<255*2){
      r= 255*2 -t;
      g = 255;
      b = 0;
    } else if(t<255*3){
      r = 0;
      g = 255;
      b = t - 255*2;
    } else{
      r = 0;
      g = 255*4 - t;
      b = 255;
    }
      gdImageSetPixel(im, i, j, gdImageColorResolve(im, r/*red*/,g/*
          green*/, b/*blue*/ ));
      }
    }
    #ifdef DEBUG
    printf("Maxvalue is %d, min value is %d\n",maxt,mint);
    #endif

    /* Open a file for writing. "wb" means "write binary", important
      under MSDOS, harmless under Unix. */
    pngout = fopen(file, "wb");

    /* Output the image to the disk file in PNG format. */
    gdImagePng(im, pngout);

    /* Close the files. */
    fclose(pngout);

    /* Destroy the image in memory. */
    gdImageDestroy(im);
}
```

## C.4 Convert

### C.4.1 convert.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  FILE* filer;
  FILE* filew;
  int read, write,i;
  float data;
  filer = fopen(argv[1],"rb");
  filew = fopen(argv[2],"wb");
  char* cdata;

  read = fread(&data,sizeof(float),1,filer);
    if(read!= 1){
      printf("Read failed\n");
    }

  while(read){
//    for(i=0;i<1000;i++){
    cdata = &data;
    write = fwrite(&cdata[3],sizeof(char),1,filew);
    if(write != 1){
      printf("Coudn't write\n");
    }
    write = fwrite(&cdata[2],sizeof(char),1,filew);
    if(write != 1){
      printf("Coudn't write\n");
    }
    write = fwrite(&cdata[1],sizeof(char),1,filew);
    if(write != 1){
      printf("Coudn't write\n");
    }
    write = fwrite(&cdata[0],sizeof(char),1,filew);
    if(write != 1){
      printf("Coudn't write\n");
    }
    read = fread(&data,sizeof(float),1,filer);
    if(read!= 1){
      printf("Read failed\n");
    }
  }
  fclose(filer);
  fclose(filew);
}
```