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.

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 Schlubeger for providing me with the motivation for this assignment, and for helping me get started and understanding the problem.
Contents

List of Figures xiii
List of Tables xv

1 Introduction 1
  1.1 Contributions ........................................... 2
  1.2 Outline .................................................. 2

2 Background Material 3
  2.1 Seismic data ............................................. 3
  2.2 Seismic event tracking by global path optimization ....... 4
  2.3 Cluster super computers .................................. 4
  2.4 Njord ..................................................... 5
  2.5 Manhattan distance ..................................... 6
  2.6 Fast Marching Method (FMM) ............................ 6
    2.6.1 FMM Initial data .................................... 7
    2.6.2 FMM main loop ...................................... 8
  2.7 Spherical vs Cartesian coordinates ........................ 8
  2.8 Parallel Fast Marching Method .......................... 10
    2.8.1 Herrmann’s Parallel algorithm 1 .................... 11
    2.8.2 Herrmann’s Parallel algorithm 2 .................... 11
    2.8.3 Herrmann’s Parallel algorithm 3 .................... 11
    2.8.4 Domain decomposition ................................ 13
    2.8.5 Herrmann’s results .................................. 13

3 Parallel implementation of FMM 15
  3.1 Fast Marching Method .................................... 15
    3.1.1 Choosing domain decomposition ....................... 15
    3.1.2 Rectangular 1:2 scale decomposition ................. 16
    3.1.3 Choosing a parallel FMM .............................. 17
    3.1.4 Choosing Spherical vs Cartesian coordinates ........ 17
  3.2 HFMM ...................................................... 17
    3.2.1 Parallel Fast Marching Method ....................... 17
    3.2.2 Initial data ........................................... 18
    3.2.3 Narrowband ........................................... 19
    3.2.4 Calculation of arrival times ......................... 19
    3.2.5 Exchanging border values ............................ 19
  3.3 PFMM ...................................................... 20
# CONTENTS

4  Modeling of the FMM algorithm ........................................... 23  
   4.1  HFMM ................................................................. 23  
   4.2  PFMM ................................................................. 25  
   4.3  Validation of PFMM .................................................. 26  
   4.4  Validation of HFMM .................................................. 29  

5  Performance analysis ...................................................... 31  
   5.1  HFMM ................................................................. 31  
      5.1.1  Analysis of HFMM .............................................. 34  
   5.2  PFMM ................................................................. 34  
      5.2.1  Analysis of PFMM .............................................. 40  
   5.3  Comparing HFMM and PFMM .......................................... 41  

6  Conclusion ................................................................. 43  
   6.1  Future work .......................................................... 44  

Bibliography ................................................................. 47  

A  Application 1 source code ................................................ 49  
   A.1  Array ................................................................. 49  
      A.1.1  array_mpi.h .................................................. 49  
      A.1.2  array_mpi.c .................................................. 51  
   A.2  Fast Marching Method .............................................. 53  
      A.2.1  fmm_mpi.h .................................................... 53  
      A.2.2  fmm_mpi.c .................................................... 54  
   A.3  Application .......................................................... 69  
      A.3.1  mpi_app.c .................................................... 69  

B  Application 2 source code ................................................ 77  
   B.1  Array ................................................................. 77  
      B.1.1  array_mpi.h .................................................. 77  
      B.1.2  array_mpi.c .................................................. 80  
   B.2  Fast Marching Method .............................................. 85  
      B.2.1  fmm_mpi.h .................................................... 85  
      B.2.2  fmm_mpi.c .................................................... 86  
   B.3  Application .......................................................... 101  
      B.3.1  mpi_app.c .................................................... 101  

C  Common files .............................................................. 109  
   C.1  Heap ................................................................. 109  
      C.1.1  heap.h ....................................................... 109  
      C.1.2  heap.c ....................................................... 111  
      C.1.3  testheap.c ................................................... 114  
   C.2  Time ................................................................. 115  
      C.2.1  time.h ....................................................... 115  
      C.2.2  time.c ....................................................... 116  
   C.3  Fileio ............................................................... 117  
      C.3.1  fileio.h ...................................................... 117  
      C.3.2  fileio.c ...................................................... 118
C.4 Convert .......................................................... 122
C.4.1 convert.c .................................................. 122
List of Figures

2.1 Border cells ................................................. 5
2.2 Manhattan distance ........................................ 6
2.3 Narrowband .................................................. 7
2.4 Serial FMM algorithm ....................................... 8
2.5 Traveltime array step n .................................... 8
2.6 Traveltime array step n+1 ................................. 9
2.7 Cartesian coordinate system. Figure from [1] .......... 9
2.8 Parallel Fast Marching Method 1. ........................ 11
2.9 Parallel Fast Marching Method 2. ........................ 12
2.10 Parallel Fast Marching Method 3. ...................... 12
2.11 Non optimal domain decomposition .................... 13
2.12 Quadratic grid decomposition ............................ 14

3.1 Rectangular 1:2 scale decomposition .................... 16
3.2 Parallel Fast Marching Method, My version ........... 18
3.3 Parallel Fast Marching Method revised ................ 20

5.1 Speedup from HFMM ........................................ 32
5.2 Speedup for HFMM with and without rollback ........ 33
5.3 Speedup for PFMM .......................................... 35
5.4 Speedup for PFMM, 256 nodes .......................... 36
5.5 Speedup for PFMM, salt data ............................ 37
5.6 Speedup for PFMM compared to theoretical speedup . 38
5.7 PFMM theoretical vs measured time .................... 39
5.8 320x320x320 matrix comparing HFMM and PFMM .. 41

6.1 Manhattan distance for pick 1 .......................... 44
6.2 Manhattan distance for pick 2 .......................... 44
6.3 Manhattan distance for both picks, bold where they collide .. 45
List of Tables

3.1 Optimal memory usage and my applications memory usage . . 19
5.1 Theoretical and measured runtime for PFMM . . . . . . . . . . 39
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) [1] 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.
CHAPTER 1. INTRODUCTION

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.

![Figure 2.1: Border cells](image)

### 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. Because 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 process. 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 it's 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](image)

2.6 Fast Marching Method (FMM)

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

\[ |\Delta u(x)| = F(x), x \in \Omega \]  

(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].
[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.

Where

\[
D_{ijk}^{-x}G = \frac{G_{ijk} - G_{i-1,jk}}{\Delta x}, D_{ijk}^{+x}G = \frac{G_{i+1,jk} - G_{ijk}}{\Delta x} \\
D_{ijk}^{-y}G = \frac{G_{ijk} - G_{ij-1,k}}{\Delta y}, D_{ijk}^{+y}G = \frac{G_{ij+1,k} - G_{ijk}}{\Delta y} \\
D_{ijk}^{-z}G = \frac{G_{ijk} - G_{ijk-1}}{\Delta z}, D_{ijk}^{+z}G = \frac{G_{ijk+1} - G_{ijk}}{\Delta z}
\]

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 -1 -1
-1 -1 -1 -1 2 -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
```

Figure 2.5: Traveltime array step n

An illustration 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 \textit{xyz} and Spherical \textit{rθφ}. Where \textit{r} is the length from origin and \textit{θ}
Figure 2.6: Traveltime array step n+1

is the angle between the x axis and the r line and $\phi$ is the angle 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.

\[
\begin{align*}
t_{ijk} &= t_{i-1jk} + \frac{\Delta x}{v}, t_{ijk} = t_{i+1jk} + \frac{\Delta x}{v} \\
t_{ijk} &= t_{ij-1k} + \frac{\Delta y}{v}, t_{ijk} = t_{ij+1k} + \frac{\Delta y}{v} \\
t_{ijk} &= t_{ijk-1} + \frac{\Delta z}{v}, t_{ijk} = t_{ijk+1} + \frac{\Delta z}{v}
\end{align*}
\]  
(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.
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 B's 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.

\[
\begin{align*}
\max(D_{ijk}^r t, -D_{ijk}^r t, 0)^2 + \\
\max(D_{ijk}^\theta t, -D_{ijk}^\theta t, 0)^2 + \\
\max(D_{ijk}^\phi t, -D_{ijk}^\phi t, 0)^2 \right)_{1/2} = 1
\end{align*}
\]

where

\[
\begin{align*}
D_{ijk}^r t &= \frac{t_{i,j,k} - t_{i-1,j,k}}{\Delta r}, D_{ijk}^r t = \frac{t_{i+1,j,k} - t_{i,j,k}}{\Delta r} \quad (2.6) \\
D_{ijk}^\theta t &= \frac{t_{i,j,k} - t_{i,j-1,k}}{r\Delta \theta}, D_{ijk}^\theta t = \frac{t_{i,j+1,k} - t_{i,j,k}}{r\Delta \theta} \quad (2.7) \\
D_{ijk}^\phi t &= \frac{t_{i,j,k} - t_{i,j-1,k}}{r\sin \theta \Delta \phi}, D_{ijk}^\phi t = \frac{t_{i,j+1,k} - t_{i,j,k}}{r\sin \theta \Delta \phi} \quad (2.8)
\end{align*}
\]

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 straightforward 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 inherent 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.

<table>
<thead>
<tr>
<th>1 1</th>
<th>2 2</th>
<th>3 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>2 2</td>
<td>3 3</td>
</tr>
<tr>
<td>4 4</td>
<td>5 5</td>
<td>6 6</td>
</tr>
<tr>
<td>4 4</td>
<td>5 5</td>
<td>6 6</td>
</tr>
<tr>
<td>7 7</td>
<td>8 8</td>
<td>9 9</td>
</tr>
<tr>
<td>7 7</td>
<td>8 8</td>
<td>9 9</td>
</tr>
</tbody>
</table>

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
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 based on Herrmann’s 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.
\[ w \times C \times 2 + h \times C \times 2 = TC \]  

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

![Table](image)

In Figure 3.1 each node has a 1:2 rectangular data area, and the node also maps 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 narrow-
       band
   (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 that sets your memory requirements.

The velocity array is floating point numbers, saved as a float array in my appli-
cation.

The second array you need is arrival times, where you 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 nar-
rowband 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.
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 sets the problem size.

The total memory usage will be according to Table 3.2.2 on my application $(\text{arraysize}^4)^5 + \text{heap filldata}$ and optimal memory usage will be $(\text{arraysize}^4)^2 + 1/4 + \text{heap filldata count}^4 + \text{filldata}$. 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 straightforward 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 it’s 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 roll-
backs will occur. I will now try to explain how one can calculate those aspects
and come up with an equation for overall execution time.

\[
\text{arraysize} = n \cdot m \cdot k \quad (4.1)
\]
\[
\text{bordersize} = n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2 \quad (4.2)
\]

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 an-
other 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 \( T_{\text{traverse}} \). \( T_{\text{traverse}} \) is the time it takes to go through the
matrix and set the band array back to a previous state. \( T_{\text{traverse}} \) 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.

\[
T_{traverse} = \text{arraysize} \cdot mtime \\
T_{traverse} = n \cdot m \cdot k \cdot mtime
\]

\[
T_{rollback} = \#\text{rollback} \cdot T_{traverse} \\
\#\text{rollback} = P(\text{rollback}) \cdot \left(bordersize + \text{extraBorderSends}\right)
\]

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 \( \text{calcn} \), 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.

\[
T_{compute} = T_{loop} \cdot \text{arraysize} \\
T_{loop} = \#\text{CalcNeighbours} \cdot \text{calcn} + \log(\text{sizeNarrowband})
\]

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 \( 2.143 \cdot 10^{-10} \).

\[
T_{comm} = (bordersize + \text{resendCount}) \cdot 20 \cdot \beta + \alpha \\
T_{comm} = ((n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) + \text{resendCount}) \cdot 20 \cdot \beta + \alpha
\]
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 $T_{\text{wavetime}}$.

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

$$T_{\text{node}} = T_{\text{wavetime}} + T_{\text{compute}} + T_{\text{comm}} + T_{\text{rollback}}$$ (4.7)

The overall time it takes to finish the HFMM is given my $T_{\text{overall}}$. 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 $T_{\text{communicate}}$. Mostly because it takes too little time to have any affect. $T_{\text{overall}}$ is shown in Equation 4.8. P is max number of processors.

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

$T_{\text{compute}}$ is the exact same as Equation 4.5 used in HFMM. That means that also $T_{\text{loop}}$ 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 $T_{\text{comm}}$ for HFMM. This is because in PFMM only the arrival time and band array are exchanged.

$$T_{\text{comm}} = \text{bordersize} \cdot 8 \cdot \beta + \alpha$$ (4.9)

$$T_{\text{comm}} = (n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) \cdot 8 \cdot \beta + \alpha$$
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 $T_{\text{rollbackcheck}}$ is given by Equation 4.10.

$$T_{\text{rollbackcheck}} = T_{\text{loop}} \cdot \text{bordersize} \quad (4.10)$$

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 $T_{\text{node}}$ 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.

$$T_{\text{rollback}} = T_{\text{node}} \cdot (\text{MAX}(#\text{rollbacks}0, #\text{rollbacks}1, \ldots, #\text{rollbacks}P - 1) - \text{MaxM + M}) \quad (4.11)$$

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 $T_{\text{node}}$ in HFMM. The main reason for $T_{\text{node}}$ time is the array size.

$$T_{\text{node}} = T_{\text{compute}} + T_{\text{comm}} + T_{\text{rollbackcheck}} \quad (4.12)$$

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

$$T_{\text{overall}} = T_{\text{node}} \cdot \text{ManhattanDistance} + T_{\text{rollback}} \quad (4.13)$$

### 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 calculations the 160x160x160 matrix was used. Its runtime for the serial code was 8.73 seconds.
\[ n = m = k = 160 \]

\[ T_{\text{overall}} = T_{\text{node}} \cdot \text{ManhattanDistance} + T_{\text{rollback}} \]

\[ T_{\text{overall}} = 8.73 \]

\[ \text{ManhattanDistance} = 1 \]

\[ T_{\text{rollback}} = 0 \]

\[ 8.73 = T_{\text{node}} \cdot 1 + 0 \]

When running only one node, the Manhattan distance will be 1. \( T_{\text{rollback}} \) is set to 0 as there is no need for a roll back on the serial version.

\[ T_{\text{node}} = T_{\text{compute}} + T_{\text{comm}} + T_{\text{rollbackcheck}} \]

\[ T_{\text{comm}} = 0 \]

\[ T_{\text{rollbackcheck}} = T_{\text{loop}} \cdot \text{bordersize} \]

\[ T_{\text{rollbackcheck}} = T_{\text{loop}} \cdot 160^2 \cdot 6 \]

\[ T_{\text{compute}} = T_{\text{loop}} \cdot \text{arraysize} \]

\[ T_{\text{compute}} = T_{\text{loop}} \cdot 160^3 \]

\[ T_{\text{node}} = T_{\text{loop}} \cdot 160^3 + 0 + T_{\text{loop}} \cdot 160^2 \cdot 6 \]

\( T_{\text{comm}} \) 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 \( T_{\text{node}} \) is only dependent on \( T_{\text{loop}} \)

\[ 8.73 = T_{\text{loop}} \cdot (160^3 + 160^2 \cdot 6) \]

\[ T_{\text{loop}} = 2.054311 \cdot 10^{-06} \]

The result can be seen above. This is the calculated time for \( T_{\text{loop}} \). The accurate value will vary depending on where in the matrix \( T_{\text{loop}} \) is executed, but this result is an average over all points. If the program ran again, \( T_{\text{loop}} \) should become the same. In a parallel version there will be more \( T_{\text{loop}} \) at the borders. Those are smaller as there is only one value that needs updating, but it shouldn’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 \( T_{\text{loop}} \) 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.
\[ T_{\text{loop}} = 2.054311 \cdot 10^{-06} \]
\[ T_{\text{rollback check}} = T_{\text{loop}} \cdot (n \cdot m \cdot 2 + n \cdot k \cdot 2 + m \cdot k \cdot 2) \]
\[ T_{\text{rollback check}} = 2.054311 \cdot 10^{-06} \cdot (80^2 \cdot 6) \]
\[ T_{\text{rollback check}} = 0.078885542 \]

For a 8 node configuration \( T_{\text{rollback check}} \) was 0.078885542 seconds.

\[ T_{\text{comm}} = \text{bordersize} \cdot 8 \cdot \beta + \alpha \]
\[ T_{\text{comm}} = 80^2 \cdot 6 \cdot 8 \cdot 2.143 \cdot 10^{-10} + 1.995 \cdot 10^{-6} \]
\[ T_{\text{comm}} = 0.001336609 \]

The communication time for Njord is very low. \( T_{\text{comm}} \) is very low, only 0.001336609 seconds.

\[ T_{\text{compute}} = T_{\text{loop}} \cdot \text{arraysize} \]
\[ T_{\text{compute}} = 2.054311 \cdot 10^{-06} \cdot 80^3 \]
\[ T_{\text{compute}} = 1.0518072 \]

\( T_{\text{compute}} \) 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.

\[ T_{\text{node}} = T_{\text{compute}} + T_{\text{comm}} + T_{\text{rollback check}} \]
\[ T_{\text{node}} = 1.0518072 + 0.001336609 + 0.078885542 \]
\[ T_{\text{node}} = 1.1320294 \]

A node will complete in 1.1320294 seconds.

\[ \text{ManhattanDistance} = 4 \]
\[ T_{\text{rollback}} = 0 \]
\[ T_{\text{overall}} = T_{\text{node}} \cdot \text{ManhattanDistance} + T_{\text{rollback}} \]
\[ T_{\text{overall}} = 1.1320294 \cdot 4 \]
\[ T_{\text{overall}} = 4.5281175 \]

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

\( T_{\text{overall}} \) was calculated to 4.52 seconds the measured time was 4.11 seconds. This means that \( T_{\text{loop}} \) 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.
CHAPTER 4. MODELING OF THE FMM ALGORITHM

4.4 Validation of HFMM

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

\[
T_{\text{loop}} = 2.054311 \cdot 10^{-06}
\]
\[
T_{\text{compute}} = T_{\text{loop}} \cdot \text{arraysize}
\]
\[
T_{\text{compute}} = 2.054311 \cdot 10^{-06} \cdot 80^3
\]
\[
T_{\text{compute}} = 1.0518072
\]

\(T_{\text{compute}}\) is 1.0518072 seconds. The same as in PFMM.

\[
T_{\text{comm}} = (\text{bordersize} + \text{resendCount}) \cdot 20 \cdot \beta + \alpha
\]
\[
T_{\text{comm}} = (80^2 \cdot 6 + \text{resendCount}) \cdot 20 \cdot 2.143 \cdot 10^{-10} + 1.995 \cdot 10^{-6}
\]
\[
T_{\text{comm}} = 0.001645824 + 4.286 \cdot 10^{-9} \cdot \text{resendCount} + 1.995 \cdot 10^{-6}
\]

\(T_{\text{comm}}\) is dependent on the resendCount, this value is difficult to predict before the application starts.

\[
T_{\text{traverse}} = \text{arraysize} \cdot \text{mtime}
\]
\[
T_{\text{rollback}} = \#\text{rollback} \cdot T_{\text{traverse}}
\]
\[
T_{\text{rollback}} = \#\text{rollback} \cdot 80^3 \cdot \text{mtime}
\]

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.

\[
T_{\text{node}} = T_{\text{wavetime}} + T_{\text{compute}} + T_{\text{comm}} + T_{\text{rollback}}
\]
\[
T_{\text{node}} = T_{\text{wavetime}} + 1.0518072 + 0.001645824 + 4.286 \cdot 10^{-9} \cdot \text{resendCount} + 1.995 \cdot 10^{-6} + T_{\text{rollback}}
\]
\[
T_{\text{node}} = 1.053455 + T_{\text{wavetime}} + T_{\text{rollback}} + 4.286 \cdot 10^{-9} \cdot \text{resendCount}
\]
\[
T_{\text{node}} = 1.053455 + T_{\text{wavetime}} + T_{\text{rollback}}
\]

This is as close to an answer to \(T_{\text{node}}\) as one can. The resend count is remove because Njord has such good interconnect that it will be negligible. The measured time for this run is 9.9 seconds. Each node uses 1.053455. This means that \(T_{\text{wavetime}}\) and \(T_{\text{rollback}}\) 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 \(T_{\text{wavetime}}\) is small. Therefore, \(T_{\text{rollback}}\) 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 Herrmann’s 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.

\[ \frac{M}{n} = speedup \]  

(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
Table 5.1: Theoretical and measured runtime for PFMM

<table>
<thead>
<tr>
<th># nodes</th>
<th>Calculated</th>
<th>Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8.7296</td>
<td>8.79</td>
</tr>
<tr>
<td>4</td>
<td>6.6669</td>
<td>6.31</td>
</tr>
<tr>
<td>8</td>
<td>4.5232</td>
<td>4.11</td>
</tr>
<tr>
<td>16</td>
<td>2.8226</td>
<td>2.4</td>
</tr>
</tbody>
</table>

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, because its so low that it won’t affect the results in a significant way, also we assume no rollbacks.

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 Analysis 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, but 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 problem 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

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.

Figure 5.8: 320x320x320 matrix comparing HFMM and PFMM
Chapter 6

Conclusion

This thesis has focused on developing a parallel method for the Fast Marching 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 asynchronous 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 synchronous communication was tested, this resulted in too much overhead in communication and performed worse than the asynchronous version.

PFMM had much better performance. The theoretical 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 theoretical 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, asynchronous 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 both 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.

![Figure 6.1: Manhattan distance for pick 1](image)

![Figure 6.2: Manhattan distance for pick 2](image)

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
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 achieved in 2-3 iterations, because 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.


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 // 2 // 448
#define SIZEY 160 // 704 // 4 // 704
#define SIZEZ 160 // 1216 // 4 // 1216

/* Calculate the strides for global array */
#define stride0 ((SIZEX+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 allocating memory */
```

/* The size of each array, for allocating memory */
```c
#define ARRSIZE (SIZEX+2)*(SIZEY+2)*(SIZEZ+2)
#define LOCALARRYSIZE (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

```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;
  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[GETINDEX(i,j,k)]);
  printf("\n");
}
```

APPENDIX A. APPLICATION 1 SOURCE CODE 51
```c
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");
    }
}

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

```c
#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 containing 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 position with band information
 */
struct me {
    float value;
    int x;
    int y;
    int z;
    int n;
} MPI_Element;

/*
 * initialize 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

```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[GETINDEX(px, py, pz)];
```
temp->x = px;
55 temp->y = py;
56 temp->z = pz;
57 heapInsert(data->heap, temp);
58 #ifdef DEBUG
59 printf("%d inserted %d, %d to heap, heap size is %d\n",
60 cartrank, px, py, pz, data->heap->heapsize);
61 #endif
62 }
63
64 /* initialize the FMM
65 * velarray is the velocity field
66 * posx, posy, posz is position of the starting point
67 * x, y, z is the size of the array, most are read from array_mpi.h
68 */
70 FmmData* initFMM(float* velarray, int posx, int posy, int posz, int x,
71 int y, int z) {
72 working = malloc(sizeof(int) * size);
73 /* init datatypes*/
74 MPI_Element e;
75 MPI_Datatype type[5] = { MPI_FLOAT, MPI_INT, MPI_INT, MPI_INT, MPI_INT,}
76 int blocklen[5] = { 1, 1, 1, 1, 1};
77 MPI_Aint disp[5];
78 disp[0] = 0;
79 disp[1] = sizeof(float);
80 disp[2] = sizeof(int) + disp[1];
81 disp[3] = sizeof(int) + disp[2];
82 disp[4] = sizeof(int) + disp[3];
83 MPI_Type_create_struct(5, blocklen, disp, type, &
84 mpi_element_struct);
85 MPI_Type_commit(& mpi_element_struct);
86 /* end init datatypes*/
87 #ifdef DEBUG
88 printf("%d done init mpi datatypes\n", cartrank);
89 #endif
90 int send = 0;
91 FmmData* data;
92 data = malloc(sizeof(FmmData));
93 if (data == 0) {
94 printf("%d Failed to allocate memory, exiting\n", cartrank);
95 exit(1);
96 }
97 data->bandarray = malloc(sizeof(int)*LOCALARRAYSIZE);
98 if (data->bandarray == 0) {
99 printf("%d Failed to allocate memory, exiting\n", cartrank);
100 exit(1);
101 }
102 data->timearray = malloc(sizeof(float)*LOCALARRAYSIZE);
103 if (data->timearray == 0) {
104 printf("%d Failed to allocate memory, exiting\n", cartrank);
105 exit(1);
106 }
107 data->velocityarray = velarray;
```c
108     #ifdef DEBUG
109     printf("%d: initializing heap\n", cartrank);
110 #endif
111     data->heap = initHeap(LOCALARRAYSIZE);
112     data->x = x;
113     data->y = y;
114     data->z = z;
115     data->posx = posx;
116     data->posy = posy;
117     data->posz = posz;
118
119     #ifdef DEBUG
120     printf("%d: clearing memory bandarray %p size %d\n", cartrank, data->bandarray, sizeof(int)*LOCALARRAYSIZE);
121     fflush(stdout);
122 #endif
123     /*
124     * setting the band array to outside
125     */
126     memset(data->bandarray, OUTSIDE, sizeof(int)*LOCALARRAYSIZE);
127     #ifdef DEBUG
128     printf("%d: Heap ok, clearing memory timearray %p size %d\n", cartrank, data->timearray, sizeof(float)*LOCALARRAYSIZE);
129 #endif
130     /*
131     * zeroing the arrival time array
132     */
133     int i = 0;
134     for(i=0;i<LOCALARRAYSIZE;i++)
135     data->timearray[i] = 0;
136     //memset(data->timearray, 0, sizeof(float)*LOCALARRAYSIZE);
137     //bzero(data->timearray, sizeof(float)*LOCALARRAYSIZE);
138     data->sentarray = malloc(sizeof(float)*LOCALARRAYSIZE);
139     bzero(data->sentarray, sizeof(float)*LOCALARRAYSIZE);
140     #ifdef DEBUG
141     printf("%d: done allocating memory, setting starting point\n", cartrank);
142     fflush(stdout);
143 #endif
144     /*
145     * inserting the starting point on the correct node and add it to the heap / narrow band
146     */
147     if(getDest(posx, posy, posz) == cartrank){
148         int *cr = getLocalCord(posx, posy, posz);
149         // insert starting point
150         data->timearray[GETLINDEX(cr[0], cr[1], cr[2])] = 0;
151         data->bandarray[GETINDEX(cr[0], cr[1], cr[2])] = BAND;
152         // printf("%d: sat pos %d %d %d, as known\n", cartrank, cr[0], cr[1], cr[2]);
153         Element* element;
154         element = malloc(sizeof(Element));
155         element->x = cr[0];
156         element->y = cr[1];
157         element->z = cr[2];
158         element->value = data->timearray[GETLINDEX(cr[0], cr[1], cr[2])];
159         heapInsert(data->heap, element);
160         send = 1;
```
return data;

float min(float per, float truls)
{
    if(per > truls)
        return truls;
    else return per;
}

float max(float per, float truls)
{
    if(per > truls)
        return per;
    else return truls;
}

void printFloatArray(int sizex, int sizey, int z, float* array)
{
    int i, j;
    printf("\nPrinting matrix\n");
    for(i=0; i<sizex; i++)
        for(j=0; j<sizey; j++)
            printf("%8.2f", array[GETLINDEX(i, j, z)]);
    printf("\n");
}

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 */
            */
        }
APPENDIX A. APPLICATION 1 SOURCE CODE

255 if (data->timearray[GETLINDEX(px, py, pz)] != 0 && data
256   ->timearray[GETLINDEX(px, py, pz)] <= sol) {
257     return;
258 }
259 /* If the point is OUTSIDE add it to the heap/
260 narrowband */
261 if (data->bandarray[GETLINDEX(px, py, pz)] == OUTSIDE) {
262     data->bandarray[GETLINDEX(px, py, pz)] = BAND;
263     add = 1;
264 }
265 /* If the point is OUTSIDE add it to the heap */
266 #ifdef DEBUG
267 printf("%d: sol is out of range s\n", cartrank, sol, px, py, pz);
268 #endif
269 /* store max and min for debug purposes */
270 #ifdef DEBUG
271 if (sol > valuemax && sol != BIGFLOAT) {
272     valuemax = sol;
273 } else if (sol < valuemin) {
274     valuemin = sol;
275 #endif
276 /* set the new arrival time */
277 data->timearray[GETLINDEX(px, py, pz)] = sol;
278 #ifdef DEBUG
279 printf("%d: x%d, y%d, z%d, \n", cartrank, data->x, data->y, data->z, px, py, pz, data->timearray[GETLINDEX(px, py, pz)]) ;
280 #endif
281 /* add it to the narrowband if it should be added */
282 if (add) {
283     addToHeap(data, px, py, pz);
284 }
285 }
286 }
287 // remove ??
288 void checkforchange(FmmData* data, int px, int py, int pz) {
289     float sol = BIGFLOAT;
290     int i;
291     int o[6] = {px-1, px, px+1, px, px};
292     int l[6] = {py, py, py, py+1, py, py};
293     int m[6] = {pz, pz, pz, pz, pz};
294     for (i = 0; i < 6; i++) {
295         if (o[i] >= 0 && o[i] < data->x && l[i] < 0 && l[i] < 0 && m[i] >= 0 && m[i] < data->z && data->bandarray[GETLINDEX(o[i], l[i], m[i])] > BAND) {
296             #ifdef DEBUG
297             printf("%d: checking for rollback at (\n", cartrank, o[i], l[i], m[i]);
298             #endif
299             sol = calcDistance(data, px, py, pz);
300         }
301     }
302 }
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, 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, int 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: already 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]};
  /* 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]);
  }

  /*
APPENDIX A. APPLICATION 1 SOURCE CODE

* set rollback values, so we can check if a rollback is necessary
*/

/*if(largest_solution > e.value && rollbacksmallest > e.value)
rollbacksmallest = e.value;
rollbackn = e.n;
}*/

/* roll back 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;
#endif

if (snorth || ssouth || swest || seast || sabove || sbelow)
    cr = getGlobalCord(x, y, z);
#if define DEBUG
    printf("%d : sending element at %d %d %d gave global coord %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)];

rollbacksmallest = BIGFLOAT;
rollbackn = 0;

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 || east || 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(east){
    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 synchronous border exchange */
void sendRecvBorderChanges(FmmData* data, int x, int y, int z, int send){
  MPI_Element e;
  int *cr;
  int east,west,north,south,above,below;
  int seast,swest,snorth,ssouth,sabove,sbelow;
rest = 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);
  }
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 BMM */

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\nsizes\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("setting\n", cartrank, e->x, e->y, e->z);
          #endif
          data->bandarray[GETLINDEX(e->x, e->y, e->z)] = KNOWN;
        }
      } /* End of working loop */
      while (sendrun) {
        MPI_Send(&end, 1, MPI_INT, i, 9, gridcomm);
        if (end) {
          sendrun = 0;
          run = 0;
        }
      } /* End of send loop */
    } /* End of while(!end) */
  } /* End of while(run) */
}
```c
int k[6] = {\text{e} \to x - 1, e \to x, e \to x + 1, e \to x, e \to x, e \to x};
int l[6] = {e \to y, e \to y - 1, e \to y, e \to y + 1, e \to y, e \to y};
int m[6] = {e \to z, e \to z, e \to z, e \to z, e \to z - 1, e \to z + 1};
int i;
posx = e \to x;
posy = e \to y;
posz = e \to z;
/*
  update largest_solution if this solution is the largest
*/
if (data \to timearray[GETINDEX(posx, posy, posz)] >
largest_solution) {
largest_solution = data \to timearray[GETINDEX(posx, posy, posz)];
}
/* check if this point has been sent before */
if (data \to sentarray[GETINDEX(posx, posy, posz)] == 0) {
senddata = 1;
} else if (data \to sentarray[GETINDEX(posx, posy, posz)] <= data \to
timearray[GETINDEX(posx, posy, posz)]) {
senddata = 0;
}
/*
  printFloatArray ( data \to x, data \to y, \to z, data \to timearray);
  printFloatArray ( data \to x, data \to y, \to z, data \to timearray);
  printFloatArray ( data \to x, data \to y, \to z, data \to timearray);
  printFloatArray ( data \to x, data \to y, \to z, data \to timearray);
  printFloatArray ( data \to x, data \to y, \to z, data \to timearray);
  for (i=0; i<6; i++) {
    if (k[i] >= 0 && k[i] < data \to x && l[i] >= 0 && l[i] < data \to y
        && m[i] >= 0 && m[i] < data \to z) {
      calcElement(data, k[i], l[i], m[i]);
    }
  }
#endif
/*
  Send changes to border and look for incoming changes to the
  border */
sendRecvBorderChanges(data, posx, posy, posz, senddata);
#endif DEBUG
```
APPENDIX A. APPLICATION 1 SOURCE CODE

731 */Do we end the loop?*/
732 if(heapGetMin(data->heap) == 0){
733     run = 0;
734 } else{
735     run = 1;
736 }
737 checkOthers();
738 #ifdef DEBUG
739     if(carrank == 0 && n%1000 == 0){
740         printf("%d: reached n %d\n", carrank, n);
741     }
742 #endif
743 }
744 /* see if border changes are coming */
745 sendRecvBorderChanges(data,0,0,0,0);
746 /* if we have work to do let's notify others and start to work
again, if not let others know we are done */
747 if(heapGetMin(data->heap) != 0){
748     run = 1;
749     // notify others I am still working
750     notifyOthers(1);
751 } else if(working[carrank] == 1){
752     // notify that I have stopped working
753     notifyOthers(0);
754 }
755 checkOthers();
756 sum = 0;
757 for(i = 0; i<size;i++){
758     sum += working[i];
759 }
760 if(sum == 0){
761     end = 1;
762 }
763 }
764 #ifdef DEBUG
765     printf("valuemax %f, valuemin %f\n", valuemax, valuemin);
766 #endif
767 }
768 }
769 }
770 /*
771 * free the variables used in the FMM
772 */
773 void freeFMM(FmmData* data){
774     //free(data->timearray);
775     free(data->bandarray);
776     free(data);
777 }
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 should 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 doesn’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[GETINDEX(i, j, k)] != value)
                    rvalue = 0;
    return rvalue;
}

/* Check if a global array doesn’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 matrices 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%ld local_y%ld local_z%ld coords%d,%d,%d \n",
          cartrank , local_x , local_y , local_z , coords[0], coords[1], coords [2]);
}

/*
 * initialize data types 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);
}
```c
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("Couldn'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: opening 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(& farray[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(& farray[GETINDEX(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 (cartrank != 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/receiving north south
    MPI_Send(&local_array[GETINDEX(0,0,0)], 1, zy_plane, north, 0, gridcomm);
    MPI_Recv(&local_array[GETINDEX(local_x,0,0)], 1, zy_plane, south, 0, gridcomm, MPI_STATUS_IGNORE);
    MPI_Send(&local_array[GETINDEX(local_x-1,0,0)], 1, zy_plane, south, 1, gridcomm);
    MPI_Recv(&local_array[GETINDEX(0,local_y-1,0)], 1, zy_plane, north, 1, gridcomm, MPI_STATUS_IGNORE);

    // sending/receiving east, west
    MPI_Send(&local_array[GETINDEX(0,0,0)], 1, xz_plane, west, 2, gridcomm);
    MPI_Recv(&local_array[GETINDEX(0,local_y,0)], 1, xz_plane, east, 2, gridcomm, MPI_STATUS_IGNORE);
    MPI_Send(&local_array[GETINDEX(0,local_y-1,0)], 1, xz_plane, east, 3, gridcomm);
    MPI_Recv(&local_array[GETINDEX(0,-1,0)], 1, xz_plane, west, 3, gridcomm, MPI_STATUS_IGNORE);
APPENDIX A. APPLICATION 1 SOURCE CODE

262 // sending/receiving above, below
263 MPI_Send(&local_array[GETLINDEX(0,0,local_z−1)],1,xy_plane,above
264 ,4,gridcomm);
265 MPI_Recv(&local_array[GETLINDEX(0,0,−1)],1,xy_plane,below,4,
266 gridcomm,MPI_STATUS_IGNORE);
267 MPI_Send(&local_array[GETLINDEX(0,0,0)],1,xy_plane,below,5,
268 gridcomm);
269 MPI_Recv(&local_array[GETLINDEX(0,0,local_z)],1,xy_plane,above,5,
270 gridcomm,MPI_STATUS_IGNORE);
271 }
272
273 void writeaf file()
274 {
275 FILE* f;
276 f = fopen("/work/idarbo/per.conv", "w");
277 int i,j,k;
278 float value;
279 for(i=0;i<x;++i)
280 for(j=0;j<y;++j)
281 for(k=0;k<z;++k){
282 // value = i+j+k;
283 value = 1;
284 if(j<8 && k<8){
285 value = 9;
286 }
287 fwrite(&value,sizeof(float),1,f);
288 }
289 fclose(f);
290 }
291
292 int main(int argc, char** argv){
293 float* array,*time1,*time2;
294 int timeusec,timesec, timeusec2, timesec2,rtimesec, rtimeusec;
295 FmmData* data;
296 x = SIZEX;
297 y = SIZEY;
298 z = SIZEZ;
299 int i,j,k;
300 /* initialize MPI*/
301 MPI_Init(&argc, &argv);
302 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
303 MPI_Comm_size(MPI_COMM_WORLD, &size);
304 MPI_Dims_create(size,ndims,dims);
305 MPI_Cart_shift(gridcomm, 0, &north, &south);
306 MPI_Cart_shift(gridcomm, 0, &west, &east);
307 MPI_Cart_shift(gridcomm, 2, &below, &above);
308 MPI_Comm_rank(gridcomm, &cartrank);
MPI_Cart_coords(gridcomm, cartrank, 3, coords);

// if(cartrank ==0)
// writeFile();
// 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 */
/* scatter data (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, timeusec2, 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

```c
#ifndef ARRAY_H
#define ARRAY_H

#include "mpi.h"

/* Defines the size of the array */
#define SIZEX 448
#define SIZEY 704
#define SIZEZ 1216

/* Calculate the strides for global array */
#define stride0 ((SIZEX+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 yz_plane;
MPI_Datatype xy_plane;
MPI_Datatype y_column;
MPI_Datatype y_column_resized;

MPI_Datatype int_xz_plane;
MPI_Datatype int_yz_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);
APPENDIX B. APPLICATION 2 SOURCE CODE

79

∗ print a local int array to stdout
80
*/
81
void printLocalIntArray(int* array);
82
/*
83
∗ Get what node a global position resides in
84
*/
85
int getDest(int x, int y, int z);
86
/*
87
∗ Get global coordinates from local coordinates
88
*/
89
int* getGlobalCord(int x, int y, int z);
90
/*
91
∗ Get local coordinates from global coordinates
92
*/
93
int* getLocalCord(int x, int y, int z);
94
/*
95
∗ Start exchanging of borders
96
*/
97
void exchangeBorders(float* array);
98
/*
99
∗ Initialize mpi data types
100
*/
101
void initMPIDatatypes();
102
/*
103
∗ Start exchanging integer borders
104
*/
105
void int_exchangeBorders(int* array);
106
/*
107
∗ Initialize integer datatypes
108
*/
109
void int_initMPIDatatypes();
110
/*
111
∗ Wait for all int exchange borders
112
*/
113
void wait_exchange_int();
114
/*
115
∗ wait for all float exchange borders
116
*/
117
void wait_exchange_float();
118
/*
119
∗ Wait for all border exchanges
120
*/
121
void wait_exchange_all();
122

#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;
    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: %f%z
", k, array[k], cartrank, k);
    for (i=-1; i<local_x; i++)
```
APPENDIX B. APPLICATION 2 SOURCE CODE

```c
57    printf("%d : ", cartrank);
58    for(j=-1;j<=local_y;j++){
59        printf("%8f",array[GETINDEX(i,j,k)]);
60    }
61    printf("\n");
62    }
63    printf("\n\n");
64 }
65 */
66 // print a local integer array to stdout
67 */
68 void printLocalIntArray(int* array){
69    int i,j,k;
70    for(k=-1;k<=local_z;k++){
71        printf("%d : array\n",cartrank,k);
72        for(i=-1;i<=local_x;i++){
73            printf("%d : ",cartrank);
74            for(j=-1;j<=local_y;j++){
75                printf("%d",array[GETINDEX(i,j,k)]);
76            }
77            printf("\n");
78        }
79    }
80    printf("\n\n");
81 }
82 */
83 // print the global array to stdout
84 */
85 void printArray(float* array){
86    int i,j,k;
87    for(k=-1;k<=z;k++){
88        printf("array\n",k);
89        for(i=-1;i<=x;i++){
90            printf("%8f",array[GETINDEX(i,j,k)]);
91        }
92    }
93    printf("\n");
94    printf("\n\n");
95 }
96 */
97 // start non blocking exchange border communication
98 */
99 void exchangeBorders(float* array){
100 // sending / receiving north south
101     MPI_Isend(&array[GETINDEX(0,0,0)],1,zy_plane,north,0,
102                  gridcomm,&request[0]);
103     MPI_Irecv(&array[GETINDEX(local_x,0,0)],1,zy_plane,south,0,
104                  gridcomm,&request[1]);
105     MPI_Waitall(2, request, &status);
106 }
107 ```
APPENDIX B. APPLICATION 2 SOURCE CODE

gridcomm,& request[1]); //MPI_STATUS_IGNORE);
111 MPI_Isend(& array [GETLINDEX(local_x−1,0,0)],1,zy_plane,south,
112 1,gridcomm,& request[2]);
113 MPI_Irecv(& array [GETLINDEX(−1,0,0)],1,zy_plane,north,1,
114 gridcomm,& request[3]); //MPI_STATUS_IGNORE);
115
116 // sending/receiving east, west
117 MPI_Isend(& array [GETLINDEX(0,0,0)],1,xz_plane,west,2,
118 gridcomm,& request[4]);
119 MPI_Irecv(& array [GETLINDEX(0,local_y,0)],1,xz_plane,east,2,
120 gridcomm,& request[5]); //MPI_STATUS_IGNORE);
121 MPI_Isend(& array [GETLINDEX(0,local_y−1,0)],1,xz_plane,east,
122 3,gridcomm,& request[6]);
123 MPI_Irecv(& array [GETLINDEX(0,−1,0)],1,xz_plane,west,3,
124 gridcomm,& request[7]); //MPI_STATUS_IGNORE);
125
126 // sending/receiving above, below
127 MPI_Isend(& array [GETLINDEX(0,0,local_z−1)],1,xy_plane,above,
128 4,gridcomm,& request[8]);
129 MPI_Irecv(& array [GETLINDEX(0,−1,0)],1,xy_plane,below,4,
130 gridcomm,& request[9]); //MPI_STATUS_IGNORE);
131 MPI_Isend(& array [GETLINDEX(0,0,0)],1,xy_plane,below,5,
132 gridcomm,& request[10]);
133 MPI_Irecv(& array [GETLINDEX(0,0,local_z)],1,xy_plane,above,5,
134 gridcomm,& request[11]); //MPI_STATUS_IGNORE);
135 }
136
137 /*
138 * initialize mpi datatypes
139 */
140 void initMPIDatatypes()
141 {
142    MPI_Type_vector(local_y,local_z,local_z+2,MPI_FLOAT,&
143                  zy_plane);
144    MPI_Type_commit(&zy_plane);
145
146    MPI_Type_vector(local_x,local_z,(local_z+2)*(local_y+2),
147                    MPI_FLOAT,&xz_plane);
148    MPI_Type_commit(&xz_plane);
149
150    MPI_Type_vector(local_y,1,local_z+2,MPI_FLOAT,&y_column);
151    MPI_Type_commit(&y_column);
152    MPI_Type_create_resized(y_column,0,(local_z+2)*(local_y+2)*
153                               sizeof(float),&y_column_resized);
154    MPI_Type_vector(1,2,1,y_column_resized,&xy_plane);
155    MPI_Type_commit(&xy_plane);
156 }
157
158 /*
159 * start exchanging integer border
160 */
161 void int_exchangeBorders(int* array)
162 {
163     // sending/receiving north south
164     MPI_Isend(& array [GETLINDEX(0,0,0)],1,int_zy_plane,north,6,
165                  gridcomm,& request[12]);
166     MPI_Irecv(& array [GETLINDEX(local_x,0,0)],1,int_zy_plane,
south, 6, gridcomm, & request[13]); //MPI_STATUS_IGNORE);
154 MPI_Isend(& array[GETLINDEX(local_x - 1, 0, 0)], 1, int_zy_plane,
south, 7, gridcomm, & request[14]);
155 MPI_Irecv(& array[GETLINDEX(-1, 0, 0)], 1, int_zy_plane, north, 7,
gridcomm, & request[15]); //MPI_STATUS_IGNORE);
156
157 // printf("%d: sending to %d receiving from %d\n", cartrank, west, east);
158 // sending/receiving east, west
159
160
161
162 MPI_Isend(& array[GETLINDEX(0, 0, 0)], 1, int_xz_plane, west, 8,
gridcomm, & request[16]);
163 MPI_Irecv(& array[GETLINDEX(0, local_y, 0)], 1, int_xz_plane, east
, 8, gridcomm, & request[17]); //MPI_STATUS_IGNORE);
164 MPI_Isend(& array[GETLINDEX(0, local_y - 1, 0)], 1, int_xz_plane,
east, 9, gridcomm, & request[18]);
165 MPI_Irecv(& array[GETLINDEX(0, -1, 0)], 1, int_xz_plane, west, 9,
gridcomm, & request[19]); //MPI_STATUS_IGNORE);
166
167 // sending/receiving above, below
168
169 MPI_Isend(& array[GETLINDEX(0, 0, local_z - 1)], 1, int_xy_plane,
above, 10, gridcomm, & request[20]);
170 MPI_Irecv(& array[GETLINDEX(0, 0, -1)], 1, int_xy_plane, below, 10,
gridcomm, & request[21]); //MPI_STATUS_IGNORE);
171 MPI_Isend(& array[GETLINDEX(0, 0, 0)], 1, int_xy_plane, below, 11,
gridcomm, & request[22]);
172 MPI_Irecv(& array[GETLINDEX(0, 0, local_z)], 1, int_xy_plane,
above, 11, gridcomm, & request[23]); //MPI_STATUS_IGNORE);
173 }
174
175 /*
176 * initialise integer mpi datatypes
177 */
178 void int_initMPIDatatypes()
179 {
180   MPI_Type_vector(local_y, local_z, local_z + 2, MPI_INT,&
int_zy_plane);
181   MPI_Type_commit(& int_zy_plane);
182
183   MPI_Type_vector(local_x, local_z, (local_z + 2)* (local_y + 2),
   MPI_INT,& int_xz_plane);
184   MPI_Type_commit(& int_xz_plane);
185
186   MPI_Type_vector(local_y, 1, local_z + 2, MPI_INT,& int_y_column);
187   MPI_Type_commit(& int_y_column);
188   MPI_Type_create_resized(int_y_column, 0, (local_z + 2)* (local_y
+ 2)* sizeof(int), & int_y_column_resized);
189   MPI_Type_vector(1, 2, 1, int_y_column_resized, & int_xy_plane);
190   MPI_Type_commit(& int_xy_plane);
191
192 /*
193 * Wait for int border exchange to finish
194 */
195 void wait_exchange_int()
196 {
197   MPI_Waitall(12, & request[12], & status[12]);
APPENDIX B. APPLICATION 2 SOURCE CODE

197 }
198 /*
199  * wait for float border exchange to finish
200 */
201 void wait_exchange_float()
202 {
203   MPI_Waitall(12, request, status);
204 }
205 /*
206  * wait for all border exchanges to finish
207 */
208 void wait_exchange_all()
209 {
210   MPI_Waitall(24, request, status);
211 }
B.2 Fast Marching Method

B.2.1 fmm_mpi.h

```c
#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 arrival times
    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;

/* initialize the FMM set velocity array, 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

```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
 * rollbacksallest is the smallest value of received border values, which dictates the rollback number */

float largest_solution, rollbacksallest;

/*
 * 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 is %d maxsize if
   \n", cartrank, px, py, pz, data->heap->heapsize, data->heap->maxsize, data->heap->maxsize);
    #endif
    temp= malloc(sizeof(Element));
    temp->value = data->timearray[GETINDEX(px, py, pz)];
```
APPENDIX B. APPLICATION 2 SOURCE CODE

```c
54     temp->x = px;
55     temp->y = py;
56     temp->z = pz;
57     heapInsert(data->heap, temp);
58     #ifdef DEBUG
59     printf("%d: inserted %d, %d, %d to heap, heap size is %d\n",
60      cartrank, px, py, pz, data->heap->heapsize);
61     #endif
62 }
63
64 } /*
65     * initialize the FMM
66     * velarray is the velocity field
67     * posx, y, z is position of the starting point
68     * x, y, z is the size of the array, most are read from array_mpi.h
69 */
70 FmmData* initFMM(float* velarray, int posx, int posy, int posz, int x,
71                   int y, int z) {
72     working = malloc(sizeof(int) * size);
73     MPI_Element e;
74     MPI_Datatype type[5] = { MPI_FLOAT, MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT };
75     int blocklen[5] = { 1, 1, 1, 1, 1 };
76     MPI_Aint disp[5];
77     disp[0] = 0;
78     disp[1] = sizeof(float);
79     disp[2] = sizeof(int) + disp[1];
80     disp[3] = sizeof(int) + disp[2];
81     disp[4] = sizeof(int) + disp[3];
82     MPI_Type_create_struct(5, blocklen, disp, type, &mpi_element_struct);
83     MPI_Type_commit(&mpi_element_struct);
84     /* end init datatypes*/
85     #ifdef DEBUG
86     printf("%d: done init mpi datatypes \n", cartrank);
87     #endif
88     int send = 0;
89     FmmData* data;
90     data = malloc(sizeof(FmmData));
91     if (data == 0) {
92     printf("%d: Failed to allocate memory, exiting \n", cartrank);
93     exit(1);
94     }
95     data->bandarray = malloc(sizeof(int)*LOCALARRAYSIZE);
96     if (data->bandarray == 0) {
97     printf("%d: Failed to allocate memory, exiting \n", cartrank);
98     exit(1);
99     }
100     data->timearray = malloc(sizeof(float)*LOCALARRAYSIZE);
101     if (data->timearray == 0) {
102     printf("%d: Failed to allocate memory, exiting \n", cartrank);
103     exit(1);
104     }
105     data->velocityarray = velarray;
```
APPENDIX B. APPLICATION 2 SOURCE CODE

108 #ifdef DEBUG
109 printf("%d: initializing heap\n", cartrank);
110 #endif
111 data->heap = initHeap(LOCALARRAYSIZE);
112 data->x = x;
113 data->y = y;
114 data->z = z;
115 data->posx = posx;
116 data->posy = posy;
117 data->posz = posz;
118
119 #ifdef DEBUG
120 printf("%d: clearing memory, bandarray, %p, size, %d\n", cartrank, data->bandarray, sizeof(int)*LOCALARRAYSIZE);
121 fflush(stdout);
122 #endif
123 /*
124 * setting the band array to outside
125 */
126 memset(data->bandarray, OUTSIDE, sizeof(int)*LOCALARRAYSIZE);
127 #ifdef DEBUG
128 printf("%d: Heap ok, clearing memory, timearray, %p, size, %d\n", cartrank, data->timearray, sizeof(float)*LOCALARRAYSIZE);
129 #endif
130 /*
131 * zeroing the arrival time array
132 */
133 int i = 0;
134 for(i=0;i<LOCALARRAYSIZE; i++)
135 data->timearray[i] = 0;
136 //memset(data->timearray, 0, sizeof(float)*LOCALARRAYSIZE);
137 //bzero(data->timearray, sizeof(float)*LOCALARRAYSIZE);
138 data->sentarray = malloc(sizeof(float)*LOCALARRAYSIZE);
139 bzero(data->sentarray, sizeof(float)*LOCALARRAYSIZE);
140 #ifdef DEBUG
141 printf("%d: done allocating memory, setting starting point\n", cartrank);
142 fflush(stdout);
143 #endif
144 /*
145 * inserting the starting point on the correct node and add it to
146 * the heap / narrow band
147 */
148 if(getDest(posx, posy, posz) == cartrank){
149 int *cr = getLocalCord(posx, posy, posz);
150 // insert starting point
151 data->timearray[GETLINDEX(cr[0], cr[1], cr[2])] = 0;
152 data->bandarray[GETLINDEX(cr[0], cr[1], cr[2])] = BAND;
153 //printf("%d: sat pos %d %d %d, as known\n", cartrank, cr[0], cr[1], cr[2]);
154 Element* element;
155 element = malloc(sizeof(Element));
156 element->x = cr[0];
157 element->y = cr[1];
158 element->z = cr[2];
159 element->value = data->timearray[GETLINDEX(cr[0], cr[1], cr[2])];
160 heapInsert(data->heap, element);
161 send = 1;
APPENDIX B. APPLICATION 2 SOURCE CODE

```cpp
161 } return data;
162 } /*
163 * return min
164 */
165 float min(float per, float truls) {
166 if (per > truls) return truls;
167 else return per;
168 }
169 /*
170 * return max
171 */
172 float max(float per, float truls) {
173 if (per > truls) return per;
174 else return truls;
175 }
176 /*
177 * prints xy plane from a float array
178 */
179 void printFloatArray(int sizex, int sizey, int z, float* array) {
180 int i, j;
181 printf("\n\nPrinting matrix\n\n")
182 for (i = 0; i < sizex; i++) {
183 for (j = 0; j < sizey; j++) {
184 printf("%8.2f", array[GETLINDEX(i, j, z)]);
185 }
186 printf("\n\n")
187 }
188 }
189 /*
190 * calculate the arrival time for a point x,y,z
191 */
192 float calcDistance(FmmData* data, int x, int y, int z) {
193 float sol;
194 sol = BIGFLOAT;
195 if (data->bandarray[GETLINDEX(x + 1, y, z)] > BAND) {
196 sol = min(data->timearray[GETLINDEX(x + 1, y, z)] + 1/data->velocityarray[GETLINDEX(x, y, z)], sol);
197 #ifdef DEBUG
198 printf("%d: sol is %f for x+1\n", cartrank, data->timearray[GETLINDEX(x + 1, y, z)] + 1/data->velocityarray[GETLINDEX(x, y, z)]);
199 #endif
200 }
201 if (data->bandarray[GETLINDEX(x - 1, y, z)] > BAND) {
202 sol = min(data->timearray[GETLINDEX(x - 1, y, z)] + 1/data->velocityarray[GETLINDEX(x, y, z)], sol);
203 #ifdef DEBUG
204 printf("%d: sol is %f for x-1\n", cartrank, data->timearray[GETLINDEX(x - 1, y, z)] + 1/data->velocityarray[GETLINDEX(x, y, z)]);
205 #endif
206 }
207 }
```
APPENDIX B. APPLICATION 2 SOURCE CODE

90

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;

void calcElement(FmmData* data, int px, int py, int pz) {
    int add = 0;
    float sol;
    Element *temp;
    /*
    * 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
    *
    */
    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 happen
                in serial version */
        }
    }
}
if (data->timearray[GETLINDEX(px, py, pz)] != 0 && data->timearray[GETLINDEX(px, py, pz)] <= sol) {
    return;
}

/* If the point is OUTSIDE add it to the heap */
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 arrival time */
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);
}

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};
    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

#define DEBUG

printf("%d: rolling back %d, old value %f, new value %f\n", cartrank, o[i], l[i], m[i], data->timearray[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
*/
#endif

void addElement(FmmData* data, MPI_Element e) {
  int* cr;
  int i, j, k;
  cr = getLocalCord(e.x, e.y, e.z);

#define DEBUG

printf("%d: adding element, to local\n", 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: already added\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]};

/* * recalculate all neighbours */

for (i=0; i<6; i++) {
  #ifdef DEBUG
  printf("%d: calc element\n", cartrank, o[i], l[i], m[i]);
  #endif
  calcElement(data, o[i], l[i], m[i]);
}
APPENDIX B. APPLICATION 2 SOURCE CODE

356  * set rollback values, so we can check if a rollback is necessary
357  */
358  /*if(largest_solution > e.value & rollallbacksmallest > e.value)
359  rollallbacksmallest = e.value;
360  rollbackn = e.n;
361  */
362 }
363
364 /* rollack all values above input value
365 */
366 void rollback(FmmData* data, float value){
367  int i,j,k;
368  for(i=0;i<local_x;i++)
369    for(j=0;j<local_y;j++)
370      for(k=0;k<local_z;k++)
371        if(data->bandarray[GETLINDEX(i,j,k)] > BAND & data->
372          timearray[GETLINDEX(i,j,k)] > value){
373            data->bandarray[GETLINDEX(i,j,k)] = BAND;
374            addToHeap(data,i,j,k);
375          }
376    }
377  }
378 }
379
380 /* send new values and check for incoming border values
381 */
382 void sendRecvBorderChanges(FmmData* data, int x, int y, int z, int
383  send){
384  MPI_Element e;
385  int run= 1;
386  int *cr;
387  int reast,rwest,rnorth,rsouth,rabove,rbelow;
388  int seast,seast,snorth,ssouth,sabove,sbelow;
389  reast = rwest = rnorth = rsouth = rabove = rbelow = 0;
390  seast = swest = snorth = ssouth = sabove= sbelow = 0;
391  /*
392   * if we are to send a value
393  */
394  if(send){
395    /*
396     * see where we have to send the value
397    */
398      if(x == 0){
399        snorth = 1;
400          }
401      if(x == local_x-1){
402        ssouth = 1;
403          }
404      if(y == 0){
405        swest = 1;
406          }
407      if(y == local_y-1){
408        seast = 1;
409          }
410      if(z == 0){
411        sbelow = 1;
if (z == local_z - 1) {
  sabove = 1;
}

#ifdef DEBUG
  printf("%d sending_to_north %d sending_to_south %d sending_to_west %d sending_to_east %d sending_to_above %d sending_to_below %d
", cartrank, x, y, z, snorth, ssouth, swest, seast, above, below, sabove, above);
#endif

if (snorth || ssouth || swest || seast || sabove || sbelow) {
  cr = getGlobalCord(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;

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);
  }
...
MPI_Iprobe(below, 1, gridcomm, &rbelow, MPI_STATUS_IGNORE);

if(rnorth || rsouth || rwest || east || rabove || rbelow)

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(east)
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 synchronous border exchage */
void sendRecvBorderChanges(FmmData* data, int x, int y, int z, int send)

MPI_Element e;

int *cr;
int east, west, north, south, above, below;
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;
  } else if (x == local_x - 1) {
    ssouth = 1;
  } else if (y == 0) {
    swest = 1;
  } else if (y == local_y - 1) {
    seast = 1;
  } else if (z == 0) {
    sbelow = 1;
  } else if (z == local_z - 1) {
    sabove = 1;
  }
  #ifdef DEBUG
  printf("%d: north %d south %d west %d east %d below %d above %d
", 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(& ssoouth, 1, MPI_INT, south, 0, gridcomm);
MPI_Recv(& rnorhth, 1, MPI_INT, north, 0, gridcomm, MPI_STATUS_IGNORE);
MPI_Send(& snorth, 1, MPI_INT, north, 0, gridcomm);
MPI_Recv(& rsouuth, 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[GETINDEX(x, y, z)];
  e.n = data->bandarray[GETINDEX(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 to known\n", cartrank, e->x, e->y, e->z);
                #endif
                data->bandarray[GETINDEX(e->x, e->y, e->z)] = KNOWN;
            } else {
                for(int 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);
                    }
                }
            }
        }
    }
```c
int k[6] = {e->x-1,e->x,e->x+1,e->x,e->x,e->x};
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;
}
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]);
    }
}
#endif
/* 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]);
    }
}
#endif
/* Send changes to border and look for incoming changes to the
border */
sendRecvBorderChanges(data, posx,posy,posz,senddata);
#endif DEBUG
if(cartrank == 0 || cartrank == -2) {printLocalArray(data->
timearray);
printLocalIntArray (data->bandarray);
}
#endif
senddata = 0;
```
APPENDIX B. APPLICATION 2 SOURCE CODE

/* Do we end the loop? */
if (heapGetMin(data->heap) == 0)
    run = 0;
else{
    run = 1;
}
checkOthers();
#endif DEBUG
if (cartrank == 0 && n%1000 == 0)
    printf("%d: reached %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;
}
#endif 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

```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 should 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 doesn't contain a value

```c
int checkArray(int x, int y, int z, float value)
```

```c
int rvalue = 1;
for(i=0; i<x; i++)
    for(j=0; j<y; j++)
        for(k=0; k<z; k++){
            if(local_array[GETINDEX(i,j,k)] != value){
                rvalue = 0;
                return rvalue;
            }
        }
    return rvalue;
```

Check if a global array doesn't contain a specific value

```c
int checkGArray(int x, int y, int z, float value)
```

```c
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

```c
void divide_matrix()
```

```c
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;
    }
}
```
APPENDIX B. APPLICATION 2 SOURCE CODE

103

110 }
111 }
112 printf("%d: local_x local_y local_z coords %d,%d,%d
",
113    cartrank, local_x, local_y, local_z, coords[0], coords[1], coords
114    [2]);
115 }
116
117 /* Initialize the program allocating local matrices and initializing
   data types */
118 void init()
119 {
120   divide_matrix();
121   local_array = malloc(sizeof(float)*LOCALARRAYSIZE);
122   if(local_array == 0){
123     printf("Couldn't allocate enough memory for local_array
");
124     exit(1);
125   }
126   bzero(local_array, sizeof(float)*LOCALARRAYSIZE);
127   initMPIDatatypes();
128 }
129
130 /* 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 matrices */
131 void scatterdata(char* filename)
132 {
133   int i, j;
134   int* cr;
135   int value;
136   char* errorstr;
137   int reslen;
138   FILE* f;
139   int offset = 0;
140   printf("%d: opening file %s
", cartrank, filename);
141   f = fopen(filename, "rb");
142   if(!f){
143     printf("%d: unable to open file %d
", cartrank, f);
144     fflush(stdout);
145     return;
146   }
147   if(DEBUG){
148     printf("%d: opened file %d
", cartrank, f);
149   }
150   for(i = 0; i<local_x; i++){
151     for(j= 0; j< local_y; j++)
152       {
153         cr = getGlobalCord(i, j, 0);
154         offset = sizeof(float)*(cr[2]+cr[1]*z+cr[0]*z*y);
155         fseek(f, offset, SEEK_SET);
156         fread(&local_array[GETINDEX(i, j, 0)], sizeof(float), local_z, f);
157         free(cr);
158       }
159   }
160   fclose(f);
APPENDIX B. APPLICATION 2 SOURCE CODE

163 }
164 /*
165 * gather all the local matrixes into a global matrix on node 0
166 */
167 float* gatherdata(float* iarray)
168 int i,j,k,dest,flag,r,t;
169 float* farray;
170 MPI_Request* requests;
171 MPI_Status* status;
172 requests = malloc(sizeof(MPI_Request)*local_y*local_x*2);
173 status = malloc(sizeof(MPI_Status)*local_y*local_x*2);
174 if(cartrank == 0){
175 if (DEBUG){
176 printf("%d: started gathering\n",cartrank);
177 }
178 farray = malloc(sizeof(float)*ARRAYSIZE);
179 bzero(farray,sizeof(float)*ARRAYSIZE);
180 }
181 }
182 if(cartrank != 0){
183 for(i=0;i<local_x;i++)
184 for(j=0; j<local_y; j++){
185 MPI_Send(&iarray[GETLINDEX(i,j,0)],local_z,MPI_FLOAT,0,i*local_y+j,gridcomm);
186 if (DEBUG){
187 printf("%d: Done sending\n",cartrank);
188 }
189 }
190 }
191 }
192 if(cartrank==0){
193 if (MPIDEBUG){
194 printf("%d: starting setting recvs\n",cartrank);
195 }
196 for(r=0;r<x;r+=local_x)
197 for(t=0; t<y; t+=local_y)
198 for(k=0;k<dims[2];k++){  
199 dest = getDest(r,t,k*local_z);
200 if (DEBUG){
201 printf("%d: receiving from %d\n",cartrank,dest);
202 }
203 for(i=0;i<local_x;i++)
204 for(j=0;j<local_y; j++){
205 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)]);
206 if (dest == 0){
207 MPI_Isend(&iarray[GETINDEX(i,j,0)],local_z,MPI_FLOAT ,0,i+local_y+j,gridcomm,&requests[(local_y*i)+j]);
208 }
209 }
210 }
211 if (dest == 0){
212 MPI_Waitall(local_y*local_x*2,requests,status);
213 }
214 if (dest != 0){
215 MPI_Waitall(local_y*local_x,requests[local_y*local_x],status);
216 }
APPENDIX B. APPLICATION 2 SOURCE CODE

216                  }
217                  }
218                  }
219                  }
220                  printf("%d: Done gathering \n", cartrank);
221                  return farray;
222                  }
223                  }
224                  
225                  /* check if the global array is the same as the array inside a file */
226                  void checkData(char* filename) {
227                  if(cartrank == 0) {
228                  int i,j,k;
229                  printf("%d: Reading file \n", cartrank);  
230                  file_array = malloc(ARRAYSIZE*sizeof(float));
231                  bzero(file_array,ARRAYSIZE*sizeof(float));
232                  readfile(file_array,filename,x,y,z);
233                  printArray(file_array);
234                  printf("%d: checking data consistency \n", cartrank, file_array[GETINDEX(0,0,0)]);
235                  for(i=0;i<x;i++)
236                  for(j=0;j<y;j++)
237                  for(k=0;k<z;k++) {
238                  if(file_array[GETINDEX(i,j,k)] != global_array[GETINDEX(i,j,k)]){
239                  printf("%d: error at %d,%d,%d file %lf global %lf\n", cartrank,i,j,k,file_array[GETINDEX(i,j,k)],
240                  global_array[GETINDEX(i,j,k)]);
241                  }
242                  }
243                  }
244                  }
245                  }
246                  }
247                  */
248                  /* a test function for writing a file with values */
249                  void writeafile() {
250                  FILE* f;
251                  f = fopen("/work/idarbo/per.conv","w");
252                  int i,j,k;
253                  float value;
254                  for(i=0;i<x;i++)
255                  for(j=0;j<y;j++)
256                  for(k=0;k<z;k++) {
257                  //value = i+j+k;
258                  value = 1;
259                  if(j < 8 & & k < 8) {
260                  value = 9;
261                  }
262                  fwrite(&value,sizeof(float),1,f);
263                  }
264                  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_Comm_create(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[GETINDEX(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);
379       */
380   }
381
382   printf("%d is done ending gracefully.\n", cartrank);
383   MPI_Finalize();
384
385  "}
Appendix C

Common files

C.1 Heap

C.1.1 heap.h

```c
#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 really min sorted
 * Only used in testing if the heap works correctly, shouldn't be used in production code
 */
float checkHeapConsistency(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 parent to a specific position in the heap
*/
```
```c
55 */
56 int inline getParent(int pos) {
57    return pos>>1; //pos/2;
58 }
59 /*
60 * Gets the left child of a specific position in the heap
61 */
62 int inline getLeft(int pos) {
63    return pos <<1; // pos * 2
64 }
65 /*
66 * Gets the right child of a specific position in the heap
67 */
68 int inline getRight(int pos) {
69    return (pos <<1) +1; // pos*2+1
70 }
71 */
72 /*
73 * Move one value to its correct position in the min sorted heap
74 * Run once for each new element in the heap
75 */
76 void MinHeapify(Heap* heap, int pos) {
77    int l, r;
78    int smallest = 0;
79    l = getLeft(pos);
80    r = getRight(pos);
81    if(l <= heap->heapsize && heap->array[l]->value < heap->array[pos]->value) {
82        smallest = l;
83    } else {
84        smallest = pos;
85    }
86    if(r <= heap->heapsize && heap->array[r]->value < heap->array[smallest]->value) {
87        smallest = r;
88    }
89    if(smallest != pos) {
90        Element* temp;
91        temp = heap->array[pos];
92        heap->array[pos] = heap->array[smallest];
93        heap->array[smallest] = temp;
94        MinHeapify(heap, smallest);
95    }
96 }
97 /*
98 * return and remove the smallest element from the heap
99 */
100 Element* heapExtractMin(Heap* heap) {
101    if(heap->heapsize <1) return NULL;
102    Element* temp = heap->array[1];
103    heap->array[1] = heap->array[heap->heapsize];
104    heap->heapsize--;
105    #ifdef DEBUG
106    printf("New heapsize %d\n", heap->heapsize);
107    #endif
108    MinHeapify(heap, 1);
109    return temp;
```
111 */
112 */
113 /∗ Return the smallest element from the heap
114 /∗ Does not remove anything from the heap
115 */
116 Element* heapGetMin(Heap* heap){
117     if(heap->heapsize <1) return NULL;
118     Element* temp = heap->array[1];
119     return temp;
120 }
121
122 */
123 /∗ Add a new element to the heap
124 /∗ this function adds a element to a position and moves it up until
125 its found its place in the heap.
126 */
127 void heapIncreaseKey(Heap* heap, int pos, Element* key){
128     Element* temp;
129     // if (key->value > array[pos]->value) {
130     if(heap->array[pos] != 0){
131         exit(2);
132     }
133     #ifdef DEBUG
134     printf("Inserting into %d\n",pos);
135     #endif
136     heap->array[pos] = key;
137     while (pos > 1 && heap->array[getParent(pos)]->value > heap->array
138         [pos]->value){
139     #ifndef DEBUG
140     printf("chaging position between %d and %d\n",pos,getParent(pos)
141         );
142     #endif
143     temp = heap->array[pos];
144     heap->array[pos] = heap->array[getParent(pos)];
145     heap->array[getParent(pos)] = temp;
146     pos = getParent(pos);
147     }
148     //checkHeapConsistency(heap);
149 }
150
151 */
152 /∗ add a new element to the heap
153 */
154 void heapInsert(Heap* heap, Element* key){
155     heap->heapsize++;
156     if(heap->heapsize > heap->maxsize)
157     {
158         printf("Exceeded heap max size of %d\n",heap->maxsize);
159         exit(5);
160     }
161     #ifdef DEBUG
162     printf("New heapsize %d array at %p\n",heap->heapsize,heap->array
163     );
164     #endif
165     heap->array[heap->heapsize] = 0;
166     heapIncreaseKey(heap,heap->heapsize,key);
167 }
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);
    }
    return 0;
}
```
C.2  Time

C.2.1  time.h

```c
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

```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

/*
 * 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

```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.%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+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("Couldn't_read_hole_file_exiting_read\n",read);
        fclose(f);
        exit(1);
      }
    }
  printf("done_read, closing_file\n");
```

APPENDIX C. COMMON FILES

56  fflush(stdout);
57  fclose(f);
58 }
59
60 /∗
61 * Reads float array from a text file
62 */
63 void readTextFile(float* array, char* file, int x, int y, int z){
64  FILE *f;
65  float temp;
66  char ctemp = 'h';
67  int i, j, t;
68  f = fopen(file, "r");
69  if (!f){
70     printf("Error opening file %s", file);
71     return;
72  }
73  bzero(array, sizeof(float)*ARRAYSIZE);
74  memset(array, 1, sizeof(float)*ARRAYSIZE);
75  fread(&ctemp, sizeof(char), 1, f);
76  for(i = 0; i < 2; i++){
77      while(c temp != '\n'){
78          printf(".");
79          fread(&ctemp, sizeof(char), 1, f);
80      }
81      printf("\n");
82  }
83  // for(t = 0; t < y*x; t++) {
84  t = 0;
85  while(fscanf(f, "%d%d%f\n", &i, &j, &temp) != EOF)
86      array[GETINDEX(i-1, j-1, z)] = temp;
87  t++;
88  }
89  printf("read %d values\n", t);
90  fclose(f);
91 }
92
93 /∗
94 * saves a float array as a png image
95 */
96 void printFloatImage(float* array, char* file, int sizex, int sizey, int z){
97  /* Declare the image */
98  gdImagePtr im;
99  /* Declare output files */
100  FILE *pngout;
101  #ifdef DEBUG
102      printf("printing image to file \n");
103  #endif
104  int max, min;
105  float max = 0, min = MAXFLOAT;
106  int i, j, r, g, b, t;
107  for(i = 0; i < sizex; i++){
108      for(j = 0; j < sizey; j++){
109          if(array[GETINDEX(i, j, z)] < min && array[GETINDEX(i, j, z)] != BIGFLOAT)
110              min = array[GETINDEX(i, j, z)];
111          printf("new min value at %d%d%d value is %f\n", i, j, z, min);
112  }
112 } else if(array[GETINDEX(i,j,z)] > max && array[GETINDEX(i,j,z)] != BIGFLOAT) {
113     max = array[GETINDEX(i,j,z)];
114 }
115 }
116 }
117 
118 for(i=0;i<szex;i++){
119     for(j=0;j<szey;j++){
120         if(array[GETINDEX(i,j,z)] == BIGFLOAT || array[GETINDEX(i,j,z)]] == BIGFLOAT*2){
121             array[GETINDEX(i,j,z)] = max;
122         }
123     }
124 }
125 #ifdef DEBUG
126     printf("Maxvalue is %f, min value is %f \n",max,min);
127 #endif
128 /* Allocate the image: 64 pixels across by 64 pixels tall */
129     im = gdImageCreate(szex, szey);
130 /* Allocate the color black (red, green and blue all minimum).
131     Since this is the first color in a new image, it will
132     be the background color. */
133     maxt = 0;
134     mint = 9999999;
135     for(i=0;i<szex;i++){
136         for(j=0;j<szey;j++){
137             //printf("color is %d\n",(int) (((float)array[GETINDEX(i,j,z)] / (float)(max−min)) *255));
138             t = (int) (((float)array[GETINDEX(i,j,z)]−min) / (float)(max−min)) *255*4))a
139                 if(maxt < t)
140                     maxt = t;
141             if(mint > t)
142                 mint = t;
143         //printf("%d\n",t);
144         /* red= yellow= green= cyan= blue */
145         /* if(t<255){
146             r = 255;
147             g = t;
148             b = 0;
149         } else if(t<255*2){
150             r = 255*2 −t;
151             g = 255;
152             b = 0;
153         } else if(t<255*3){
154             r = 0;
155             g = 255;
156             b = t − 255*2;
157         } else{
158             r = 0;
159             g = 255*4 − t;
160             b = 255;
161         }*/ /* blue = cyan = green = yellow = red */
162     t = 255*4 −t;
166  r= 255;
167  g= t;
168  b=0;
169  } else if(t<255*2){
170    r= 255*2 -t;
171    g = 255;
172    b = 0;
173  } else if(t<255*3){
174    r = 0;
175    g = 255;
176    b = t - 255*2;
177  } else{
178    r = 0;
179    g = 255*4 - t;
180    b = 255;
181  }
182  gdImageSetPixel(im, i, j, gdImageColorResolve(im, r/red/, g/green/, b/blue/));
183 }
184 }
185 #ifdef DEBUG
186 printf("Maxvalue is %d, min value is %d\n", maxt, mint);
187 #endif
188 /* Open a file for writing. "wb" means "write binary", important
189  under MSDOS, harmless under Unix. */
190 pngout = fopen(file, "wb");
191 /* Output the image to the disk file in PNG format. */
192 gdImagePng(im, pngout);
193 /* Close the files. */
194 fclose(pngout);
195 /* Destroy the image in memory. */
196 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);
}
```