# Implementation of Parallelization

## In the Full Waveform Inversion Process

by

Paul Sebastiaan Werker


Student number: 5021774
Project duration: April 1st, 2021 – July 1st, 2021
Thesis committee:

| | |
|---|---|
| Dr. ir. D.J. Verschuur, | TU Delft, Supervisor |
| Dennis Karmelk, | Alten, Supervisor |
| Liban Abdulkadir, | Alten, Daily Supervisor |

TU Delft

# ALTEN

In this section, a short introduction is given to the company Alten, at which my internship has taken place. It includes an overview of the company, the work they do and lastly an overview of the project.

Alten's origin lies in France, where it was founded by three engineers: Laurent Schwarz, Thierry Woog and Simon Azoulay. The latter is still the CEO of the company. In 2002 the expansion started to the Netherlands, where it started with technical software solutions. Worldwide, the company has nearly 40 000 employees of which 75% in Europe. With 1000+ employees in the Netherlands, they serve other companies in providing the right people for their software solutions. They have expertise in research and development projects as well as technical and IT services. Consultants that are currently between projects, work on internal projects at Alten. This allows the consultant to expand their knowledge before moving to other projects.

The biggest customers of Alten are ASML, Philips and Shell on various technical software projects. They also serve smaller companies with specialized needs and are always searching for new clients. At Alten, training sessions are given to the consultants to keep their knowledge up to date. These training sessions are given internally so interns can also benefit from these courses. During this internship, it was possible to follow an advanced C++ course. Not only programming courses are given and range from organization to dutch workshops. Outside work, Alten employees organizes knowledge sharing sessions to gain knowledge in the advancements of programming or technologies. On top of that, a monthly meeting takes place to update the Alten employees on the state of the company.

The internal project I worked on is the full waveform inversion. The full waveform inversion is a project that uses pressure sensors to measure the acoustic wave response and from that reconstruct the layers of the earth. The full waveform inversion is a relatively new technique used in for example imaging for geotechnical site characterization. The calculations needed for this project are often demanding and are limited by processing power and memory usage. In this project we consider a 2D approximation of the full waveform inversion to significantly reduce the computation time. The goal of this internship was to implement parallelization into the project to speed up the calculation time. The OpenMP implementation was successfully implemented where a more than 2.5x speedup was accomplished while using eight threads. For MPI the calculation time has been reduced, but the additional sending and receiving of data happened to be a limiting factor. The internship lasted 3 months and was part of the Msc Applied Physics Curriculum at TU Delft.

# Contents

# 1

# Introduction

The world's energy production currently depends for the most part on oil and gas. Currently, a shift towards more renewable energy sources is desired towards 2050. In the meantime, the drilling for oil and gas continues. This may require the search of new fields at which the cost for exploration plays a more significant role as renewable energy poses as an economical competitor. A technique for discovering new oil fields is to drill a well several kilometres deep into the earth. This gives an overview of the layers of the earth and could give an indication of the presence of a new source for oil or gas. This method can be expensive and provides only limited information about the subsurface. In the ideal situation, an image of the layers could be generated without damaging it. The technique that describes this approach is the full waveform inversion (FWI). This method can be applied to various fields including medical and seismic imaging. By illuminating the surface with acoustic (sound) waves, the inside of a surface can be measured. By placing sensors on top of the surface, the reflected waves can be measured. The images of the subsurface can be reconstructed from the measurements. An example of medical imaging is the construction of an image of an unborn baby with ultrasound. The discovery of new oil or gas fields with this method can be more competitive compared to the traditional method.

The FWI is a data fitting method based on full-wavefield modelling to extract quantitative information from appropriate measurements [1]. The first occurrence of this method arose in the late '70s with a first attempt at seismic applications [2]. This technique has a heavy reliance on computational power so only in recent years it has become feasible in practice. This application can model the interior of a medium in a non-destructive manner. The execution of this process starts with placing acoustic sensors at the surface of the earth. An acoustic source, like an explosion, causes the waves to travel through the earth and reflect and refract inside the subsurface. The sensors can detect the reflected waves which can be used to generate a detailed image of the subsurface. An estimation of the subsurface is required in order to find an appropriate solution using FWI.

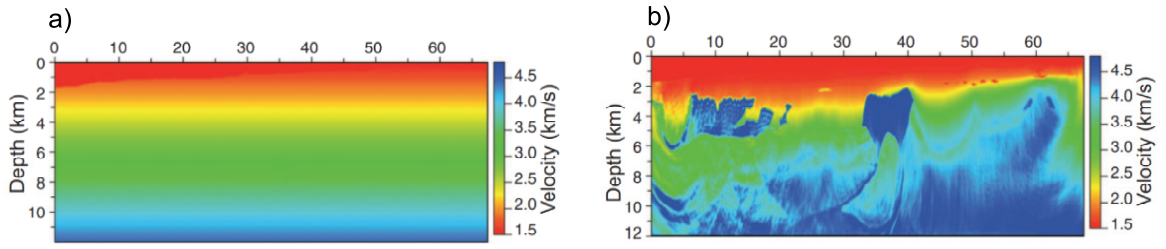


Figure 1.1: The full waveform inversion process where (a) is the initial guess of the subsurface and (b) is the final detailed model after FWI. Figure Adapted from [1]

Figure 1.1 shows the power of the full waveform inversion. An initial guess of the subsurface properties is shown in Figure 1.1 (a). After the calculations, the resultant velocity model is shown in Figure 1.1 (b) where much more detail about the underlying earth layers are uncovered.

Solving large scale 3D problems directly is still beyond the scope of many applications as it requires a lot of computational power and memory. In this implementation, a simplified full waveform inversion is used. First, a 2D version of the Helmholtz equation is utilized to model acoustic waves only. A single parameter, the relative wave propagation speed, is calculated to limit the memory and computational constraints. Secondly, the frequency domain is used to limit the non-linearity of the problem, so we can assume a linear solution. Lastly, the scattering integral formulation of the finite difference method is used for modelling. The solution to the model will only give a good result when the underground model has moderately variations in the subsurface properties.

This constraint on the system will not give very reliable results when used on actual earth layers, as they can be very heterogeneous. The project goal is however to create a single program with multiple FWI implementations for demonstrating purposes. Currently, the program is implemented with different versions of forward and inversion methods. Any model can currently be swapped by another. This allows for comparison between different implementations. This way it allows the building of new applications on top of the current implementation, like parallelization. The possible downside of this implementation is that it might be more difficult to write a completely new implementation as you are limited by the current data flows and structures. The previously mentioned implementation is considered as the best performing among the ones in the project.

Part of this internship on FWI will be the implementation of parallelization within the current framework. The goal is to speed up the calculations and allow for demonstrable parallelization gains. This means no new method will be added, but the current program will be adapted to implement different parallelization methods. In the end, any form of parallelization could be implemented in this software. In this report, the focus lies on the implementation of OpenMP and MPI. The first will allow parallelization with shared memory on a single device, while MPI allows for the distribution of tasks over multiple instances of the program.

Another part of this internship is to explore working in a commercial company. To gain experience in working in teams, communication and meeting deadlines. During this project, multiple idle consultants have come and go where we have helped each other in working and understanding the current tasks set out for this project.

# 2

# Background & Approach

In this section, the background and approach for this internship project are discussed. First, the basics of the full waveform inversion process are explained in detail. Next, the different ways of parallelization are considered. Lastly, the implementation strategy for these methods is laid out.

## 2.1. Full Waveform Inversion

The full waveform inversion process consists of two major parts, the forward- and the inversion method. For the forward modelling the finite difference method is used and for the inversion process, the conjugate gradient method is used. Since the data from our sources will be simulated, the forward modelling will be used to generate the data for the receivers. To solve the problem using these models the computational domain is discretized. A grid is created with the coordinates $\mathbf{x}_{ij} = \{x_i, z_j\}$ with $1 \leq i, j \leq n$. On this grid several acoustic sources are placed as well as receivers.

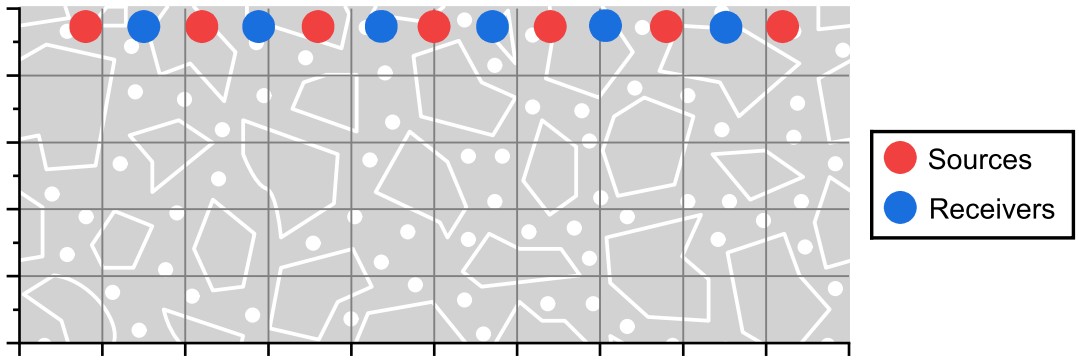


Figure 2.1: Caption

In Figure 2.1 a representation of the grid is displayed. The grid size in each direction is determined by the maximum length divided by the number of grid points. This results in a step size $h_x$ in the x-direction and $h_z$ in the z-direction. For the modelling of the sources, a version of a point source will be used. Since we have discretized our grid with spacing $h_x$ and $h_z$, a point source would introduce errors in rounding it to the nearest grid point. The implementation described in [3] is used and the source is then given by

$$\mathbf{s}(x,z) = \frac{\sin(\pi d(x))}{\pi d(x)} \frac{\sin(\pi d(z))}{\pi d(z)}, \tag{2.1}$$

where $d(y)$ denotes the distance in grid points from the source position. This function is then multiplied by the so-called window function, which determines the number of grid points over which the source is spread out.

In this implementation of the full waveform inversion, the frequency domain is used instead of the time domain. Using the frequency domain allows us to discretise the frequencies and consider only a few frequencies. It was found that a surprisingly good answer can be found with a relatively low number of frequencies [4]. The frequency group used is discretized into $m$ groups where we have a linear space between our minimal frequency $f_{min}$ and our maximum value $f_{max}$. The medium properties will be expressed in the contrast function which is given by the following equation

$$\chi(\mathbf{x}) = 1 - \left(\frac{c_0(\mathbf{x})}{c(\mathbf{x})}\right)^2. \tag{2.2}$$

This contrast function is dependant on the known background medium $\mathrm{c}_0(\mathbf{x})$ $(= 2000 m/s)$ and the unknown subsurface modal $\mathrm{c}(\mathbf{x})$. The iterative process will try to determine the contrast function at every step. Note that the contrast function results in zero when $c(\mathbf{x}) = c_0(\mathbf{x})$.

Modeling the seismic data requires solving the wave equation in a numerical sense. If the finite difference method is used the discretisized subsurface can be solved on the differences between the neighbouring grid-cells [5]. Using this finite difference model comes with drawbacks among which its instability at high frequencies, its subject to dispersion and the fact that it is rather computational expensive. To overcome this problem, the scattering integral formulation of the wave equation is used. The integral formulation sums the contribution of all grid points.

$$p_{\text{data}}\ (\mathbf{x}_\mathrm{r}, \mathbf{x}_\mathrm{s}, \omega) = \int G\,(\mathrm{x_r}, \mathrm{x}, \omega)\, p_{\text{tot}}\,(\mathrm{x}, \mathrm{x_s}, \omega)\, \chi(\mathrm{x})\mathrm{dx}, \tag{2.3}$$

$p_{data}$ describes the pressure data for each source, receiver and frequency. This equation solved by using Green's function $G$, the total pressure data $p_{tot}$ at each grid point and the contrast function $\chi$. A source transmits a wavefield that propagates to every point in the subsurface. The wavefield creates secondary sources in all points where the contrast function $\chi$ is non-zero. The secondary sources transmit energy through a smooth background medium to the receivers, represented by the Green's function $G$. The measured seismic data at the receivers are a summation of all secondary sources. The direct waves coming from the sources are removed from the measured data to obtain the final answer $p_{data}$. The integral formulation can thus be used to get a solution without having to run expensive calculations. The solution to the Green's function is complex valued and is expressed as

$$G(\mathrm{x}, \mathrm{y}) = \frac{i}{4} H_0^{(1)}\left(k_0 \|\mathrm{x} - \mathrm{y}\|\right) = -\frac{1}{4} Y_0\left(k_0 \|\mathrm{x} - \mathrm{y}\|\right) + \frac{i}{4} J_0\left(k_0 \|\mathrm{x} - \mathrm{y}\|\right). \tag{2.4}$$

where $H_0$, $J_0$ and $Y_0$ are defined as Hankel's functions [5].

Calculating the $p_{data}$ directly from the equation is the forward method used in this project. To create a linear solution, the dependancy of the total field on the contrast function is neglected. This is the Born-approximation and allows to do a linear inversion to obtain the approximate subsurface properties. In this approximation it is assumed that the true total field can be approximated by a simple, non-scattering background and hence $p_{tot} = p_0$.

### 2.1.1. Inversion Process

The inversion is the second part of the process. In the forward method we assume that the media properties are known and so thus, we can model the data at the receivers. The data collection step will obtain the data from the receivers and can be used in the inversion process to find the media properties. The inversion scheme used here will be the conjugate gradient method. This method is applied as an iterative algorithm to minimize the difference between the measured data $p_{data}$ and the modelled data. The residual between the modelled data is obtained as

$$r_0\,(\mathbf{x}_\mathrm{r}, \mathbf{x}_\mathrm{s}, \omega) = p_{\text{ref}}\,(\mathbf{x}_\mathrm{r}, \mathbf{x}_\mathrm{s}, \omega) - p_{\text{data}}\ (\mathbf{x}_\mathrm{r}, \mathbf{x}_\mathrm{s}, \omega)\,, \tag{2.5}$$

where $p_{data} = A\chi_i$ with $A$ a positive-definite matrix which represents the data of $G(x, y) * p_{tot}$ and the input vector $\chi_0$ is the approximate initial solution. The $\mathbf{x}_r$ and $\mathbf{x}_s$ are the location for the sources and receivers. The frequency is expressed as $\omega$ and $p_{ref}$ is defined as the initial input pressure data from the receivers. When the residual $r_i$ is sufficiently small it can be assumed that the minimization was successful. In a loop the following calculations are done to update the inversion

$$\alpha_i := \frac{\mathbf{r}_i^\top \mathbf{r}_i}{\mathbf{q}_i^\top \mathbf{A}\mathbf{q}_i}, \tag{2.6}$$

with $q_0 = r_0$ and $\alpha$ as scalar. From this the answer can be updated using the following formula

$$\chi_{i+1} = \chi_i + \alpha_i \mathbf{q}_i. \tag{2.7}$$

The updated answer for $\chi_i$ can be used to to calculated the next residual $\mathbf{r}_i$. From this residual the scalar $\beta$ is calculated to update the search direction $\mathbf{q}$

$$\beta = \frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{r}_i}, \tag{2.8}$$

$$\mathbf{q}_{i+1} = \mathbf{r}_{i+1} + \beta_i \mathbf{q}_i. \tag{2.9}$$

Using the updated direction the process can be repeated. The process can be stopped when the residual has reached the desired level. The $\chi$ value is then returned as the result for the conjugate gradient scheme. Above a simplified version of the conjugate gradient scheme was used to illustrate the process.

## 2.2. Parallelization

To solve these rather computation intensive equations we can make use of parallelization. When transforming these equations into code and letting our program calculate the solution, it will be done sequentially. For more complicated arithmetic, the calculation time will dominate the process. For this, parallel programming can be used to divide up tasks among different threads. The initialization and coupling of the results of the parallelization could take up additional CPU time, so not every step is suitable for parallelization. Some calculations may depend on the result of other threads, so a communication protocol needs to be setup which can slow down the total time. A possible pitfall are race conditions in which threads do not finish tasks sequentially [6]. To prevent this, operations like locking and unlocking memory are required. These regions create places where only a single processor can operate inside the memory.

For this thesis a few implementations of parallelization were considered. In the used forward and inversion methods, the calculations are independent and can be relatively straightforward parallelised. There are many ways of doing parallelization using CPU and GPU, but in this project it is limited to CPU parallelization. Two of these CPU methods will be used in this project, namely the technique OpenMP [7] and MPI [8]. While the goal of these two methods is the same, at the core they are distinct from each other. While OpenMP uses shared-memory within a single node, MPI implements parallelism between different nodes. This requires a different implementation technique which is discussed in the following sections. In the end a comparison can be made between both methods for varying amount of threads.

Each implementation will have a limit to the amount of time it can speed up the calculation. This is based on the fact that only parts of the program can be parallelized while other parts have to be done sequentially. This theoretical limit is called Amadahl's law. This law dictates the maximum improvements for the execution time after implementation [6]. The following equation expresses the law

$$S_{\text{speedup}}(s) = \frac{1}{(1-p) + \frac{p}{s}}, \tag{2.10}$$

where $S_{\text{speedup}}$ is the theoretical speedup of the complete tasked performed. $s$ Is the expected speedup and $p$ is the proportion of the execution time of the part of the task that benefits from the parallelization. It will be assumed that the number of nodes $n$ will speedup that part of the program $n$ times such that $s = n$.

### 2.2.1. OpenMP

OpenMP (Open Multi Processing) is one of the most straightforward ways of implementing parallelization on a single computer. It is a library that supports shared-memory multiprocessing programming. A non-profit organization OpenMP ARB manages the releases and updates of the library [7]. The first release of version 1.0 was in October 1997 and the latest version 5.1 was released in November 2020. For C++, a version of OpenMP is built in the GCC (GNU Compiler Collection) and can be enabled on compilation time. The 5.0 version is

used to build and compile this application.

It is the implementation of multithreading where tasks are duplicated and will run concurrently. From there it will be allocated to the available threads on the computer. The core program is started by the following preprocessor directive (pragma):

```
#pragma omp parallel
```

When a task needs access to shared-memory data, the pragma critical can be called to allow for the program to lock and unlock the memory.

```
#pragma omp critical
```

This is necessary as it is not possible to write to the same memory from different threads. These are the most straightforward examples of the usage of OpenMP. The actual implementation will take place in the loop that sums over the different frequencies. Whenever a calculation is done within that loop, it might be useful to declare the index variables private. This will place the variables on a separate stack, ensuring the thread with the correct data and thus without any data loss. A similar process holds for the reduction clause that lets you declare variables that undergo operations at the end of the parallel region. This was implemented to ensure no data is lost throughout the calculations at the cost of adding additional calculation time. An example of the implementation declaration of a parallel region looked as follows:

```
#pragma omp declare reduction(addition : class DataGrid2D : omp_out += omp_in) ...
    initializer(omp_priv(omp_orig))
int l_i, l_j, k;
#pragma omp parallel for reduction(addition : kRes) private(l_i, l_j, k)
for(int l_i,l_j,k ...) {...}
```

### 2.2.2. MPI

The MPI (Message Passing Interface) is a communication protocol for parallel computing. It is not an implementation but specifies syntax and semantics. This allows for multiple different implementations of MPI, which in the end should all compile the same code. This communication protocol had its first version 1.0 released June 1994.

The implementation used in this report is Open MPI. Its supports many operating systems, which makes the implementation portable. Unlike OpenMP, the whole program will have to run multiple times. This means that each program should have a unique set of instructions on what each program should do. Since they are run separately, there is no shared memory between them. To eventually combine the results a communication between each program should be set up. The MPI framework takes care of the communication protocol. The parallelization can be initialized by calling the following code:

```
MPI_Init(&argc, &argv);
```

The input arguments will process any possible configuration. The master and slave conditions can be defined by giving a rank to each process. For each process calling this rank, a unique id will be returned

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &size); /* get the total size of the id */
```

These ranks are then used to determine what parts of the program are executed. In a for loop parallelization, the size and rank can be used to give the unique instructions. The total number of processes is used to split up the total calculations into equal parts and the rank is used to determine a start and stop condition. In the following example, this is demonstrated:

```
int loop_size = total_loop_size / size;
int start = rank * loop_size;
int stop = (rank + 1) * loop_size;
for(int i = start; i < stop; i++){...}
```

In the example the code will start and stop depending on which process is running. Often things are calculated during the for loop which in the case of MPI will stay within each process. The processes can communicate the results back to the master process with the commands *send()* and *receive()*:

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_COMM_WORLD)
```

In this command, the specific data will be communicated with the specified process. The results can be combined into a single answer. The downside of this method is that the main process is mostly occupied with receiving and sending data instead of calculations.

While MPI is built for both C and C++, the FWI implementation relies heavily on standard C++ data structures like *std::vector<>* and *std::complex<>*. The MPI implementation only accepts 'primitive' C data structures, like *bool, int, float* etc. The data sent to and from the nodes are is a list of complex numbers *std::vector<std::complex<double»*. While on their own a translation can be made between these structures and the MPI library, a combination of the two is rather complicated. The library Boost.MPI can be used to solve this problem. This library is an interface over the currently installed MPI and simplifies the sending of data by taking care of the serialization. This means the currently used data structures All of the above will have a slightly different syntax while using boost, but do hold for each MPI implementation. In the following code sample, the usage of boost is displayed [9]:

```

#include <boost/serialization/string.hpp>
namespace mpi = boost::mpi;

mpi::environment env;
mpi::communicator world;

  if (world.rank() == 0) {
    world.send(1, 0, std::string("Hello"));
    std::string msg;
    world.recv(1, 1, msg);
    std::cout << msg << "!" << std::endl;
  } else {
    std::string msg;
    world.recv(0, 0, msg);
    std::cout << msg << ", ";
    std::cout.flush();
    world.send(0, 1, std::string("world"));
  }
```

## 2.3. System Parameters

The calculations were executed on a Macbook Pro (2013) i7 2,3 GHZ with turbo boost to 3,5 GHz, 8 core processor. The laptop contained 8GB of 1600 MHz DDR3L memory. Ubuntu was used as a native OS at version 20.04. The code was written and compiled in the C++ 17 standard. OpenMP version was set to 5.0 by the installed compiler. The Open MPI implementation was downloaded and installed at the latest version of 4.1.1[8]. On top of the Open MPI library the Boost.MPI and Boost Serialization was installed with version 1.76.0. The number of threads for OpenMP was set systemwide with the command *OMP_NUM_THREADS=<int>*. The MPI implemented executable was run with the accompanying executable *mpirun -np <int>* to allow multiple processes to run on a single machine for testing purposes.

### 2.3.1. Paralelization location

The approach was taken to analyse the scattering integral and the conjugate gradient method to find calculations with the longest duration. The possibility of parallelization at each step was considered. It was

found that the actual calculation time was dependent on two individual functions. In the forward method this function was the calculation of the pressure data and for the conjugate gradient method it was the process of calculating the gradient. The calculation of the gradient can be done in any method, but the pressure calculation requires too much data to be sent so it only makes sense in a shared-memory implementation.

### 2.3.2. Benchmark

In this project we use the forward method once to generate a subsurface model. This requires the input of $\chi$ values for which a layered composition was chosen. The values used ranges from 0 (as background) to 0.18. The layers are embedded with a constant acoustic velocity of $c_0 = 2000 m/s$. An example of this subsurface is displayed in the following image.

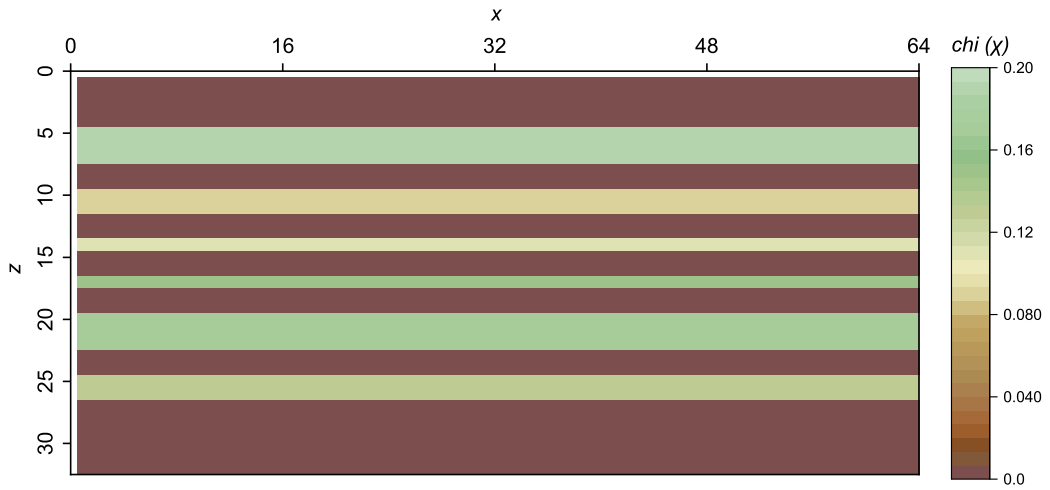


Figure 2.2: Subsurface layer used to generate the initial pressure field used as input for the FWI process.

On top of the surface, 17 sources and 17 receivers are placed. The number of frequency groups used is 15, between 10 Hz en 40 Hz. Using these parameters, two benchmarks can be executed by comparing the methods of parallelization. The first benchmark is the comparison between both methods with a varying number of threads or nodes. The original, single-core application, can be used as a reference. The second benchmark can be executed with different grid sizes. Similar subsurface models are generated with varying number of grid points. These grid sizes varied from 1000-7000 grid points.

# 3

# Results and Conclusions

The completed inversion process results in estimated values for the subsurface. In the following figure the results for the inversion process are displayed as well as the difference between the generated and estimated model. The process is started with a grid that contained $\chi = 0$ at every point.

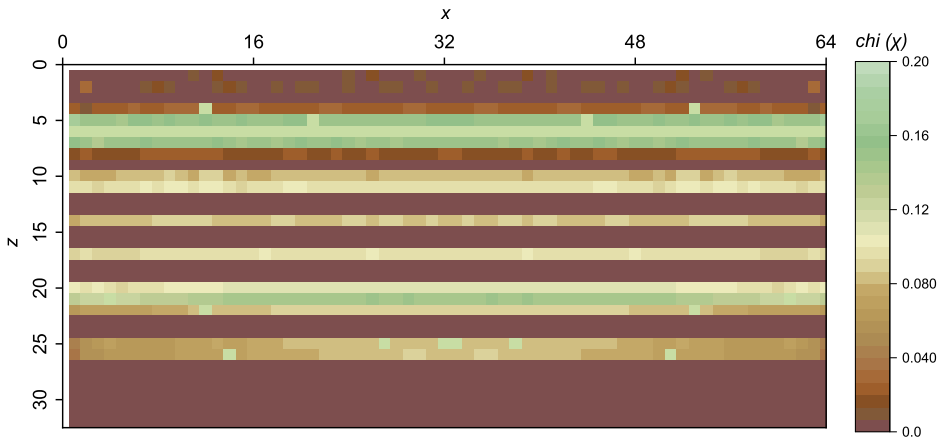


Figure 3.1: Full waveform inversion process executed using the scattering integral and conjugate gradient method. The resultant chi is displayed in this graph.

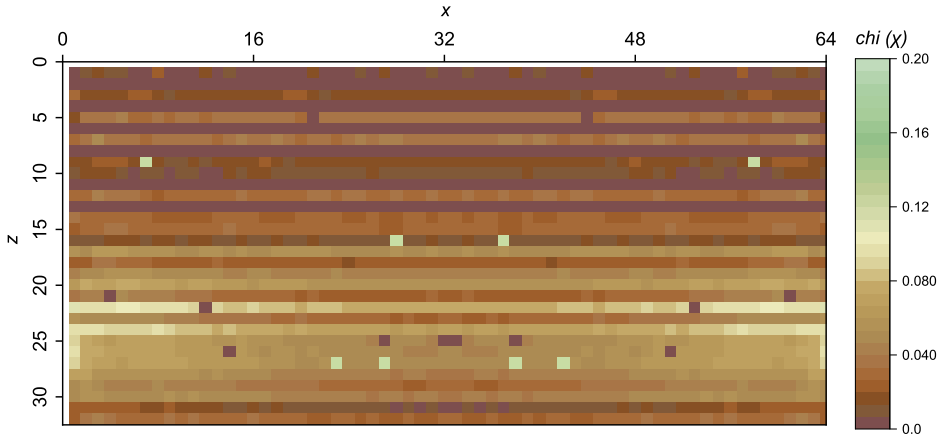


Figure 3.2: Full waveform inversion process resultant between the estimated and calculated chi values.

## 3.1. Time mapping

The start point of implementation started by finding the best locations for parallelization. Some parts of the program are unable to use parallelization such as the reading of files and initialization of the variables. System logs were used to separate the different parts of the program and estimate the time. Using this method a complete run of the program was inspected and categorized on the ability to parallelize.

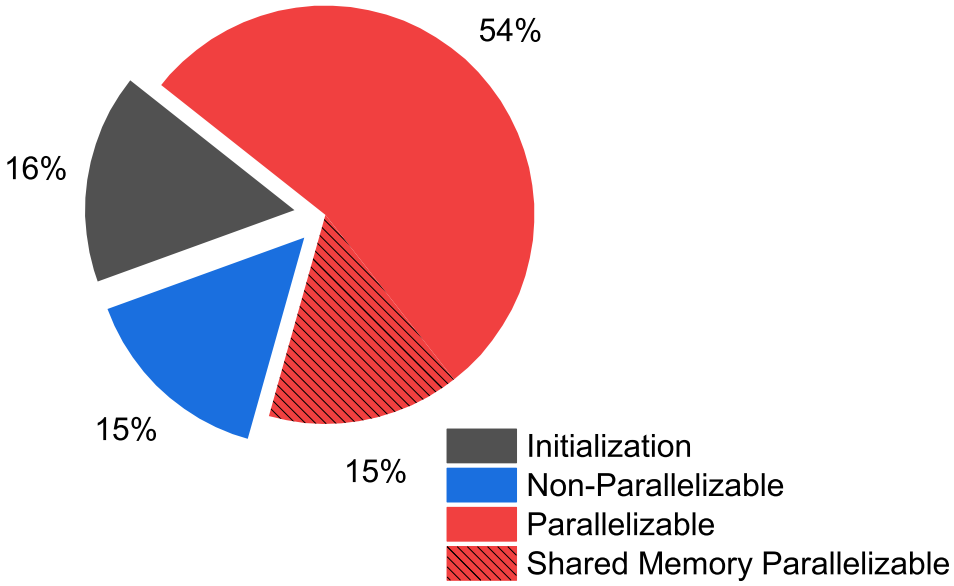


Figure 3.3: Full wavefrom inversion total calculation time breakdown. The time for initialization before the process, and the non-parallelizable part of the calculation are sequential calculations that would not benefit from parallelization for a total of 31.3%.

In Figure 3.5 the total time of the full waveform inversion process is mapped and labelled. This process was executed by calculating the total time of the inversion process as well as the individual processes. The assumption was made that the measurement did not have an impact on the calculation time. Before the actual inversion process starts, 16.2% of the total time is spent on the initialization of variables and reading/writing to and from files. The actual inversion calculation takes up 83.8% of the time.

The calculations themselves are divided into three categories. Inside the program, several calculations are made in sequential order. The forward and inversion process contains several steps for logging and keeping track of data. These general and simple calculations are not as easily divided up into different threads. No speed enhancements can be made here and if tried will likely result in an increase in time rather than reducing it. These calculations are categorized as 'Non-Parallelizable' and take up 15.1% of the inversion process.

The other 68.7% of the calculation time has the potential of parallelization. In this case, the goal is to distribute the work among different threads or nodes. Distribution of work means distribution and gathering of data. For some tasks, many variables are necessary to calculate a result. These tasks are categorized as 'Shared Memory Parallelizable' which will only profit from parallelization when the memory between these objects are shared. This is the case in the forward model where $p_{data}$ is calculated. This function takes up 14.9% of the calculation time. That leaves 53.8% of (almost) independent calculation time. This percentage is traced back to the inversion process where the conjugate is calculated for each grid point. By sending the appropriate data, the nodes can calculate the result and send back a single variable. For this reason, this process is seen as generally well parallelizable.

Combining this data, the maximum amount of speedup can be calculated for each specific method. The actual speedup could be lower depending on the time it takes to share/send data to and from the different threads.

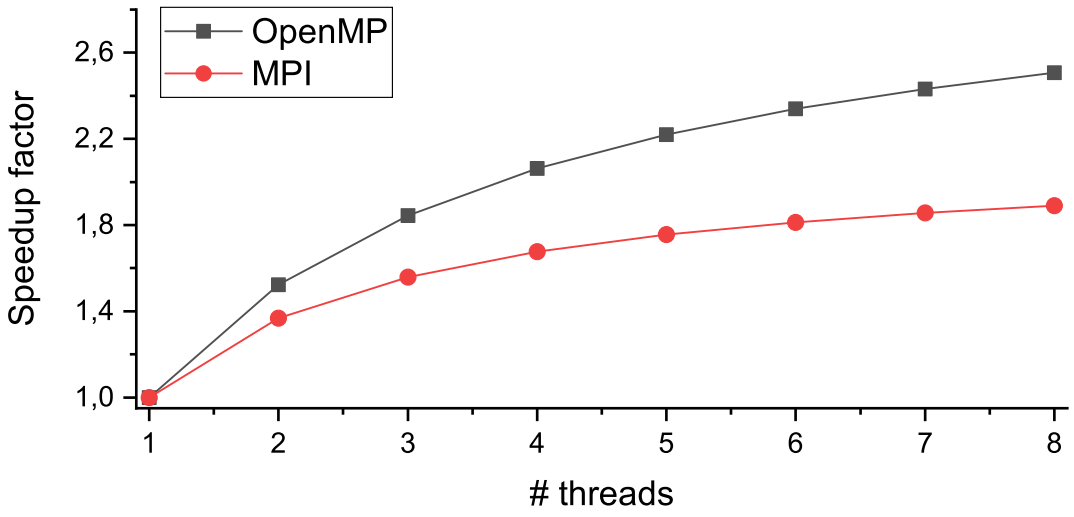


Figure 3.4: Full waveform inversion theoretical speedup factors. The maximum speedup factor for OpenMP is 2.5 for 8 threads, for MPI it is calculated at 1.9 for 8 threads.

In Figure 3.4 the speedup factor for both parallelization methods is displayed. These values are calculated using Equation 2.10. OpenMP uses shared memory parallelization, while MPI does not. This results in a higher speedup factor for OpenMP. The test setup has a maximum amount of threads and thus it is not calculated further than eight threads. From the graph, it can be concluded that the maximum amount of speed up expected is 2.5 for OpenMP and 1.9 for MPI. When these calculations are performed on a supercomputer or in a cluster, the graph of speed up will converge at 3.2 for OpenMP and 2.2 for MPI.

## 3.2. Parallelization

From the time mapping, different parallelization methods could be implemented. The two parallelizable categories time take up a single function. For MPI only the category parallelizable has been implemented, while for OpenMP both categories have been implemented.

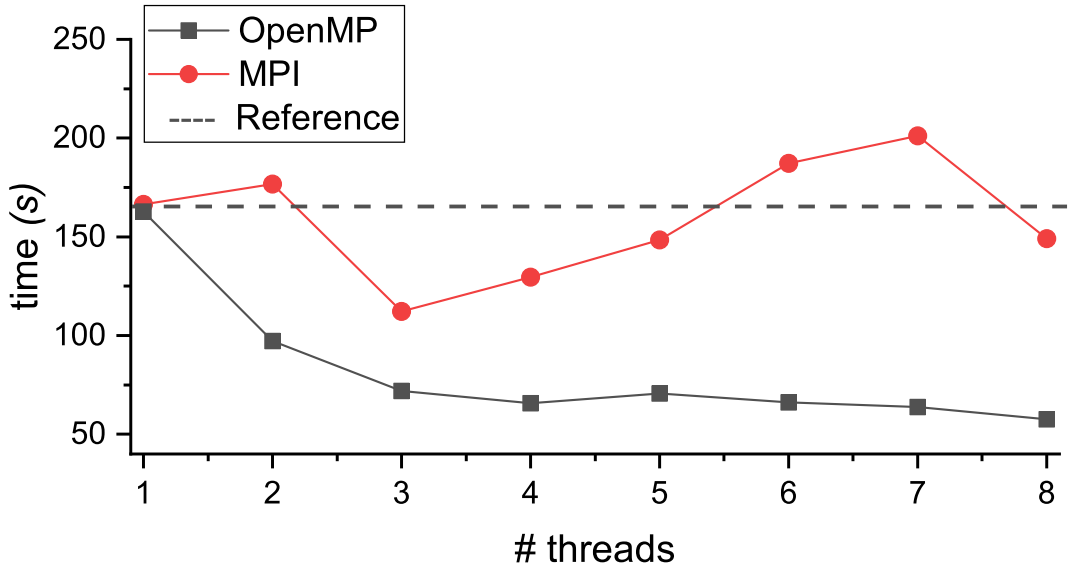


Figure 3.5: Inversion process run on a single-core, OpenMP and MPI implementation. The grid contained 5000 points.

In Figure 3.5 the total calculation time for each method is displayed. Each method has been executed with a varying number of threads or nodes. The dotted horizontal line displays the time it takes for the single-core version to complete the FWI process. For OpenMP the trend is a general speed up with an increasing number of threads. The method MPI does not show a clear pattern. For MPI it seems that with fewer nodes a faster time can be expected. The reason for this could be traced back to the implementation. For the inversion process, a calculation is done over a for loop over the frequencies. For the 15 frequencies, three and five threads will have the most equal distribution among the nodes. Taking into account that sending and receiving data to more nodes takes additional time, it is expected that with three nodes there is an equal distribution in combination with minimal sending and receiving of data.

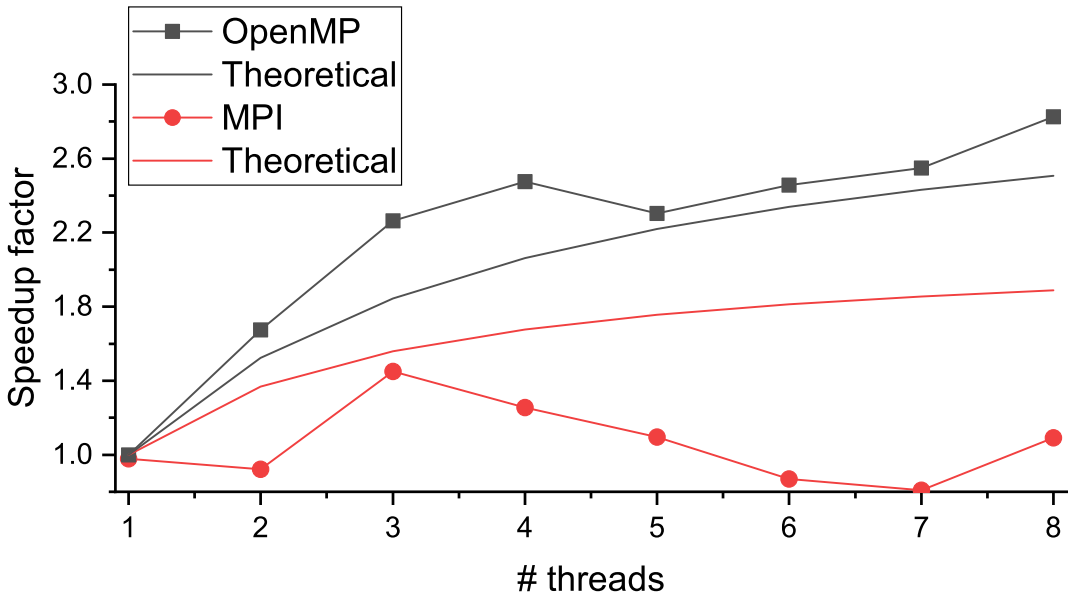


Figure 3.6: Speedup factor for the parallelization. The maximum speedup factor for OpenMP is at 2.7 for 8 threads and for MPI at 1.4 at 3 threads

In Figure 3.6 the speedup factors for both methods are displayed. Alongside the values for the previous calculated theoretical values using Amdahl's law. For the OpenMP implementation, it seems that it is always slightly faster than the theoretical values predicted. For MPI the opposite is the case, where it never exceeds or comes close to the expected values. The trend of the line does also not follow the expected speedup. In the following subsections, the results for each method will be highlighted.

### 3.2.1. OpenMP

For the implementation of OpenMP, a comparison between different grid sizes has been made. This was done for eight threads, as they resulted in the largest speedup factor. In the following figure the time for the number of grid points has been displayed

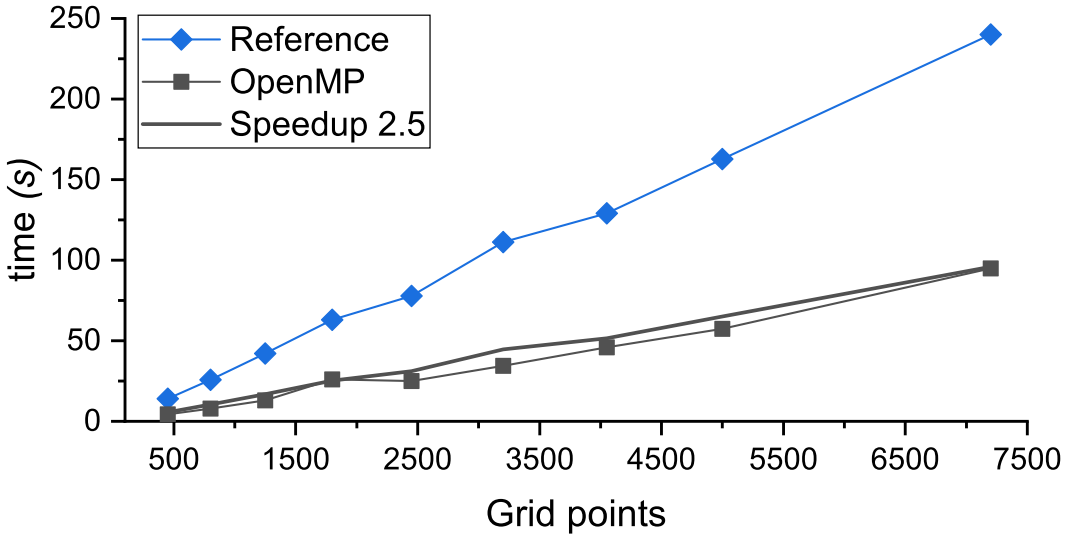


Figure 3.7: FWI process using the OpenMP implementation for different grid points. The single core application is displayed as reference.

In Figure 3.7 the time it takes for the calculation is set out for the OpenMP implementation. From the figure it is clear that both the reference and the parallelization method show a straight line. This indicates that the program scales linearly with the number of grid points. The scaling factor scales like $\mathcal{O}(n)$, with n the number of grid points. This conclusion is logical as each grid point is calculated independently.

The end result of OpenMP was compared to that of the single-core application. There was no difference in the result between them both except for time.

### 3.2.2. MPI

For the implementation of MPI a comparison is made for different grid sizes. Since the MPI implementation showed the best results for three threads, the program was run at that number for each grid.

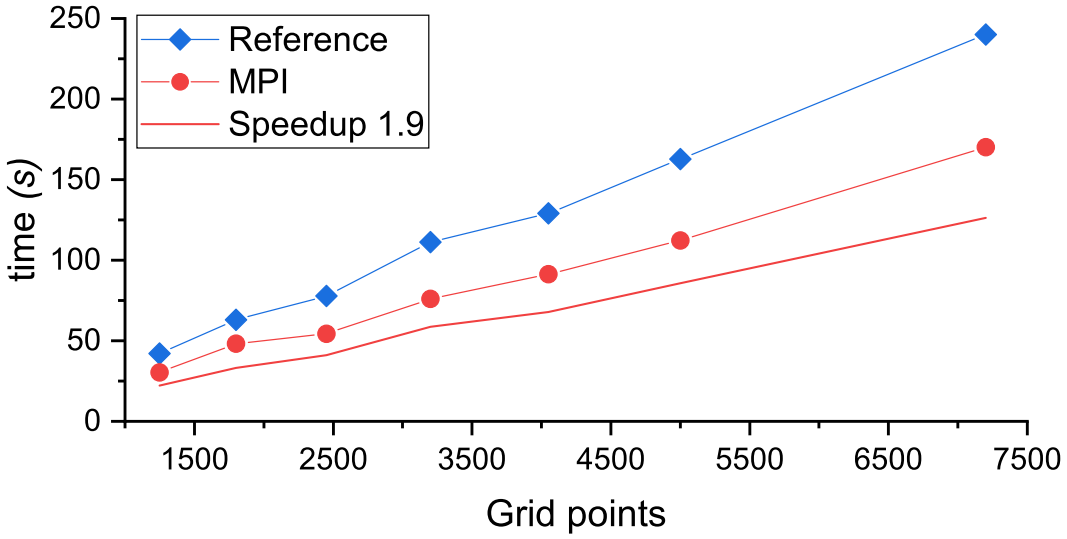


Figure 3.8: FWI process using the MPI implementation for different grid points. The single core application is displayed as reference.

In Figure 3.8 the MPI implementation for different grid sizes is displayed. The reference time for the single-core application is included. Similar to OpenMP shows the MPI implementation a constant gain in speed. This was expected from the found speedup factor from Amdahl's law. From the graph it can be shown that the MPI application scales linearly with $\mathcal{O}(n)$.

The calculated results with MPI show a difference from the single-core application. Depending on the number of nodes, a different result is calculated.

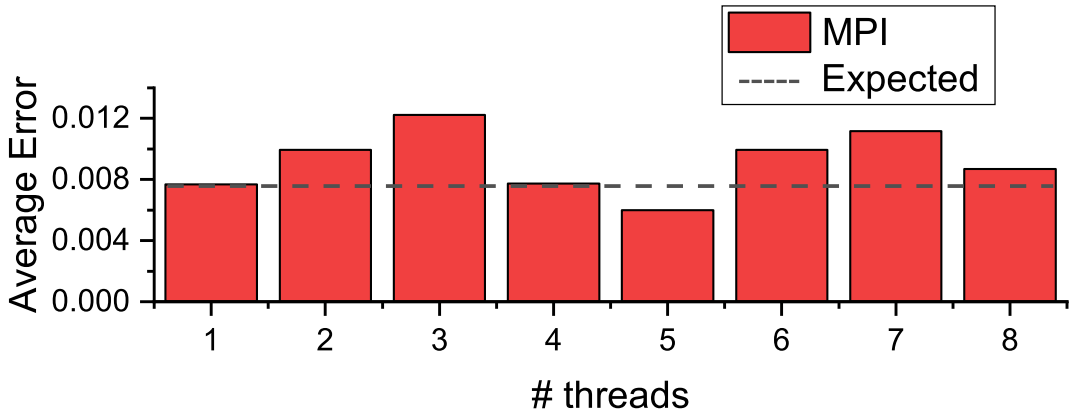


Figure 3.9: Average error using the MPI implementaiton for different amount of nodes. The difference between the expected value indicates loss of information

In Figure 3.9 the average error is displayed for the MPI implementation. From the figure it can be concluded that for a varying number of nodes a different result is calculated. The main reason could be pointed to a loss of information during the sending and receiving of information. The implementation sends all the required data to the nodes for them to calculate the answer and the node sends back the results. It is expected that somewhere in this protocol, not all information is correctly transmitted

# 4

# Discussions

The opportunity was given by Alten to investigate the parallelization implementation in their internal project. The assignment was broadly formulated such that I could make the most decisions on my own. The only constraint was that it should work with their current implementation and should function as an extension. For both methods, MPI and OpenMP, a working implementation has been built. These methods can eventually function as a reference for consultants at Alten interested in the design of parallelization.

### OpenMP

The OpenMP implementation is a straightforward implementation. Using the data structures inside the current program required some modifications for it to work. In the end, the result of OpenMP is a fast, reliable method of calculating shared-memory applications.

Using Amadahl's law the maximum speed up factor was calculated for OpenMP. The results, however, showed that the speedup factor always outperformed that of the calculation. This is displayed in Figure 3.6. The reason for this could be the fact that the time mapping itself took a significant amount of time. The assumption was made that logging time did not affect the calculation time. By fitting Amadahl's law again with the speedup factors found results in a varying contribution percentage of the program affected by parallelization. The $p$ values in Equation 2.10 then range from 71% to 84% for the different number of threads. The $p$ value is considered as a constant and should not vary at all, not even between threads, a look is taken at the value $s$. This value is considered as the maximum speedup factor. The idea behind OpenMP that each individual calculation is split up, which means in our case that 4000 calculations are distributed among the cores. In the single-core program all the calculations inside the loop have to be completed sequentially, but with OpenMP the different threads can already continue to the next calculation. This may allow a more efficient use of the calculation time. A value of $n = 2$ results in a value for $s = 2.4$ using Amadahl's law.

### MPI

The MPI implementation was a more involved implementation than OpenMP. Where the latter was already built into the compiler, MPI required the installation of several packages. Using these packages the actual implementation became simpler, as no changes needed to be made to the current data structures. With the accompanying application 'mpirun', the method was easily tested similar to OpenMP.

The result from MPI was different from the single-core application. No reason behind this could be found as all the information was sent to and from the nodes. The implementation was much slower than expected from Amdahl's law. Using three nodes resulted in the largest speedup factor. This can be traced back to the equal division of tasks over the nodes. More nodes resulted in fewer calculations per node, but an increase in sending, receiving and combining of results. There were some attempts done to reduce the overhead. For example, the data sent was split into smaller data blocks. This resulted in a longer calculation time, due to the splitting of the data taking more time than sending the complete data block. It was also tried to combine the results on each node, each node sending it down the 'chain'. This also did not affect the result or time of calculation. The conclusion should be that the implementation of MPI to decrease calculation time is not suitable for such a small application as the overhead will dominate the advantage of parallelization.

# References


[1] Jean Virieux and Stéphane Operto. “An overview of full-waveform inversion in exploration geophysics”. In: *Geophysics* 74 (Nov. 2009), WCC1–WCC26. DOI: 10.1190/1.3238367.

[2] A Bamberger, Guy Chavent, and P Lailly. “Etude mathématique et numérique ?un problème inverse pour l’équation des ondes à une dimension”. In: (Jan. 1977).

[3] Graham J. Hicks. “Arbitrary source and receiver positioning in finite-difference schemes using Kaiser windowed sinc function”. In: *Geophysics* 67 (Jan. 2002). DOI: 10.1190/1.1451454.

[4] Morten Jakobsen and Bjørn Ursin. “Full waveform inversion in the frequency domain using direct iterative T-matrix methods”. In: *Journal of Geophysics and Engineering* 12.3 (May 2015), pp. 400–418. DOI: 10.1088/1742-2132/12/3/400. URL: https://doi.org/10.1088/1742-2132/12/3/400.

[5] Peter Haffinger. *Seismic broadband full waveform inversion by shot*. s-Hertogenbosch: Boxpress, 2012. ISBN: 978-90-8891-559-8.

[6] David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface*. RISC-V edition. OCLC: ocn993666159. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018. ISBN: 9780128122754.

[7] *OpenMP About Us*. July 2018. URL: https://www.openmp.org/about/about-us/.

[8] *Open MPI Versions*. URL: https://www.open-mpi.org/software/ompi/v4.1/.

[9] URL: https://www.boost.org/doc/libs/1_76_0/doc/html/mpi/tutorial.html.