In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission for extensive copying of my thesis for scholarly purposes may be granted by the Dean of the Graduate School, Dean of my academic division, or by the University Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

High Order Parallel FFT-Type Algorithms

Ronald Gonzales

A thesis submitted in partial fulfillment of the

requirements for the degree of Master of Science in the

Department of Mathematics and Statistics Idaho State University

Fall 2017

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of Ronald Gonzales find it satisfactory and recommend that it be accepted.

Dr. Yury Gryazin,
Major Adviser

Dr. Tracy Payne,
Committee Member

Dr. Glenn Thackray,
Graduate Faculty Representative

Contents

List of Figures

List of Tables

Abstract

In recent years the progress of parallel technologies on both personal computers and large clusters renders sequential numerical algorithms a thing of the past. The following thesis will establish how the use of parallel programming is essential in comparison to running calculations sequentially on a single processor. This will be demonstrated by calculating the approximate solution to the three dimensional Helmholtz equation using multiple central processing units and a combination of both central and graphics processing units. The second and fourth order compact schemes for approximating solutions to the three dimensional Helmholtz equation are developed. In addition, the resulting algorithms are shown. Finally, a presentation of the results of the parallelization of these algorithms from both Idaho National Laboratories computer clusters and personal computers is provided.

Chapter 1

Introduction

This thesis will consider parallel algorithms for the approximate solution of the two and three dimensional Helmholtz equations. These are discretized by high-order compact finite-difference schemes. The matching order compact non-reflecting boundary conditions are applied to preserve the high accuracy of the numerical solution. This thesis does not present the convergence of these algorithms. For detail on the convergence see [10]. The methods developed here can be applied to Krylov-FFT type high-resolution algorithms for subsurface electromagnetic scattering problems.

The focus of this thesis is to demonstrate efficient parallel implementation of the proposed algorithms for approximating the solution to the three-dimensional Helmholtz equation in both a shared and distributed memory environment of a machine's central processing unit (CPU). In addition, the efficiency will be examined on a machine's graphics processing unit (GPU) that has its own memory. The parallelization of the algorithms used to approximate the solution will demonstrate the necessity of parallel computing in numerical solvers.

The high resolution of the iterative method is achieved by two contributing factors. Firstly, the application of a higher order scheme, the standard fourth order Padé approximation [2]. Note that this scheme is not restricted to the use of uniform grid size. The other contributing factor is the ability to use finer computational grids due to the increased computational power of computer clusters.

A parallel algorithm refers to either an algorithm or section of an algorithm, in which, an individual calculation is completely independent of all other calculations within that algorithm or section. In theory, many computationally expensive parallel algorithms can see a linear speed up in calculation time with an increase in the number of processors. That is, if the number of processors is doubled, then the work done by each is cut in half. This would ideally reduce the calculation time to half of its original time. The reduction in calculation time will be referred to as acceleration throughout this paper. This perfectly linear acceleration is typically not observed in practice as the increase in number of processors also increases the amount of communication between these processes. Thus, there is a limit to the acceleration as the latency, the delay due to communication, can outweigh the benefit of reducing the number of computations done by a process. Regarding acceleration of computation time, the hardware called a GPU can accelerate without the addition of multiple processors. Therefore, it is logical to examine the use of three parallel technologies: Open Multi-Processing (OpenMP) for shared memory, Message Passing Interface (MPI) for distributed memory and Compute Unified Device Architecture (CUDA) for use of GPUs.

In the area of high performance computing there are typically two programming languages used, FORTRAN and C. In the development of programs for the algorithms considered in this thesis, C is utilized. For reasons explained in detail later, these algorithms require the use of Fast Fourier Transforms (FFT). These transforms were calculated using an open source C subroutine library developed at Massachusetts Institute of Technology, namely FFTW [4]. This subroutine is currently considered the standard in FFT calculation. It should be noted that the number of grid points used in this thesis are powers of two. This was done to use FFT as efficiently as possible to better examine the effects of the parallel tools.

OpenMP is a tool that can automatically divide the calculation among a set number

of threads. A thread is a separate processing unit that can access the same memory as all other threads, yet all threads can do separate computations simultaneously. On the surface OpenMP is easily implemented into C code. However, one has to make several considerations. That is, the section of code must truly be parallelizable and a phenomenon known as a race condition must be avoided. A race condition is the updating of a variable's value in the incorrect order. OpenMP makes use of shared memory architecture and thus allowing every thread to access the same variables in the computation. The use of shared memory in OpenMP contributes to its value, but also presents limitations. Since OpenMP requires all threads to have access to the same memory it can only be used on a single computer or node. A node is a single computer within a cluster of computers. Therefore, OpenMP is a great tool for parallelizing relatively small computations. However, once computations require more memory than is available on one machine, other tools are required.

MPI is currently the standard for message, or data, passing between processors. The most important aspect of this is that the processors do not need to share memory and therefore can be on separate nodes. This is ideal for large scale computations that would overload the memory of a single node. These computations can then be run on several nodes simultaneously.

The final parallel computing platform implemented is CUDA. This tool allows the use of GPUs for computations. The benefit of a GPU relative to a CPU is that it can perform multiple calculations simultaneously, that is, performing the calculations as vector operations. It should be noted that this is not as widely available as the previous tools, as it requires a specific architecture. To use CUDA the machine must have a GPU with its own memory to run computations entirely separately from the CPU. This structure is common among large clusters that have GPUs, but not on personal computers.

In this thesis, parallelization will refer to the dividing of a for-loop's iterations among separate processing units. In a shared memory environment a processing unit will be referred to as an OpenMP thread or thread. In a distributed memory environment this will be called a MPI process or process. When working with GPUs there is shared memory with respect to the device itself. To avoid ambiguity, a processing unit on a GPU will be referred to as a CUDA thread.

To begin, this thesis will investigate a relatively simple two dimensional problem similar to the three-dimensional Helmholtz equation, the Laplace equation. This will be followed by the derivation of the second and fourth order schemes for approximating the solution to the three dimensional Helmholtz equation. The next chapter will give details of the sequential algorithm prior to any parallelization. Finally, a detailed demonstration of the parallelization of the schemes using all three parallel technologies, OpenMP, MPI and CUDA, is provided.

Chapter 2

Two Dimensional Test Problem

2.1 Introduction

This chapter will derive the discretization of the two dimensional Laplace equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 1, \text{ on } \Omega, \tag{2.1}$$

where $\Omega = [x_0, y_0] \times [x_1, y_1] = [0, 1] \times [0, 1]$ is the two dimensional domain with boundary conditions on $\partial\Omega$, defined by $u(0, y) = u(1, y) = \frac{1}{4}y(y - 1)$ and $u(x, 0) = u(x, 1) = \frac{1}{4}x(x - 1)$. These boundary conditions were chosen for simplicity. The three previously mentioned parallel technologies will be implemented with this problem to find the best approach for implementing them in the three dimensional problem.

Let N_x and N_y be the number of steps in the x and y directions respectively. Also, let $h_x = (x_1 - x_0)/(N_x + 1)$ and $h_y = (y_1 - y_0)/(N_y + 1)$ be the number of grid steps in the x and y directions respectively. Then the computational grid is defined as

$$\Omega_G = \{(x_i, y_j) \mid x_i = x_0 + ih_x,\ y_j = y_0 + jh_y, 1 \leq i \leq N_x,\ 1 \leq j \leq N_y, \}.$$

Taylor expansion gives

$$u_{i\pm1,j} = u_{i,j} \pm \frac{\partial u_{i,j}}{\partial x} h_x + \frac{1}{2}\frac{\partial^2 u_{i,j}}{\partial x^2} h_x^2 \pm \frac{1}{6}\frac{\partial^3 u_{i,j}}{\partial x^3} h_x^3 + \frac{1}{24}\frac{\partial^4 u_{i,j}}{\partial x^4} h_x^4 \pm \dots$$

where $u_{i,j} = u(x_i, y_j)$.

Then in the addition of $u_{i-1,j}$ and $u_{i+1,j}$ the odd terms will cancel, that is

$$u_{i-1,j} + u_{i+1,j} = 2u_{i,j} + \frac{\partial^2 u_{i,j}}{\partial x^2} h_x^2 + \mathcal{O}(h_x^4).$$

So
$$-\frac{\partial^2 u_{i,j}}{\partial x^2} h_x^2 = -u_{i-1,j} + 2u_{i,j} - u_{i+1,j} + \mathcal{O}(h_x^4)$$

$$\frac{\partial^2 u_{i,j}}{\partial x^2} = \frac{1}{h_x^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) + \mathcal{O}(h_x^2).$$

The same can be done with the expansion with respect to y. From these, the following second order scheme for the approximation of (2.1) can be obtained:

$$\begin{aligned}
\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} &= 1 \\
\frac{h_y^2}{h_x^2}(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) + u_{i,j-1} - 2u_{i,j} + u_{i,j+1} &= h_y^2 \\
\left(-2 - 2\frac{h_y^2}{h_x^2}\right) u_{i,j} + \frac{h_y^2}{h_x^2}(u_{i-1,j} + u_{i+1,j}) + u_{i,j-1} + u_{i,j+1} &= h_y^2, \qquad (2.2)
\end{aligned}$$

for $1 \le i \le N_x$ and $1 \le j \le N_y$. Let $U_j = \begin{bmatrix} u_{1,j} & u_{2,j} & \cdots & u_{N_x,j} \end{bmatrix}^T$, let $F_j = \begin{bmatrix} h_y^2 - \frac{h_y^2}{4h_x^2} y(y-1) & h_y^2 & \cdots & h_y^2 & h_y^2 - \frac{h_y^2}{4h_x^2} y(y-1) \end{bmatrix}^T \in \mathbb{R}^{N_x}$ and let $A_1 = (a_{n,m}) \in \mathbb{R}^{N_x \times N_x}$ be such that $a_{n,m} = 1$ if $n - 1 = m$ or $n + 1 = m$, and $a_{n,m} = 0$ otherwise. If A is defined by

$$A = \frac{h_y^2}{h_x^2} A_1 + \left(-2 - 2\frac{h_y^2}{h_x^2}\right) I$$

then (2.2) can be written as

$$U_{j-1} + AU_j + U_{j+1} = F_j. \qquad (2.3)$$

Theorem 2.1.1. Let $B = (b_{ij}) \in \mathbb{R}^{N_x \times N_x}$. Suppose $b_{i,j} = 1$ when $i + 1 = j$ or $i - 1 = j$ and $b_{i,j} = 0$ otherwise. Let $\beta_n^l = \sin\left(\frac{n\pi}{N_x+1} \cdot l\right)$ where $1 \le n, l \le N_x$. Define $v_l = \begin{bmatrix} \beta_1^l & \beta_2^l & \cdots & \beta_{N_x}^l \end{bmatrix}^T$. Then v_l is an eigenvector of B with corresponding eigenvalue $\lambda = 2\cos\left(\frac{\pi}{N_x+1} \cdot l\right)$.

Proof. Let B, v_l, λ and β_n^l be defined as above. It needs to be shown that $Bv_l = \lambda v_l$. Also, recall the trigonometric identity $2\sin(x)\cos(y) = \sin(x-y) + \sin(x+y)$ for $x, y \in \mathbb{R}$

[8]. Then

$$\begin{aligned}\lambda\beta_n^l &= 2\cos\left(\frac{\pi}{N_x+1}\cdot l\right)\sin\left(\frac{n\pi}{N_x+1}\cdot l\right)\\ &= \sin\left((n-1)\frac{\pi}{N_x+1}\cdot l\right)+\sin\left((n+1)\frac{\pi}{N_x+1}\cdot l\right)\\ &= \beta_{n-1}^l+\beta_{n+1}^l.\end{aligned}$$

Note that $\beta_{N_x+1}^l = 0 = \beta_0^l$. Thus

$$Bv_l = \begin{bmatrix}\beta_2^l & \beta_1^l+\beta_3^l & \cdots & \beta_{N_x-2}^l+\beta_{N_x}^l & \beta_{N_x-1}^l\end{bmatrix}^T = \lambda v_l. \qquad \square$$

Theorem 2.1.2. Let v and λ_i be an eigenpair of the matrix A_i for $1 \le i \le n$. Then $\sum_{i=1}^{n}\lambda_i$ is an eigenvalue of the matrix $\sum_{i=1}^{n}A_i$ with corresponding eigenvector v.

Proof. Let A_i, λ_i and v be as defined above. Then

$$\left(\sum_{i=1}^{n}A_i\right)v = \sum_{i=1}^{n}A_iv = \sum_{i=1}^{n}\lambda_iv = \left(\sum_{i=1}^{n}\lambda_i\right)v. \qquad \square$$

Now consider the eigenpairs, that is, the corresponding eigenvalues and eigenvectors, of A. Note that any nonzero vector is an eigenvector of the identity matrix. Thus $\left(-2-2\frac{h_y^2}{h_x^2}\right)$ and any nonzero vector in $\mathbb{R}^{N_x}$ is an eigenpair of $\left(-2-2\frac{h_y^2}{h_x^2}\right)I$. Hence for each $1 \le l \le N_x$ the vector v_l from Theorem 2.1.1 is an eigenvector of A_1 with corresponding eigenvalue $2\cos\left(\frac{\pi}{N_x+1}\cdot l\right)$. It follows that $2\frac{h_y^2}{h_x^2}\cos\left(\frac{\pi}{N_x+1}\cdot l\right)$ and v_l form an eigenpair of the matrix $\frac{h_y^2}{h_x^2}A_1$. Let $\lambda_l = \left(-2-2\frac{h_y^2}{h_x^2}\right)+2\frac{h_y^2}{h_x^2}\cos\left(\frac{\pi}{N_x+1}\cdot l\right)$. Then λ_l and v_l form an eigenpair of A by Theorem 2.1.2.

Lemma 2.1.3. Let v_l be as in Theorem 2.1.1. Then $||v_l||_2^2 = \langle v_l, v_l\rangle = (N_x+1)/2$ for $1 \le l \le N_x$.

Proof. Let $l \in \{1,\ldots,N_x\}$. Recall the trigonometric identity $2\sin(x)\sin(y) = \cos(x-y)-\cos(x+y)$ [8]. Note that $e^{i\alpha} = \cos\alpha + i\sin\alpha$ where $i = \sqrt{-1}$. Also, consider the

geometric series $\sum_{k=1}^{n} z^k = \frac{1-z^{n+1}}{1-z} - 1$ for $z \in \mathbb{C}$ [1]. Then

$$\begin{aligned}
\langle v_l, v_l \rangle &= \sum_{n=1}^{N_x} \left(\beta_n^l\right)^2 \\
&= \sum_{n=1}^{N_x} \left[\sin\left(\frac{n\pi}{N_x+1}l\right)\right]^2 \\
&= \frac{1}{2}\sum_{n=1}^{N_x} \left[1 - \cos\left(\frac{2n\pi}{N_x+1}l\right)\right] \\
&= \frac{N_x}{2} - \frac{1}{2}\sum_{n=1}^{N_x} \cos\left(\frac{2n\pi}{N_x+1}l\right) \\
&= \frac{N_x}{2} - \frac{1}{2}\sum_{n=1}^{N_x} Re\left(e^{\frac{2n\pi i}{N_x+1}l}\right) \\
&= \frac{N_x}{2} - \frac{1}{2} Re\left[\frac{1-e^{2l\pi i}}{1-e^{\frac{2\pi i}{N_x+1}l}} - 1\right] \\
&= \frac{N_x+1}{2} - \frac{1}{2} Re\left[\frac{1-e^{2l\pi i}}{1-e^{\frac{2\pi i}{N_x+1}l}}\right].
\end{aligned}$$

Now let $\sigma = 1 - e^{2l\pi i}$ and $\eta = 1 - e^{\frac{2\pi i}{N_x+1}l}$. Then

$$\begin{aligned}
Re\left[\frac{1-e^{2l\pi i}}{1-e^{\frac{2\pi i}{N_x+1}l}}\right] &= \frac{Re(\sigma)\cdot Re(\eta) - Im(\sigma)\cdot Im(\eta)}{Re(\eta)^2 + Im(\eta)^2} \\
&= \frac{(1-\cos(2l\pi))\cdot Re(\eta) - (\sin(2l\pi))\cdot Im(\eta)}{Re(\eta)^2 + Im(\eta)^2} \\
&= 0.
\end{aligned}$$

Thus $||v_l||_2^2 = \langle v_l, v_l \rangle = \frac{N_x+1}{2}$ for $l \in \{1, \ldots, N_x\}$. □

Lemma 2.1.4. Let v_l be as in Theorem 2.1.1. If $l \neq l'$ then v_l and $v_{l'}$ are orthogonal vectors.

Proof. Let v_l and $v_{l'}$ be as defined above. Suppose $l \neq l'$. Then

$$\begin{aligned}
\langle v_l, v_{l'} \rangle &= \sum_{n=1}^{N_x} \beta_n^l \beta_n^{l'} \\
&= \sum_{n=1}^{N_x} \sin\left(\frac{n\pi}{N_x+1} \cdot l\right) \sin\left(\frac{n\pi}{N_x+1} \cdot l'\right) \\
&= \frac{1}{2} \sum_{n=1}^{N_x} \left[\cos\left([l-l']\frac{n\pi}{N_x+1}\right) - \cos\left([l+l']\frac{n\pi}{N_x+1}\right)\right] \\
&= \frac{1}{2} \sum_{n=1}^{N_x} \left[Re\left(e^{i\left([l-l']\frac{n\pi}{N_x+1}\right)}\right) - Re\left(e^{i\left([l+l']\frac{n\pi}{N_x+1}\right)}\right)\right] \\
&= \frac{1}{2} Re\left(\sum_{n=1}^{N_x} e^{i\left([l-l']\frac{n\pi}{N_x+1}\right)} - \sum_{n=1}^{N_x} e^{i\left([l+l']\frac{n\pi}{N_x+1}\right)}\right) \\
&= \frac{1}{2} Re\left(\frac{1-e^{i[l-l']\pi}}{1-e^{i\left([l-l']\frac{n\pi}{N_x+1}\right)}} - \frac{1-e^{i[l+l']\pi}}{1-e^{i\left([l+l']\frac{n\pi}{N_x+1}\right)}}\right).
\end{aligned}$$

Now consider the case in which $l - l'$ is even. It follows $l + l'$ is even as well. Thus $\langle v_l, v_{l'} \rangle = 0$ since

$$Re\left(1-e^{i[l-l']\pi}\right) = 1 - \cos([l-l']\pi) = 0 = 1 - \cos([l+l']\pi) = Re\left(1-e^{i[l-l']\pi}\right).$$

Now consider the case in which $l - l'$ is odd and hence $l + l'$ is odd. Then

$$Re\left(1-e^{i[l-l']\pi}\right) = 1 - \cos([l-l']\pi) = 2 = 1 - \cos([l+l']\pi) = Re\left(1-e^{i[l-l']\pi}\right).$$

It follows

$$\begin{aligned}
\langle v_l, v_{l'} \rangle &= Re\left[\frac{1}{1-e^{i\left([l-l']\frac{n\pi}{N_x+1}\right)}} - \frac{1}{1-e^{i\left([l+l']\frac{n\pi}{N_x+1}\right)}}\right] \\
&= Re\left[\frac{1-e^{\frac{i\pi(l+l')}{N_x+1}} - \left(1-e^{\frac{i\pi(l-l')}{N_x+1}}\right)}{\left(1-e^{\frac{i\pi(l-l')}{N_x+1}}\right)\left(1-e^{\frac{i\pi(l+l')}{N_x+1}}\right)}\right] \\
&= Re\left[\frac{e^{\frac{i\pi(l-l')}{N_x+1}} - e^{\frac{i\pi(l+l')}{N_x+1}}}{\left(1-e^{\frac{i\pi(l-l')}{N_x+1}}\right)\left(1-e^{\frac{i\pi(l+l')}{N_x+1}}\right)}\right] \\
&= Re\left[\frac{e^{\frac{i\pi l}{N_x+1}}\left(e^{\frac{-i\pi l'}{N_x+1}} - e^{\frac{i\pi l'}{N_x+1}}\right)}{e^{\frac{i\pi(l-l')}{2(N_x+1)}}\left(e^{\frac{-i\pi(l-l')}{2(N_x+1)}} - e^{\frac{i\pi(l-l')}{2(N_x+1)}}\right)e^{\frac{i\pi(l+l')}{2(N_x+1)}}\left(e^{\frac{-i\pi(l+l')}{2(N_x+1)}} - e^{\frac{i\pi(l+l')}{2(N_x+1)}}\right)}\right] \\
&= Re\left[\frac{e^{\frac{i\pi l}{N_x+1}}\left(e^{\frac{-i\pi l'}{N_x+1}} - e^{\frac{i\pi l'}{N_x+1}}\right)}{e^{\frac{i\pi l}{N_x+1}}\left(e^{\frac{-i\pi(l-l')}{2(N_x+1)}} - e^{\frac{i\pi(l-l')}{2(N_x+1)}}\right)\left(e^{\frac{-i\pi(l+l')}{2(N_x+1)}} - e^{\frac{i\pi(l+l')}{2(N_x+1)}}\right)}\right] \\
&= Re\left[\frac{-2i\sin\left(\frac{\pi l'}{N_x+1}\right)}{2i\sin\left(\frac{\pi(l-l')}{2(N_x+1)}\right)\cdot 2i\sin\left(\frac{\pi(l+l')}{2(N_x+1)}\right)}\right] \\
&= Re\left[\frac{i\sin\left(\frac{\pi l'}{N_x+1}\right)}{2\sin\left(\frac{\pi(l-l')}{2(N_x+1)}\right)\cdot\sin\left(\frac{\pi(l+l')}{2(N_x+1)}\right)}\right] \\
&= \frac{Re\left[i\sin\left(\frac{\pi l'}{N_x+1}\right)\right]}{2\sin\left(\frac{\pi(l-l')}{2(N_x+1)}\right)\cdot\sin\left(\frac{\pi(l+l')}{2(N_x+1)}\right)} = 0.
\end{aligned}$$

Thus $\langle v_l, v_{l'} \rangle = \sum_{n=1}^{N_x} \sin\left(\frac{n\pi}{N_x+1}\cdot l\right)\sin\left(\frac{n\pi}{N_x+1}\cdot l'\right) = 0$ when $l \neq l'$. □

These lemmas show the relationship between these eigenvectors with respect to their inner product. Note that Lemma 2.1.4 holds for the normalized eigenvectors. Now consider the following theorems that are essential to the derivation of this two dimensional scheme.

Theorem 2.1.5. Let v_l be as in Theorem 2.1.1. Define $w_l = ||v_l||^{-1}v_l$. Let $V = \begin{bmatrix} w_1 & w_2 & \cdots & w_{N_x} \end{bmatrix} \in \mathbb{R}^{N_x \times N_x}$. Then V is an orthogonal matrix.

Proof. Let V be as defined above. It needs to be shown that $V^TV = I$. Note that this implies $VV^T = I$ as a left inverse of a matrix must also be a right inverse [7]. Observe that

$$\begin{aligned} V^TV &= V^T \begin{bmatrix} w_1 & w_2 & \cdots & w_{N_x} \end{bmatrix} \\ &= \begin{bmatrix} V^Tw_1 & V^Tw_2 & \cdots & V^Tw_{N_x} \end{bmatrix} \\ &= \begin{bmatrix} \langle w_1, w_1\rangle & \langle w_1, w_2\rangle & \cdots & \langle w_1, w_{N_x}\rangle \\ \langle w_2, w_1\rangle & \langle w_2, w_2\rangle & & \\ \vdots & & \ddots & \vdots \\ \langle w_{N_x}, w_1\rangle & \langle w_{N_x}, w_2\rangle & \cdots & \langle w_{N_x}, w_{N_x}\rangle \end{bmatrix} \\ &= I \end{aligned}$$

by Lemmas 2.1.4 and 2.1.3. Thus V is orthogonal. □

Theorem 2.1.6. Let $B \in \mathbb{R}^{n\times n}$ for any $n > 0$. For $1 \leq i \leq n$ let v_i and λ_i be eigenpairs of B. Define $V = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix}$. If V is orthogonal then $V^TBV = \Lambda$ where Λ is the diagonal matrix $\begin{bmatrix} \lambda_1 & \cdots & \lambda_n \end{bmatrix} I$.

Proof. Let B, λ_i, v_i and V be defined as above. Then

$$BV = \begin{bmatrix} Bv_1 & \cdots & Bv_n \end{bmatrix} = \begin{bmatrix} \lambda_1 v_1 & \cdots & \lambda_n v_n \end{bmatrix} = V\Lambda.$$

Where Λ is the diagonal matrix $\begin{bmatrix} \lambda_1 & \cdots & \lambda_n \end{bmatrix} I$. Thus $V^TBV = V^TV\Lambda = \Lambda$. □

Let A be as defined in (2.3), $U_{j-1} + AU_j + U_{j+1} = F_j$. Let V and Λ be defined by A as in Theorem 2.1.6. So $V^TAV = \Lambda$. Then (2.3) can be written as

$$\begin{aligned} V^TU_{j-1} + V^TAVV^TU_j + V^TU_{j+1} &= V^TF_j \\ W_{j-1} + \Lambda W_j + W_{j+1} &= \overline{F_j} \end{aligned} \tag{2.4}$$

where $\overline{F_j} = V^T F_j$ and $W_j = V^T U$. This tridiagonal system can be solved using the LU decomposition [6], which yields N_x independent systems

$$\begin{aligned} \lambda_i w_{i,1} + w_{i,2} &= \overline{F_{i,1}} \\ w_{i,j-1} + \lambda_i w_{i,j} + w_{i,j+1} &= \overline{F_{i,j}} \qquad \text{for } j = 2, 3, \ldots, N_y - 1 \\ w_{i,N_y-1} + \lambda_i w_{i,N_y} &= \overline{F_{i,N_y}} \end{aligned}$$

for $i = 1, 2, \ldots, N_x$. The solution can be parallelized with respect to the x direction of the computational domain. Prior to solving this system, $\overline{F}_j = V^T F_j$ must be found. To calculate this transformed vector, consider the eigenvectors that are the columns of V.

Definition 2.1.7. The *discrete sine transform* (DST) of the vector $x = \begin{bmatrix} x_1 \ldots x_n \end{bmatrix}^T \in \mathbb{R}^n$ is given by $\overline{x} = \begin{bmatrix} \overline{x}_1 \ldots \overline{x}_n \end{bmatrix}^T$ where

$$\overline{x}_k = \sum_{l=1}^{n} \sin\left(\frac{\pi k}{n+1} \cdot l\right) x_l$$

for $1 \leq k \leq n$ [6].

Thus the modified right hand side, $\overline{F}_j = V^T F_j$, is simply the DST of the right hand side F_j times a scalar that normalizes the columns of V. To see how this is calculated, consider the following definition. Note that $i = \sqrt{-1}$ and is not an index in the following computations.

Definition 2.1.8. The *discrete Fourier transform* (DFT), of the real vector $y = \begin{bmatrix} y_1 \ldots y_n \end{bmatrix}^T \in \mathbb{R}^n$ is given by $\overline{y} = \begin{bmatrix} \overline{y}_1 \ldots \overline{y}_n \end{bmatrix}^T$ where

$$\overline{y}_k = \sum_{l=0}^{n} e^{\frac{-2\pi i}{n}(l-1)(k-1)} y_l$$

for $1 \leq k \leq n$ [6].

Now consider

$$\overline{y}_k = \sum_{l=0}^{N-1} e^{\frac{-\pi i l k}{N/2}} y_l = \sum_{l=0}^{N-1} \left[\cos\left(\frac{\pi l k}{N/2}\right) - i \sin\left(\frac{\pi l k}{N/2}\right)\right] y_l$$

for $0 \leq k \leq N-1$. Let $N/2 = n+1$, that is $N = 2n+2$. Define $y_0 = 0$, $y_{n+1} = y_{n+2} =$

$\cdots = y_{2n+1} = 0$ and $y_l = x_l$ for $1 \leq l \leq n$. Then

$$-Im(\overline{y}_k) = -Im\left(\sum_{l=0}^{2n+1}\left[\cos\left(\frac{\pi lk}{N/2}\right) - i\sin\left(\frac{\pi lk}{N/2}\right)\right]\right) y_l = \sum_{l=1}^{n}\sin\left(\frac{\pi lk}{n+1}\right) x_l = \overline{x_k}$$

for $1 \leq k \leq n$. Therefore, to find the DST of some $x \in \mathbb{R}^n$ define $y = \begin{bmatrix} y_0 & \dots & y_{2n+1} \end{bmatrix}^T \in \mathbb{R}^{2n+2}$ such that $y_0 = y_{n+1} = y_{n+2} = \cdots = y_{2n+1} = 0$ and $y_l = x_l$ for $1 \leq l \leq n$. Then calculate the DFT of y, namely $\overline{y}$. Then $\overline{x} = \begin{bmatrix} -Im(\overline{y}_1) & \cdots & -Im(\overline{y}_n) \end{bmatrix}^T$. It is essential to note that calculating the DFT and hence the DST, is independent with respect to the y direction and therefore can be parallelized.

Given the DST of the right hand side, (2.4) can be solved. Note that the solution, $W_j = V^T U_j$, is the DST of U_j. So to find U_j consider the following

$$VW_j = VV^T U_j = U_j$$

since V is orthogonal. Note that VW is simply the DST of W. Finding the DST of W will be referred to throughout this paper as the reverse transform and the forward transform will refer to the DST of original the right hand side, F_j. Both the forward and reverse transforms are independent with respect to the y direction in the domain.

In conclusion, to solve (2.1) one must compute the DST of the right hand side using the DFT. Solve the tridiagonal system using the LU decomposition. This step will be referred to as the tridiagonal solver. Then find the reverse transform to that solution. These three steps will collectively be referenced to as the solver. This is outlined in Algorithm 1.

Algorithm 1 Sequential 2D Laplace Solver

1: **for** $k = 1, \ldots, N_y$ **do**
2: 1D forward DST in $x-$direction
3: **end for**
4: **for** $i = 1, \ldots, N_x; j = 1, \ldots, N_y$ **do**
5: Solve the tridiagonal system using LU decomposition
6: **end for**
7: **for** $k = 1, \ldots, N_y$ **do**
8: 1D reverse DST in $x-$direction
9: **end for**

As noted previously the forward and reverse DST calculations are independent with respect to the y direction and the tridiagonal solver is independent with respect to the x direction. Therefore, these three computations must be parallelized separately. Starting with this sequential solver algorithm the parallel tools OpenMP, MPI and CUDA, can now be implemented.

2.2 OpenMP

The solver was first parallelized using OpenMP with the luxury of a shared memory structure. As previously mentioned this tool is relatively easy to implement with careful planning. That is, the calculations within the for-loops must be independent with respect to each iteration. In addition, there must be no possibility of a race condition. In the derivation of the scheme it is apparent that the for-loops in Algorithm 1 are indeed parallelizable. The creation of private variables protects against race conditions. A private variable is a variable that is only accessible to a single thread [3]. It should be noted that only variables subject to race conditions should be made private as the creations of private variables can increase the time required to complete the calculation. Algorithm 2 does not explicitly show the use of private variables, but they are essential for the correct calculation.

OpenMP, by default, automatically divides the iterations of a for-loop evenly across

threads [3]. Unless specified, a program using this tool will run sequentially, on a single thread. The parallel sections of the code must be defined by the programmer with the use of compiler directives. Only the parallel section will be executed on multiple threads. To parallelize a for-loop with OpenMP a directive is placed prior to the loop of interest. Algorithm 2 demonstrates an abridged version of the directives used in the program.

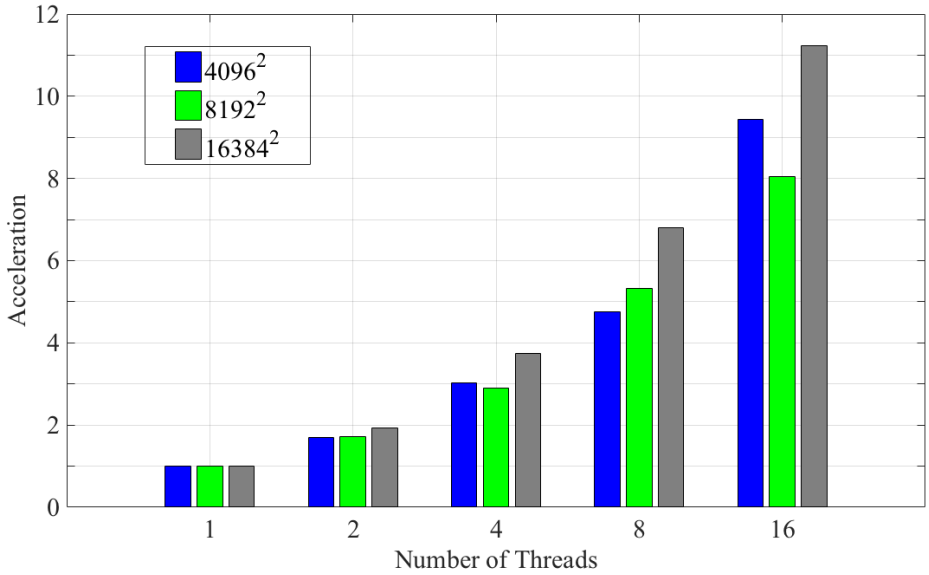
Algorithm 2 OpenMP 2D Laplace Solver

```
1: #pragma omp parallel for
2: for k = 1, ..., N_y do
3:    1D forward DST in x−direction
4: end for
5: #pragma omp parallel for
6: for i = 1, ..., N_x; j = 1, ..., N_y do
7:    Solve the tridiagonal system using LU decomposition
8: end for
9: #pragma omp parallel for
10: for k = 1, ..., N_y do
11:    1D reverse DST in x−direction
12: end for
```

Algorithm 2 was successfully programmed in C and run on Idaho National Laboratories (INL) clusters Falcon and Falconviz as well as several personal computers. The results that follow are those recorded from Falconviz and use uniform computational grids. It should be noted that this program was successfully tested on nonuniform grids as well.

Figure 2.1: 2D OpenMP Acceleration

Acceleration is the Sequential Time Divided by Parallel Time



The graph in Figure 2.1 shows the acceleration from one to sixteen threads, utilizing three separate computational grids $N_x = N_y = 4096$, $N_x = N_y = 8192$, and $N_x = N_y = 16384$. These results are promising. As the number of threads increases, an acceleration of the computation time is observed. This acceleration is approximately linear in all three grid sizes. It should be noted that the acceleration is measured with respect to the computation time with one thread. For example, if the computation time was 40 seconds on one thread and 10 seconds on four threads then the accelerations is $40/10 = 4$.

2.3 MPI

Although Figure 2.1 shows desirable results, OpenMP, by itself, is limited to single machine where the CPU's share memory. For a large enough computational grid the memory required to allocate the necessary arrays can overrun the random access memory (RAM) of the node or machine. Thus there is a need to spread the work across multiple nodes on a cluster. This can be accomplished using MPI. This is a standard-

ized and portable means of passing messages or data between processors whether they share memory or not. The use of this tool, although not as intuitive as OpenMP, should allow for a decrease in computation time while having the capability to use larger computational grids.

Unlike OpenMP, a program utilizing MPI does not have specific sections of parallelization. Each process runs the entire program and only communicates with another when explicitly specified by the programmer [5]. Therefore, the program must be modified to only perform the appropriate calculations and communicate the information back and forth as efficiently as possible. Each process in MPI is assigned a unique positive integer value starting with zero, called a rank. The rank is used to determine which calculations need to be executed on that process.

As an example, consider a for-loop with n iterations running on p MPI processes. For simplicity, assume that p divides n. Let r be the rank of the process. Then the start and end positions are $r \cdot n/p$ and $(r+1) \cdot n/p$ respectively. Thus the for-loop in a typical sequential program would be written as **for** $i = 0, \ldots, n-1$ while the same loop modified for MPI would be written as **for** $i = r \cdot n/p, \ldots, (r+1) \cdot n/p - 1$. The case in which p does not divide n is considered in Chapter 4.

In the implementation of OpenMP the only modified section was shown in Algorithm 2. MPI, however, required modification to the entire program. The arrays were reduced to only allocate the specific length required for each process to perform its respective calculations. In addition, the LU factorization parallelized as described in the previous process. This was not done to accelerate its computation, but to adapt to the reduced length of the arrays. The section of interest, the solver, was parallelized as shown in Algorithm 3.

Algorithm 3 MPI 2D Laplace Solver

1: Find $start_x, start_y, end_x$, and end_y
2: **for** $k = start_y, \ldots, end_y$ **do**
3: 1D forward DST in $x-$direction
4: **end for**
5: Send and receive data via MPI functions
6: **for** $i = start_x, \ldots, end_x; j = 1, \ldots, N_y$ **do**
7: Solve the tridiagonal system using LU decomposition
8: **end for**
9: Send and receive data via MPI functions
10: **for** $k = start_y, \ldots, end_y$ **do**
11: 1D reverse DST in $x-$direction
12: **end for**
13: Send and receive data via MPI functions

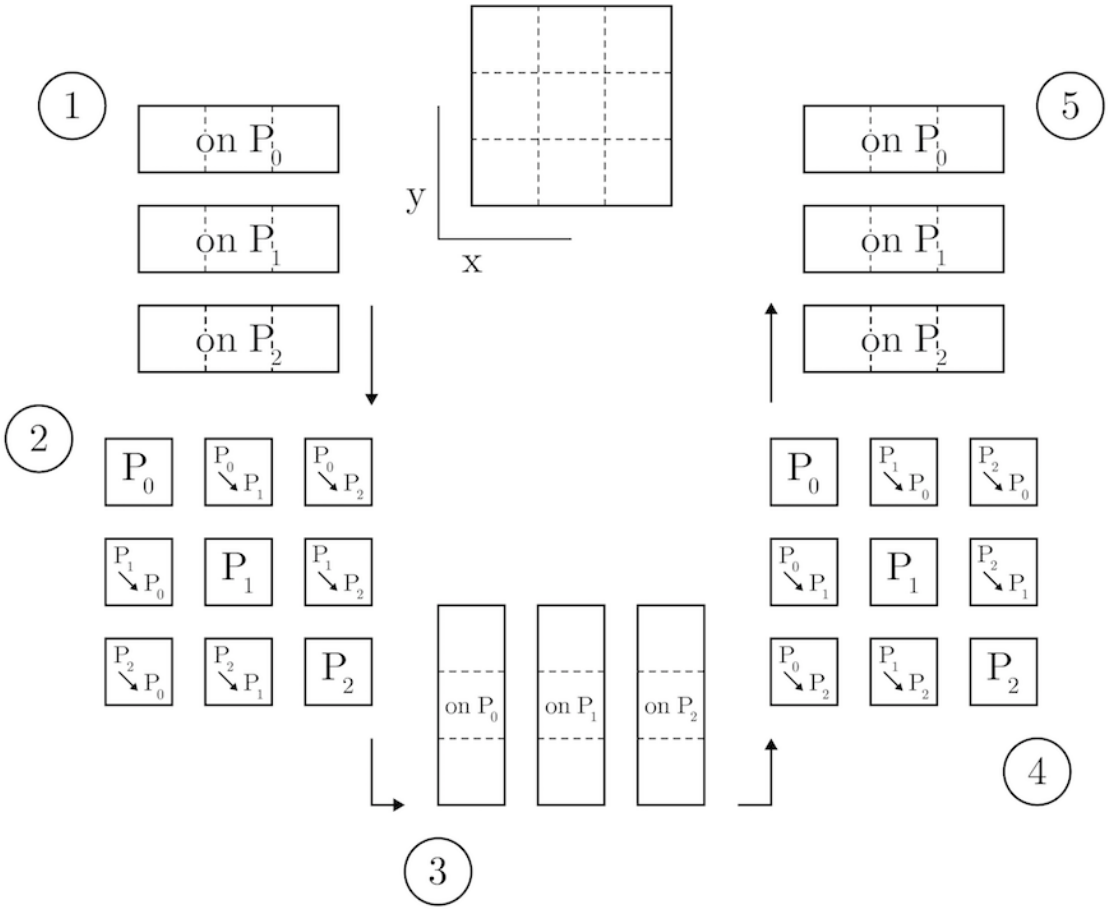
Figure 2.2 gives a visual example of how the transfer of data between processes was achieved. For ease of illustration the figure assumes the use of three processes. The processes are denoted P_0, P_1 and P_2. The computational domain, in its whole, is displayed in the center of Figure 2.2. The first step shows how the domain is divided as evenly as possible among the three processes with respect to the y direction. This is where the forward transform is computed since the calculations do not depend on y.

Once the FFT is computed the domain needs to be divided with respect to the x direction since the tridiagonal solver is independent in this direction. To accomplish this division, parts of the domain need to be sent to different processes. As an example, examine P_0. To solve the tridiagonal system P_0 needs a portion of the information that currently resides on P_1 and P_2. Also, P_0 has information that processes P_1 and P_2 need. To reduce memory and communication only the necessary parts of the arrays are sent and the memory that is no longer needed is released. The second step illustrates the specific sections that need to be sent and their destination process. The sections along the diagonal will not be sent, as they currently reside on the appropriate process.

The third step shows the sections of the domain assembled on the appropriate process after receiving the messages sent in the second step. Now the domain is divided as

evenly as possible among the processes with respect to x direction so that the tridiagonal solver can be executed in parallel. The fourth step is simply the reverse of step two. The fifth and final step has domain divided with respect to the y direction and can now calculate the reverse transform.

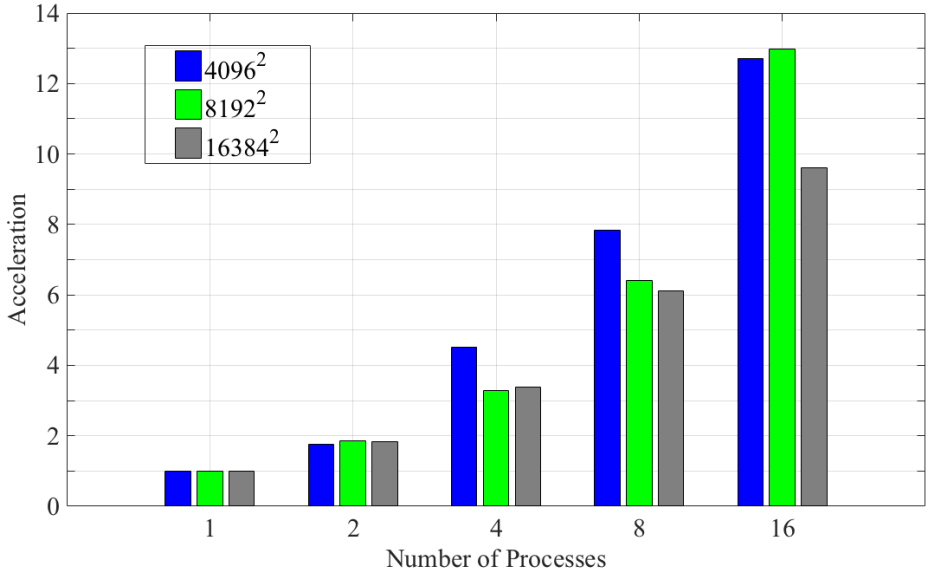
Figure 2.2: 2D Transfer of Information between MPI Processes



This program was run in two ways. Firstly, it was run on one node to compare against OpenMP. Then several nodes were used, testing the ability to utilize very large computational grids. In the case of running on one node or machine, a personal computer can be used. However, to use several, a cluster of machines is needed. As in the case of OpenMP, the MPI program was successfully run on INLs clusters Falcon and Falconviz, as well as several personal computers. The results that follow are those recorded

from Falconviz and use the following uniform computational grids, $N_x = N_y = 4096$, $N_x = N_y = 8192$, and $N_x = N_y = 16384$.

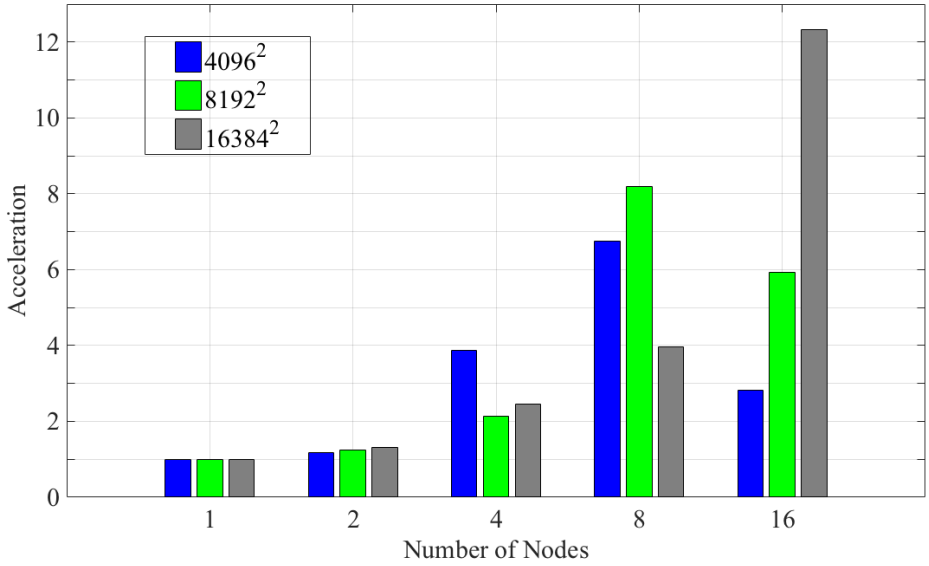
Figure 2.3: 2D MPI Acceleration by Processors on One Node
Acceleration is the Sequential Time Divided by Parallel Time



The graph in Figure 2.3 shows the acceleration on a single node ranging from one to sixteen processes. With respect to the consistency for all three grid sizes these results are not as desirable as those in the implementation of OpenMP. However, the acceleration is approximately linear and the final acceleration is greater than that in OpenMP except for the case of 16384^2.

Figure 2.4: 2D MPI Acceleration by Nodes

Acceleration is the Sequential Time Divided by Parallel Time



For comparison, Figure 2.4 shows the acceleration by use of one to sixteen nodes, rather than processors. This illustrates the addition of more nodes is not necessarily beneficial. Examining the acceleration of the 4096^2 and 8192^2 grids reveals the addition of more nodes actually hindered the performance after eight nodes. This can be explained by the increased latency outweighing the benefit of using more computational processes. The extreme acceleration from eight to sixteen nodes on the 16384^2 grid can be explained by more of the appropriate data fitting in the CPU's cache. Therefore, drastically reducing the amount of latency on an individual node. More detail on a machine's cache is given in Chapter 4.

2.4 CUDA

The last parallel technology considered in this chapter is CUDA. This allows for potential acceleration by utilizing a machines GPU. The benefit of calculating on a GPU is that it can simultaneously carry out hundreds of floating point operations. The exact number

is dependent on the specific GPU. A program can not be executed solely on a GPU. Thus the implementations of CUDA will take the original sequential program and place certain functions on the GPU. A function that is run on a GPU is called kernel function. As previously mentioned, the use of CUDA requires the GPU to have its own memory [3]. This means that the GPU will not be able to access any arrays or variables being used by the CPU. Thus any arrays needed in a kernel function must be allocated on both the CPU and GPU. Then the data from the array on the CPU is copied into the array on the GPU. This processes is reversed after the kernel function is executed so that the CPU can use the data computed by the kernel function.

The algorithm that outlines the implementation of the CUDA program is the same as the sequential program, Algorithm 1. CUDA has designed libraries that include FFT functions that, in theory, are implemented identically to the functions in FFTW libraries [9]. This allowed for the ability to simply change the libraries used to compile the program without changing any code. These functions perform the FFT calculation on a machines GPU rather than a CPU. Unlike OpenMP and MPI, CUDA does not possess the ability to add more CUDA threads. Including a GPU is either all or nothing. For this reason a table is used, rather than, a graph as in the previous sections. The following results were computed using INL's Falconviz.

Grid Size	Acceleration
4096^2	1.136
8192^2	1.126
16384^2	1.376

Table 2.1: 2D CUDA Acceleration
Acceleration is the Sequential Time Divided by Parallel Time

This findings are not as impressive as in the use of MPI and OpenMP. Note that

the use of CUDA's libraries that mimic FFTW accelerates the forward and reverse transforms, but leaves the tridiagonal solver untouched. This suggests that further steps must be taken in the implementation of CUDA for the three dimensional equation.

2.5 Conclusion

This chapter has developed a second order scheme for approximating the solution to the two dimensional Laplace equation. The three parallel technologies, OpenMP, MPI and CUDA, were used to create three variant programs. A comparison of all three was reviewed.

Chapter 3

Discretization

3.1 Introduction

The subsurface scattering problem in consideration is formulated in the form of the Helmholtz equation

$$\nabla^2 u + k^2(z)u = f(x, y, z), \quad \text{in } \Omega, \tag{3.1}$$

with the Dirichlet, Neumann or Sommerfeld-like boundary conditions

$$\Gamma u = g, \text{ on } \partial\Omega, \tag{3.2}$$

where $k(x, y, z)$ is a complex valued variable coefficient, Ω is a three dimensional rectangular domain, $\partial\Omega$ is the boundary of Ω and Γ is a differential operator corresponding to the Dirichlet, Neumann or Sommerfeld-like boundary conditions. It is assumed that k is constant throughout any plane in the x and y direction.

In this chapter, both second and fourth order compact approximation finite-differences schemes are developed. To introduce these compact schemes for the solution of the three dimensional Helmholtz equation (3.1) with boundary conditions (3.2) consider the following. The computational domain is $\Omega = [x_0, x_1] \times [y_0, y_1] \times [z_0, z_1]$. Let N_x, N_y, and N_z be the number of grid points in the x, y, and z directions respectively. Let $h_x = (x_1 - x_0)/(N_x + 1)$, $h_y = (y_1 - y_0)/(N_y + 1)$, and $h_z = (z_1 - z_0)/(N_z + 1)$ be the grid steps in the x, y, and z directions respectively. Then the computational grid is

defined by

$$\Omega_G = \{(x_i, y_j, z_l) \mid x_i = x_0 + ih_x,\ y_j = y_0 + jh_y,\ z_l = z_0 + lh_z,$$

$$1 \le i \le N_x,\ 1 \le j \le N_y,\ 1 \le l \le N_z\}.$$

Consider the cases with Dirichlet or Neumann boundary conditions on $\partial\Omega$. That is, $u(0,y,z) = u(a,y,z) = u(x,0,z) = u(x,a,z) = 0$ or $\frac{\partial u}{\partial x}\big|_{x=0} = \frac{\partial u}{\partial x}\big|_{x=a} = \frac{\partial u}{\partial y}\Big|_{y=0} = \frac{\partial u}{\partial y}\Big|_{y=a} = 0$ where $z = a$ and $z = 0$ at the top and bottom of the Ω respectfully. The second order central finite-difference approximation of the first derivative on all boundaries is utilized. The second order central finite-difference approximation of the first derivative on all boundaries is utilized for the Neumann and Sommerfeld-like boundary conditions. The following notation will be used for the second order central differences at the $(i,j,l)-$th grid point,

$$\delta_x^2 u_{i,j,l} = \frac{u_{i-1,j,l} - 2u_{i,j,l} + u_{i+1,j,l}}{h_x^2},$$

$$\delta_y^2 u_{i,j,l} = \frac{u_{i,j-1,l} - 2u_{i,j,l} + u_{i,j+1,l}}{h_y^2} \text{ and}$$

$$\delta_z^2 u_{i,j,l} = \frac{u_{i,j,l-1} - 2u_{i,j,l} + u_{i,j,l+1}}{h_z^2}$$

where $u_{i,j,l} = u(x_i, y_j, z_l)$.

3.2 A Second Order Compact Scheme

Consider the three dimensional Helmholtz equation (3.1). That is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + k^2 u = f. \tag{3.3}$$

Recall the Taylor expansion

$$u_{i\pm1,j,l} = u_i \pm \frac{\partial u_{i,j,l}}{\partial x} h_x + \frac{1}{2}\frac{\partial^2 u_{i,j,l}}{\partial x^2} h_x^2 \pm \frac{1}{6}\frac{\partial^3 u_{i,j,l}}{\partial x^3} h_x^3 + \frac{1}{24}\frac{\partial^4 u_{i,j,l}}{\partial x^4} h_x^4 \pm \dots.$$

The addition of $u_{i-1,j,l}$ and $u_{i+1,j,l}$ shows that the odd terms will cancel, that is

$$u_{i-1,j,l} + u_{i+1,j,l} = 2u_{i,j,l} + \frac{\partial^2 u_{i,j,l}}{\partial x^2} h_x^2 + \mathcal{O}(h_x^4), \text{ so}$$

$$-\frac{\partial^2 u_{i,j,l}}{\partial x^2} h_x^2 = -u_{i-1,j,l} + 2u_{i,j,l} - u_{i+1,j,l} + \mathcal{O}(h_x^4)$$

$$\frac{\partial^2 u_{i,j,l}}{\partial x^2} = \frac{1}{h_x^2}(u_{i-1,j,l} - 2u_{i,j,l} + u_{i+1,j,l}) + \mathcal{O}(h_x^2)$$

$$\frac{\partial^2 u_{i,j,l}}{\partial x^2} = \delta_x^2 u_{i,j,l} + \mathcal{O}(h_x^2).$$

This argument can be repeated for both the y and z derivatives. Then by substituting these approximations into (3.3) the following second order scheme is obtained

$$\delta_x^2 u_{i,j,l} + \delta_y^2 u_{i,j,l} + \delta_z^2 u_{i,j,l} + k_l^2 u_{i,j,l} = f_{i,j,l}. \tag{3.4}$$

Now consider the following manipulations to Equation (3.4). Let $R_{zx} = h_z^2/h_x^2$ and $R_{zy} = h_z^2/h_y^2$. Then the scheme can be written as

$$\begin{aligned} h_z^2 f_{i,j,l} &= R_{zx}\left(u_{i-1,j,l} - 2u_{i,j,l} + u_{i+1,j,l}\right) + R_{zy}\left(u_{i,j-1,l} - 2u_{i,j,l} + u_{i,j+1,l}\right) \\ &\quad + \left(u_{i,j,l-1} - 2u_{i,j,l} + u_{i,j,l+1}\right) + h_z^2 k_l^2 u_{i,j,l} \\ &= -\left(2R_{zx} + 2R_{zy} + 2 + h_z^2 k_l^2\right) u_{i,j,l} \\ &\quad + R_{zx}\left(u_{i-1,j,l} + u_{i+1,j,l}\right) + R_{zy}\left(u_{i,j-1,l} + u_{i,j+1,l}\right) + u_{i,j,l-1} + u_{i,j,l+1}. \end{aligned} \tag{3.5}$$

Note that it is assumed that k_l^2 is constant for a fixed l. Let $\alpha_l = -\left(2R_{zx} + 2R_{zy} + 2 + h_z^2 k_l^2\right)$ and $U_l = \begin{bmatrix} u_{1,1,l} & u_{2,1,l} & \cdots & u_{N_x,1,l} & u_{1,2,l} & u_{2,2,l} & \cdots & u_{N_x,2,l} & \cdots & u_{N_x,N_y,l} \end{bmatrix}^T$. Define $A_1 = \alpha I$ where I is the identity matrix in $\mathbb{R}^{N_x \cdot N_y \times N_x \cdot N_y}$. Let $A_2 = (a_{i,j}) \in \mathbb{R}^{N_x \cdot N_y \times N_x \cdot N_y}$ be such that $a_{i,j} = 1$ when $|i-j| = 1$ and $a_{i,j} = 0$ otherwise. Also, let $A_3 = (a_{i,j}) \in \mathbb{R}^{N_x \cdot N_y \times N_x \cdot N_y}$ be such that $a_{i,j} = 1$ when $|i-j| = N_x$ and $a_{i,j} = 0$ otherwise. Now define $A = A_1 + R_{zx}A_2 + R_{zy}A_3$. Then Equation (3.5) can be written as

$$U_{l-1} + AU_l + U_{l+1} = h_z^2 f_l. \tag{3.6}$$

Now examine the eigenpairs of A.

Theorem 3.2.1. Let $B = (b_{ij}) \in \mathbb{R}^{N_x \cdot N_Y \times N_x \cdot N_Y}$. Let $\beta_{n,m}^{l,k} = \sin\left(\frac{n\pi}{N_x+1} \cdot l\right) \sin\left(\frac{m\pi}{N_y+1} \cdot k\right)$ where $1 \leq n, l \leq N_x$ and $1 \leq m, k \leq N_y$. Define $v_{l,k} = \begin{bmatrix} \beta_{1,1}^{l,k} & \beta_{2,1}^{l,k} & \cdots & \beta_{N_x,1}^{l,k} & \beta_{1,2}^{l,k} & \cdots & \beta_{N_x,N_y}^{l,k} \end{bmatrix}^T$. Suppose $b_{i,j} = 1$ when $i + 1 = j$ or $i - 1 = j$ and $b_{i,j} = 0$ otherwise. Then $v_{l,k}$ is an eigenvector of B with corresponding eigenvalue $\lambda = 2\cos\left(\frac{\pi}{N_x+1} \cdot l\right)$.

Proof. Let B, $v_{l,k}$, λ and $\beta_{n,m}^{l,k}$ be as defined above. Recall the trigonometric identity $2\sin(x)\cos(y) = \sin(x-y) + \sin(x+y)$ for $x, y \in \mathbb{R}$ [8]. Then

$$\begin{aligned}
\lambda\beta_{n,m}^{l,k} &= 2\cos\left(\frac{\pi}{N_x+1} \cdot l\right) \sin\left(\frac{n\pi}{N_x+1} \cdot l\right) \sin\left(\frac{m\pi}{N_y+1} \cdot k\right) \\
&= \left[\sin\left((n-1)\frac{\pi}{N_x+1} \cdot l\right) + \sin\left((n+1)\frac{\pi}{N_x+1} \cdot l\right)\right] \sin\left(\frac{m\pi}{N_y+1} \cdot k\right) \\
&= \beta_{n-1,m}^{l,k} + \beta_{n+1,m}^{l,k}.
\end{aligned}$$

Note that $\beta_{N_x+1,m}^{l,k} = 0 = \beta_{0,m}^{l,k}$. Thus

$$Bv_{l,k} = \begin{bmatrix} \beta_{2,1}^{l,k} & \beta_{1,1}^{l,k} + \beta_{3,1}^{l,k} & \cdots & \beta_{N_x-1,1}^{l,k} + \beta_{1,1}^{l,k} & \beta_{N_x,2}^{l,k} + \beta_{2,2}^{l,k} & \cdots & \beta_{N_x-1,N_y}^{l,k} \end{bmatrix}^T = \lambda v_{l,k}.$$

□

Define $v_{l,k}$ as in Theorem 3.2.1. As a result, $v_{l,k}$ is an eigenvector of A_2 with corresponding eigenvalue $2\cos\left(\frac{\pi}{N_x+1} \cdot l\right)$. It follows that $\lambda_l = 2R_{zx}\cos\left(\frac{\pi}{N_x+1} \cdot l\right)$ and $v_{l,k}$ form an eigenpair of $R_{zx}A_2$. To find an eigenpair of A_3 consider the following theorem.

Theorem 3.2.2. Using the same set up as Theorem 3.2.1, suppose $b_{i,j} = 1$ when $i + N_x = j$ or $i - N_x = j$ and $b_{i,j} = 0$ otherwise. Then $v_{l,k}$ is an eigenvector of B with corresponding eigenvalue $\lambda = 2\cos\left(\frac{\pi}{N_y+1} \cdot k\right)$.

Proof. First, observe that

$$\begin{aligned}\lambda\beta_{n,m}^{l,k} &= 2\cos\left(\frac{\pi}{N_y+1}\cdot k\right)\sin\left(\frac{n\pi}{N_x+1}\cdot l\right)\sin\left(\frac{m\pi}{N_y+1}\cdot k\right)\\ &= \left[\sin\left((m-1)\frac{\pi}{N_y+1}\cdot k\right)+\sin\left((m+1)\frac{\pi}{N_y+1}\cdot k\right)\right]\sin\left(\frac{n\pi}{N_x+1}\cdot l\right)\\ &= \beta_{n,m-1}^{l,k}+\beta_{n,m+1}^{l,k}.\end{aligned}$$

Note that $\beta_{n,N_y+1}^{l,k}=0=\beta_{n,0}^{l,k}$. Thus

$$Bv_{l,k}=\begin{bmatrix}\beta_{1,2}^{l,k} & \beta_{2,2}^{l,k} & \cdots & \beta_{N_x,2}^{l,k} & \beta_{1,1}^{l,k}+\beta_{1,3}^{l,k} & \cdots & \beta_{N_x,N_y-1}^{l,k}\end{bmatrix}^T=\lambda v_{l,k}. \qquad \square$$

Theorem 3.2.2 shows that $v_{l,k}$ is an eigenvector of $R_{zy}A_3$ with corresponding eigenvalue $\lambda_k = 2R_{zy}\cos\left(\frac{\pi}{N_x+1}\cdot k\right)$. Note $(\alpha, v_{l,k})$ is an eigenpair of A_1. Thus $(\alpha_l+\lambda_l+\lambda_k, v_{l,k})$ is an eigenpair of A by Theorem 2.1.2. Consider the following lemmas.

Lemma 3.2.3. Let $v_{l,k}$ be as in Theorem 3.2.1. Then $||v_{l,k}||_2^2=\langle v_{l,k},v_{l,k}\rangle=(N_x+1)(N_y+1)/4$ for $1\le l\le N_x$ and $1\le k\le N_y$.

Proof. Let $l\in\{1,\dots,N_x\}$ and $k\in\{1,\dots,N_y\}$. Then

$$\begin{aligned}\langle v_{l,k},v_{l,k}\rangle &= \sum_{n=1}^{N_x}\sum_{m=1}^{N_y}\left(\beta_{n,m}^{l,k}\right)^2\\ &= \sum_{n=1}^{N_x}\sum_{m=1}^{N_y}\left[\sin\left(\frac{n\pi}{N_x+1}\cdot l\right)\sin\left(\frac{m\pi}{N_y+1}\cdot k\right)\right]^2\\ &= \sum_{n=1}^{N_x}\sin^2\left(\frac{n\pi}{N_x+1}\cdot l\right)\left[\sum_{m=1}^{N_y}\sin^2\left(\frac{m\pi}{N_y+1}\cdot k\right)\right]\\ &= \sum_{n=1}^{N_x}\sin^2\left(\frac{n\pi}{N_x+1}\cdot l\right)\left[\frac{Ny+1}{2}\right] \quad \text{by Lemma 2.1.3}\\ &= \frac{Ny+1}{2}\sum_{n=1}^{N_x}\sin^2\left(\frac{n\pi}{N_x+1}\cdot l\right)\\ &= \left(\frac{Ny+1}{2}\right)\left(\frac{Nx+1}{2}\right) \quad \text{by Lemma 2.1.3.}\end{aligned}$$

Thus $||v_{l,k}||_2^2=\langle v_{l,k},v_{l,k}\rangle=(N_x+1)(N_y+1)/4$ for $1\le l\le N_x$ and $1\le k\le N_y$. $\square$

Lemma 3.2.4. Let $v_{l,k}$ be as in Theorem 3.2.1. If $l \neq l'$ or $k \neq k'$ then $v_{l,k}$ and $v_{l',k'}$ are orthogonal vectors.

Proof. Let $v_{l,k}$ and $v_{l',k'}$ as defined above. Suppose $k \neq k'$. Let

$$\alpha_n^{l,l'} = \sin\left(l \cdot \frac{n\pi}{N_x+1}\right)\sin\left(l' \cdot \frac{n\pi}{N_x+1}\right).$$

Note that $\alpha_n^{l,l'}$ does not depend on m and

$$\sum_{m=1}^{N_y} \sin\left(k \cdot \frac{m\pi}{N_y+1}\right)\sin\left(k' \cdot \frac{m\pi}{N_y+1}\right) = 0$$

by Lemma 2.1.4. Then

$$\begin{aligned}
\langle v_{l,k}, v_{l',k'}\rangle &= \sum_{n=1}^{N_x}\sum_{m=1}^{N_y} \beta_{n,m}^{l,k}\beta_{n,m}^{l',k'} \\
&= \sum_{n=1}^{N_x}\sum_{m=1}^{N_y} \alpha_n^{l,l'} \sin\left(k \cdot \frac{m\pi}{N_y+1}\right)\sin\left(k' \cdot \frac{m\pi}{N_y+1}\right) \\
&= \sum_{n=1}^{N_x}\left[\alpha_n^{l,l'} \sum_{m=1}^{N_y} \sin\left(k \cdot \frac{m\pi}{N_y+1}\right)\sin\left(k' \cdot \frac{m\pi}{N_y+1}\right)\right] \\
&= \sum_{n=1}^{N_x}\left[\alpha_n^{l,l'} \cdot 0\right] = 0.
\end{aligned}$$

The same can be shown for $l \neq l'$. □

These lemmas show the relationship between these eigenvectors with respect to their inner product. Note that Lemma 3.2.4 holds for the normalized eigenvectors. Now consider the following theorems that are essential to the derivation of both the second and fourth order schemes.

Theorem 3.2.5. Let $v_{l,k}$ be as in Theorem 3.2.1. Define $w_{i,j} = ||v_{i,j}||^{-1}v_{i,j}$. Let $V = \begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{N_x,1} & w_{1,2} & \cdots & w_{N_x,N_y} \end{bmatrix} \in \mathbb{R}^{N_x \cdot N_y \times N_x \cdot N_y}$. Then V is an orthogonal matrix.

Proof. Let V be as defined above. Then

$$\begin{aligned}
V^TV &= V^T\begin{bmatrix} w_{1,1} & w_{2,1} & \cdots & w_{N_x,1} & w_{1,2} & \cdots & w_{N_x,N_y}\end{bmatrix} \\
&= \begin{bmatrix} V^Tw_{1,1} & V^Tw_{2,1} & \cdots & V^Tw_{N_x,1} & V^Tw_{1,2} & \cdots & V^Tw_{N_x,N_y}\end{bmatrix} \\
&= \begin{bmatrix} \langle w_{1,1}, w_{1,1}\rangle & \langle w_{1,1}, w_{2,1}\rangle & \cdots & \langle w_{1,1}, w_{N_x,N_y}\rangle \\ \langle w_{2,1}, w_{1,1}\rangle & \langle w_{2,1}, w_{2,1}\rangle & & \\ \vdots & & \ddots & \vdots \\ \langle w_{N_x,N_y}, w_{1,1}\rangle & \langle w_{N_x,N_y}, w_{2,1}\rangle & \cdots & \langle w_{N_x,N_y}, w_{N_x,N_y}\rangle \end{bmatrix} \\
&= I
\end{aligned}$$

by Lemmas 3.2.4 and 3.2.3. Thus V is orthogonal. □

Let V be the matrix defined in Theorem 3.2.5 and let $W_l = V^TU_l$. Then

$$U_{l-1} + AU_l + U_{l+1} = h_z^2 f_l$$

$$V^TU_{l-1} + V^TAVV^TU_l + V^TU_{l+1} = h_z^2V^Tf_l$$

$$W_{l-1} + \Lambda W_l + W_{l+1} = \overline{F_l}$$

where $\Lambda = V^TAV$ is the diagonal matrix of eigenvalues by Theorem 2.1.6 and $\overline{F_l} = h_z^2V^Tf_l$. This yields $N_x \cdot N_y$ independent systems

$$\begin{aligned}
\lambda_i w_{i,1} + w_{i,2} &= \overline{F_{i,1}} \\
w_{i,l-1} + \lambda_i w_{i,l} + w_{i,l+1} &= \overline{F_{i,l}} \qquad \text{for } l = 2, 3, \ldots, N_z - 1 \\
w_{i,N_z-1} + \lambda_i w_{i,N_z} &= \overline{F_{i,N_z}}
\end{aligned}$$

for $i = 1, 2, \ldots, N_x \cdot N_y$.

These systems can be solved using the LU decomposition of the generated tridiagonal matrix. The computations in this solution are independent with respect to both x and y direction of the computational domain. Therefore, it can now be parallelized with respect to the either direction.

Prior to solving this system, $\overline{F}_l = V^TF_l$ must be found. This differs from the two

dimensional test problem in that it $\overline{F}_l$ is a DST in two directions, x and y. Therefore, to find $\overline{F}_l$ the DST is calculated in one direction, then the other. The process is then the same as the two dimensional case: find the DST of the right hand side, solve the tridiagonal system and find the reverse transform of the result. This will be outlined in Algorithm 4.

3.3 A Fourth Order Compact Scheme

Consider an equivalent alteration to the three dimensional Helmholtz equation,

$$-\frac{\partial^2 u}{\partial x^2}-\frac{\partial^2 u}{\partial y^2}-\frac{\partial^2 u}{\partial z^2}-k^2u=f. \tag{3.7}$$

Recall the Taylor expansion

$$u_{i\pm1,j,l}=u_{i,j,l}\pm\frac{\partial u_{i,j,l}}{\partial x}h_x+\frac{h_x^2}{2}\frac{\partial^2 u_{i,j,l}}{\partial x^2}\pm\frac{h_x^3}{3!}\frac{\partial^3 u_{i,j,l}}{\partial x^3}+\frac{h_x^4}{4!}\frac{\partial^4 u_{i,j,l}}{\partial x^4}\pm\frac{h_x^5}{5!}\frac{\partial^5 u_{i,j,l}}{\partial x^5}+\frac{h_x^6}{6!}\frac{\partial^6 u_{i,j,l}}{\partial x^6}+\dots.$$

In the addition of $u_{i-1,j,l}$ and $u_{i+1,j,l}$ the odd terms will cancel, that is

$$u_{i-1,j,l}+u_{i+1,j,l}=2u_{i,j,l}+\frac{\partial^2 u_{i,j,l}}{\partial x^2}h_x^2+\frac{1}{12}\frac{\partial^4 u_{i,j,l}}{\partial x^4}h_x^4+\mathcal{O}(h_x^6)$$

$$\frac{1}{h_x^2}\left(u_{i-1,j,l}-2u_{i,j,l}+u_{i+1,j,l}\right)=\frac{\partial^2 u_{i,j,l}}{\partial x^2}+\frac{1}{12}\frac{\partial^4 u_{i,j,l}}{\partial x^4}h_x^2+\mathcal{O}(h_x^4)$$

$$\delta_x^2 u_{i,j,l}=\frac{\partial^2 u_{i,j,l}}{\partial x^2}+\frac{h_x^2}{12}\cdot\frac{1}{h_x^2}\delta_x^2\frac{\partial^2 u_{i,j,l}}{\partial x^2}+\mathcal{O}(h_x^4)$$

$$\delta_x^2 u_{i,j,l}=\left(I+\frac{1}{12}\delta_x^2\right)\frac{\partial^2 u_{i,j,l}}{\partial x^2}+\mathcal{O}(h_x^4)$$

$$\frac{\partial^2 u_{i,j,l}}{\partial x^2}=\left(I+\frac{1}{12}\delta_x^2\right)^{-1}\delta_x^2 u_{i,j,l}+\mathcal{O}(h_x^4).$$

This process can be repeated for both the y and z derivatives. Then by substituting these approximations into (3.7) the following fourth order scheme is obtained

$$\begin{aligned}&-\left(I+\frac{h_x^2}{12}\delta_x^2\right)^{-1}\delta_x^2 u_{i,j,l}-\left(I+\frac{h_y^2}{12}\delta_y^2\right)^{-1}\delta_y^2 u_{i,j,l}\\&-\left(I+\frac{h_z^2}{12}\delta_z^2\right)^{-1}\delta_z^2 u_{i,j,l}-k_l^2 u_{i,j,l}=f_{i,j,k}.\end{aligned} \tag{3.8}$$

Theorem 3.3.1. The terms $\left(I + \frac{h_x^2}{12}\delta_x^2\right)$ and $\left(I + \frac{h_y^2}{12}\delta_y^2\right)$ commute.

Proof. Let u be a vector. First note that the operators δ_x^2 and δ_y^2 commute since

$$\begin{aligned}
\delta_x^2\delta_y^2 u_{i,j,l} &= \delta_x^2\left(\frac{u_{i,j-1,l} - 2u_{i,j,l} + u_{i,j+1,l}}{h_y^2}\right) \\
&= \frac{1}{h_y^2}\left(\delta_x^2 u_{i,j-1,l} - 2\delta_x^2 u_{i,j,l} + \delta_x^2 u_{i,j+1,l}\right) \\
&= \frac{1}{h_y^2}\frac{1}{h_x^2}\left(\left[u_{i-1,j-1,l} - 2u_{i,j-1,l} + u_{i+1,j-1,l}\right]\right. \\
&\quad \left. -2\left[u_{i-1,j,l} - 2u_{i,j,l} + u_{i+1,j,l}\right] + \left[u_{i-1,j+1,l} - 2u_{i,j+1,l} + u_{i+1,j+1,l}\right]\right) \\
&= \frac{1}{h_x^2}\frac{1}{h_y^2}\left(\left[u_{i-1,j-1,l} - 2u_{i-1,j,l} + u_{i-1,j+1,l}\right]\right. \\
&\quad \left. -2\left[u_{i,j-1,l} - 2u_{i,j,l} + u_{i,j+1,l}\right] + \left[u_{i+1,j-1,l} - 2u_{i+1,j,l} + u_{i+1,j+1,l}\right]\right) \\
&= \frac{1}{h_x^2}\left(\delta_y^2 u_{i-1,j,l} - 2\delta_y^2 u_{i,j,l} + \delta_y^2 u_{i+1,j,l}\right) \\
&= \delta_y^2\left(\frac{u_{i-1,j,l} - 2u_{i,j,l} + u_{i+1,j,l}}{h_x^2}\right) = \delta_y^2\delta_x^2 u_{i,j,l}.
\end{aligned}$$

Thus

$$\begin{aligned}
\left(I + \frac{h_x^2}{12}\delta_x^2\right)\left(I + \frac{h_y^2}{12}\delta_y^2\right)u_{i,j,l} &= \left(I + \frac{h_y^2}{12}\delta_y^2\right)u_{i,j,l} + \frac{h_x^2}{12}\delta_x^2\left(I + \frac{h_y^2}{12}\delta_y^2\right)u_{i,j,l} \\
&= u_{i,j,l} + \frac{h_y^2}{12}\delta_y^2 u_{i,j,l} + \frac{h_x^2}{12}\delta_x^2 u_{i,j,l} + \frac{h_x^2}{12}\frac{h_y^2}{12}\delta_x^2\delta_y^2 u_{i,j,l} \\
&= u_{i,j,l} + \frac{h_x^2}{12}\delta_x^2 u_{i,j,l} + \frac{h_y^2}{12}\delta_y^2 u_{i,j,l} + \frac{h_y^2}{12}\frac{h_x^2}{12}\delta_y^2\delta_x^2 u_{i,j,l} \\
&= \left(I + \frac{h_x^2}{12}\delta_x^2\right)u_{i,j,l} + \frac{h_y^2}{12}\delta_y^2\left(I + \frac{h_x^2}{12}\delta_x^2\right)u_{i,j,l} \\
&= \left(I + \frac{h_y^2}{12}\delta_y^2\right)\left(I + \frac{h_x^2}{12}\delta_x^2\right)u_{i,j,l}.
\end{aligned}$$

□

Note that the theorem holds for all combinations of the terms $\left(I + \frac{h_x^2}{12}\delta_x^2\right)$, $\left(I + \frac{h_y^2}{12}\delta_y^2\right)$ and $\left(I + \frac{h_z^2}{12}\delta_z^2\right)$ as the proof is not dependent on the choice of x, y or z. To remove the terms with inverses in (3.8), left multiply the equation by $\left(I + \frac{h_x^2}{12}\delta_x^2\right)\left(I + \frac{h_y^2}{12}\delta_y^2\right)\left(I + \frac{h_z^2}{12}\delta_z^2\right)$.

Then using Theorem 3.3.1 gives

$$
\begin{aligned}
&-\left(I+\frac{h_y^2}{12}\delta_y^2\right)\left(I+\frac{h_z^2}{12}\delta_z^2\right)\delta_x^2 u_{i,j,l}\\
&-\left(I+\frac{h_x^2}{12}\delta_x^2\right)\left(I+\frac{h_z^2}{12}\delta_z^2\right)\delta_y^2 u_{i,j,l}\\
&-\left(I+\frac{h_x^2}{12}\delta_x^2\right)\left(I+\frac{h_y^2}{12}\delta_y^2\right)\delta_z^2 u_{i,j,l}\\
&-\left(I+\frac{h_x^2}{12}\delta_x^2\right)\left(I+\frac{h_y^2}{12}\delta_y^2\right)\left(I+\frac{h_z^2}{12}\delta_z^2\right)k_l^2 u_{i,j,l}\\
&=\left(I+\frac{h_x^2}{12}\delta_x^2\right)\left(I+\frac{h_y^2}{12}\delta_y^2\right)\left(I+\frac{h_z^2}{12}\delta_z^2\right)f_{i,j,l}. \qquad (3.9)
\end{aligned}
$$

Now multiply out and drop all terms with $h_x^2h_y^2$, $h_y^2h_z^2$, $h_x^2h_x^2$ and $h_x^2h_y^2h_z^2$. This is justified as the fourth order approximation scheme is considered and only the second order terms need to remain. It follows that

$$
\begin{aligned}
&-\left[\delta_x^2+\delta_y^2+\delta_z^2\right]u_{i,j,l}-\tfrac{1}{12}\left[h_x^2+h_z^2\right]\delta_x^2\delta_z^2-\tfrac{1}{12}\left[h_x^2+h_y^2\right]\delta_x^2\delta_y^2\\
&-\tfrac{1}{12}\left[h_y^2+h_z^2\right]\delta_y^2\delta_z^2-k_l^2u_{i,j,l}-\tfrac{1}{12}\left(h_x^2\delta_x^2+h_y^2\delta_y^2+h_z^2\delta_z^2\right)k_l^2u_{i,j,l}\\
&=f_{i,j,l}+\tfrac{1}{12}\left(h_x^2\delta_x^2+h_y^2\delta_y^2+h_z^2\delta_z^2\right)f_{i,j,l}. \qquad (3.10)
\end{aligned}
$$

Let $\overline{\delta_x}^2u_{i,j,l}=h_x^2\delta_x^2u_{i,j,l}=u_{i-1,j,l}-2u_{i,j,k}+u_{i+1,j,l}$, $\overline{\delta_y}^2u_{i,j,l}=h_y^2\delta_y^2u_{i,j,l}$ and $\overline{\delta_z}^2u_{i,j,l}=h_z^2\delta_z^2u_{i,j,l}$. Now, multiply both sides of Equation (3.10) by h_z^2. Let $R_{zx}=h_z^2/h_x^2$ and $R_{zy}=h_z^2/h_y^2$. Then

$$
\begin{aligned}
&-\left[R_{zx}\overline{\delta}_x^2+R_{zy}\overline{\delta}_y^2+\overline{\delta}_z^2\right]u_{i,j,l}\\
&-\frac{1}{12}\left[1+R_{zx}\right]\overline{\delta}_x^2\overline{\delta}_z^2u_{i,j,l}-\frac{1}{12}\left[R_{zx}+R_{zy}\right]\overline{\delta}_x^2\overline{\delta}_y^2u_{i,j,l}-\frac{1}{12}\left[1+R_{zy}\right]\overline{\delta}_y^2\overline{\delta}_z^2u_{i,j,l}\\
&-h_z^2k_l^2u_{i,j,l}-\frac{h_z^2}{12}\left(\overline{\delta}_x^2+\overline{\delta}_x^2+\overline{\delta}_z^2\right)k_l^2u_{i,j,l}\\
&=h_z^2f_{i,j,l}+\frac{h_z^2}{12}\left(\overline{\delta}_x^2+\overline{\delta}_x^2+\overline{\delta}_z^2\right)f_{i,j,l}.
\end{aligned}
$$

Now rewrite this scheme to group the coefficients of $l-1$, l and $l+1$ layers in the

following form,

$$
\begin{aligned}
&-\left[I+\tfrac{1}{12}\left(\left[1+R_{zx}\right]\overline{\delta}_x^2+\left[1+R_{zy}\right]\overline{\delta}_y^2\right)\right]u_{i,j,l\mp1}-\tfrac{1}{12}k_{i,j,k\mp1}^2u_{i,j,k\mp1}\\
&\quad-\left[\left(R_{zx}\overline{\delta}_x^2+R_{zy}\overline{\delta}_y^2-2I\right)-\tfrac{2}{12}\left[1+R_{zx}\right]\overline{\delta}_x^2\right.\\
&\quad+\tfrac{1}{12}\left[R_{zx}+R_{zy}\right]\overline{\delta}_x^2\overline{\delta}_y^2\overline{\delta}_x^2-\tfrac{2}{12}\left[1+R_{zy}\right]\overline{\delta}_y^2\\
&\quad\left.+k_{i,j,l}^2I+\tfrac{1}{12}\left[\overline{\delta}_x^2+\overline{\delta}_y^2-2I\right]k_l^2\right]u_{i,j,l}=F_{i,j,l},
\end{aligned}
\tag{3.11}
$$

where $F_{i,j,l}=h_z^2f_{i,j,l}+\frac{h_z^2}{12}\left(\overline{\delta}_x^2+\overline{\delta}_x^2+\overline{\delta}_z^2\right)f_{i,j,l}$. Now consider the fact that k^2 depends only on the level l in the z direction. Recall the definitions of A_2 and A_3 from the previous section: $A_2=(a_{i,j})\in\mathbb{R}^{N_x\cdot N_y\times N_x\cdot N_y}$ be such that $a_{i,j}=1$ when $|i-j|=1$ and $a_{i,j}=0$ otherwise and $A_3=(a_{i,j})\in\mathbb{R}^{N_x\cdot N_y\times N_x\cdot N_y}$ be such that $a_{i,j}=1$ when $|i-j|=N_x$ and $a_{i,j}=0$ otherwise. Now define the matrices C_1,C_2 and C_3 by

$$
C_1=-\left[I+\frac{1}{12}\left(\left[1+R_{zx}\right]A_2+\left[1+R_{zy}\right]A_3\right)+\frac{1}{12}k_{l-1}^2I\right],
$$

$$
\begin{aligned}
C_2=-\left[\left(-2+k_l^2-\frac{1}{6}k_l^2\right)I+\left(R_{zx}-\frac{1}{6}-\frac{1}{6}R_{zx}+\frac{1}{12}k_l^2\right)A_2\right.\\
\left.+\left(R_{zy}-\frac{1}{6}-\frac{1}{6}R_{zy}+\frac{1}{12}k_l^2\right)A_3+\frac{1}{12}\left[R_{zx}+R_{zy}\right]A_2A_3\right]\text{ and}
\end{aligned}
$$

$$
C_3=-\left[I+\frac{1}{12}\left(\left[1+R_{zx}\right]A_2+\left[1+R_{zy}\right]A_3\right)+\frac{1}{12}k_{l+1}^2I\right].
$$

Also, let $U_l=\begin{bmatrix}u_{1,1,l} & u_{2,1,l} & \cdots & u_{N_x,1,l} & u_{1,2,l} & \cdots & u_{N_x,N_y,l}\end{bmatrix}^T$. Thus (3.3) can be written as

$$
C_1U_{l-1}+C_2U_l+C_3U_{l+1}=F_l. \tag{3.12}
$$

Now analyze the eigenpairs of C_1, C_2 and C_3.

Theorem 3.3.2. Let $B_1,B_2\in\mathbb{R}^{n\times n}$ for $n>1$. If (λ_1,v) and (λ_2,v) are eigenpairs of B_1 and B_2 respectively then $(\lambda_2\lambda_1,v)$ is an eigenpair of B_1B_2.

Proof. Let B_1 and B_2 be as defined above. Let (λ_1,v) and (λ_2,v) be eigenpairs of B_1

and B_2 respectively. Then

$$B_1B_2v = B_1(B_2v) = B_1(\lambda_2 v) = \lambda_2 B_1 v = \lambda_2\lambda_1 v. \qquad \square$$

Let $\beta_{n,m}^{l,k} = \sin\left(\frac{n\pi}{N_x+1}\cdot l\right)\sin\left(\frac{m\pi}{N_y+1}\cdot k\right)$ where $1 \le n, l \le N_x$ and $1 \le m, k \le N_y$ and define $v_{l,k} = \begin{bmatrix}\beta_{1,1}^{l,k} & \beta_{2,1}^{l,k} & \cdots & \beta_{N_x,1}^{l,k} & \beta_{1,2}^{l,k} & \cdots & \beta_{N_x,N_y}^{l,k}\end{bmatrix}^T$. Then $v_{l,k}$ is an eigenvector of A_2, A_3 and A_2A_3 by Theorems 3.2.1, 3.2.2 and 3.3.2 respectively. Thus $v_{l,k}$ is an eigenvector of C_1, C_2 and C_3 by Theorem 2.1.2. Then C_1, C_2 and C_3 can be diagonalized simultaneously. Let $V = \begin{bmatrix}w_{1,1} & w_{2,1} & \cdots & w_{N_x,1} & w_{1,2} & \cdots & w_{N_x,N_y}\end{bmatrix} \in \mathbb{R}^{N_x\cdot N_y \times N_x\cdot N_y}$ where $w_{i,j} = ||v_{i,j}||^{-1}v_{i,j}$. Note that V is orthogonal by Theorem 3.2.5. Reformulate (3.12) as follows

$$\begin{aligned} C_1U_{l-1} + C_2U_l + C_3U_{l+1} &= F_l \\ VC_1V^TVU_{l-1} + VC_2V^TVU_l + VC_3V^TVU_{l-1} &= VF_l \\ \Lambda_1W_{l-1} + \Lambda_2W_l + \Lambda_3W_{l+1} &= \overline{F}_l. \end{aligned} \tag{3.13}$$

Where $\Lambda_1 = VC_1V^T$, $\Lambda_2 = VC_2V^T$, $\Lambda_3 = VC_3V^T$, $W_l = VU_l$ and $\overline{F}_l = VF_l$.

This provides a tridiagonal system that can be solved using the LU decomposition. As in the second order scheme, the computations in this solution are independent with respect to both x and y direction of the computational domain. Therefore, it can now be parallelized with respect to the either direction. As the eigenvectors are the same as the second order scheme, the process in solving this system is the same. That is, find the DST of the right hand side, solve the tridiagonal system and find the reverse transform of the result.

3.4 Conclusion

Chapter 3 presented the problem that is the focus of this thesis, the three dimensional Helmholtz equation. The second and fourth order schemes for approximating the so-

lution to the equation were developed. These schemes included forward and reverse transforms that are independent in the z direction and a tridiagonal solver that is independent with respect to the y direction. As a result, these sections can be parallelized.

Chapter 4

Parallelization

4.1 Introduction

This chapter provides a detailed examination of the parallelization of second and fourth order compact algorithms for approximating solutions to the three dimensional Helmholtz equation. The results found in this chapter will show the accelerations and the immense benefit for implementing parallel technologies. In development, the program was run on several variations of computational grid size and number of threads and processes. However, for the results displayed in this chapter only grid sizes that are powers of two are utilized. This optimized the FFT calculation and therefore making the study of acceleration due to parallelization more direct [6].

The goal of implementing OpenMP and MPI is to observe near linear acceleration with the addition of threads or processes respectively. The work must be divided as evenly as possible among the threads or processes. To elaborate, consider a for-loop with n iterations and let p be the number of MPI processes or OpenMP threads. Only consider the case where $n > p$ as in practice n will be very large and p is limited by the hardware. If p divides n the number of iterations performed by each process or thread is simply p/k. On the other hand, if $n \mod p = r \neq 0$ then one of the remaining iterations is given to r processes. This prevents one process from doing significantly more work than the others.

4.2 Sequential Algorithm

This section will outline and identify sections of the algorithm for approximating the solution to (3.1) to communicate the process of parallelization efficiently. Several steps must first be taken prior to the actual approximation. Firstly, the domain and the medium coefficients, k, must be defined. The analytic solution is calculated at each grid point for error checking. Finally, the right hand side of the system is defined and the LU decomposition is computed. The primary area of focus is the solver, the section of code where the approximation is calculated. This consists of three parts; the forward transformation, the tridiagonal solver and the reverse transformation. The following algorithm details this solver and the remainder of this chapter will refer to these sections as such.

Algorithm 4 Sequential 3D Helmholtz Solver

1: **for** $k = 1, \ldots, N_z$ **do**
2: 2D forward DST in $x-, y-$direction
3: **end for**
4: **for** $j = 1, \ldots, N_y; i = 1, \ldots, N_x; k = 1, \ldots, N_z$ **do**
5: Solve the tridiagonal system using LU decomposition
6: **end for**
7: **for** $k = 1, \ldots, N_z$ **do**
8: 2D reverse DST in $x-, y-$direction
9: **end for**

The most computationally expensive sections in the solver are the forward and reverse transformations. Thus making these sections the primary concern in accelerating the calculation time. The methods in which they were modified will be elaborated on later on in this chapter. After the transforms were successfully parallelized and a desirable acceleration observed, the focus turned to the tridiagonal solver. Two different options were considered for the parallelization of the tridiagonal solver.

Prior to discussing the differences in the options, consider a machine's processor. There is very quick memory on a processor called a cache. This is where the data that the processor is working on is stored. Data is loaded into the cache from the RAM of the

machine. When this is done a section of memory is transferred to fill the entire cache. This process is not instantaneous. Thus it is wise to arrange the needed arrays in such a way that the required information for the next computation is already in the cache.

With this concept in mind, the first option organizes the arrays in a way that more relevant data is on the cache. This provides a relatively short computation time when run sequentially. However, this did not accelerate well when parallelized. The second option reorganized the for-loops in a way that would parallelize well. This increases the calculation time in serial, but a near linear acceleration was observed in parallel. When considering the acceleration of the entire solver it was the second options that performed slightly better. Therefore, the results that follow only consider the second option.

4.3 OpenMP

During the first implementation of OpenMP into the solver the anticipated linear acceleration of the forward and reverse transforms was not observed. After several experiments the cause was found to be the immense expense of creating FFTW plans. A FFTW plan is a function that sets up the calculations of the FFT [4]. The solution was to minimize the creation of these plans reducing the number down to only two plans. This led to another issue in that the creation of the plans are not thread safe. For a function to be thread safe it must be free of race conditions [3]. The functions written by the developers of FFTW can not be modified to use private variables as is the typical solution for a race condition. To fix this issue in both the forward and reverse transforms, the two FFTW plan creations were defined within a critical region. A critical region is a location in the code in which all the threads must reach prior to any other computations [5]. This fixed the issue and near linear acceleration was observed in both the forward and reverse transforms. The creation of the FFTW plans aside, the forward and reverse transforms contain many variables that are subject to race conditions. As in the two

dimensional problem these variables were made private. Note that the use of the private variables is omitted from Algorithm 5.

Algorithm 5 OpenMP 3D Helmholtz Solver

```
#pragma omp parallel for
for k = 1, ..., N_z do
    2D forward DST in x−, y−direction
end for
#pragma omp parallel for
for j = 1, ..., N_y; i = 1, ..., N_x; k = 1, ..., N_z do
    Solve the tridiagonal system using LU decomposition
end for
#pragma omp parallel for
for k = 1, ..., N_z do
    2D reverse DST in x−, y−direction
end for
```

Algorithm 5 was coded in C and successfully run with a near linear acceleration. These results are displayed in Figure 4.1. Despite finding desirable results, different strategies were examined to look for significant improvements.

A benefit in the implementation of OpenMP is that it automatically distributes the work as evenly as possible among the threads by default. This does come at a cost as it takes time for OpenMP to divide the for-loops. However, this tool does include the ability to manually divide the number of iterations among threads. Despite having desirable results with the automatic division a manual version was tested in hope to observe an increase in acceleration. After several experiments with various grid sizes and number of threads it was found to make no significant difference. Therefore, the results given in Figure 4.1 are found using the default division.

In another attempt to improve upon the observed acceleration a different plan was created for FFTW. This original OpenMP program used an FFTW plan that calculated the forward and reverse transforms using one dimensional FFT of a single column in a x, y plane of the domain. The FFTW subroutine has a function that allows for the one dimensional FFT of the entire x, y plane in a single execution. In addition, there is

the ability to included multithreading without explicitly adding OpenMP directives [5]. This technique was implemented and accelerations was observed. However, the original method had a much larger acceleration in every case and this technique was abandoned.

Figure 4.1: 3D OpenMP Acceleration

Acceleration is the Sequential Time Divided by Parallel Time

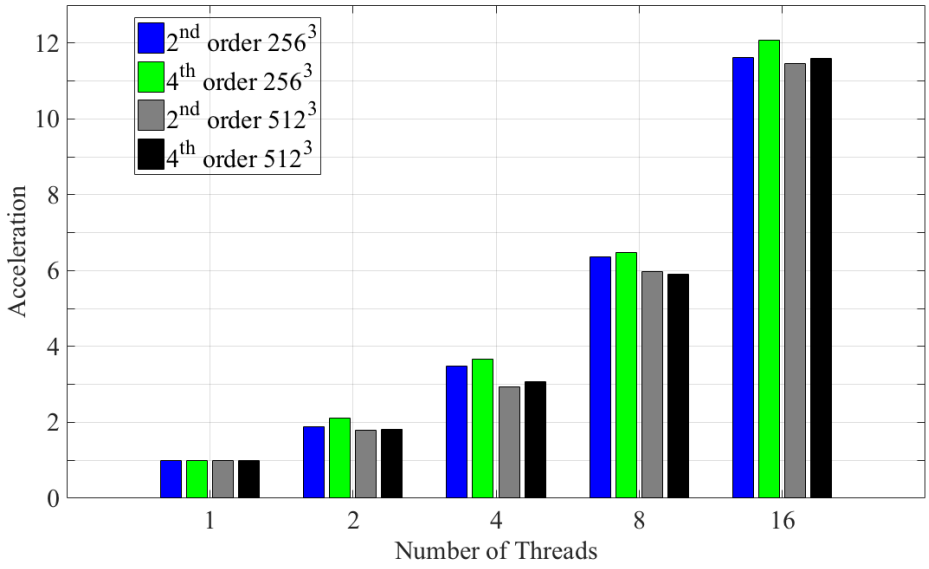


Figure 4.1 displays the acceleration in calculation time for both the second and fourth order programs. These programs used default OpenMP loop division and calculates the forward and reverse transforms using one dimensional FFT of a single column in a x, y plane of the domain. The results come from INL's Falconviz cluster using two uniform computational grids, $N_x = N_y = N_z = 256$ and $N_x = N_y = N_z = 512$. These results show the desired approximately linear acceleration.

4.4 MPI

In this section a process will refer to either an individual processor or a compute node of a cluster as both can be used in MPI. As described in Chapter 2, large enough

computational grid sizes require the use of MPI. Therefore, the sequential program was modified to run on several nodes allocating only the minimum required memory on each. This was accomplished by dividing the computational domain as evenly as possible as described in the introduction section of this chapter. In turn, this enables the ability to run with much larger computational grids, as the program is no longer limited by the memory of a single node.

Algorithm 6 MPI 3D Helmholtz Solver

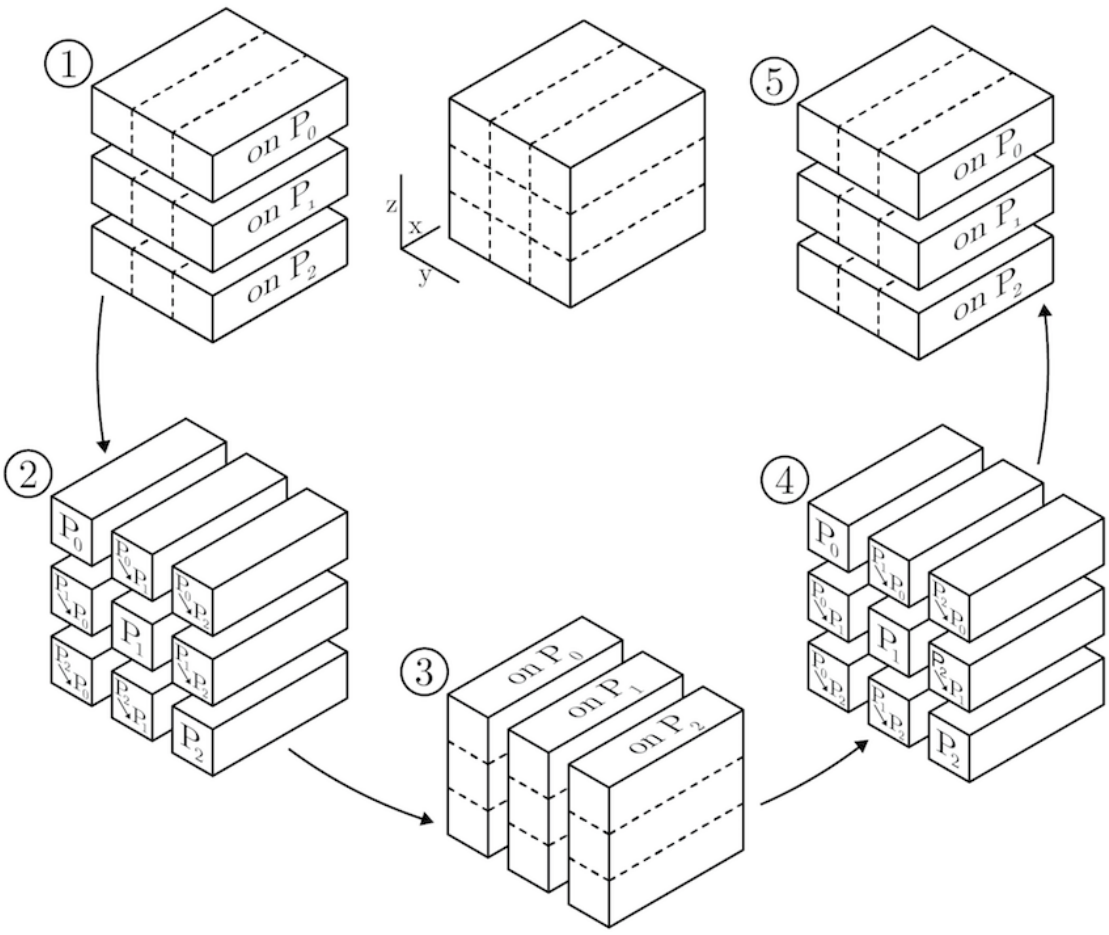
1: Find $start_y, start_z, end_y$, and end_z
2: **for** $k = start_z, \ldots, end_z$ **do**
3: 2D forward DST in $x-, y-$direction
4: **end for**
5: Send and receive data via MPI functions
6: **for** $j = start_y, \ldots, end_y; i = 1, \ldots, N_x; k = 1, \ldots, N_z$ **do**
7: Solve the tridiagonal system using LU decomposition
8: **end for**
9: Send and receive data via MPI functions
10: **for** $k = start_z, \ldots, end_z$ **do**
11: 2D reverse DST in $x-, y-$direction
12: **end for**
13: Send and receive data via MPI functions

A major difference from the OpenMP implementation is that the entire program must be parallelized, including the LU decomposition. The reasoning is the L and U arrays are length $N_x \times N_y \times N_z$, requiring vast amount of memory for large grid sizes. To successfully run on multiple nodes, each node needs only the necessary parts of the arrays.

Figure 4.2 gives a graphical example of the transfer of data between processes. For ease of illustration the figure assumes the use of three processes. The processes are denoted P_0, P_1 and P_2. The three dimensional computational domain, in its whole, is displayed in the center of Figure 4.2. The first step shows how the domain is divided as evenly as possible among the three processes with respect to the vertical, z direction. This is where the forward transform is computed since the calculations do not depend on

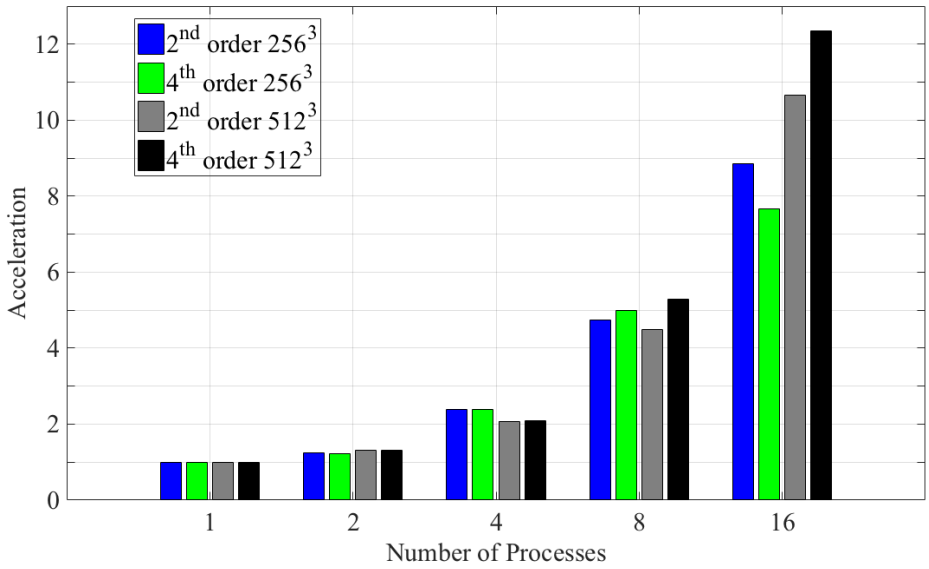
z. Once the FFT is computed, certain parts of the domain need to be sent to different processes as the tridiagonal solver is independent with respect to the y direction. The second step illustrates the specific sections that need to be sent and their destination process. The sections along the diagonal will not be sent as they currently reside on the appropriate process. The third step shows the sections of the domain assembled on the appropriate process after receiving the messages sent in the second step. Now the domain is divided as evenly as possible among the processes with respect to the y direction so that the tridiagonal solver can be calculated in parallel. The fourth step is simply the reverse of step two. The fifth and final step has domain divided with respect to the z direction and can now calculate the reverse transform.

Figure 4.2: 3D Transfer of Information between MPI Processes



As in the implementation of the two dimensional MPI algorithm, this program was run in two ways. Firstly, it was run on one machine to compare to OpenMP. Then several compute nodes were used to test the ability to approximate with very large computational grids. The results that follow are those recorded from Falconviz and use the uniform computational grids, $N_x = N_y = N_z = 256$ and $N_x = N_y = N_z = 512$. Both second and fourth order are shown.

Figure 4.3: 3D MPI Acceleration by Processors on One Node
Acceleration is the Sequential Time Divided by Parallel Time



The graph in Figure 4.3 shows the acceleration on a single node ranging from one to sixteen processes. Similar to the two dimensional case, these accelerations are not as consistent as in the implementation of OpenMP. However, the acceleration is roughly linear.

Order	Grid Size	Acceleration
Second	1024^3	1.2167
Fourth	1024^3	1.1074

Table 4.1: 3D MPI Acceleration by Nodes
Acceleration is the Sequential Time Divided by Parallel Time

Arguably the greatest attribute of MPI is that it can be used to run calculations across several nodes. The memory of a single node on INL's Falconviz is overrun when running the program with a grid size of 1024^3. To run such a memory intense computation more nodes are needed. With this grid size, the program was run using one to sixteen nodes. The required memory was too great to complete the computations until the number of nodes reached eight. The results in Table 4.1 show the accelerations from eight to sixteen nodes.

4.5 CUDA

The final parallel technology considered is CUDA. This is used to perform calculations on a GPU. This provides great potential for accelerating algorithms. As stated in Chapter 2, a program utilizing CUDA can not be executed solely on a GPU. In the two dimensional case the only the forward and reverse transforms were calculated on the GPU. In this section, the implementation moves the computation of both forward and reverse transforms to a kernel function. In addition, the tridiagonal solver is computed in a kernel function on the GPU. There is significant latency when transferring data between the CPU and GPU. Therefore, this communication must be minimized to observe a better acceleration. The following results were computed using INL's Falconviz and demonstrate the benefit of using GPUs to acclerate a normally sequential program.

Order	Grid Size	Acceleration
Second	256^3	2.792
	512^3	4.0543
Fourth	256^3	2.9019
	512^3	3.4775

Table 4.2: 3D CUDA Acceleration
Acceleration is the Sequential Time Divided by Parallel Time

The accelerations displayed in Table 4.2 show a drastic improvement over those in the two dimensional case. This improvement is made possible by computing the tridiagonal solver on a GPU. It is possible that the kernel functions perform better than the use of CUDA's functions that mimic those in FFTW.

4.6 Results

The results of these three different parallelizations demonstrated the impressive improvements in computation time verses the sequential run time. It should be noted that the presented implementations of the algorithms were successfully run on several personal computers and several clusters, including: Idaho State Universitys Leibniz, the Falcon, Falconviz and Bechler clusters at INL and the Blue Waters cluster funded by the National Center for Supercomputing Applications. The Falconviz cluster was chosen to compare results in this thesis for several reasons, primarily, its ability to run CUDA programs. In addition, the Falconviz cluster has relatively low traffic and it has a large memory capacity. Another noteworthy feature is it has four NVIDIA Quadro K6000 GPUs available for calculation. The Falconviz cluster has one terabyte of RAM with eight sixteen core 2499.799 MHz processors making it incredibly powerful tool in testing the speed up provided by doubling the number of OpenMP threads or MPI processes.

4.7 Conclusion

This chapter has shown the power of parallel computing when applied to numerical algorithms. The three parallel technologies, OpenMP, MPI and CUDA, were used to create three variant programs. A comparison of all three was reviewed.

Chapter 5

Future Work

There are several ways to expand upon the work done in this thesis. Firstly, the CUDA implementation can be expanded to include the use of more than just one GPU. Though it is uncommon among clusters and especially personal computers to have more than one GPU per node there is the ability to include multiple GPUs per node. INL's Falconviz cluster includes four GPUs on each node allowing the investigation of acceleration due to multiple GPUs.

In this thesis the second and fourth order approximations were considered. There is a plan to implement a sixth order approximation algorithm. This will be done with methods similar to the methods already presented. That is, the sixth order scheme will be programmed to use OpenMP, MPI and CUDA. The accelerations found will then be compared to those presented in this thesis.

OpenMP was shown to be of great use on a single node of a cluster, but in the use of multiple nodes, MPI is required. It is possible to combine the two into a hybrid program. MPI can be used to divide the work up among multiple nodes and OpenMP then used to divide the work among the threads to each processor on the node. Future work will examine the benefits of such an implementation.

As mentioned in Section 4.3, a function in the FFTW subroutine using multithreading for a shared memory structure was investigated. There is a similar FFTW function for distributed memory. This was not considered in this thesis and future research will

look for a benefit over the current method.

This thesis considered methods that can be applied to parallel Krylov-FFT type high-resolution algorithms for subsurface electromagnetic scattering problems. The development of this iterative method is an implementation of a fast multigrid solver for the convection-diffusion part of the discretized Navier-Stokes system. Future work will implement the methods developed in this thesis into this system.

The results presented in the previous chapter demonstrated the power of parallel computing. Not only can the presented algorithms be expanded to use in the discretized Navier-Stokes system, but in countless other algorithms. The development of sequential numerical algorithms should become a thing of the past to give way for the development of parallelizable numerical algorithms.

Acronyms

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DFT Discrete Fourier Transform.

DST Discrete Sine Transform.

FFT Fast Fourier Transform.

GPU Graphics Processing Unit.

MPI Message Passing Interface.

OpenMP Open Multi-Processing.

RAM Random Access Memory.

Glossary

acceleration Reduction in calculation time.

FFTW Open source C subroutine library for calculating FFT.

parallel Running computations across multiple computational units simultaneously.

race condition The updating of a variable's value in the incorrect order.

Bibliography

[1] L. Ahlfors, *Complex analysis*, Third Edition, McGraw-Hill, Inc., 1979.

[2] I. Babuska and S. Sauter, *Is the pollution effect of the FEM avoidable for the Helmholtz equation considering high wave numbers?*, SIAM Journal on Applied Mathematics **42** (2000), pp. 451–484.

[3] V. Eijkhout, *Introduction to high performance scientific computing*, Second Edition, Lulu, 2015.

[4] M. Frigo and S. Johnson, *FFTW Manual*, Massachusetts Institute of Technology, 2003.

[5] G. Hager and G. Wellein, *Introduction to high performance computing for scientists and engineers*, First Edition, CRC Press, 2010.

[6] R. Kress, *Numerical analysis*, First Edition, Springer, 1998.

[7] S. Lang, *Linear algebra*, Third Edition, Springer, 1987.

[8] R. Larson and R. Hostetler, *Trigonometry*, Fifth Edition, Houghton Mifflin Company, 2001.

[9] *CUDA Manual*, NVIDIA Corporation, 2017.

[10] Y. Gryazin, *Preconditioned Krylov subspace methods for sixth order compact approximations of the Helmholtz equation*, unpublished manuscript (2012).