2018

A Parallel Spectral Method Approach to Model Plasma Instabilities

Kevin S. Scheiman
Wright State University

A PARALLEL SPECTRAL METHOD APPROACH TO MODEL PLASMA INSTABILITIES

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science

By

KEVIN S. SCHEIMAN
B.S., Wright State University, 2017

2018
Wright State University

Wright State University

GRADUATE SCHOOL

April 27, 2018

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERIVISION BY Kevin S. Scheiman ENTITLED A Parallel Spectral Method Approach to Model Plasma Instabilities BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

_____

Amit Sharma, Ph. D.

Thesis Advisor

_____

Jason Deibel, Ph. D.

Chair, Department of Physics

Committee on Final Examination

_____

Amit Sharma, Ph. D.

_____

Brent Foy, Ph. D.

_____

Ivan Medvedev, Ph. D.

_____

Barry Milligan, Ph. D.

Interim Dean of the Graduate School

Abstract

Scheiman, Kevin S. M.S. Department of Physics, Wright State University, 2018. A Parallel Spectral Method Approach to Model Plasma Instabilities.

The study of solar-terrestrial plasma is concerned with processes in magnetospheric, ionospheric, and cosmic-ray physics involving different particle species and even particles of different energy within a single species. Instabilities in space plasmas and the earth's atmosphere are driven by a multitude of free energy sources such as velocity shear, gravity, temperature anisotropy, electron, and, ion beams and currents. Microinstabilities such as Rayleigh-Taylor and Kelvin-Helmholtz instabilities are important for the understanding of plasma dynamics in presence of magnetic field and velocity shear.

Modeling these turbulences is a computationally demanding processes; requiring large memory and suffer from excessively long runtimes. Previous works have successfully modeled the linear and nonlinear growth phases of Rayleigh-Taylor and Kelvin-Helmholtz type instabilities in ionospheric plasmas using finite difference methods. The approach here uses a two-fluid theoretical ion-electron model by solving two-fluid equations using iterative procedure keeping only second order terms. It includes the equation of motion for ions and electrons, the continuity equations for both species, and the assumption that the electric drift and gravitational drift are of the same order.

The effort of this work is to focus on developing a new pseudo-spectral, highly-parallelizable numerical approach to achieve maximal computational speedup and efficiency. Domain decomposition along with Message Passing Interface (MPI) functionality was implemented for use of multiple processor distributed memory computing. The global perspective of using Fourier Transforms not only adds to the accuracy of the differentiation process but also limits memory calling when performing calculations. An original method for calculating the Laplacian for a periodic function was developed that obtained a maximum speedup of 2.98 when run on 16 processors, with a theoretical max of 3.63. Using this method as a backbone for parallelizing the RT-KH solution, the final program achieved a speedup of 1.70 when calculating only first order terms, and 1.43 when calculating up to second order.

Table of Contents

List of Figures

steps surpasses 17,000 (shown here for 17400, 18000, 18600, and 19200), the anomaly can be seen to develop everywhere but the middle of the x grid space, where the shear velocity is approximately zero. The same anomaly occurs for the electric potential.

List of Tables

Chapter 1

Introduction

Plasmas in general make up most of matter in the universe. As a result, there are many different categories of plasmas, ranging from solar, to interstellar, to atmospheric. A sub-branch of space plasma is the Earth's ionosphere – the region of Earth's atmosphere that is ionized.

The abundance of plasma in vast region around the earth has a great influence on communications: both surface based and surface-to-space. As a result, the understanding of the physics principles of this plasma is a topic of constant study. Since experimental setups for plasmas are costly endeavors, computational approaches are a far more common means of analyzing plasma behavior. However, programs capable of such simulations tend to be rather cumbersome due to the sheer volume of calculations being performed and large variation in spatial and temporal scale of plasma processes. A concerted effort has been made to optimize the process beyond simple limitations to memory storage. The need for a faster, more accurate means of modelling is discussed in the present work.

1.1 Ionospheric Plasmas

Earth's ionosphere – ranging from 80 km (50 miles) to 800 km (500 miles) – is an atmospheric region encapsulating a large volume of space around the earth and within the magnetosphere. The reason for this special categorization is that this region is highly populated by electrically charged particles. Light from the Sun – specifically ultraviolet and high x-rays – bombards Earth's upper atmosphere and excites electrons to leave their accompanying atom, resulting in an ionized gas called a *plasma*.

The concentrations of plasmas vary throughout the ionosphere, being separated between the D, E, and F layers, with respect to altitude. Each layer has different concentrations of electrons due to the elements present within it: heaviest being the lowest. These layers lie in the already familiar atmospheric layers, as shown in Figure 1.1, but are not strictly defined as their altitudes change throughout the day and between seasons. Since the Sun is a primary contributor to this excitation, the characteristics of the ionosphere change from day to night. During the night, cosmic rays ionize the ionosphere far less strongly than the Sun during the day, so it is less charged [1].

At heights of 80km (50 miles), in the thermosphere, the atmosphere is thin and electrons can exist for short period before being captured by a positive ion, however, number of electrons in this region is sufficient to effect radio communications. The D region (50-90 km) of the ionosphere, due to low level of ionization is quite different from that of weakly collisional or collisionless plasmas. The D-layer is produced by the hydrogen Lyman-alpha line at 121.5 nm and hard X-rays.

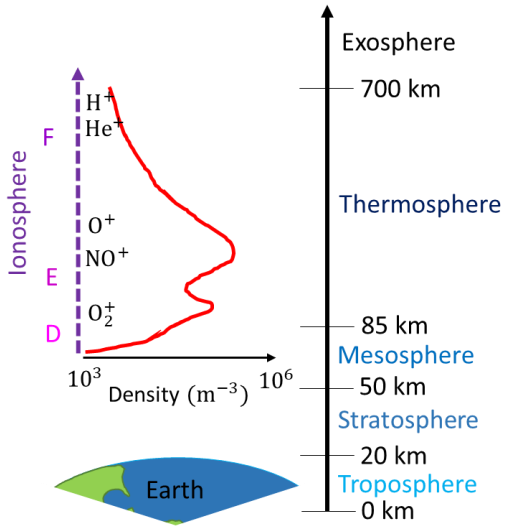


Fig. 1.1. Earth's atmosphere layers, including the ionosphere. The ionosphere can be divided into three regions – D, E, and F – distinguished by the density in the region [3].

The Mesosphere (45-85 km) and lower Thermosphere constitutes the E-layer (90-120 km) formed by photoionization of molecular oxygen by radiation in the 100-150mn range and by soft X-rays of 1-10nm. The ion composition of the E-layer is mostly O_2^+ and NO^+ ions. The F-layer of the ionosphere ranging from approximately 120-500km is largest region of ionized atmosphere produced by photoionization of atomic oxygen by extreme UV radiation in the 10-100nm range.

Plasmas, unlike other gases, can conduct electric charges and are susceptible to magnetic fields. Due to this conduction property, plasmas are highly susceptible to Thomson scattering, where the free electrons emit radiation of the same frequency as that of the incident wave. The direction of the scattered signal is determined by the angle of the electron's rotatory motion about its axis (i.e. its spin) [2]. Radio signals and other forms of communication use ionosphere to reflect electromagnetic waves off the charged particles in the ionosphere to extend signals beyond the visible curvature of the Earth.

1.2 Rayleigh-Taylor & Kelvin-Helmholtz Instabilities

In fluid mechanics, Rayleigh-Taylor and Kelvin-Helmholtz (RT-KH) instabilities are associated with layers of fluid with a discontinuous change in tangential velocity and/or density across an interface. In hydrodynamics, RT instabilities can be observed when a heavier fluid is suspended above a lighter one under the influence of gravity, as shown in Figure 1.2.1. This class of instability accounts, for example, for the over-turning of fluid and also accounts for buckling of interfaces subject to shear motions. A common example is water suspended above oil. Such a system is naturally unstable (so long as no magnetic fields are present), as due to gravity or acceleration, the lighter fluid (oil) will rise and penetrate through the heavier (water). If a magnetic field is present, then the system can be stabilized along its direction, although perpendicular perturbations are still unstable [4].

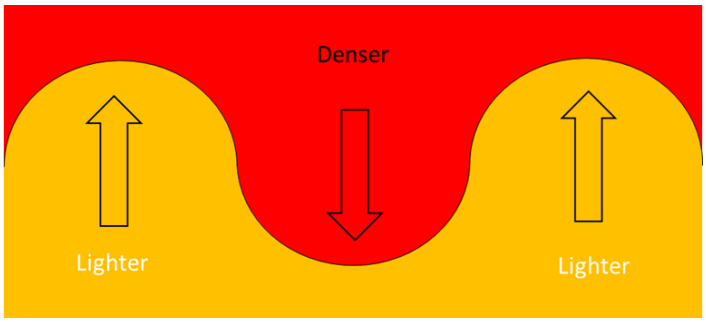


Fig. 1.2.1. Basic example of Rayleigh-Taylor type instability, with arrows showcasing the direction of motion of the perturbations.

KH instabilities are the exact opposite to RT in that they occur parallel to the boundary plane between the two mediums. This occurs when one or both of the fluids experiences motion parallel to the other. If one begins with a small perturbation upwards, the flow above the perturbation begins to speed up in order for the same amount of medium to pass

through while the flow below slows down since there's more space. This is similar to the Bernoulli effect creating lift for an airfoil. As a result, the perturbations grow and are swept along by the flow, resulting in curls, shown in Figure 1.2.2. Wind blowing across a lake is an obvious example, as the velocity of the wind parallel to the water's surface creates a rippling effect in the water – or, in the case of the ocean, waves.

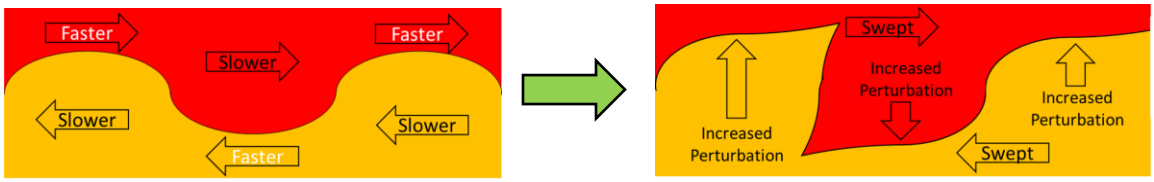


Fig. 1.2.2. Basic example of Kelvin-Helmholtz type instability. The left image shows the initial perturbations and the right image shows the eventual evolution of the system.

For plasmas, in absence of magnetic field, the shear velocity – which is proportional to the velocity differences between the mediums – always results in KH instabilities. However, the presence of a magnetic field and its direction has different effects on plasmas depending on whether it is fully or partially ionized. In particular, partially ionized plasmas – such as those found in Earth's ionosphere – still experience instabilities when the magnetic field is applied along the flow direction [4].

1.3 Turbulence Equations

The RT-KH instabilities examined here are governed by two primary equations relating the charge density and electric potential of the plasma with respect to time. They are used to study the evolution of very low frequency $\omega/\omega_{ci} \ll 1$ electrostatic waves (low in relation to the plasma's cyclotron frequency ω_{ci}) and can be seen as a negatively charged electron fluid moving against a positively charged ion background. For their derivation, a slab of plasma in an equilibrium state was first assumed (shown in Figure 1.3), where the ion and electron charge densities were initially inhomogeneous. The slab can be thought of as being an equatorial plane around the Earth so that Earth's magnetic field is perpendicular to the plane of interest and homogenous, ensuing that RT phenomenon are not suppressed, as shown in Figure 1.3.

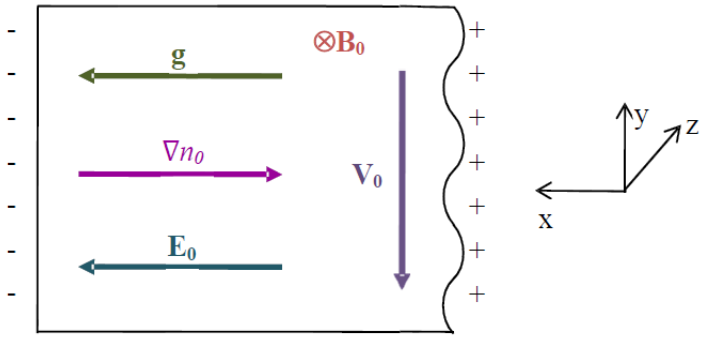


Fig. 1.3. Equilibrium geometry used in deriving equations of motion.

The electric field, E_0, and number densities of ions, n_{i0}, and electrons, n_{e0}, were inhomogeneous at initial equilibrium, with the density gradients in the opposite direction of the gravitational acceleration. The velocity profile of the fluid, V_0, is set within the slab to be perpendicular to the gravitational pull and represents the shear strength between the ion and electron fluids.

Collisional effects and the inertia of electrons are neglected (assuming the electron mass is approximately zero). The equations of motion for the ions and electrons becomes:

$$m_i n_i \left[\frac{\partial V_i}{\partial t} + (\boldsymbol{V}_i \cdot \nabla)\boldsymbol{V}_i\right] = eZn_i \left[\boldsymbol{E} + \frac{1}{c}(\boldsymbol{V}_i \times \boldsymbol{B})\right] - T_i \nabla n_i - \nabla \cdot \boldsymbol{\Pi} + m_i n_i \boldsymbol{g} \tag{1.3.1}$$

$$0 = -en_e \left[\boldsymbol{E} + \frac{1}{c}(\boldsymbol{V}_e \times \boldsymbol{B})\right] - T_e \nabla n_e \tag{1.3.2}$$

Where V_i and V_e are the ion and electron velocities, T_i and T_e are the ion and electron temperatures, and n_i and n_e are the ion and electron charge densities, respectively. The two systems must also satisfy their continuity equations:

$$\frac{\partial n_i}{\partial t} + \nabla \cdot (n_i \boldsymbol{V}_i) = 0 \tag{1.3.3}$$

$$\frac{\partial n_e}{\partial t} + \nabla \cdot (n_e \boldsymbol{V}_e) = 0 \tag{1.3.4}$$

Where the electron charge density is given by $n_e = Zn_i$ for ion charge Z. Overall charge neutrality for the system will also be assumed:

$$\boldsymbol{J} = Zen_i \boldsymbol{V}_i - en_e \boldsymbol{V}_e \tag{1.3.5}$$

The electric drift and gravitational drift will also be assumed to be of the same order. The final time dependent equations are then derived using collinear theory, where the final state of the system is given by the sum of the initial state and the perturbation.

$$\frac{\partial \Delta\delta\varphi}{\partial t} + \left[V_0 + \rho_i^2\omega_{ci}\kappa_n - \frac{g}{\omega_{ci}} - \frac{\rho_i^2}{2}(V_0'' + \kappa_n V_0'')\right]\frac{\partial \Delta\delta\varphi}{\partial y} - (V_0'' + \kappa_n V_0')\frac{\partial \delta\varphi}{\partial y}$$
$$- \frac{\omega_{ci}B_{0z}}{n_{i0}c}\rho_i^2\frac{g}{\omega_{ci}}\frac{\partial \Delta\delta n}{\partial y} + \frac{\omega_{ci}B_{0z}}{n_{i0}c}\left(\frac{g}{\omega_{ci}} - \rho_i^2 V_0''\right)\frac{\partial \delta n}{\partial y}$$
$$- \frac{\omega_{ci}B_{0z}}{n_{i0}c}\rho_i^2 V_0'\frac{\partial^2 \delta n}{\partial x \partial y} \tag{1.3.6}$$
$$= -\frac{c}{B_{0z}}\{\delta\varphi, \Delta\delta\varphi\} + \frac{\rho_i^2\omega_{ci}}{n_{i0}}\nabla\cdot\{\nabla\delta\varphi, \delta n\}$$

$$\frac{\partial \delta n}{\partial t} + V_0\frac{\partial \delta n}{\partial y} - \frac{n_0 c}{B_{0z}}\kappa_n\frac{\partial \delta\varphi}{\partial y} = \frac{c}{B_{0z}}\{\delta n, \delta\varphi\} \tag{1.3.7}$$

Here, $\delta\varphi$ and δn are the electric potential and charge density perturbations (respectively) to the initial equilibrium state of the system. The dimensionless form of these equations is then obtained after normalizing and using a simple variable substitution:

$$\frac{\partial \Delta\delta\varphi}{\partial t} + \left[V_0 + \tau_i\kappa_n - \nu_g - \frac{\tau_i}{2}(V_0'' + \kappa_n V_0'')\right]\frac{\partial \Delta\delta\varphi}{\partial y} - (V_0'' + \kappa_n V_0')\frac{\partial \delta\varphi}{\partial y}$$
$$- \tau_i g\frac{\partial \Delta\delta n}{\partial y} + \left(\nu_g - \tau_i V_0''\right)\frac{\partial \delta n}{\partial y} - \tau_i V_0'\frac{\partial^2 \delta n}{\partial x \partial y} \tag{1.3.8}$$
$$= -\{\delta\varphi, \Delta\delta\varphi\} + \tau_i\nabla\cdot\{\nabla\delta\varphi, \delta n\}$$

$$\frac{\partial \delta n}{\partial t} + V_0 \frac{\partial \delta n}{\partial y} - \kappa_n \frac{\partial \delta \varphi}{\partial y} = \{\delta n, \delta \varphi\} + D\nabla^2 \delta n \tag{1.3.9}$$

The ion gravitational drift, v_g (which is assumed to be of the same order as the gravitational drift, g, and electric drift), the shear velocity V_0, the plasma density scale length κ_n, and τ_i (given by $\rho_i^2 \omega_{ci}$ for the ion cyclotron frequency ω_{ci} and ion mass density ρ_i) are all constants with respect to time. V_0 can – and in nonuniform cases will – vary spatially along the x-direction but be constant along the y-direction, as shown in Figure 1.3.

For the purpose of generalization, the dimensionless equations 1.3.8 and 1.3.9 will be used for simulations and further analysis. It is obvious to note that they are a pair of coupled differential equations. They are also written so as to separate the linear (with respect to a variable to the first power) and nonlinear (with respect to a variable to the second power) components to the left and right-hand sides of the equations, respectively. The separation of terms allows for calculations of solutions only with linear terms (linear approximation), ignoring the non-linear terms. It may be noted that the $\frac{\partial^2 \rho}{\partial x \partial y}$ term is second order in nature, but because of its outlying coefficient dependent on the shear velocity (or namely its x-derivative) and its job in coupling the equations, it is considered a mandate to be calculated even when excluding the other non-linear terms.

An important feature to note for Equation 1.3.8 is the recurrence of potential's Laplacian, $\Delta\delta\varphi$, which is also known as the vorticity. Vorticity is typically a vector quantity describing the rotation along the edges of RT instabilities, but here it is a scalar quantity.

Although this is how the system is evolved in time, in application, the vorticity itself is largely an uninterested factor. The electric potential, charge density, and electric field are the three most common parameters sought after. Charge density is provided from equation 1.3.9, but the potential and electric field must be solved for:

$$\Delta\delta\varphi = \nabla^2\delta\varphi = \frac{\partial^2\delta\varphi}{\partial x^2} + \frac{\partial^2\delta\varphi}{\partial y^2} \tag{1.3.10}$$

$$\boldsymbol{E} = -\boldsymbol{\nabla}\delta\varphi = -\frac{\partial\delta\varphi}{\partial x}\hat{\boldsymbol{x}} - \frac{\partial\delta\varphi}{\partial y}\hat{\boldsymbol{y}} \tag{1.3.11}$$

Both 1.3.10 and 1.3.11 are given for a two-dimensional space. Equation 1.3.10 is strictly Poisson's equation, solving for $\delta\varphi$. The electric field's vector components (Equation 1.3.11) and magnitude are then easily obtained from the potential using 1.3.10.

Chapter 2

Computational Differentiation

Most physical processes are governed either by ordinary or partial differential equations. Solving ODEs or PDEs equations is an essential part of solving even the simplest time dependent computer simulations. Their abundance in physical problems has produced many different techniques to obtain a solution. From fast executable equations to spot-on transformations, the routes these processes take is important to consider in higher level programming to ensure efficient memory allocation, runtimes, and solution accuracies. Here, two vastly different approaches to solving differential equations numerically will be examined: the widespread and simplistic finite difference method, and the highly precise spectral method.

The discussion of all finite difference methods is one that has been the subject of countless books and research topics in of themselves, and so a brief description of their application process will be given. Their accuracy and performance compared to spectral differentiation will be the primary focus of this chapter.

2.1 Finite Difference Method

The main concept in any finite difference method is largely explained by the name itself: using a finite basis to solve differential equations through the use of difference equations. These equations are relatively simple in nature, including basic algebraic operations dependent on functional values. The discrete values become a subject of intense analysis regarding accuracy and executable runtimes in programming. Computers, although capable of thousands upon millions of calculations per second, are still finite machines, and so performing perfect differentiation on a function thought of as being smooth and continuous will always result in an approximation of the solution. Even the best finite difference techniques will still have an inherent amount of error since they do not take into account the entire function in their calculations. The reason behind this is in their means of turning the derivative into a difference expression.

There are several different ways to produce finite difference equations, but the most common is by using a Taylor series expansion. Expanding the desired function into a Taylor series, solving for the appropriate derivative, and making a general approximation to remove excess terms gives a solution that consists of only values of the original function. However, since the Taylor series itself is an infinite sum, further discretization errors occur due to computational limits [5].

The use of the Taylor series also brings up the subject of accuracy. Expanding the series using more and more terms naturally results in a better approximation to the original function. As a result, simple operations (such as derivatives) tend to have multiple finite

difference equations for varying degrees of accuracy. Some finite difference equations are shown in Table 2.1 as examples.

Table 2.1. Various examples of finite difference equations. The subscripts denote indexed values of the function's grid-space based off position i. The distance between two points are a constant value, $|f_i - f_{i+1}| = h$ [6].

Operation	Finite Difference Expression	Error
$\frac{df(x)}{dx}$	$\frac{f_{i+1} - f_i}{h}$	$O(h)$
	$\frac{f_{i+1} - f_{i-1}}{2h}$	$O(h^2)$
	$\frac{-f_{i+2} + 8f_{i+1} - 8f_{i-1} + f_{i-2}}{12h}$	$O(h^4)$
$\frac{d^2f(x)}{dx^2}$	$\frac{f_{i+2} - 2f_{i+1} + f_i}{h^2}$	$O(h^2)$
	$\frac{f_{i+1} - 2f_i + f_{i-1}}{h^2}$	$O(h^2)$
	$\frac{-f_{i+2} + 16f_{i+1} - 30f_i + 16f_{i-1} - f_{i-2}}{h}$	$O(h^4)$
$\frac{\partial f(x,y)}{\partial x \partial y}$	$\frac{1}{4h^2}\left(f_{i+1,i+1} - f_{i+1,i-1} - f_{i-1,i+1} + f_{i-1,i+1}\right)$	$O(h^2)$
$\nabla^2 f(x,y)$	$\frac{1}{12h^2}\left(-60f_{i,i} + 16\left(f_{i+1,i} + f_{i,i+1} + f_{i-1,i} + f_{i,i-1}\right) - \left(f_{i+2,i} + f_{i,i+2} + f_{i-2,i} + f_{i,i-2}\right)\right)$	$O(h^4)$

This idea of using more terms for better results not only is applicable to the equation itself, but also to the function. Computationally, the discrete points of a function are most easily thought of as lying on a uniform grid, regardless of dimensionality (1D, 2D, or 3D). So, the more points defined within the function, the better its resolution. Using a uniform grid-space with relatively few points while applying a finite difference scheme to it will give poor numerical results.

Whereas the grid-space is an attribute that can be fiddled with to provide the best balance between performance and accuracy, any lone finite difference method completely lacks the ability to compute at least one border of its space. Since any given point in the grid-space, i, is dependent on at least one of its neighbors, $i \pm 1$, the very edge(s) of the space is a fundamental flaw due to lack of points. Therefore, even though the higher accurate expressions will give better results, it will simultaneously increase the point width (the width in respect to grid points; not necessarily actual numerical values of the function) of the non-calculatable border. This is shown in Figure 2.1.1 for a basic sine function and using the $\mathrm{O}(h)$ and $\mathrm{O}(h^4)$ equations for the first derivative in Table 2.1.

The greatest hinderance this border inaccessibility causes is in relation to boundary conditions. If a very specific set of boundaries need to be maintained, then the finite difference method will be hard to use effectively as it can never truly reach the very edge of the grid-space. As the number of points in the function's domain is increased (thereby decreasing Δx, or h), the valid region of numerical results from a finite difference scheme will continually approach the boundaries, but at an exponentially decreasing rate (i.e. a diminishing return). There comes a point when adding more points to the grid-space makes a negligible difference in the accuracy of the method used. An example of this effect is shown in Figure 2.1.2 for the same sample sine function used previously.

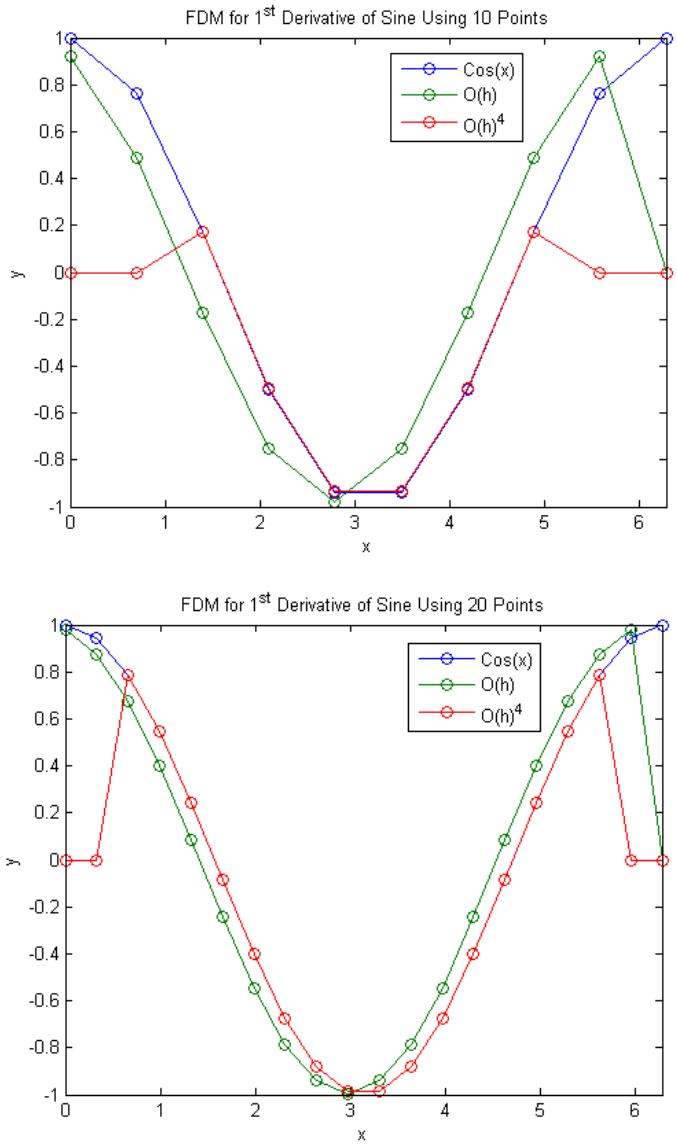


Fig. 2.1.1. Two finite difference methods for calculating $\frac{d}{dx}\sin(x)$. Whereas the $O(h)$ scheme is less accurate for the bulk of the function, it gives more values on the given domain. As seen with the $O(h^4)$ equation, two points bordering the edge of the grid-space are unable to be calculated unless another method is used. Increasing the overall number of points increased the valid domain of the $O(h^4)$ equation from $1.369 \rightarrow 4.887$ to $0.6614 \rightarrow 5.622$, corresponding to a valid domain increase of 4.1% per additional point.

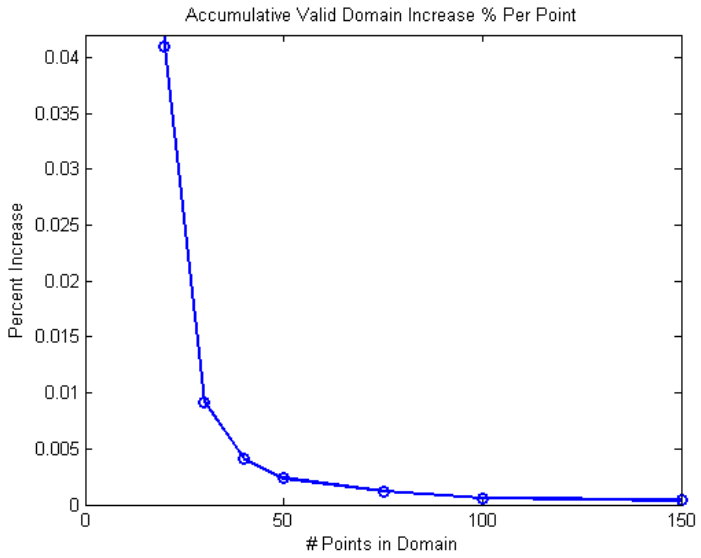


Fig. 2.1.2. The effect of adding more points to a uniform grid-space. As more points are added, the valid range of the $O(h)^4$ finite difference method for $\frac{d}{dx}\sin(x)$ increases, but the increase each additional point provides decreases exponentially. The values plotted here are accumulative, showcasing the per-point percent increase from each successive grid-space expansion – from 10 to 20, 20 to 30, 30 to 40, and so on.

Although this is fine in theory, computationally it becomes cumbersome and inefficient. Adding more points equally increases the amount of storage necessary to run the program., which becomes severally detrimental for algorithms that are already computationally and memory intensive.

The simplistic algebraic nature of finite difference methods also poses another hindrance: execution time. Although the implementation will be finished in $O(n)$ operations, computers must look up each value for the equation individually. Despite the points being relatively local to one another in the grids-space, to a computer each value of the function holds its own space in memory that must be read from to proceed with calculations. This constant calling in memory – looking up the function value at a specific point – may seem

innocent enough but hinders the program's performance. Multiple points in memory must be drawn for calculating just one point of the derivative. Since finite difference equations are most commonly used in loops (calculating the derivative of every point of the function across the grid-space), programs performing thousands of high-precision calculations experience longer execution times than what might be expected because of this continual memory communication.

2.2 Spectral Differentiation Method

Whereas finite difference methods are seen from a local perspective (requiring sets of smaller subsets around distinct points), spectral methods of differentiation take on a global approach to generating solutions. The most common implementation is using the Fourier Transform, converting the entire function from real space to frequency space to make differentiation far simpler. Taking the derivative of a function in real space is rather complicated (as shown with the finite difference method), but in frequency space (or k-space), differentiation can be done by simply multiplying by a constant. The value of this constant determines what operation is being performed on the function – whether differentiation or integration. This is identical to the $O(n)$ operations of finite difference methods but is far more advantageous as there is very limited memory calling – something that finite difference methods lack. Table 2.2 shows the k-space coefficients that would be used to calculate the first or second derivative of a function [8].

Table 2.2. Frequency space algorithms for spectral differentiation. The Fourier Transform of the dataset, Y_k, is first obtained for $0 \leq k < N$ total discrete points, spanning length L. The value of k along the domain determines the coefficient to multiply Y_k by. Y_k is then brought back by an inverse Fourier Transform to obtain the derivative [8].

$k < N/2$	$k = N/2$	$k > N/2$
1st Derivative Coefficients		
$Y_k = Y_k \cdot 2\pi i/L$	$Y_k = 0$	$Y_k = Y_k \cdot 2\pi i(k-N)/L$
2nd Derivative Coefficients		
$Y_k = -Y_k \cdot (2\pi k/L)^2$	$Y_k = -Y_k \cdot (2\pi k/L)^2$	$Y_k = -Y_k \cdot (2\pi i(k-N)/L)^2$

Since the spectral method uses the entire function for determining a solution, they are extremely accurate. It's not uncommon for spectral methods to achieve an accuracy of ten digits where finite difference schemes would get only reach two or three [7]. Figure 2.2.3 shows a simple example of the accuracy variances between spectral and finite difference methods, where for a basic sine function the spectral method of differentiation yields far

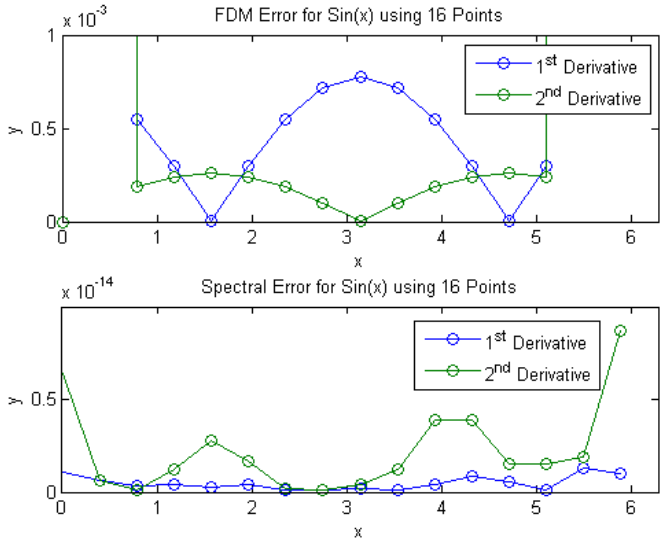


Fig. 2.2.3. The error associated with a sine function from 0 to 2π when using a spectral method (bottom) and a $O(h^4)$ finite difference method (top) from Table 2.1. Double variables were used, which have a maximum of 17 digits. The highest errors for the spectral method are seen near the beginning and end of the function, which is attributed to the Fourier transform's attempt at matching the boundary points. The other error is primarily a result of machine precision. As discussed in Section 2.1, the end points using a FDM are likely going to be invalid due to the lack of adjacent function values.

more accurate results.

Although all the work done in k-space is straightforward, the process of getting there is subject of concern. The most common means of doing so is through the use of the Fast

Fourier Transform (FFT), which many data processing libraries already have included within their basic function calls. The FFT provides a simple and reliable means of implementing a rather complicated procedure. One of the most proliferated FFT libraries is FFTW – aptly named the Fastest Fourier Transform in the West – which even hosts its own parallel methods to further increase its performance. However, even with being highly optimized for performance, spectral differentiation can still take far longer than using finite difference methods specifically because of the forward and backward transformations into k-space.

The forward and backward transformations are the most computationally intensive part, but once in k-space the transformed function can be easily manipulated multiple times and brought back into real space at different points to give the solutions of several different operations all with just one forward transform. This technique is called an out-of-place transform as the transformed data is allocated to new memory and then manipulated. The original function can still be used elsewhere without hinderance. In-place transforms can also be done though in case memory is limited, where the transformed values overwrite the original function. This ability to overwrite in memory is another feature that finite difference methods fail at since they must be able to pull from memory any value of the original function's values. Despite these improvements, spectral differentiation can still take far longer than using finite difference methods specifically because of the forward and backward transformations into k-space.

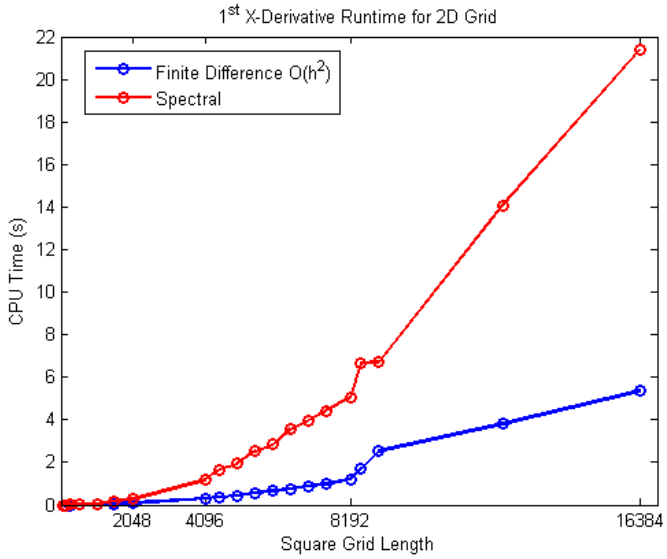


Fig. 2.2.4. CPU time for finite difference and spectral method of calculating the first x-derivative of $\sin(x+y)$ on a periodic square grid with side lengths ranging from 64 to 16384 (2^n for $6 \leq n \leq 14$ with additional points chosen for continuity).

Figure 2.2.4 shows how for relatively small domains (consisting of less than 1024^n points for n dimensions) spectral differentiation is almost identical to finite difference methods (runtimes varying by milliseconds). As the grid size is increased, both methods experience growth, with finite difference being a largely linear increase and spectral being slightly exponential. Both methods experience a sudden jump in runtimes between grid sizes of 8192^2 and 9000^2 which could use more analysis as to the cause. It is therefore important to consider and find the perfect balance between performance and accuracy when choosing between spectral or finite difference methods.

The biggest and most apparent problem with spectral methods is obvious for anyone with knowledge of Fourier Transforms: periodic domains. If the function being operated upon is not periodic and the range being analyzed does not lie on one full period, the Fourier Transform will have artifacts due to the discontinuities at the endpoints. However, there

are methods that enable the use of Fourier Transforms on non-periodic functions. The Chebyshev interpolation is one of the most common and can still be done by using FFTs, so long as the function is still continuous and smooth [8].

Chapter 3

Parallel Computing

The understanding of basic programming techniques is a necessity when wanting to generate complex mathematical or physical simulations. The most generic and well-known method of doing so is simple serial programming: writing a program that will execute one line at a time until it is finished. As computational models have evolved, these basic serial approaches have become impractical due to incredibly long runtimes. As a result, researches and programmers alike transitioned to using multiple computers to work on one problem at the same time. The effort eventually led computing industries such as Intel to create systems that ran on multiple processors instead of just one [9].

The fundamental idea of parallel computing is simple: break apart the serial program and distribute the workload amongst several other processors. With each processor working on a different part of the program simultaneously, computation times decrease substantially. Of course, the largest detriment to parallel processing involves its complex nature, requiring extra hardware and specialized software to run smoothly. However, the necessity for increased computational power has continued to outweigh the cost of specialized equipment.

One of the key subtitles in parallelized programming is memory allocation. In most cases, working on parallelizing a program will only be done if the program is performing complicated calculations, or more typically, handling large amounts of data. Small programs that already run in fractions of a second will not see much performance increase. For larger programs though, the sheer amount of data may be too much for any one processor to hold at a time, hence the inherent need to divide the problem into multiple chunks. As a result, for programs dealing with large amounts of data it is important to dynamically allocate the memory in the heap – the global memory storage. Otherwise, the program may fail entirely due to overloading (or overflowing) the stack – the statically allocated local memory space – which is inherently smaller than the heap.

Due to the intricate mathematical terms in simulating RT-KH instabilities in the ionosphere, the usual easy transition from a serial to parallel implementation proved rather cumbersome. The need for a basic understanding of parallel processing is therefore a necessity before one approaches analysis of the final program, which is outlined here.

The machine used for the performance statistics in this chapter and subsequent was a single 64-bit computer consisting of two Intel Xeon E5-2620 v2 processors, each operating at 2.10 GHz. The machine had 64 GB RAM (8 x 8GB DDR3) operating at a maximum of 1333 MT/s with a 15 MB cache. With two sockets on the motherboard, each supporting two, six core processors, and each core having dual thread capabilities, the system has a total of 24 CPUs. Every program was run up to a minimum of 16 processors.

3.1 Amdahl's Law – Speedup & Efficiency

In basic terms, the main purpose of parallelizing any program is to decrease its runtime. This idea is simple enough, but depending on the situation, the actual improvement is not always so blatantly obvious as just looking at how long it took the program to run. Other factors may be hindering – or facilitating – its performance, and so different tools must be used to analyze the program. In parallel processing, these tools are known as the speedup and efficiency.

The speedup S of a program is a ratio between its serial and parallel execution times, given that both are measured in the same units:

$$S(n) = \frac{T(1)}{T(n)} \tag{3.1.1}$$

The function $T(n)$ represents the amount of time it takes for a program to run on n processors. Therefore, $T(1)$ is the same as running the program in a serial fashion (with one processor performing all calculations). Generally, $T(1) > T(2) > T(3) > \cdots > T(n)$, and so the speedup will always be a quantity greater than one and increasing with increasing number of processors. In theory, $T(n) = T(1)/n$, so the runtime is cut proportional to the number of processors. Of course, the result of (3.1) assuming this perfect scenario is simply a linear relationship between the number of processors and speedup [10].

This brings forth the second tool of analysis: efficiency. Suppose the perfectly ideal parallel execution time $T(n) = T(1)/n$ is observed, and so the number of processors is equivalent to the speedup of the program. In this case, each processor is being used to its fullest extent. In other words, the program is very efficient at doing work; no other factors are limiting the execution time and so maximum speedup is achieved. Numerically, the efficiency can be described as:

$$E(n) = \frac{S(n)}{n} \tag{3.1.2}$$

Where again, $S(n)$ is the speedup and n is the number of processors. So, in a perfect world, the efficiency of all parallel programs would be equal to one no matter how many processors were used. However, the real world does not always produce this perfect product [11].

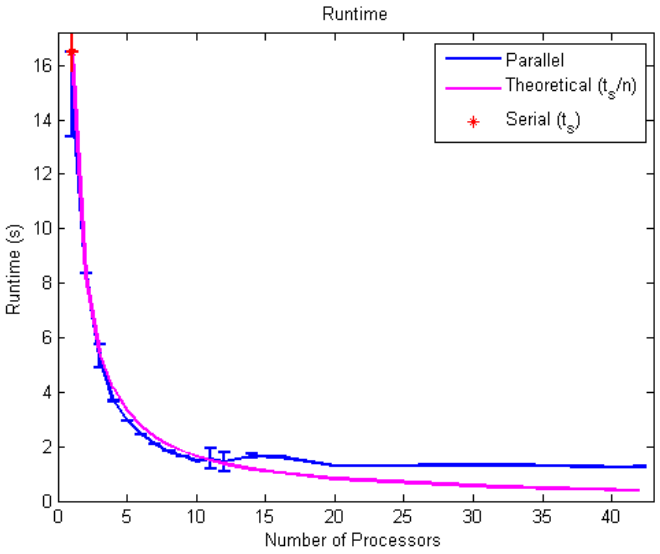
To demonstrate, consider a very simple parallel program that will perform a total of 10^{10} additions, distributed amongst n processors. Each processor will then do $10^{10}/n$ number of calculations. An example pseudo-code in C that each processor will run is shown below:

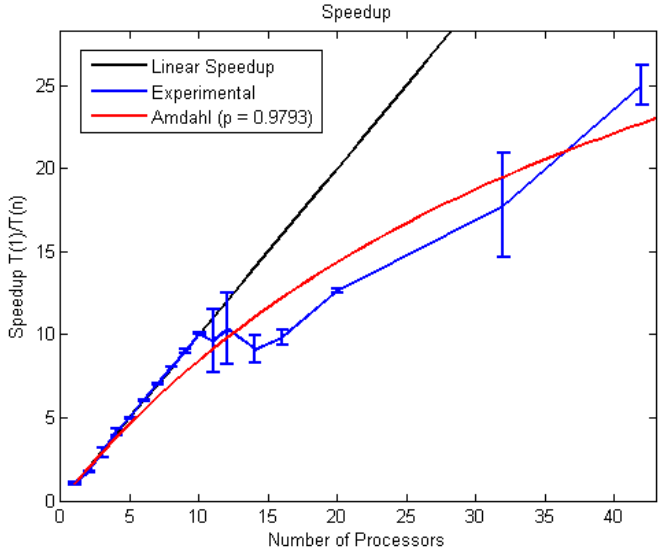
Program 3.1.

```
1.  int main (int argc, char* argv[]) {
2.          int procID, numProcs;
3.          double calcs = 1e10, counter, sum, i;
4.          // . . . Initialize all processors and start runtime clock . . .
5.          for (i = 0; i < calcs/numProcs; i++) {
6.                  counter = counter + 1;
7.          }
8.          MPI_Reduce(&counter, &sum, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
9.          // . . . Output runtime . . .
```

The MPI_Reduce function works to collect and reduce values from all processors down to a single value on the receiving processor. In this case, it is summing together all the "counter" variables from all processors and outputting the results to processor 0. Also note that double variables were used since 10^{10} exceeds the maximum int size in C. This code can also be easily modified to run for any number of processors instead of for just numProcs that evenly divides calcs.

Since this is a largely parallel code (i.e. there is little serial portion to this program), a perfectly linear speedup and a constant efficiency of one is expected. However, this is not exactly the case. First, the runtime plot of Figure 3.1.1 shows that the time it takes the program to run with multiple processors does indeed decrease. As the number of processors is increased though, the runtime seems to approach an asymptotic minimum of one second, showcasing that there is a diminishing return. This is expected, since in theory the serial execution time is simply being divided by the number of processors. As the processor count goes past ten, the experimental and theoretical runtimes begin to vary. This feature can be analyzed better by looking at the speedup and efficiency.


Runtime
Parallel
Theoretical (t_s/n)
Serial (t_s)
Runtime (s)
Number of Processors


Speedup
Linear Speedup
Experimental
Amdahl (p = 0.9793)
Speedup T(1)/T(n)
Number of Processors

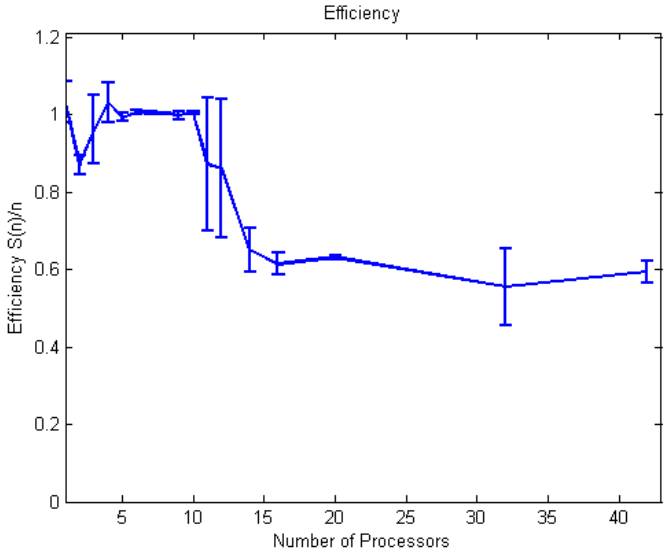

Efficiency
Efficiency S(n)/n
Number of Processors

Figure 3.1.1. Runtime, speedup, and efficiency of Program 3.1 on a 2.1 GHz, 24 CPU (parallel threads) Linux system. Error bars show the standard deviation of three different trials for each n number of processors (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20, 32, and 42). The program was run in serial every time to ensure that parallel runs were compared with their own serial runs (i.e. the program would run in serial and then immediately run in parallel, after recording and resetting the runtime clock). The reason for this is the same reason as to why there are such large error bars: the machine was not running only Program 3.1 at the time. Various background processes were running simultaneously, resulting in varying execution times depending on the intensity of the other background tasks. The Amdahl's Law curve fit for speedup was achieved using Matlab's curve fitting tool, varying the value of p and obtaining a final $R^2 = 0.9502$.

Note that the speedup does appear to be perfectly linear if when the number of processors is relatively low – again below ten. This is a good sign as it shows the program is highly efficient when dividing amongst only a few processors (the point when largest reduction in runtimes should and do occur). As more than ten processors begin being used, the linear trend falls off, and the efficiency of the program begins to diminish. This effect is described in Amdahl's Law.

$$S(n) = \frac{1}{1 - p + p/n} \tag{3.1.3}$$

This relation gives a new representation of the speedup, now completely independent of runtimes. Here, p is the percentage of the program that has be parallelized [12].

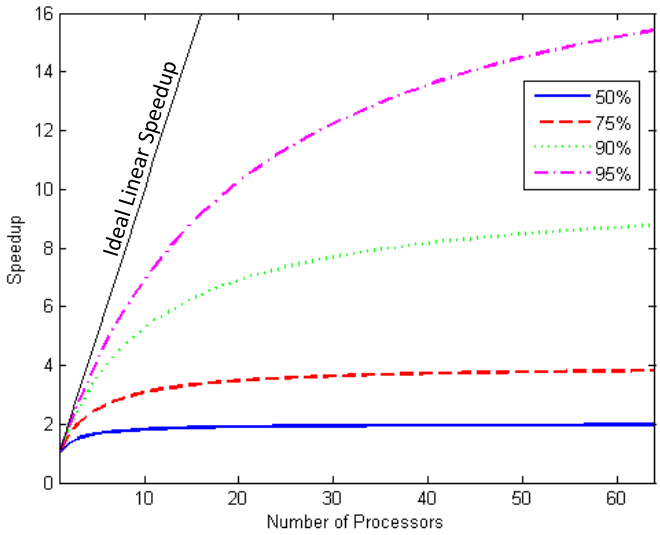


Figure 3.1.2. Amdahl's Law for different percentages of parallelization of a given program. As p approaches 100%, the speedup will become more linear.

Why does the variable p matter? The speedup of a program is limited by its least parallelized component. In other words, any serial part of the program will immediately hinder the overall speedup that can be achieved because that part cannot be run in parallel, shown in Figure 3.1.2. Therefore, according to the curve fitting, Program 3.1 is 97.93% parallelized.

So what accounts for that missing 2.07% of parallelization? First, it should be noted that speedup examinations are typically done under very controlled circumstances. The machine's background processes and hidden operations are stripped to the bare minimum to ensure the most accurate runtime of the program itself. Since the Linux machine used was not setup for this ideal set of testing, there are some inconsistencies.

Although it is difficult to see visually, the serial runtimes are not equivalent for each run. To help minimize the error associated with hidden background processes on the machine, each execution of the program (for each n number of processors) actually involved running the same method twice back-to-back: once serially and once in parallel for n processors. This way each n processor runtime was compared with its own serial runtime at that instant (or close to it), ensuring that no random background process started to chug up processing power half way through testing all values of n. This was done three different times and then averaged to give a good representation of the program's runtime. So, in theory, the execution of the program with $n = 1$ should output two identical runtimes since it's just performing two back-to-back serial operations. However, this is not the case, as shown in Table 3.1.1:

Table 3.1.1. Sample runtime values (in seconds) for Program 3.1 when running on one processor.

Serial Runtime	**Parallel Runtime ($n = 1$)**	**Speedup**
14.043807	13.954585	1.006393741
15.452439	14.097064	1.096145907
16.725916	16.714468	1.000684916
Averages		
15.40738733	14.922039	1.034408188

With only one processor, the speedup should be simply 1.0. The slight variance means that even in the immediate back-to-back execution of the program there is some discrepancy in execution time. This can could be attributed to random machine background operations or to the MPI functions. Although in this example all the parallel $n = 1$ runtimes are faster

than their serial counterparts, it will be shown later (in Chapter 3.4) that this is not necessarily always true.

In this case, the averaged measured speedup is off by a factor of 0.9677 (with a maximum of 0.9993 and minimum of 0.9123) from the theoretical speedup of 1.0 (an error of 3.4408%). Therefore, when executing with n processors, the serial runtime will not be a perfect representation of the parallel n processor's runtime for $n = 1$. As a result, speedups and efficiencies will experience offsets. More importantly, the percent of parallelization p will not be accurate since the serial runtimes are not perfectly comparable. To counter this problem, a new variable will be introduced to Amdahl's Law:

$$S(n) = \frac{a}{1 - p + p/n} \tag{3.1.4}$$

Where the value of a is directly related to the factored offset experience in the serial, T_{serial}, and parallel $n = 1$, $T(1)$, runtimes. Again, in theory (and in highly controlled practice) T_{serial} should be identical to $T(1)$, but because the machine still had unknown operations being performed in the background, the values are different. Since $a > 1$ would mean it'd be possible to achieve a speedup greater than a 1:1 (linear) ratio, it must be mandated that $a \leq 1$, which is given by the following expression:

$$a = 1 - \left|1 - \frac{T_{serial}}{T(1)}\right| \tag{3.1.5}$$

If $T_{serial} > T(1)$, meaning that the serial execution is slower than the $n = 1$ parallel execution (resulting in a speedup greater than 1.0), then 3.1.5 simplifies to just:

$$a = \frac{T_{serial}}{T(1)} \tag{3.1.6}$$

If $T_{serial} < T(1)$, meaning that the serial execution is faster than the $n = 1$ parallel execution (resulting in a speedup less than 1.0), then 3.1.5 ensures that the scaling factor does not violate Amdahl's Law. If $a = 1$, then the serial and parallel $n = 1$ runtimes will be identical, which is identical to the theoretical outcome.

The value of a can be thought of as a normalization factor, ensuring that the serial and parallel $n = 1$ runtimes are identical, thereby aiding in finding a more accurate p value. In our example, Program 3.1, an average factor offset of 0.9667 was experienced for the speedup, meaning that if Equation 3.1.4 were fitted to the experimental speedup data, it should be seen that $a \approx 0.9667$:

Table 3.1.2. Fit data for the speedup of Program 3.1.

$\boldsymbol{p}$	$\boldsymbol{a}$	$\boldsymbol{R^2}$
0.9793 ± 0.0041	1	0.9502
0.9812 ± 0.0039	0.9667364	0.9502
0.9802 ± 0.0107	0.9838 ± 0.1734	0.9503

The values shown in Table 3.1.2 were achieved from using Matlab's curve fitting tool. Since Program 3.1 is already simplistic and largely parallelized, there isn't much of an

improvement to be seen here between $a = 1$ and $a = 0.9667$. If a is allowed to be varied to achieve the best fit possible, $R^2 = 0.9503$ is achieved for $a = 0.9838$. This is slightly off from our experimental averaged offset of 0.9667, but still lies in the maximum/minimum bounds of 0.9123 to 0.9993. The final parallelization percentage can then be best approximated as 98.02%, which is only a small increase from the initial fit using Equation 3.1.3. This gives a maximum speedup of around 49.68, which would occur for $n > 10^6$. This is of course a ridiculous and non-real-world scenario, but it does exhibit how even a highly parallelized program reaches an asymptotic maximum speedup rather than the idealized linear increase.

The missing 1.98% can be attributed to both the small amount of serial code (such as the recording of time) and the communication time between processors. Up until now it has been assumed that the processors can send and receive data instantaneously, which is not the case. Although communication is fast, it is still finite, meaning that there is serial time spent transferring data between processors. This is hidden from our view in the MPI_Reduce function, which both collects and sums all the parts together. It is obvious that inter-processor communication occurs at some point, and that a summation occurs at some point, but the ordering or actual means of doing so is unknown.

There are forms of Amdahl's Law that account for things such as communication times and other serial processes, but experimentally it is difficult to measure these durations on a machine not under strict controls simply because of their small values [10].

3.2 Message Passing Interface

Since the hardware specifications for parallel supercomputers vary so tremendously, the key similarity between applications lie in their means of programming. Parallel computing can be done in a wide array of languages (C, Python, Java, Fortran, etc.) but does require a different thought process than standard serial codes. Many languages have their own built in parallel processing methods, but they may not be fully fleshed out, have few function calls, or their inner workings are hidden from the user behind magic curtains (something that becomes troublesome for complicated and delicate data structures and manipulations). As a result, often it is simpler to default to one of the most widely used parallel programming libraries to get the job done: Message Passing Interface (MPI).

The very first release of MPI was in 1994, and since then the library has expanded considerably, all the way up to a third standard release in 2012, with the latest edits in 2015. Since it is largely considered the standard in message passing libraries (replacing many of its ancestors), its diversity has continued to improve, giving it the customizability wanted by higher level programmers while maintaining relatively a basic structure. Since the library is so vast, this section will focus on the techniques necessary for understanding codes presented later.

MPI is fundamentally based on two means of message passing between multi-processor machines: sending and receiving. These communications can happen from strictly one processor to another (point-to-point communication), or from one processor to multiple

(collective communication). As a result, the entire structure of MPI can be broken down once two functions are understood: MPI_Send and MPI_Recv.

As their names suggest, MPI_Send is responsible for sending data away from a specific processor and MPI_Recv receives the data on another. Therefore, it is not enough for one processor to simply send information; the target processor must also receive the information. The sender will wait until the target has received the data (i.e. a successfully completed communication) before continuing any further in the program, and likewise the target will wait until it has received its data. This can often be a source of frustration where a program simply sits at one point, endlessly waiting for a receive (or send) that is never fully completed. This point-to-point communication style can be thought of as two-step procedure, shown in Fig. 3.2.1.

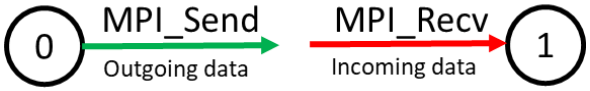


Fig. 3.2.1. Processor 0 calls the MPI_Send function with the intent of sending out data to processor 1. Once this is called, processor 0 sits and waits until processor 1 calls the MPI_Recv function for the specific data that processor 0 sent.

The arguments of both the MPI_Send and MPI_Recv functions are practically identical except that one needs to know where it is sending the data to (MPI_Send) and the other needs to know where the data is coming from (MPI_Recv). Full detail of the function calls will not be discussed here as they are unnecessary in the context of theory and can be easily found in MPI documentation (www.open-mpi.org).

Although MPI_Send and MPI_Recv can be combined in numerous ways to perform larger communication schemes, it is often far easier to implement one collective communication instead of several point-to-point transfers. Doing so not only simplifies the code itself but also improves performance as these MPI functions have be optimized to minimize communication time between processors [13].

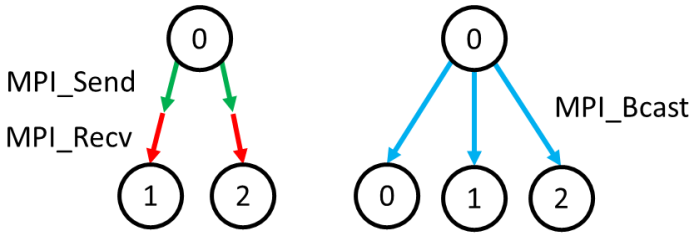


Fig. 3.2.2. A collective broadcasting maneuver (to the right) as compared to multiple point-to-point communications (on the left) for three processors. Note that here, processor 0 is not only initializing but also performs work, hence in the point-to-point fashion it does not communicate with itself. In the collective format though, it does, simply telling itself to keep a copy of the data in addition to sending it to all other processors.

Two of the most obvious forms of collective communication would be sending/receiving one piece of information to/from all processors. This is known as broadcasting and gathering. Instead of every single processor spending time initializing and defining the same variable over and over (a rather inefficient procedure), one processor can do the work and then send it to all the others, shown in Fig. 3.2.2. This idea of having a host or master processor is a fundamental principle in parallel programming. Have one processor (the master) that initializes and broadcasts the information to the working processors, who then perform all calculations, and then gather the final results back on the master processor.

Since the whole point of parallel processing is to reduce runtimes though, broadcasting may seem a bit counterintuitive. If the master initialized the original array, broadcasted it, and then had each processor calculate the new array, the same exact operation will be repeated for every processor, resulting in zero decomposition of the problem. Each processor should instead calculate the new values of only part of the array. This can still do this with a broadcast maneuver, but every processor now has the entire original array when it only needs its part of the array – again, very inefficient. Gathering the results from all the processors wouldn't be very straightforward since each processor has both updated and non-updated array values.

A far more effective process would be to send only parts of the array to all the processors, have them manipulate the parts, and then gather all the parts back. This can be done through a few loops and using send/receive commands, but luckily MPI has its own built in tools to do exactly this process: MPI_Scatter, shown in Figure 3.2.3.

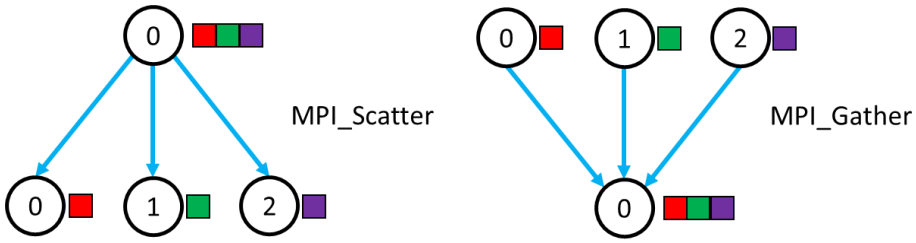


Fig. 3.2.3. An example of MPI_Scatter and MPI_Gather for three processors. The square blocks represent parts of an array that are either distributed or gathered chronologically.

Since MPI_Scatter and MPI_Gather are exact reversal operations, they are often used in tandem in parallel processing: scattering that data, performing work on the data, and then gathering the data. As a result, the MPI_Gather function combines all the elements into a collective array. The elements can also be gathered into a single value without the use of an addition loop by using other specialized functions such as MPI_Reduce, which was used in the previous section in Program 3.1.

3.3 FFTW-MPI Functionality

The accepted norm of MPI has made it not only useful for parallel computing itself but has also allowed it to evolve and be merged into other programming libraries that require specific techniques to be successfully implemented. FFTW is one of these many libraries (briefly mentioned in in Chapter 2.2), offering a simpler solution to the problem of parallelizing Fourier Transforms. This makes processes such as spectral differentiation far easier to manage in program development. Whereas finite difference method is simple to implement in a parallel scheme by just making sure each processor has the necessary points to fill its equation, spectral methods have restrictions that make it harder to grasp in certain instances.

A common and logical means of dividing up a two-dimensional grid with $nx \times ny$ number of points amongst processors is similar to a basic coordinate axis: four quadrants whose origin lies at the center of the grid space, but not necessarily at the origin point for the distribution. Although this is easy for conceptual understanding, it is far from the simplest means of implementation. This method of domain decomposition limits the performance of the program due to the basic four processor distribution and causes issues with spectral methods.

The top left quadrant of the equally divided four-quadrant system will have data ranging from $0 \rightarrow \frac{nx}{2}$ along the x-axis and from $0 \rightarrow \frac{ny}{2}$ along the y-axis. In any serial code, this is can be done through a simplistic nested for-loop, ranging from the appropriate bounds

listed above. This chunk could be stored in an entirely new array and then sent to the appropriate processor, but in doing so the memory usage of the program is increased: one array containing the full grid and at least one other array that holds the to-be-distributed grid chunk (that placeholder array would then be updated for each quadrant of the grid before being sent off to its appropriate processor). In an ideal situation, the initial grid should be directly distributed to all processors without having to allocate more memory. However, using MPI commands such as send, receive, or scatter cannot accomplish such a task by themselves as there is no way to identify the start or end of a specific row in the array to send. Calling such a function with the intent of sending a chunk with data size of $\frac{nx}{2} \times \frac{ny}{2}$ will result in sending a distribution that is actually of dimension $nx \times \frac{ny}{4}$ from the perspective of the entire grid layout.

Figure 3.6 shows the visual outcome of the master processor calling a function such as MPI_Send with the intention of sending a square quadrant of data to any desired "destination" processor. The starting x and y points of the grid are defined as (0,0) in this case, and the size of the data wanting to be sent is $\frac{nx}{2} \cdot \frac{ny}{2}$.

There are specific datatypes native to MPI that can be constructed to procedurally create quadrant layouts like the one above – such as MPI subarrays – but in the case of spectral differentiation it is a moot point. As noted previously, spectral differentiation tackles the problem with a global approach, requiring an entire range of the dataset to be used. This means that for each processor to perform a spectral method on its chunk of the data, that

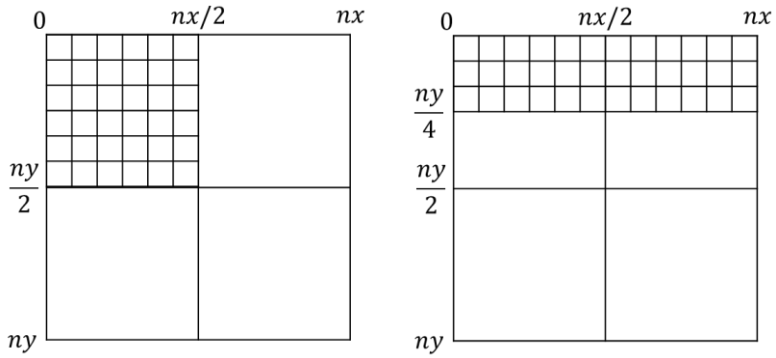


Fig. 3.3. An example of the outcome of calling a command such as MPI_Send. The left grid layout is the desired way to separate the grid, but because there is no way to define the breaking point along the x-direction, the data sent is actually a continuous row of values that loops around to the beginning until the specific size has been met, shown on the right.

chunk must be a continuous region that spans at least one entire dimension of the original whole grid. Therefore, dividing processors amongst quadrants of the grid in the above manner will not work for spectral methods since only half of the entire x range and half of the entire y range is used. Finite difference methods are still appropriate, but just require a few additional row/columns to ensure that their boundary calculations can reach the borders of the designated grid chunks.

It may be noted though that the continuity requirement of the spectral method is not violated in the second (right) distribution shown in Figure 3.3. So instead of separating the grid out into quadrants it can be separated by entire rows. Each processor will then have continuous horizontal chunks of the data and discontinuous vertical chunks.

This solves the problem for performing a simple 1D FFT of a 2D function, but what about a 2D or higher spatial dimensional FFT? In the above approach, a straightforward 2D FFT performed by each processor would result in invalid values since there is not a full sample

of all y values. The only way to perform such an operation is if the entire grid is known for the processor. For a 3D grid-space this approach works just fine. However, this is entirely nonsensical for parallelizing a 2D space as each processor would simply be performing the same calculation.

One possible solution is to have the master process perform the 2D Fourier transform first and then distribute the k-space grid amongst all processors. After each processor is finished with its manipulation, the k-space grid is then brought back to the master, who then transforms the grid back to real space. An analogous process would be used for performing a 3D FFT on a 3D space. This is an entirely valid tactic, but it is preferred in parallel computing that communication between the master and slave processes is limited to only scattering the initial grid and gathering the results. This is done to limit the overall communication time between processors, which is critical for simulations involving time evolution where multiple iterations of the same calculation will occur.

If communication back to the master is necessary to obtain the solution, then any operations performed by the master should be significantly less intensive than anything done by the processors. Doing so ensures that the workload is most effectively distributed, and that the majority of the program is parallelized (i.e. a higher p value), resulting in better speedup from Amdahl's Law. In this case, the master would be performing a 2D Fourier transform over the entire grid whereas the slaves would be performing a simple multiplication on only parts of the grid – the exact opposite of effective workload distribution.

The other option to solving these problems is using FFTW's built in MPI functionality. Here, the inner workings are kept hidden from the user and put into a nicely wrapped package of a function that not only performs the Fourier Transform but also distributes the results to all desired processors. In the end, the transformed (k-space) data is distributed amongst all the processors [14]. Which comes first: the FFT or the distribution? According to FFTW documentation, a multi-dimensional parallel FFT is performed in the following manner:

> In the MPI version of FFTW, we assume that a multi-dimensional array is distributed across the rows (the first dimension) of the data. To perform the FFT of this data, each processor first transforms all the dimensions of the data that are completely local to it (e.g. the rows). Then, the processors have to perform a transpose of the data in order to get the remaining dimension (the columns) local to a processor. This dimension is then Fourier transformed, and the data is (optionally) transposed back to its original order [15].

As explained further in the documentation, the problem with such a method becomes the transposing of the data. However, the language is still rather ambiguous as to which data is being transposed: the k-space data or the original. Chapter 3.4 makes an attempt to mimic this above process and gives another solution to the 2D FFT problem by considering the Laplacian.

3.4 The Laplacian – An Example MPI FFTW Program

One of the most complicated terms in the dimensionless RT-KH equations is the vorticity, seeing as it is found from the Laplacian of the electric potential. The repeated use of it in Equation 1.3.8 makes it a useful starting point in parallelization and serves as an excellent example in the introduction to parallel computing, especially for spectral methods of differentiation. In the case of calculating the 2D Laplacian (Equation 3.4.1), several difficulties arise that must be dealt with in unique ways.

$$\Delta = \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{3.4.1}$$

The easiest implementation would be to perform a straightforward 2D FFT of the 2D space and then manipulate the k-space elements, as described by Johnson [8]. As previously discussed in Section 3.3, the problem then becomes parallelizing a 2D Fourier Transform on a 2D dataset. In the spirit of avoiding the mystical hidden processes of FFTW's built in MPI functions, the only other means of implementation is through 1D differentiation for each processor. Luckily the Laplacian is an operation that can be separated into a series of single dimensional problems. Each 1D problem can then be distributed amongst the processors, combined, and then sent back to the master for the solution.

The process of row distribution described in Section 3.3 already solves half of our 2D problem: the $\frac{\partial^2}{\partial x^2}$ contribution. Each processor can perform a 1D Fourier transform on each of its rows (i.e. each row in its chunk), multiply by a constant, and then transform back.

The y-component becomes slightly trickier. In order to ensure each processor receives a continuous column of data from using MPI functions, the initial grid must be transposed so that the columns are now rows. The process for the x derivative then becomes identical for the y derivative.

After manipulations, each processor then has the $\frac{\partial^2}{\partial x^2}$ and $\frac{\partial^2}{\partial y^2}$ terms for two chunks of the grid. However, it only has the Laplacian for a square of size $\frac{nx}{\#\ Procs} \times \frac{ny}{\#\ Procs}$. The reason for this is because of the transpose for the y-component. The chunks used for the x and y derivatives do not line up, as they fundamentally can't due to the continuity necessity. If a processor were to have the Laplacian for its whole row-based chunk, then it would also have to have every column of the grid. This would mean that each processor would be calculating the $\frac{\partial^2}{\partial y^2}$ term for the entire grid even though it really only needs it for the rows it was given.

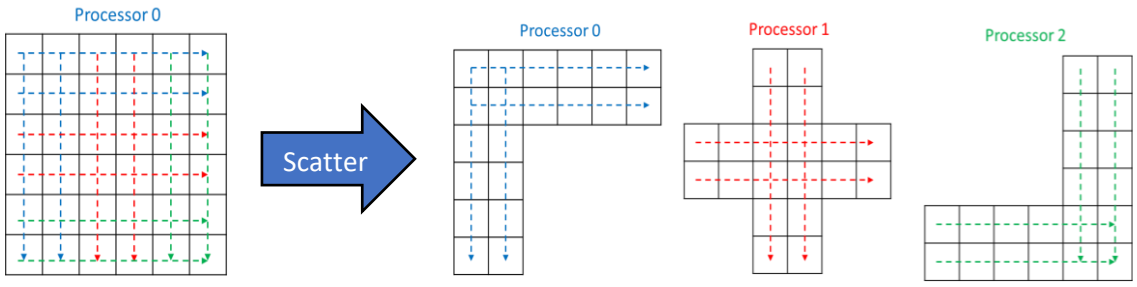


Fig. 3.4.1. Master processor 0 initializes and distributes (scatters) the original grid to all processors (in this case, the master is also performing calculations). Columns are scattered after transposing the original grid. Each processor then calculates $\frac{\partial^2}{\partial x^2}$ and $\frac{\partial^2}{\partial y^2}$ for its chunk. However, they only have the 2D Laplacian for a small portion of their entire chunk: where the rows and columns overlap.

By this point, all calculations are finished and everything needed for the Laplacian is obtained, but just not collected correctly. As shown in Figure 3.4.1, each processor has pieces that are required for the other processors to know the full Laplacian of their horizontally row-distributed chunk. It should be noted that in this approach, the complete Laplacian (i.e. both the $\frac{\partial^2}{\partial x^2}$ and $\frac{\partial^2}{\partial y^2}$ terms) for each continuous row is the goal for each processor. The columns are the information needed to be communicated. This can be reversed where processors work to achieve complete columns and communicate row information.

This introduces the third problem of collecting the distributed data. There are two immediate solutions to this: inter-processor communication and master-gathering. Either each processor communicates with every other processor to obtain all of its missing pieces, or each processor communicates back to the master, which then single handedly combines all the elements.

As stated previously, in parallel processing it is usually best that the master process needs only to initially send and finally gather the components to and from all processors without having to perform any calculations that are in excess of the working processors. Therefore, inter-processor communication seems the way to go. In order for this to be achieved for the Laplacian, each processor must communicate with all others to scatter and gather the missing blocks necessary for said processor to have the Laplacian of its horizontal chunk. This is shown visually in Figure 3.4.2.

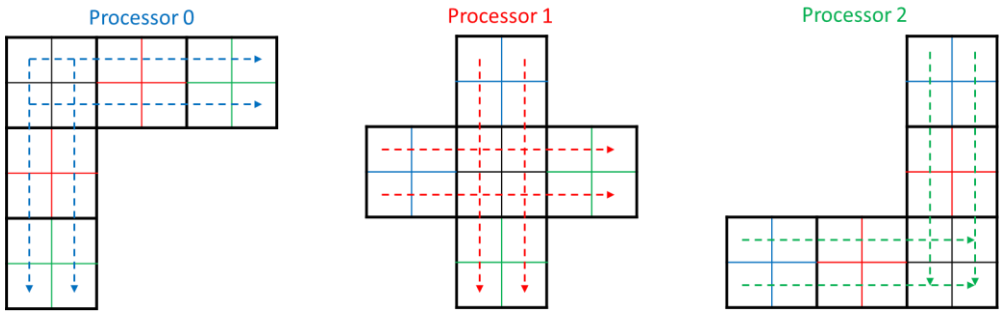


Fig. 3.4.2. Each processor must communicate with all other processors in order to obtain the Laplacian for its row-chunk. This color-coded image shows which blocks must be either sent or gathered to specific processors. In this case, row-oriented blocks are received and column-oriented blocks are sent (the dashed lines represent orientation). So the top middle red block of Processor 0 will be received from Processor 1, hence why Processor 1's top middle block

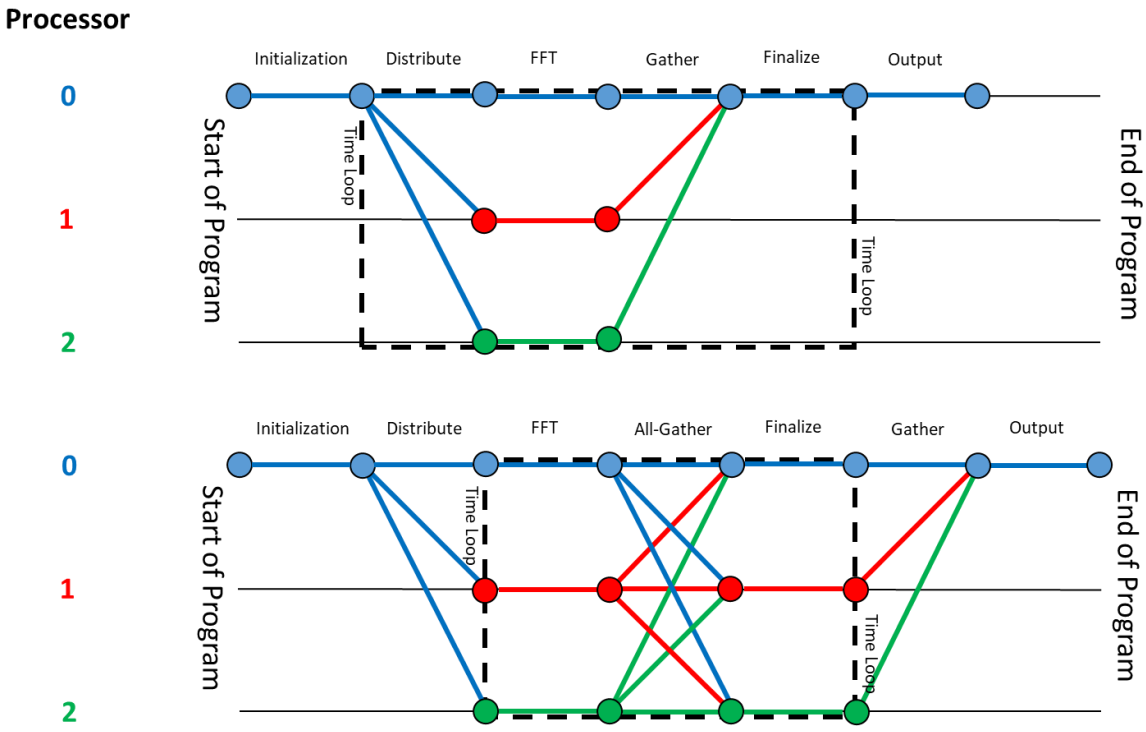


Fig. 3.4.3. A master-gathering method (top) vs. inter-processor method (bottom) of communication. The color circular nodes represent key points in the program. Colored lines for a specific processor represent when that processor is doing work (slave processors only perform work once they have data received from the master). Colored lines connecting between processors represent communications from that processor. Time simulations are a common application, hence the dashed block representing the area where a time-loop would occur.

This method, although does limit communication to the master, also inherently has a tremendous amount of communication between processors. This communication time must be taken into account especially for programs that involve time simulations where looping over this communication phase can result in unexpectedly large runtimes.

Due to the universal sending and receiving from all n processors, the inter-processor method (the bottom visualization in Figure 3.4.3) results in n^2 communications per time iteration. With the additional master requirements at the start and end, any program implementing this method would have a total of $2n + n^2t$ communications for n number of processors and t number of time-loop iterations. Keep in mind that a self-communication is still considered a communication (the distribution phase, although starting on processor 0, results in three communications between 0, 1, and 2). The master-gather approach (the top visualization in Figure 3.4.3) is substantially less intensive, resulting in a mere $2n$ communications per time iteration, and a grand total of $2nt$ communications for the entire program. Therefore, in respect to communication times, the master-gather approach is far superior.

Based off this same argument, one could say that a simple serial program is thereby the fastest as it has no communication between different processors at all. Of course this isn't the case since one processor is performing all the work, but it brings up a good point in: Is it truly better to minimize communication when the result is an increased workload on a single processor? In most cases no, since the time spent sending data is far less than the time computing. However, in this case a simple addition of components is being performed

while drastically reducing our communication time from $O(n^2)$ to $O(n)$. So for the purpose of simplicity and minimizing communication, the master-gather implementation will be focused on.

To begin analysis on the program's performance, the speedup will be analyzed first. As already discussed in Chapter 3.1, since the machine used for testing was not under precise controls, the a coefficient for Equation 3.1.4 must first be found, which can be done from the serial and parallel $n = 1$ runtimes of the program:

Table 3.4.1. Sample discrepancies in runtime values (in seconds) for the master-gather MPI approach to the Laplacian solution when running on one processor.

Serial Runtime	**Parallel Runtime ($n = 1$)**	**Speedup**
10.944555	14.467633	0.756485529
10.921053	13.267305	0.823155343
9.155819	12.307454	0.743924698
Averages		
10.34047567	13.347464	0.774521857

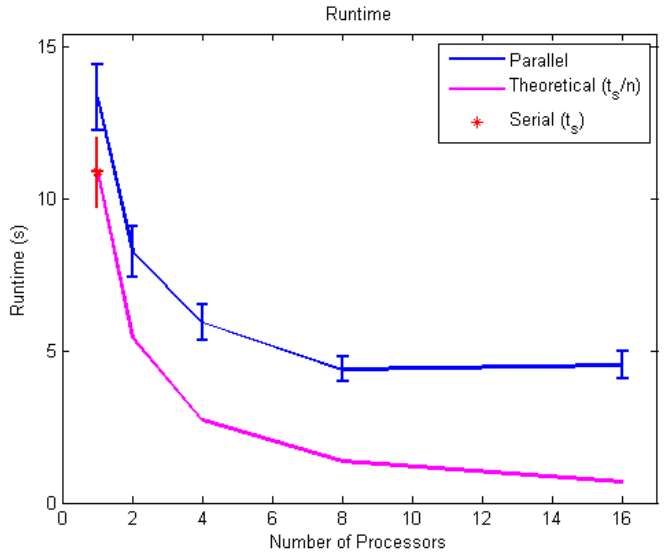
Table 3.4.1 gives the results of the three trial executions for $n = 1$ processors. Using 3.1.5, the normalization factor is determined to be $a = 0.7088$, with a maximum and minimum value of 0.6558 and 0.7852. Using Matlab's curve fitting tool for 3.1.4, the best fit is achieved for when $a = 0.7052$, which is only a 0.5092% difference from the expected value.

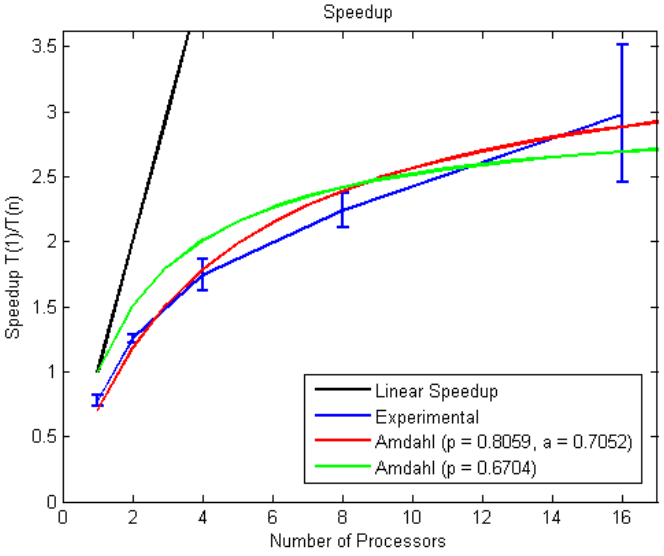
Table 3.4.2. Fit data for the speedup of the Laplacian master-gather program.

p	a	R^2
0.6704 ± 0.08385	1	0.8977
0.8043 ± 0.02145	0.7088	0.9852
0.8059 ± 0.0845	0.7052 ± 0.1855	0.9853

The master-gather approach's parallelization of 80.59% is fairly reputable, considering how the process still requires communication to the master for each time iteration. This means that, as designed, the FFT manipulations performed on the distributed data is far more expensive in execution time than the final combination done on the master process. As can be seen in Figure 3.4.4, the runtime of the parallel program is always higher than the theoretical but follows a similar trend. The reason for this is because of the $n = 1$ runtime offsets, previously explained in Chapter 3.1. Here, the offset is more substantial due to the increased complexity of the program.

However, despite the high parallelization of the program, the 3.1.4 speedup trendline (shown in Figure 3.4.4) using the best fit values for a and p reaches a horizontal asymptote of around only 3.63 when taken out to $n \sim 10^4$. This can be attributed to the efficiency of the program, which in fact never achieves a perfect 1.0 and experiences an exponential decrease. Therefore, in theory there could be far better methods of parallelizing the solution to the Laplacian.


Runtime
Parallel
Theoretical (t_s/n)
Serial (t_s)
Runtime (s)
Number of Processors


Speedup
Linear Speedup
Experimental
Amdahl (p = 0.8059, a = 0.7052)
Amdahl (p = 0.6704)
Speedup T(1)/T(n)
Number of Processors

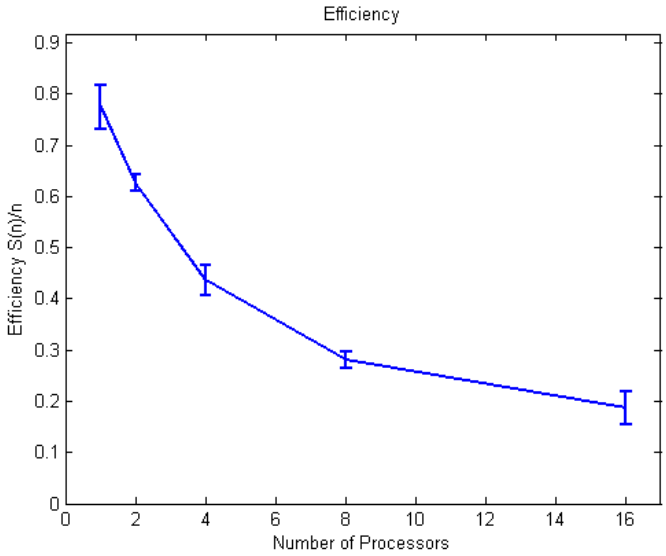

Efficiency
Efficiency S(n)/n
Number of Processors

Figure 3.4.4. Runtime, speedup, and efficiency of the master-gather MPI solution to solving the Laplacian using MPI on a 2.1 GHz, 24 CPU (parallel threads) Linux system. Error bars show the standard deviation of three different trials for each n number of processors (1, 2, 4, 8, and 16). The program was run in serial every time to ensure that parallel runs were compared with their own serial runs.

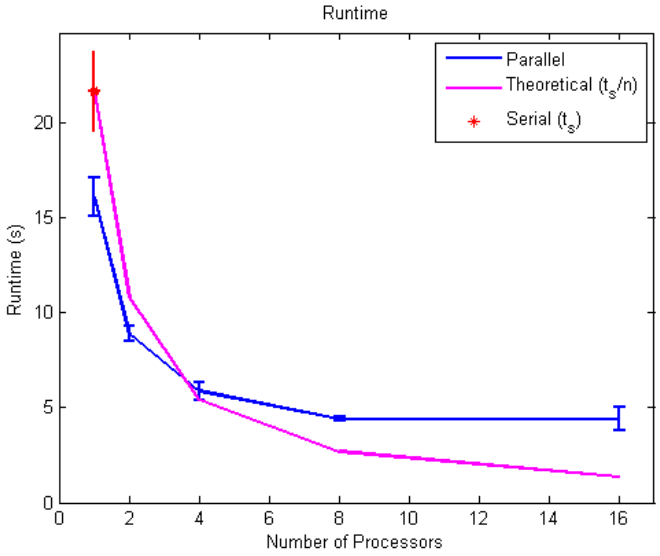


Figure 3.4.5. Runtime of the master-gather solution to solving the Laplacian as compared to a strictly serial 2D FFT with no parallelization. The pink "theoretical" line shows the best possible runtime improvement for the serial code based off the serial runtime. Error bars show the standard deviation of three different trials for each n number of processors (1, 2, 4, 8, and 16). The program was run in serial every time to ensure that parallel runs were compared with their own serial runs.

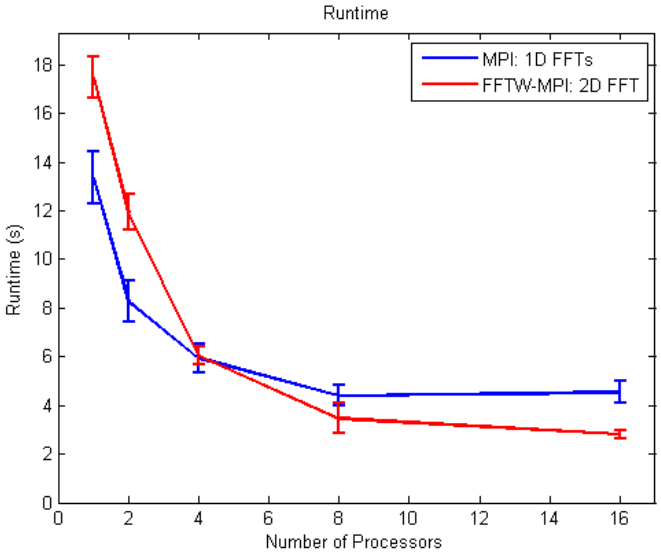
If parallelization is simply ignored and the serial 2D FFT method described by Johnson is used, that the master-gather parallel method is far superior in runtime even for only a single processor, shown in Figure 3.4.5. This implies that the forward and backward process of taking a 2D FFT is more costly than repeated use of 1D FFTs along each dimension on the same dataset. Further examination of this phenomena could be performed to confirm or deny this claim, in addition to extending the case to a 3-dimensional space.

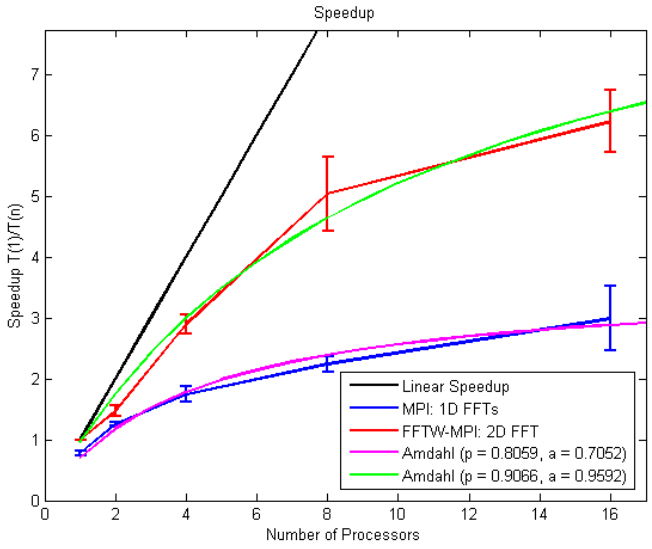
The theoretical runtime of the serial 2D FFT solution also brings up an interesting point on the parallelization of the current solution. For relatively low number of processors, the master-gather method continues to surpass the expected best-case scenario for the serial 2D FFT's parallelization runtime. However, as the number of processors surpasses four, the theoretical runtimes become faster, meaning that there could be a more effective routine for parallelizing a 2D FFT other than the master-gather approach.

A serial 2D approach to the Laplacian problem is obviously not the most efficient means of execution, and because of the problem's complexity, surely there are other methods to solve it. So how about using a 2D parallelized method, such as FFTW's built in parallel 2D FFT? Here, instead of having to break the problem into x-components and y-components, the entire 2D grid can be considered and equally divided amongst the number of processors available. This is possible thanks to the hidden inner-workings of FFTW's MPI protocols, which allows the user to treat the program as being classically parallelized: each processor has a chunk of the grid of size $nx \times \frac{ny}{\#\ Procs}$ starting at a local y_0 point. The performance analysis of this method can be seen in Figure 3.4.6.

Beforehand, it should be noted that when analyzing the speedup and efficiency of the 2D FFT method, the parallel $n > 1$ executions were compared simply with its $n = 1$ execution. This is not the same approach used for the 1D FFT (master-gather) solution because the actual means of how the parallelized 2D FFT performs its transformations is unknown. It is therefore also unknown if it is proper to compare its parallel runtimes to

the non-parallelized 2D FFT or some other serial method. As a result, both the speedup and efficiency have an ideal value of 1.0 for $n = 1$ processors. This creates a non-smooth relation when examining the speedup and efficiency for comparing $n = 1$ to $n = 3$ processors.





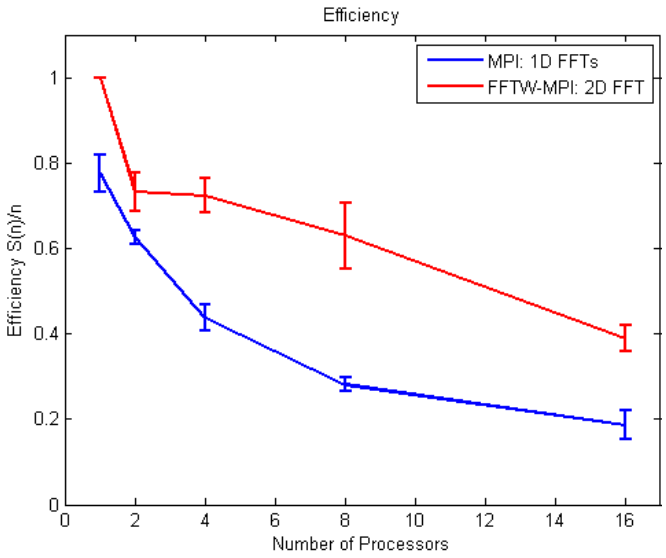


Figure 3.4.6. Runtime, speedup, and efficiency of the master-gather solution to solving the Laplacian using 1D FFTs MPI compared to using FFTW's built in MPI functionality to implement a 2D FFT on a 2.1 GHz, 24 CPU (parallel threads) system. Error bars show the standard deviation of three different trials for each n number of processors (1, 2, 4, 8, and 16). The 2D FFT was compared to its own serial runtimes, resulting in the ideal speedup and efficiency values of 1.0 for $n = 1$.

Due to this approach, the idealized 1.0 speedup and efficiency for $n = 1$ should result with $a = 1$ when fitting 3.1.4 to the data. However, because the runtimes will inevitably still experience noise, it is expected for this value to vary by small degrees. Table 3.4.3 showcases how letting a vary to achieve the best fit (which occurs for $a = 0.9592$) changes the final parallelization percentage (p) and R^2 by very little, which is expected since the initial $n = 1$ value is set to be the idealized standard.

Table 3.4.3. Fit data for the speedup of the Laplacian using FFTW's built in MPI functionality.

p	a	R^2
0.8989 ± 0.0176	1	0.9854
0.9066 ± 0.0668	0.9592 ± 0.33275	0.9862

Using the best fit values for a and p, the speedup maximizes at round 10.26 for $n \sim 10^5$. This is substantially higher than the 1D FFT approach (almost by a factor of three), implying that FFTW's MPI functions are a far better solution. The efficiency also yields a similar conclusion. However, when looking strictly at runtimes, the pattern shifts. For a low number of processors ($n < 4$), the 1D master-gather approach has an execution time remarkably lower than its 2D FFTW-MPI counterpart. The average runtimes are practically identical when running on four processors: 5.878 and 6.042 seconds for the 1D and 2D approaches, respectively. It can also be seen between Figure 3.4.5 and 3.4.6 how the runtimes of FFTW's 2D FFT matches the theoretical best runtime of the strictly serial 2D FFT slightly better than the master-gather approach.

This key difference showcases that FFTW's MPI solution is better designed to achieve maximum speedup than the master-gather approach presented here. This way systems running on numerous processors (i.e. $n > 4$) obtain the fastest runtimes possible. However, if the intent of the user is to use only a few processors – simply because of hardware restrictions or to minimize communication time – then the master-gather approach will actually run in a shorter timeframe.

Since there are certain attributes of both the solutions presented here that make each superior, it could be inferred that an even better solution to a problem such as the 2D FFT (and thereby the Laplacian) may exist: one that maximizes speedup and efficiency while simultaneously achieving the lowest runtimes possible.

Chapter 4

Plasma RT-KH Program

With the understanding of the fundamental mechanics of spectral methods and parallel computing, discussion on the actual program for simulating Rayleigh-Taylor and Kelvin-Helmholtz instabilities in ionospheric plasmas can begin.

Although the methods used here for calculations and achieving speedup are new techniques, the basic structure of the simulation is something that has already been done before. David V. Rose, I. Paraschiv, and T. C. Genoni from Voss Scientific developed a serial approach to evolving Equations 1.3.8 and 1.3.9 in time on a periodic domain using finite difference methods for differentiation. Their program is therefore incapable of running on multiple processors, meaning its performance is limited strictly by the fastest processor of the machine. The basic structure of the serial code was used as a guideline in the steps for time evolution and will also be used for comparison as to simulation accuracy and runtime of the parallelized solution.

4.1 Outline & Methods

When simulating Equations 1.3.8 and 1.3.9, a generic step method was used to make small discrete steps forward in time. To further increase the accuracy of time evolution, the iterations were separated into two parts. Generally known as a predictor-corrector method, a half-time step is first taken to aide in the "prediction" of the full-time step. The updated values from the predictor step are then used to calculate the final "corrected" values for the full-time step. The time step for the predictor part must always be half that of the full-time step, but the full time-step can be adjusted to account for rapidly or slowly changing functions. Although the program developed at Voss Scientific is capable of dynamic time-stepping, the program made here was kept simpler and uses a constant, linear time evolution.

Since the time stepping in of itself is rather trivial, focus will instead be shifted on the outline of the computations involved within each time step. Due to the nature of spectral differentiation with MPI as discussed previously, the program can be thought of in a series of steps determined by the inter-processor communication, which varies depending on what terms of 1.3.8 and 1.3.9 are calculated.

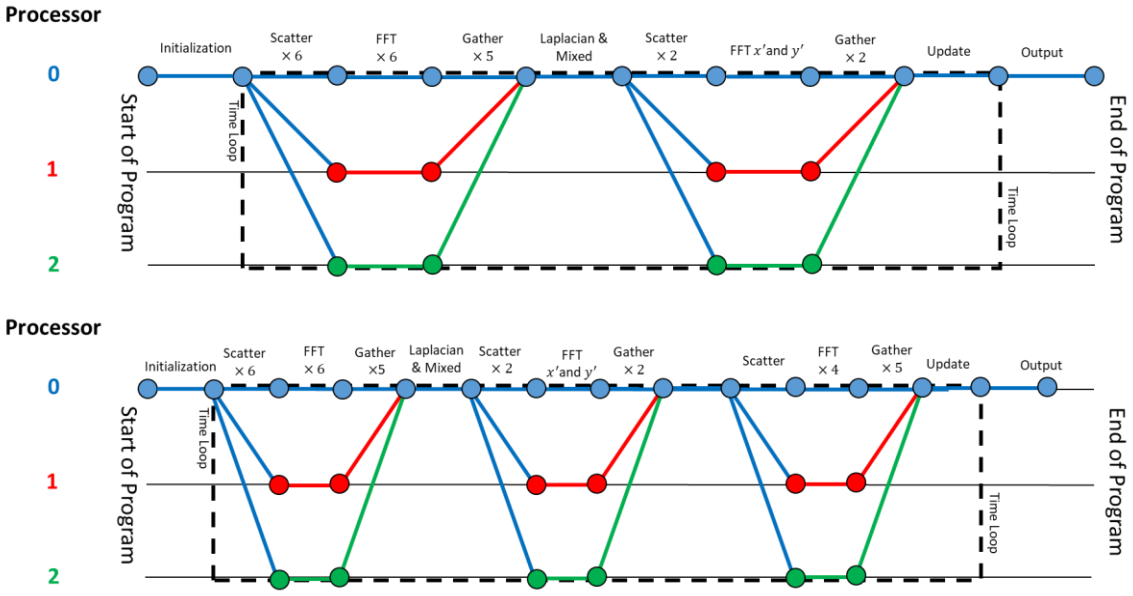


Figure 4.1. Communication maps for the RT-KH plasma simulation program. The top image showcases the communications necessary when dealing with only the linear terms of Equation 1.2.1 and 1.2.2. The bottom image shows the communications need when non-linear terms are also calculated.

Due to the $\frac{\partial^2 \rho}{\partial x \partial y}$ and $\Delta\rho$ terms in 1.3.8, even the nonlinear procedure requires a dual stage scatter/gather approach, shown in Figure 4.1. The mixed derivative requires the y derivative to be calculated first and then the x derivative taken subsequently after. The Laplacian ρ term follows the same method outline in Chapter 3.4. As a result, any simulation ignoring the nonlinear terms (specifically on the right-hand-side of 1.3.8 and 1.3.9) will have a total of $15nt$ calculations per time step, t, for n number of processors (i.e. $15nt$ for each half-time step in the predictor-corrector method). If the nonlinear terms are wanted to be calculated, then the communications increase to $21nt$.

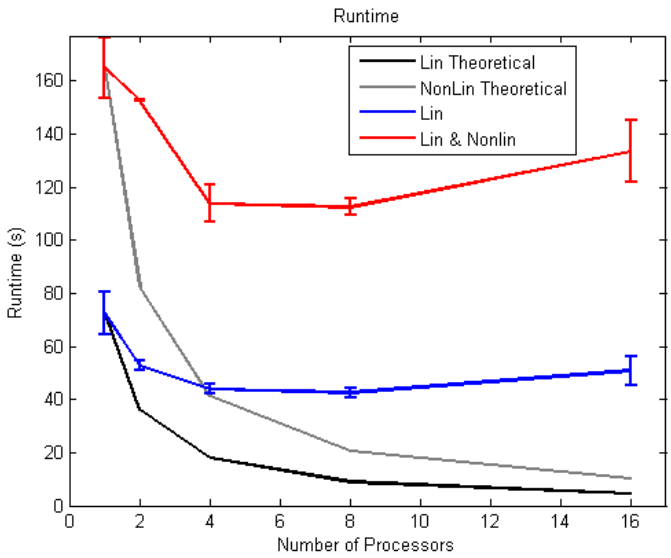
If one pays close attention it can be seen during the nonlinear calculations (the third "_/" formation on the bottom of Figure 4.1) that even though only four values are being

calculated (resulting in four FFTs), five values are being gathered. The fifth is the $\frac{\partial \rho}{\partial x}$ term, calculated in the first processor manipulation, and is an artifact of the functions used to calculate the derivatives. This could easily be edited to reduce the number of FFTs performed by the linear method (which still calculates this $\frac{\partial \rho}{\partial x}$ but does nothing with it, resulting in 6 FFTs and only five gathers).

4.2 Performance

Since the RT-KH MPI implementation varies depending on whether nonlinear terms are included or not, it is important to consider both possibilities when analyzing performance. Another key attribute that will greatly affect the speedup and runtime of the program that is not explicitly outlined is writing the data to a file. The optimization of this process was not attempted, and so any file saving done by the program is an inherently serial process that will greatly hinder the performance of the program. Therefore, to analyze strictly the methods for calculations, it will be assumed that no output files are created to obtain the best representation of the program's numerical performance.

Figure 4.2.1 showcases the runtime, speedup, and efficiency of the MPI RT-KH program when excluding and including nonlinear terms. Similar to FFTW's MPI speedup tests, the parallel $n > 1$ executions were compared simply with the $n = 1$ execution, again resulting in an ideal speedup of 1.0 for $n = 1$.



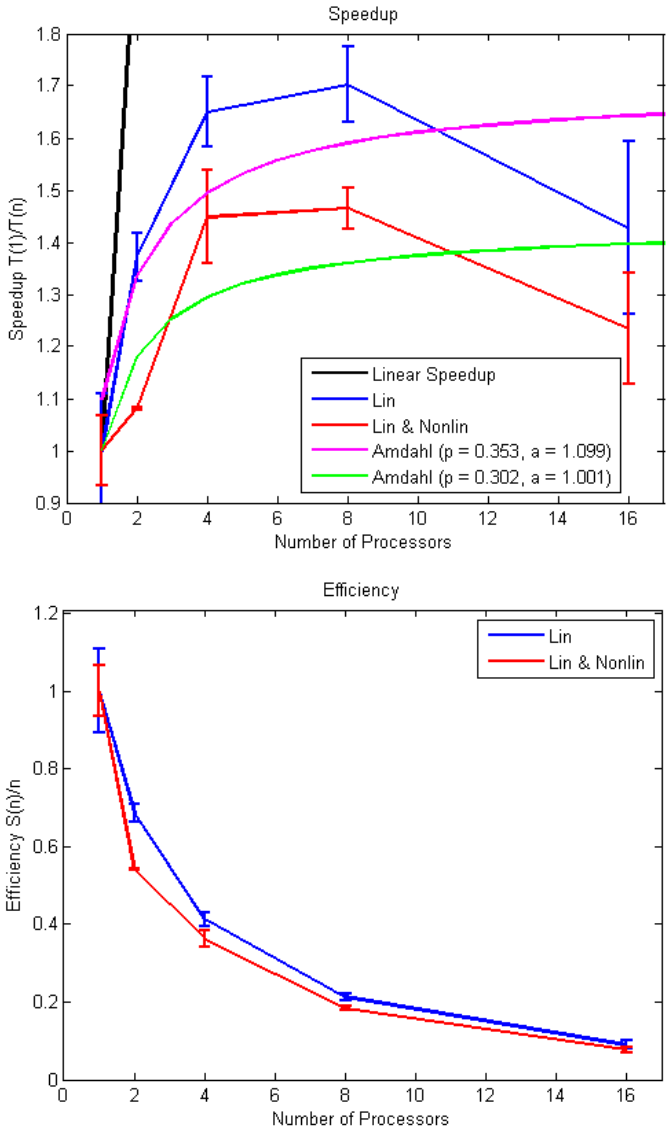


Figure 4.2.1. Runtime, speedup, and efficiency of the MPI RT-KH program for 5000 time steps. Error bars show the standard deviation of three different trials for each n number of processors (1, 2, 4, 8, and 16).

When dealing with just linear terms, a maximum speedup of 1.70 is achieved for $n = 8$ processors. For $n = 16$, the speedup drops to 1.43. The reason for this decrease is the balance between load distributions. Although fewer FFTs are being calculated per processor, there is more time spent communicating and distributing the data. Due to this change in an already small speedup, the best fit possible when including $n = 16$ gives a

maximum theoretical speedup of 1.69 for $n \sim 10^3$ processors (hence the relatively low R^2, shown in Table 4.2.1). Although this maximum does agree with the maximum speedup achieved experimentally, it occurs at $n = 8$. So despite the relative poorness of the fit, the maximum does match what is actually achieved. Excluding the $n = 16$ term does give far better fits for Amdahl's Law (up to $R^2 = 0.9771$), but also portrays the data in a biased fashion.

Table 4.2.1. Fit data for the speedup the MPI RT-KH program when calculating only the linear terms of 1.2.1 and 1.2.2.

p	a	R^2
0.4257 ± 0.1194	1	0.6442
0.353 ± 0.38903	1.099 ± 0.4643	0.6977

When looking at the performance of the program when calculating the nonlinear terms, the drastic increase in runtime is immediately noticable – over double that when they are excluded. The efficiencies are almost identical though, showcasing that workload distribution remains relatively consistent despite calculating new terms. For the speedup, a maximum occurs for $n = 8$ once again, but with a slightly decreased value of 1.46. When fitted with Amdahl's Law, the appropriate $a = 1$ is once again achieved (shown in Table 4.2.2) just like with the linear terms. Despite the goodness of fit, the theoretical maximum speedup of 1.43 occurs for $n \sim 10^3$, which is again a very similar result to the linear term test. The parallelization percentages are reasonably close to one another as well, with a percent error of 15.57%.

Table 4.2.2. Fit data for the speedup the MPI RT-KH program when including the nonlinear terms of 1.2.1 and 1.2.2.

p	a	R^2
0.3028 ± 0.1301	1	0.6016
0.302 ± 0.40765	1.001 ± 0.4114	0.6016

When compared to strictly the runtimes of the RT-KH program at Voss Scientific, the results vary depending on whether or not the nonlinear terms are calculated, as shown in Figure 4.2.2. Although more in depth analysis could be made, a general assumption would be that the finite difference method is largely faster than spectral methods. Neither of the MPI runs come close to matching the theoretical parallel runtimes of the Voss program, but they do perform better when computing the nonlinear terms.

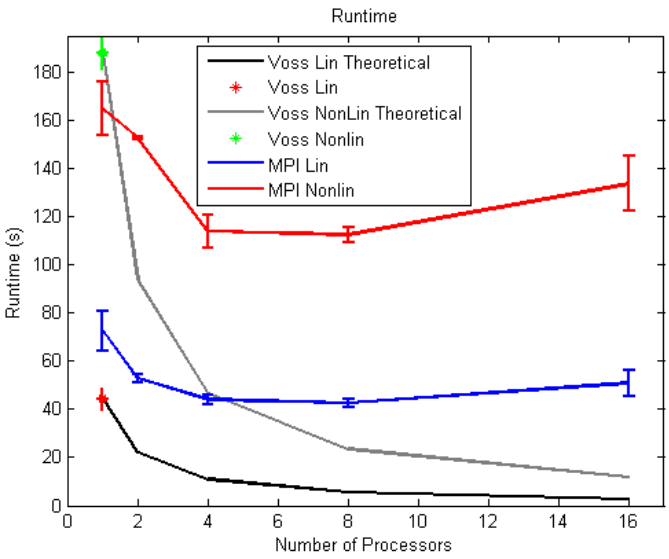


Figure 4.2.2. Runtimes of the MPI RT-KH program compared with the program developed at Voss Scientific for 5000 time steps.

4.3 Simulation Accuracy & Brief Analysis

It is important to first examine and validate the proper effects of the linear and nonlinear terms of 1.3.8 and 1.3.9 under the influence of a default shear velocity profile, shown in Figure 4.3.1. Strictly periodic functions were used to ensure that the FFTs could differentiate appropriately. The initial density and potential were made identical and the simulation was run up to 10000 time steps. Figures 4.3.2 and 4.3.3 show the evolution of such a system created by the MPI RT-KH program. The program by Voss Scientific produces identical results, showcasing that the methods used here are accurate for the linear terms in 1.3.8 and 1.3.9.

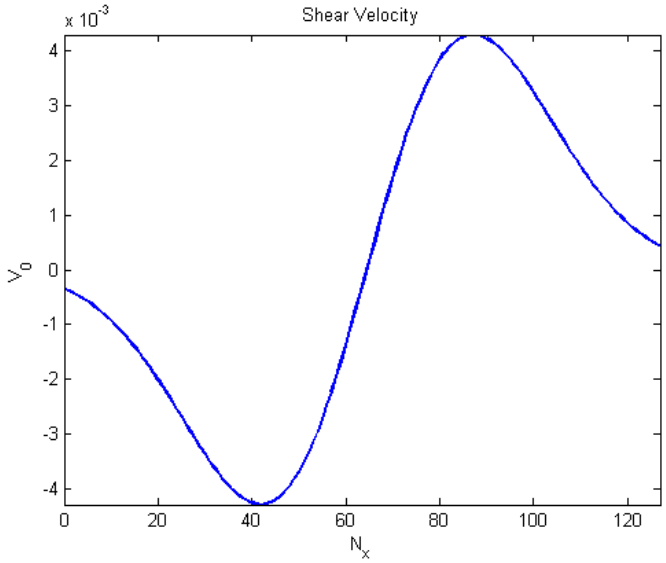


Figure 4.3.1. Default shear velocity profile.

Including the nonlinear terms in calculations also produces results that match those created by Voss Scientific. As expected, the values do not vary much from the linear term simulation, as shown in Figure 4.3.4. Similar to adding the second order terms of a Taylor series, the additional terms here should not have any large effects on the distributions.

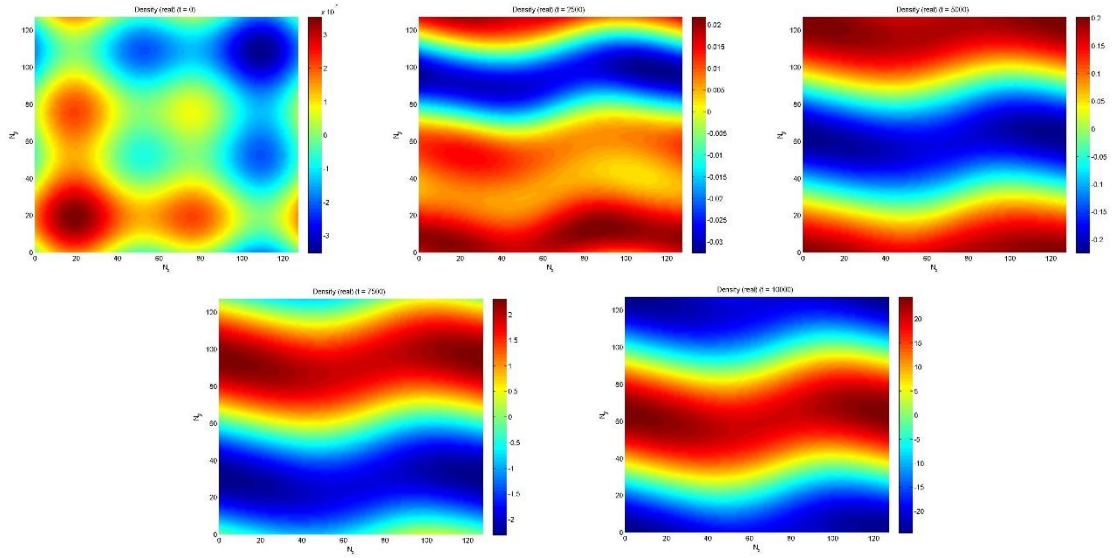


Figure 4.3.2. Plasma charge density evolution using with the default velocity profile for only linear terms. Snapshots taken for time steps at 0, 2500, 5000, 7500, and 10000.

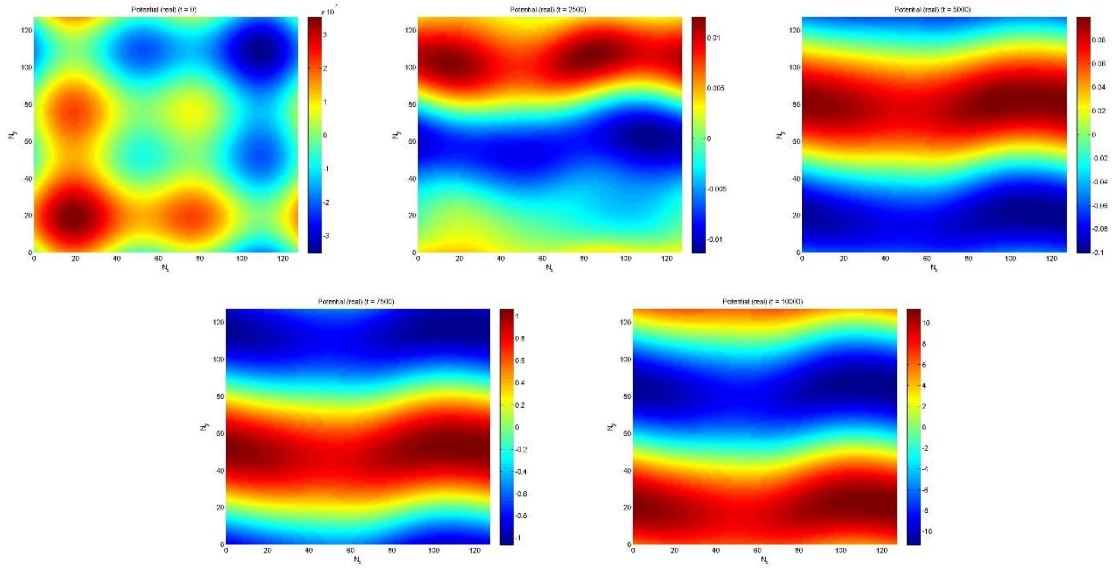


Figure 4.3.3. Electric potential evolution with the default velocity profile for only linear terms. Snapshots taken for time steps at 0, 2500, 5000, 7500, and 10000.

As noticeable in all the images, the plasma density and potentials rapidly stray from their initial distributions to form horizontal band structures. The channels of negative (blue) and positive (red) polarity are generally called “streamers” and can be viewed as negative electron charge distributions moving against a positive ion background. An initial

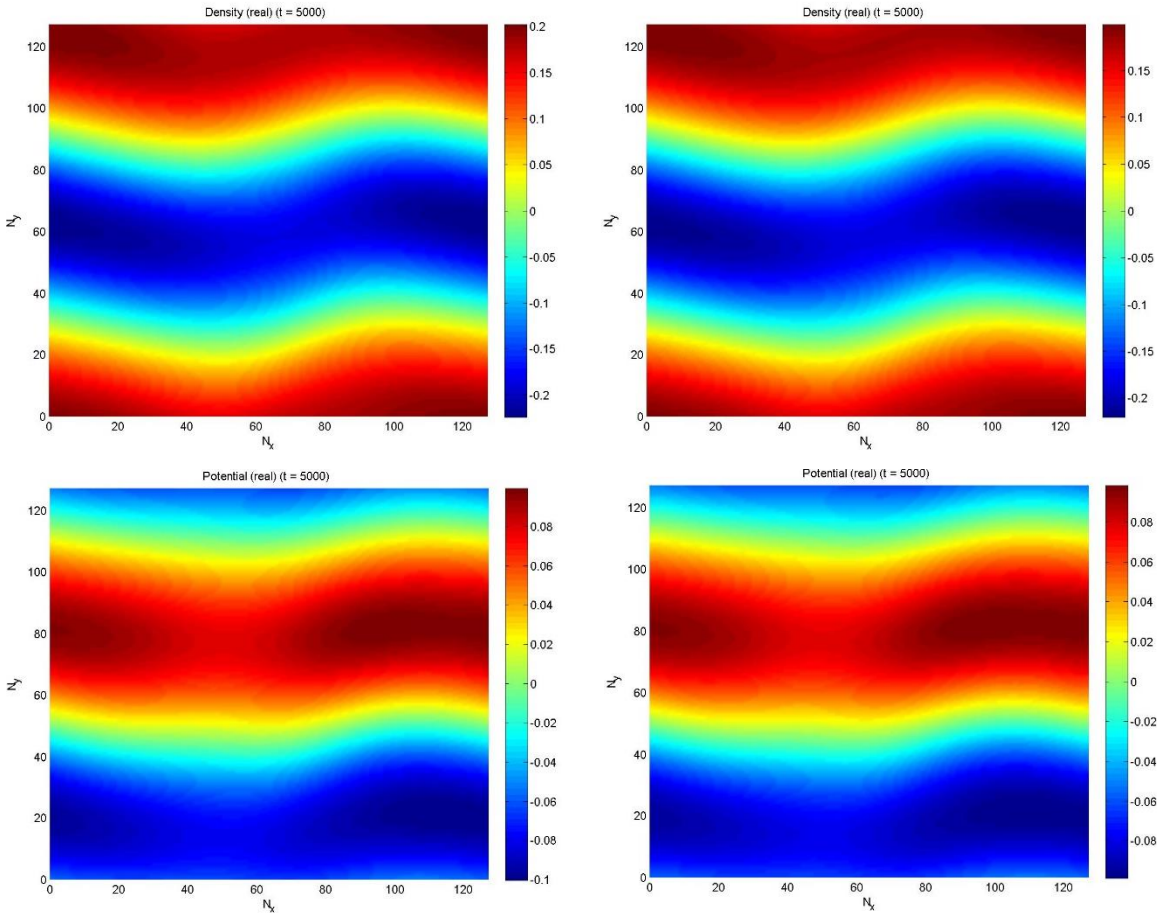


Figure 4.3.4. Sample values of plasma charge density (top) and potential (bottom) after 5000 time steps. The left images include only linear terms from 1.3.8 and 1.3.9, where as the right images include both linear and nonlinear terms. Slight variances occur near the midpoint of the x-axis, making the bands more uniform when including nonlinear

perturbation of free electrons (i.e. negative charge density) is necessary to achieve the streamer-like distributions. The evolution of streamers is dependent on an ambient electric field (provided by the potential gradient) to accelerate the free electrons within the plasma. The positive channels propagate along the electric field so long as an external source of electrons is present (provided by background ionization or due to incident photons), and the negative streamers propagate against the electric field due to the acceleration of electrons [16].

The streamers shown here are classified as the linear phase of the instability's evolution. As time progresses, the streamers dissipate as the transition from linear to nonlinear stages take place as zonal flows and vortex structures appear. Research done by Sotnikov at the Air Force Research Laboratory showcases the nonlinear features and will not be discussed in depth here due to anomalies discussed in Chapter 4.4 [2].

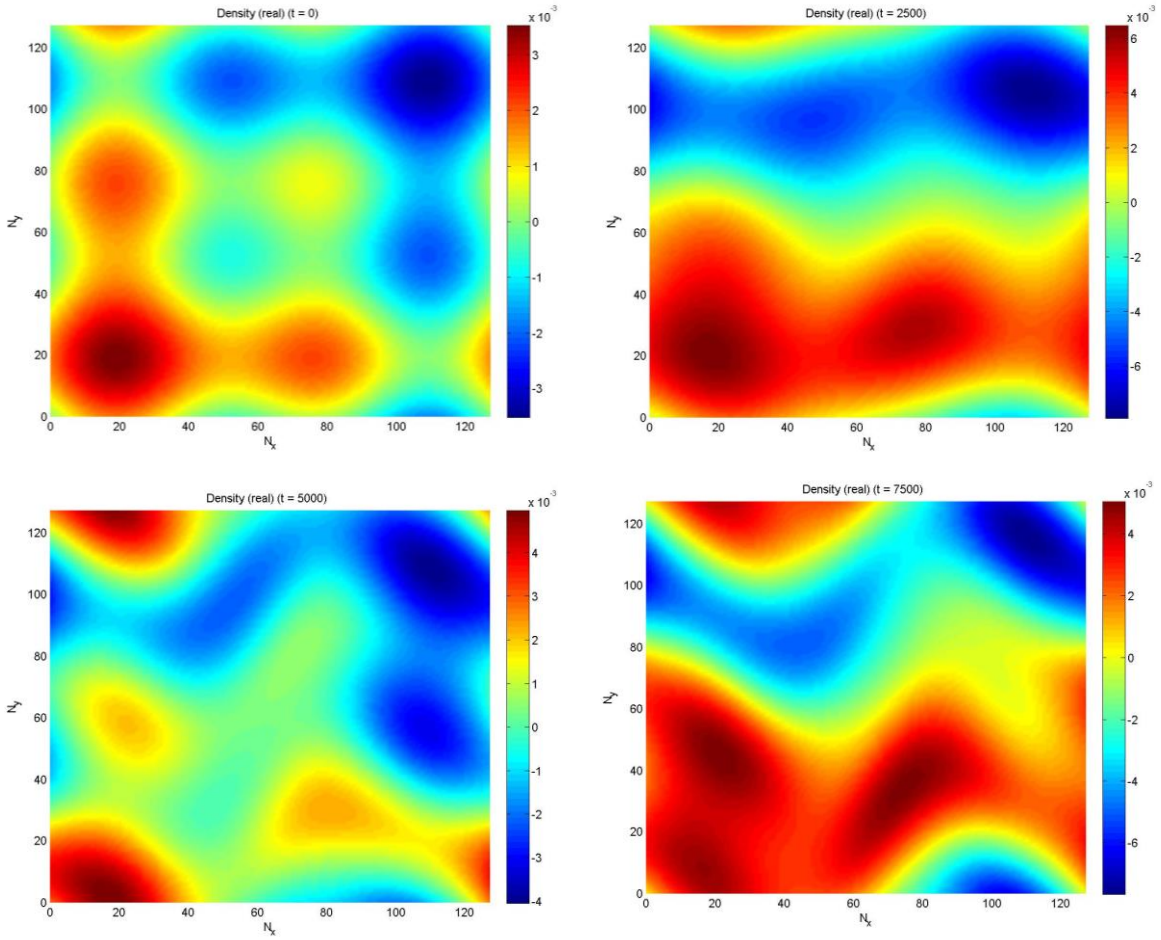


Figure 4.3.5. Plasma charge density evolution using with the default velocity profile for only linear terms without the presence of ion drift, $v_g = 0$. Snapshots taken for time steps at 0, 2500, 5000, and 7500. Notice the final distributions begin to form zonal flows dependent on the shear velocity profile, typical of the nonlinear instability phase.

One way to make the nonlinear stages occur sooner in the evolution is to decrease the ion cyclotron frequency (ω_{ci} in Equation 1.3.6) or correspondingly the gravitational drift speed of the ions (v_g in Equation 1.3.8). This technique is not useful for exact results as it will violate the $\omega/\omega_{ci} \ll 1$ condition for Equations 1.3.8 and 1.3.9. It does, however, give a basic form as to the nonlinear phase, shown in Figures 4.3.5 and 4.3.6.

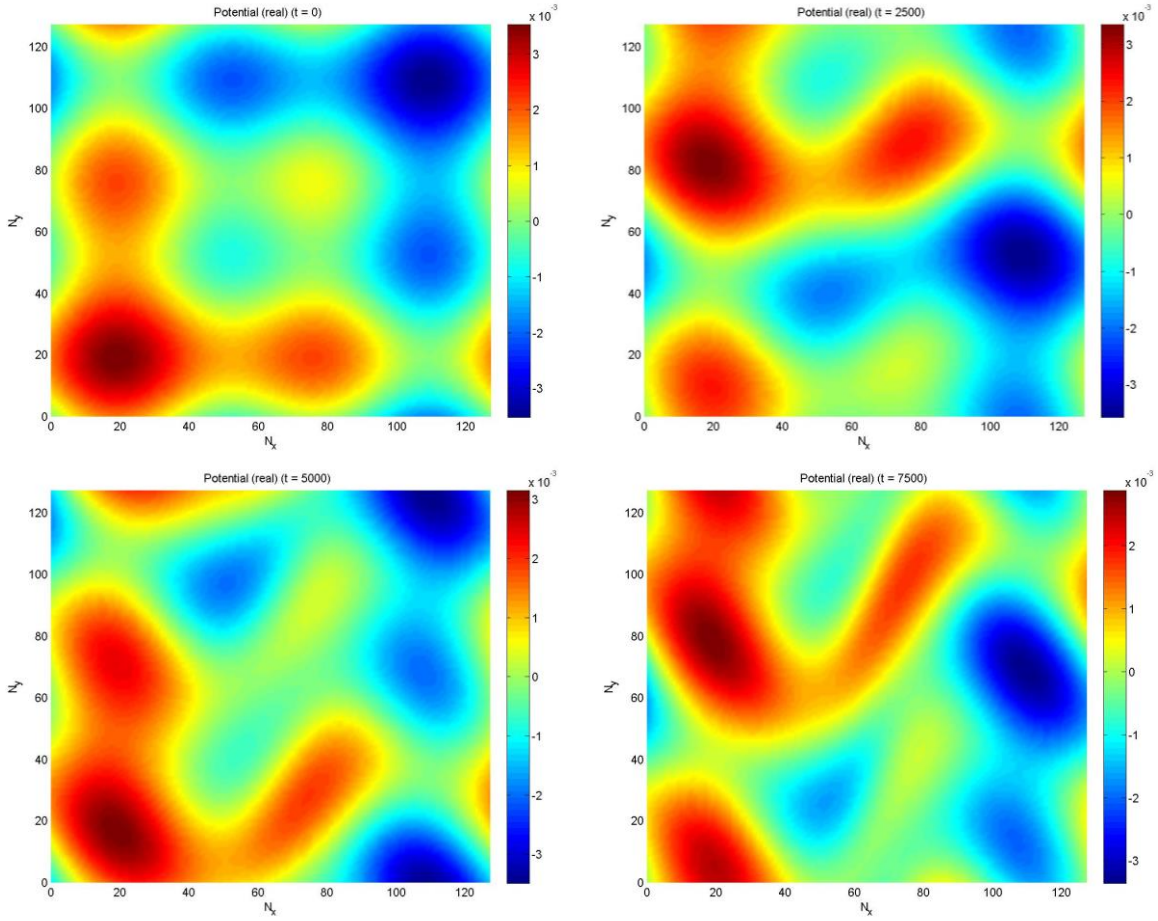


Figure 4.3.6. Electric potential evolution using with the default velocity profile for only linear terms without the presence of ion drift, $v_g = 0$. Snapshots taken for time steps at 0, 2500, 5000, and 7500. Notice the final distributions begin to form zonal flows dependent on the shear velocity profile, typical of the nonlinear instability phase.

4.4 The Spectral Anomaly

As demonstrated in Chapter 4.3, the simulations produced by the MPI RT-KH program are agreeable to the ones produced at Voss Scientific. However, this is only true for relatively few time steps for low shear velocity magnitudes. Not yet visible in the densities and potentials shown in Chapter 4.3 lies a phenomenon that is both unexpected (as it does not occur with the program developed by Voss Scientific) and completely destructive to the simulation itself.

As the simulation is allowed to progress forward, an anomaly repeatedly occurs at the minimum of the shear velocity – so in the case of the first shear velocity in Figure 4.3.1, it occurs for all y at $x \approx 42$ with an x with arbitrary width. This anomaly – shown in Figure 4.4.1 – grows in intensity (rapidly approaching infinity) but not spatially, until it completely dominates the function. If the minimum of the shear velocity is moved, the anomaly moves with it.

Numerous attempts to debug this problem have been attempted and with little success. The localization of the anomaly suggests an immediate problem with the derivative of the shear velocity (i.e. where the derivative is zero). This would explain the exponential growth of error if there was a divide-by-zero occurrence. It does not explain, though, why the anomaly is not apparent at the maximum of the shear velocity. The maximum too has a derivative of zero and should experience a similar issue, but it does not. The anomaly is only ever strictly at the minimum. The derivatives of the shear velocity (both first and

second, which are used throughout the vorticity time evolution) were analyzed for errors and they were indeed calculated correctly using spectral differentiation.

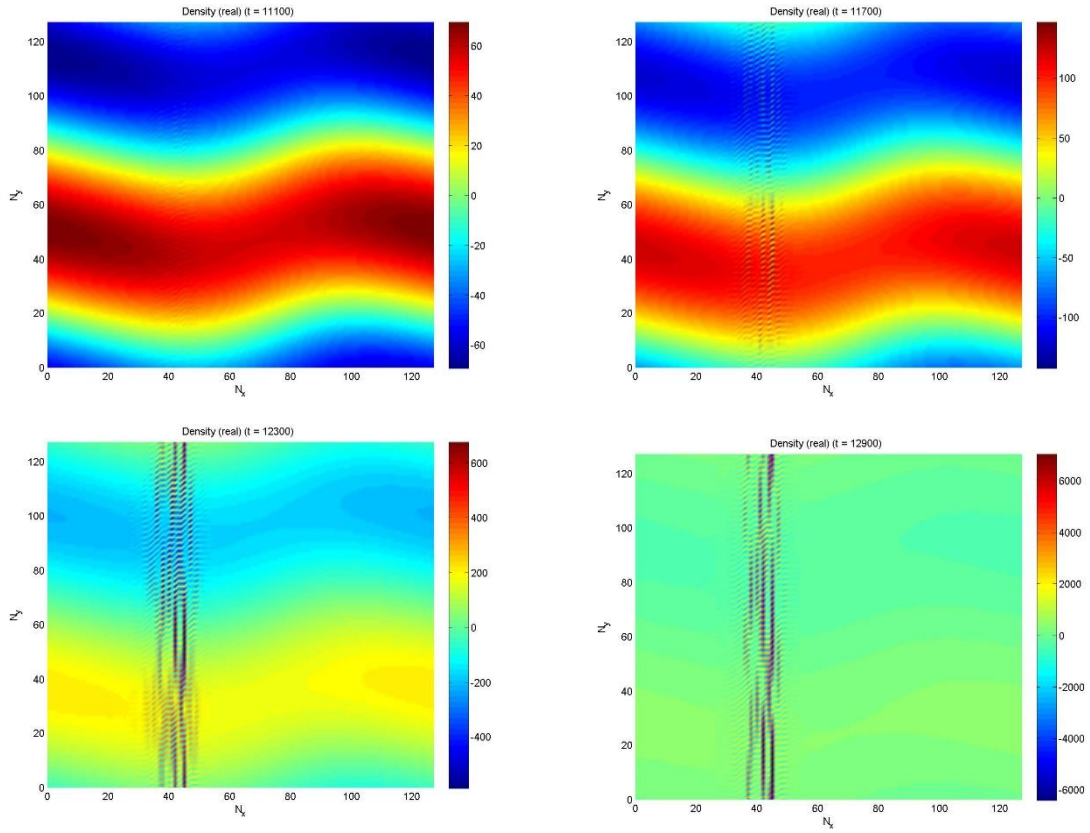


Figure 4.4.1. Time evolution of the plasma density RT-KH simulation with the same initial conditions as in Figure XXX. As the number of time steps surpasses 10000 (shown here for 11100, 11700, 12300, and 12900), an anomaly at the minimum of the shear velocity begins to develop and dominate the function. The same anomaly occurs for the electric potential.

It was briefly considered that the location of the artifact was a pure coincidence due to the separation of the grid-space when implementing MPI methods. If improper send/receive commands were being used, then the anomaly could be easily explained and solved by ensuring that data was properly transferred between processors. This turned out not to be the case, as running with a single processor and even removing all MPI functionality produced the same anomaly.

The idea was also proposed that perhaps the anomaly only occurred at the minimum of the velocity because that was the first zero derivative point to be reached when indexing the array, arguing that the maximum too would cause the error, but it was just failed to be reached when indexing the array. If the first zero derivative (and the first error) occurred at the point n_0 lying within the bounds $0 < n_0 < nx$, and the points were indexed sequentially from 0 to nx, then it could be a possibility that any values after the n_0 (so lying in the $n_0 < nx$ region) could simply not be calculated if the operation was terminated prematurely. This proposal of course does not explain why the anomaly has a definite width, but was disproven nonetheless by both indexing the array backwards (from nx to 0) and by testing a simple sine wave shear velocity where the maximum was traversed first before the minimum. In both instances, the anomaly occurred only at the minimum of the function.

One solution may come to mind to simply introduce a shear velocity that does not have a single minimum, but only a maximum, such as a Gaussian distribution (or bell-shaped curve). Ensuring that the function reaches approximately zero at the boundaries of the grid so that spectral methods don't immediately fail (i.e. making the function periodic) is a simple enough task. Yet implementing the Gaussian velocity produces symmetric anomalies along both boundaries of the grid, where the velocity reaches its lowest values, displayed in Figure 4.4.2.

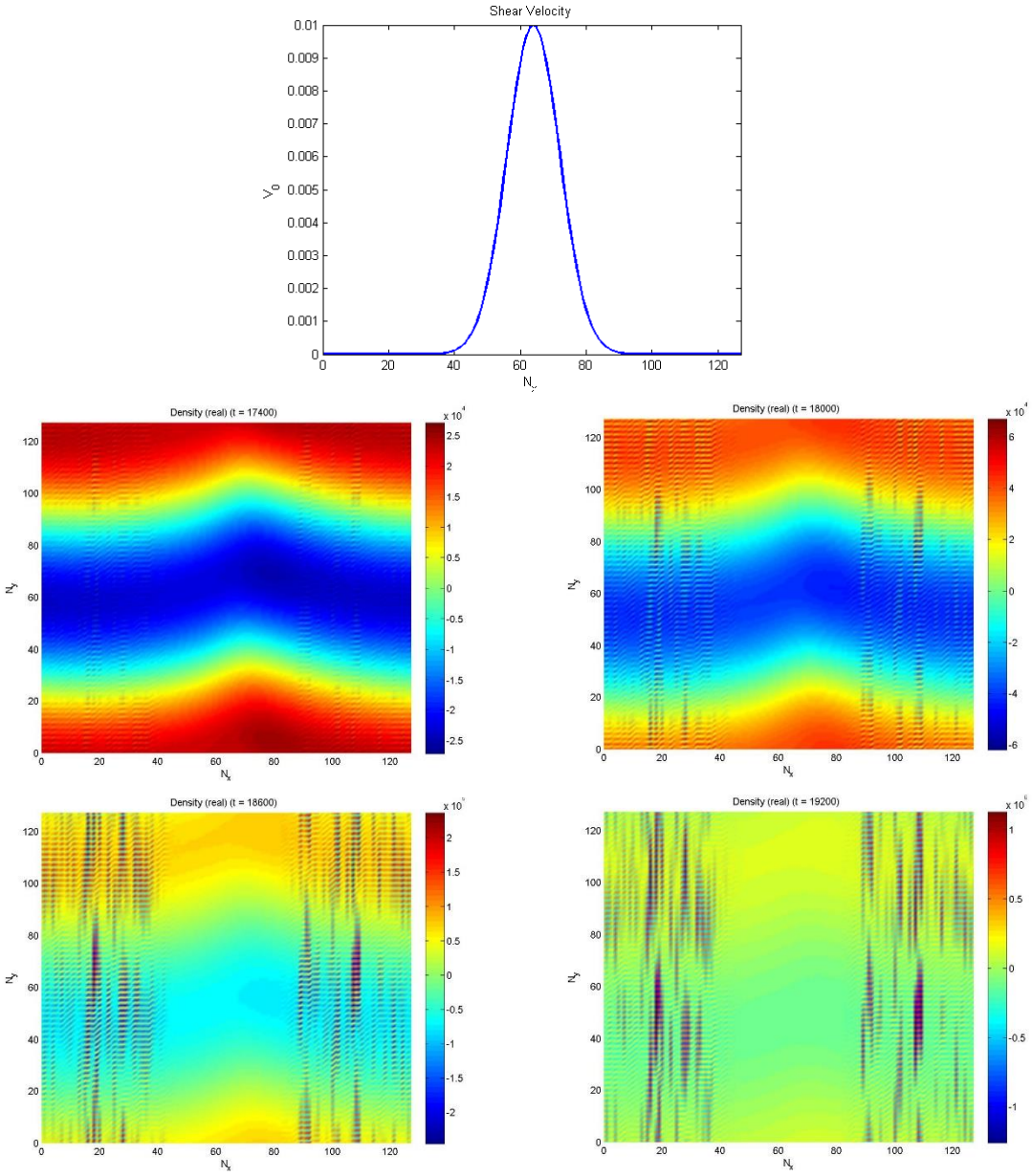


Figure 4.4.2. Time evolution of the plasma RT-KH simulation when given a bell-shaped (Gaussian) shear velocity, shown in the top figure. As the number of time steps surpasses 17,000 (shown here for 17400, 18000, 18600, and 19200), the anomaly can be seen to develop everywhere but the middle of the x grid space, where the shear velocity is approximately zero. The same anomaly occurs for the electric potential.

With the velocity fully analyzed, attention then shifted to the vorticity and density function themselves. If the velocity wasn't the problem, then something that uses the velocity must

be causing the issue. If all terms involving the shear velocity were removed (set to zero), then the functions evolved without incident. To further explore the issue, Equation 1.3.9 was first decoupled from 1.3.8 by removing the φ dependency. This way, if the problem lied in the vorticity time evolution then it would stay localized to the vorticity; the density should evolve without incident. This is indeed what happens, implying that a term in 1.3.8 was the culprit.

Removing the dependency of each variable one a time, term by term, $\frac{\partial \Delta\varphi}{\partial y}$ was localized to be the issue. In the wishful thinking that it was merely the method of taking the derivative of the Laplacian that was the problem, $\frac{\partial \Delta\varphi}{\partial y}$ was replaced by $\frac{\partial \Delta n}{\partial y}$, which is already used in 1.3.8 and derived in the exact same fashion. Alas, this replacement saw no anomaly occur, meaning that the issue was not in the derivative methods but in the vorticity itself.

The source of the anomaly was therefore localized to the first, four-term coefficient of 1.3.8 and the vorticity's y-derivative. In other words, solving the differential equation shown below:

$$\frac{\partial \Delta\varphi}{dt} + \left(V_0 + \tau_i \kappa_\rho - \nu_g - \frac{\tau_i}{2}\left(V_0'' + \kappa_\rho V_0'\right) \right) \frac{\partial \Delta\varphi}{dy} = 0 \qquad (4.4.1)$$

Removing individual V_0, V_0', or V_0'' dependency still resulted in various anomalies occurring, outline in Table 4.4. Interestingly, removing all velocity dependent terms (so

keeping only the $\tau_i \kappa_\rho - \nu_g$ coefficient) also results in anomalies, even though this is a fairly simple PDE to solve.

Table 4.4. Anomaly descriptions when manually setting shear velocity components and evolving 4.4.1 in time. The vorticity varied only in the y-direction ($-\sin(2\pi y/ny)$). Anomaly directions are given when viewed from a standard $nx \times ny$ plot (x-axis horizontal and y-axis vertical). All anomalies occurred after roughly 10,000 steps.

V_0	V_0'	V_0''	**Anomaly Description**
0	Nonzero	Nonzero	"Thin" Periodic Diagonals Across Grid ∴
0	0	Nonzero	"Thick" Periodic Horizontals Across Grid ⋯
0	Nonzero	0	"Thin" Periodic Diagonals Across Grid ∴
0	0	0	"Thick" Periodic Horizontals Across Grid ⋯
Nonzero	0	Nonzero	V_0 Minimum Localized Vertical ⋮
Nonzero	0	0	V_0 Minimum Localized Vertical ⋮
Nonzero	Nonzero	0	V_0 Minimum Localized Vertical ⋮
Nonzero	Nonzero	Nonzero	V_0 Minimum Localized Vertical ⋮

Due to its simplicity, it's easy to delve into the triple-zero case a little deeper. If the vorticity is only dependent on x, then its y-derivative will be zero. In this case the y-derivative should be nonzero, so it can be assumed that the vorticity is only dependent on y. Since the vorticity is determined by the Laplacian of the potential, then the potential is also only dependent on y. Setting the initial potential at $t = 0$ to be a simple sine function, $\sin(y)$, the initial vorticity becomes $-\sin(y)$. Solving 4.4.1 by hand then becomes trivial, giving a simple oscillating sine wave with respect to time:

$$\Delta\varphi = \sin\left(y + t\left(\nu_g - \tau_i \kappa_\rho\right)\right) \tag{4.4.2}$$

If 4.4.1 is simulated with respect to time given $V_0 = V_0' = V_0'' = 0$, the exact result given by Equation 4.4.2 is initially observed: an oscillating sine wave in the y-direction. However, as time progresses the anomaly returns, spanning all values of y and oscillating in time; exactly what Table 4.4 tells us. Yet 4.4.2 has no point of failure in its evolution, regardless of what variables are chosen, and the other differential equations do not exhibit this same phenomenon when allowed to evolve on their own.

So what else could be the source of this anomaly? Going back to Figure 2.3, a new feature may be noticed that wasn't beforehand: the periodicity of the error in spectral methods. Although this is indeed largely machine precision, it's clear that at the end points the error is the highest, but there are also two points relating to the relative extrema of the function that also experience high errors. In particular, this error is unquestionably highest (when ignoring end points) at the function's minimum value, regardless of the order of the derivative (in the case of Figure 2.2.3, the test function was a sine wave with a minimum of -1 at $3\pi/2 \approx 4.71$). Sequential use of the spectral method for differentiating will only result in the accumulation of machine errors, until eventually the noise dominates the function. Since the maximum noise seems to occur specifically at the minimum of the function and does have definite width, it would explain why all the anomalies also appear at the shear velocity's minimum. The rapid increase of the anomaly is characteristic of an accumulative error, also supportive of this theory.

Why then does the error only occur when solving for the vorticity? Considering the outlying coefficient is the only one that includes all three velocity terms (V_0, V_0', and V_0''), this term would naturally have the highest error at the velocity minimum since the velocity derivatives were also found using spectral methods. However, this does not explain the inability of the spectral method in solving simplistic PDEs that produce solutions such as 4.4.2. A different time-stepping algorithm may be needed instead of the predictor-corrector method, or there may be some other error associated with solving time dependent differential equations. Further effort could be made to validate or disprove these claims, or to simply further understand the origin of the spectral noise and the anomaly itself.

Chapter 5

Conclusion

The previous work done by the programmers at Voss Scientific provided an excellent ground basis for modeling RT-KH instabilities in ionospheric plasmas. Their program, although capable of proper simulations, did leave room for optimization.

The conversion from finite basis methods to spectral differentiation posed some difficulties when calculating terms such as the Laplacian. The solution that was finally used minimized computational runtimes and achieved a maximum theoretical speedup of 3.63. This was substantially lower than the built in MPI functionality of FFTW, which could theoretically reach a max speedup of 10.26, but still had faster runtimes for four or fewer processors. This implies that an even faster method of calculating the Laplacian of a 2D function could be created that both minimizes runtime while giving maximum speedup.

The MPI RT-KH program was then largely built around the fundamental core structure of the custom Laplacian solution. The final program achieved a maximum speedup of 1.40 when ignoring the nonlinear components of its equations, but eventually failed as the simulation progressed forward in time. This failure was due to an anomaly that occurred at the minimum of the shear velocity. The error was traced to the y-derivative of the vorticity. Solving the PDE through spectral differentiation appeared to be the issue. The high machine error at the function's minimum accumulated to the point of dominating the

entire output. This behavior is not fully understood, and further analysis should be done to examine the source of the spectral error.

Chapter 6

Bibliography

[1] "The Ionosphere." *University Corporation for Atmospheric Research Center for Science Education,* Colorado, 2014.

[2] Sotnikov, V. et al. "Low Frequency Plasma turbulence as a Source of Clutter in Surveillance and Communication." *Air Force Research Laboratory*.

[3] Cairns, Iver. "Earth's Ionosphere." *University of Sydney*, Sep.1999.

[4] Diaz, Antonia J., Roberto Soler, Jose L. Ballester, and Marcel Goossens. "Kelvin-Helmholtz and Rayleigh-Taylor instabilities in partially ionised prominences." *X Scientific Meeting of the Spanish Astronomical Society*, Jul. 2012, pg. 776-781.

[5] Frey, Pascal. "The finite difference method." Sorbonne Univeristy, Chile, 2017.

[6] Li, Shengtai and Hui Li. "Spatial Discretization." *Los Alamos National Laboratory*.

[7] Trefethen, Lloyd N. "Spectral Methods in Matlab." *MathWorks*.

[8] Johnson, Steven G. "Notes on FFT-based differentiation." *MIT Applied Mathematics,* 2011.

[9] Asanovic, Krste et al. "The landscape of parallel computing research: a view from Berkeley." *Electrical Engineering and Computer Sciences University of California at Berkeley*, Dec. 2006.

[10] Brown, Robert G. "Amdahl's Law & Parallel Speedup." *Duke Trinity College,* 2000.

[11] Michalove, Aaron. "Amdahl's Law." *Washington and Lee University*.

Lewis, Ted G. and Hesham El-Rewini. "Introduction to Parallel Computing." Pr*entice-Hall Inc*., 1992, pg. 31-32, 38-39.

[12] Amdahl, Gene M. "Validity of the single processor approach to achieving large scale computing capabilities." *IBM*, Sunnyvale, Caifornia, 1967.

[13] Kendall, Wes, Dwaraka Nath, and Wesley Bland. "A Comprehensive MPI Tutorial Resource." 2018.

[14] "Parallel FFTW." *Massachusetts Institute of Technology*, Nov. 2003.

[15] "Parallel Versions of FFTW." *Massachusetts Institute of Technology*, Nov. 2003.

[16] Kohn, C., et al. "Streamer properties and associated x-rays in perturbed air." *Plasma Sources Science and Technology*, Jan. 2018.