Senior Project: Parallel Programming

Natalie Loebner

May 15, 2006

Abstract

After years of technological advances the speed of single processors are beginning to meet their physical limitations. Thus, parallel programming has become an increasingly important tool in scientific research and commercial industries. This paper includes an exploration of how the size and number of calculations in various problems benefit from parallelization in a distributed-memory MIMD environment.

1 Introduction

Parallel programming is responsible for many advances in high powered computing, specifically concerning computing speed. The processing speeds of computers are beginning to reach their physical limits, while the problems left to conquer are not getting any smaller. Linking multiple processors together in simultaneous computation has given programmers the tools to tackle some incredibly large computational barriers.

This paper is an exploration of how and when to employ parallel programming practices. These methods provide the means to make certain computations faster through the use of simultaneous computations; however, not every program will necessarily benefit from a division of labor between processors. The particulars of parallel programming differ greatly depending on the physical set up the computers and the nature of the given problem. This paper will touch on how these factors change the approach or even the applicability of computing in parallel.

With the advent of parallel programming, large scale problems that were once considered an impossible problem have become more accessible to those in

the programming world. Decreasing the time it takes to arrive at a result can actually determine whether or not the answer is even useful. For example, Peter Pacheco, in his book *Parallel Programming with MPI*, describes how the factors needed to accurately predict the next day's weather are so complex that it would take over a day to arrive at a final analysis on a single processor (Pacheco, 2-3). Obviously the solution becomes obsolete when a program will tell us tomorrow's weather the day after tomorrow. Thus, utilizing parallel programming allows meteorologists to run their computations in a timely manner so that they are able to predict tomorrow's weather today. This is just one example of why parallel programming has become paramount in many of today's scientific research and business endeavors.

The paper is structured as follows. Section 2, is a background section that covers important information necessary for understanding parallel programming, specifically the parallel programming library MPI. This section will also discuss the four main types of architectures used in parallel processing and how to implement MPI. In Section 3, we will go into detail about conditions for effectively parallelizing a given problem. The section will outline three different problems, giving both good and bad examples of processes running in parallel. These problems include calculating π, integration using the trapezoid rule, and the pendulum problem. In Section 4, we will take a look at programming efficiency of the previous problems by applying Amdahl's Law. In the concluding section, there will be a discussion on the future directions of this research.

2 Preliminaries for Parallel Programming

Parallel programming uses the idea of divide-and-conquer to provide more computing power than the conventional sequential computer. By increasing the number of processors working on the same problem, thereby sharing the workload, parallel programming offers a cost-effective solution for the constantly increasing demand for high-powered computing. Probably the most influential factor for the ever-increasing growth of the parallel programming field is its potential for use in the commercial sector. Thus, the demand for better technology and new innovations are driven by financial motivation.

In this section, there will be an introduction to some common set-ups of physical computer systems; these are called system architectures. The physi-

cal networks of computer clusters have become significantly more sophisticated since the early IBM computers. We will look at these systems starting from a very simple design and progress to the more complex. This will include the different ways that computers are connected for communication and the different ways computer memory can be set up. Then, we will discuss how these factors affect the implementation of parallel programs. After an introduction to MPI (Message Passing Interface), which is a library that allows data to be processed in a distributed-memory environment, we will describe how MPI was implemented in the Whitman Mathlab.

2.1 Single Instruction Single Data (SISD)

A single-instruction single-data machine is also commonly called a classical von Neumann Machine. These systems are separated into two divisions which are the memory and the CPU (central processing unit). The memory portion holds both the program instructions and the data while the CPU interprets and executes the commands in the program. In the SISD model, the CPU is further divided into two more sections called the control unit and the arithmetic-logic unit (ALU). The control unit is in charge of executing the programs and the ALU does the actual computations called for by the program, refer to Figure 1. Instructions on SISD machines are done in a sequential manner. Since all of the instructions and data are stored in the main memory, they then must also be stored in what are called registers or fast memory during execution of the program; otherwise, the CPUs would not be able to access the information. The transfer between the memory unit and the registers in the CPUs dictates how fast the computer(s) will be able to arrive at a result. The path where the information travels is called a bus and it is typically where bottlenecks can occur.

A bottleneck is one of the most common drawbacks for parallel programming. The bottleneck in a bus is the place where information has a tendency to get backed up, consequently slowing down the flow of data. If the CPUs are computing at speeds faster than the bus can transfer the data, then the program will run only as fast as the slowest portion of the system.

The most common application for the von Neumann design is called pipelining. This is where the execution of commands can be done simultaneously on multiple CPUs producing a result every instruction cycle. The trick is to, as

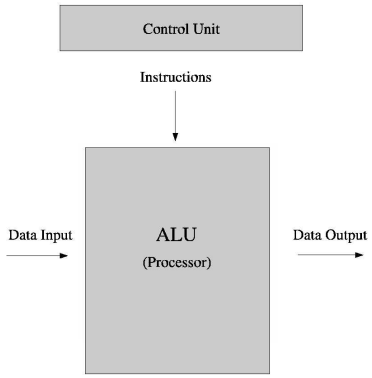

Figure 1: SISD Architecture: The data is transfered to the CPU (shown in the figure) from the main memory unit.

programmers say, keep the pipeline *full*, namely having enough computations so that each CPU is being utilized yet not so many as to cause a bottleneck. Since information must be transfered to each CPU in order for the program to execute, the addition of more CPUs for more computations will not necessarily yield faster results, which means that the SISD design does not scale well. More often than not, a program will not lend itself to such a simple division of labor. However, the SISD architecture tends to be the architecture of choice for today's complex problems because developing programs for this model is well understood.

2.2 Single-Instruction Multiple-Data (SIMD)

The single-instruction multiple-data design is divided differently than the SISD; mainly it lacks the main memory component. This system has only one CPU acting as the control unit and a number of ALUs which execute the given commands, with a limited amount of personal memory. The CPU will broadcast the same command to all the ALUs, which will either respond by computing or remain idle. For example, the CPU may want to do a calculation on an array and do the same calculation on each index. A different index can be sent to a different ALU and the entire array can be tested in one cycle of instructions. A diagram of this architecture can be seen in Figure 2

SIMD machines do not do well with programs with a lot of conditionals

because this could result in many ALUs staying idle for long periods of time. However, for tasks that are effective with this system, it does seem to scale easily to larger problems. SIMD machines handle vector and matrix operations efficiently, which makes this design well-suited for scientific computing.

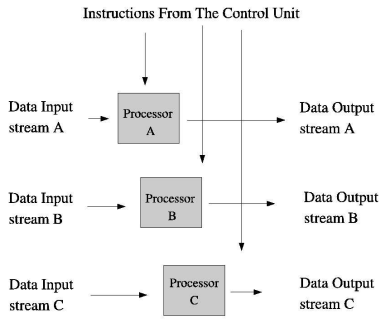

Figure 2: SIMD Architecture: The boxes labeled A,B, and C represent the ALU components.

2.3 Multiple-Instruction Single-Data (MISD)

Multiple-instruction single-data systems are considered by some to be vector processors; however, there is disagreement between experts that this type of system is only a modification of the SIMD. For this paper, we will define MISD computing systems as multiple processors executing different instructions on different data simultaneously. Refer to Figure 3 to view diagram of this architecture. An example of an MISD architecture would be a system where each machine would perform different operations on the same data set. This is not a common architecture because there are not many applications where this type of computation is useful. MISD machines tend to be used for intellectual exercises rather than real world problems.

2.4 Multiple-Instruction Multiple-Data (MIMD)

The multiple-instruction multiple-data systems have autonomous processors that each have a control unit and an ALU, allowing each processor to run its own programs independently if needed. Thus, this system can execute mul-

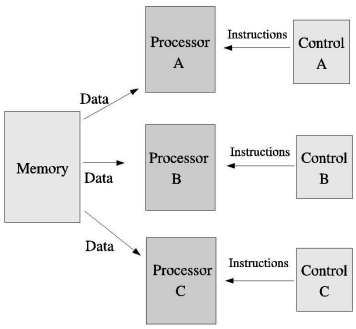

Figure 3: MISD Architecture

tiple instructions on multiple sets of data. This kind of system is considered *asynchronous* and only operates synchronously if specifically programed to operate that way. Since this design has separate instruction and data stream, it is well suited for a wide variety of applications. There are two main categories for MIMD machines which are based on how processing elements are linked to the main memory. These categories are shared-memory MIMD and distributed-memory MIMD.

2.4.1 Shared-Memory MIMD

The shared-memory systems are often also called multiprocessors. They are usually set up to have many processors which share the same memory module and are interconnected through a network. There are two main kinds of architectures for these networks, where the simplest one is a bus-based system (Pacheco, 16). However, this model can have problems when the memory is being accessed by several processors, causing delay time in fetching and storing data because there is only one path for the memory to travel. This design is often limited to a relatively small number of processors, refer to Figure 4.

Switch-based architectures seem to be the most common model used. Like the name implies, there is a switch-based interconnected network, which allows more than one path through which information may travel. These systems also have a tendency to be small due to the cost of the equipment. However, there are different modifications which have their own benefits and limitations. We will not go into detail about the types of modifications because there are so

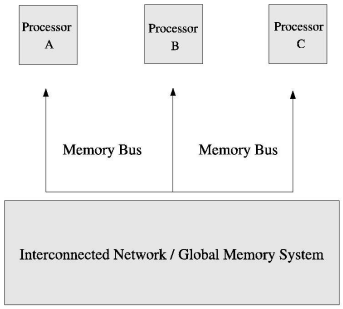

Figure 4: Shared-Memory MIMD Architecture: This figure shows a bus-based design.

many of them and tend to be problem specific.

Cache *clashing* is a very common and troublesome problem with shared-memory models. This means that different machines are altering the same variable. An example of this would be if each processor is supposed to take the same variable x and increase it by some amount. If processor A increases x and then processor B increases this new value of x then processor B's calculation is not correct according to the intended outcome of the program. Since it is nearly impossible to ensure that the processors are accessing the same information at the exact same time, great care is needed to eliminate these particular errors.

Shared-memory MIMD machines are used because they are easy to build, easy to program, and often little overhead communication is necessary. The drawbacks to this model are that a failure of any memory component or any processing element will affect the entire system and it is rather difficult to scale because more processors will require more memory.

2.4.2 Distributed-Memory MIMD

Distributed-memory MIMD systems' CPUs have their own private memory and operate as isolated units in the overall program. In the most ideal case it would be best to have a fully connected system where communication would not cause delays, where there would be no interference in communications, and simultaneous communicating can take place at any time. Since this ideal system is impractical and expensive, there are three different common layout possibilities: static networks, dynamic networks, and bus-based networks.

A dynamic interconnected network maybe closest to ideal layout. The network is made up of switches where communication is possible only where the physical hardware allows. The system can have as many or as few connects as the creators want or are able to make, refer to Figure 5.

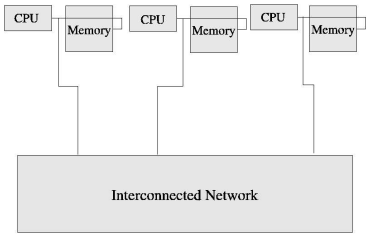

Figure 5: Distributed-Memory MIMD Architecture

A static interconnection network is considered a linear array or can be modified into a ring. There is less direct access to any given machine than in the previous model. However, depending on the type of communications necessary for a particular program, a direct connection to every machine may not be needed. The hypercube is another design which allows for relatively easy communication among machines. They are pretty simple to scale to larger sizes. There are also layouts called meshes and tori which add more dimensions to the computing process.

Bus-based networks are a cluster of workstations. They tend to be slow because the communication paths are easily filled. They also tend to be small networks.

Distributed-memory MIMD machines are also easily built and are well-suited for real time applications but harder to program than shared-memory machines. Unlike shared-memory systems, a failure in a given component will not necessarily affect the entire system, and the problems are often easily isolated. Another difference from the shared-memory design is that the distributed-memory systems lend themselves very well to massive scaling.

2.5 The Whitman MathLab

The Whitman MathLab is a Distributed-Memory MIMD system. Each processor is a fully functioning independent computer which can execute a different

program from any other computer in the network. The hub which connects the computers is a dynamic system which uses switches. Thus, any computer can communicate with any other computer in the network directly. The hub is Cisco System's Catalyst 2950 series version $12.1(19)EA1c$. The hub has a one gigabyte bandwidth which allows for fairly fast communications. The Math-Lab's computers are Dell OptiPlex $GX280$ small mini-tower machines. Each of the computers is running the Centos 4.2 operating system with Linux version $2.6.9-22.0.2.c1smp$. The compiler is gcc version 3.4.4 with the open source `SSL` version $0.9.7a$. All the data generated in this paper is done using this system.

2.6 MPI

MPI, as we said earlier, is a Message Passing Interface which allows data to be processed in a distributed-memory environment. This library is compatible with both C and Fortran, which makes this library extremely useful and versatile. For the sake of this paper, when we mention specifics about MPI we are referencing the use of open source MPICH version 1.2.7.[1] This library allows a user to compile a program and then run on a selected number of processors. This library also provides a reasonable level of portability so the program can be executed on a different cluster of computers without having to change the source code.

The main idea of simultaneous computation using MPI begins with the idea that each processor is given a *copy* of an executable program with which each processor will perform the same operations. The only differences between any two processors are their identifications, or ranks. The rank is the computer's identification which falls between 0 and the number of processors minus one. The respective rank of each machine provides a means with which a programmer can specify different instructions for each CPU.

The typical process for writing a program with MPI starts with writing a program that will run on one machine and then adapting it to work with multiple ones. The MPI commands are only recognized by a computer's compiler when MPI is initialized. Namely, there are begin and end commands which tell the computers when to reference the MPI library. In between these start and end lines, the programmer will be able to perform operations on the computers

[1]This free version of MPICH, developed at Argonne National Lab and Mississippi State University, is available at `ftp://info.mcs.anl.gov` which has all the necessary instructions to get MPICH up and running.

and can differentiate between machines using only the rank. During the MPI initialized portion of the code, the machines are able to communicate with each other. So if the program requires a computation from processor rank 0 to be shared with the other machines it becomes possible to do so using message passing.

Message passing is useful on many different levels. A parallel program is capable of dividing up the work load and then recollecting the data at the end of the computations, as well as sharing information during the computation process. The notion of message passing is very similar to the notion of writing and sending a letter. The data is like the physical letter itself, while the envelope and address are how the data is stored and where the data is ultimately sent. The analogy may be taken further by comparing the ideas of first and second class mail (Pacheco, 45-47). Some information may be tagged as important and forces the computer to either immediately send or receive the data, while other information may not be as important and will be dealt with after all the important data has been dealt with. The ranks of the machines determine who sends what data and which machines will receive the data. The size of the data must also be indicated so the machines know how much space to allot so nothing is lost. The kind of information being sent is also a required set of information in the sending and receiving process.

3 To Parallelize Or Not To Parallelize? That Is the Question.

In this section we will use three different example programs to illustrate the effectiveness of parallel programming with different applications in an MIMD environment. The first example uses a program designed to calculate π, the second uses the trapezoid rule to calculate the area under a curve, and the third example uses second-order differential equations to solve a pendulum problem. All three of these examples are particularly useful because they all lend themselves to easy parallelization, yet the effectiveness of the parallelization differs. The big difference between the programs is the amount of work each processor eventually does, namely the size or amount of calculations. The amount of work has a big impact on whether or not parallelization is useful.

An analogy to make this point clearer would be to consider a laundromat.

Assume we have a large load of laundry to wash: it is logical to assume that running several loads simultaneously would make the job go faster. But maybe we need to wait for another person to finish their load, or maybe it takes a long time to separate the darks from the lights at one point in the process. These and other considerations may lead to lengthy delays. Thus, it ends up being faster to do simultaneous loads only if the other factors do not end up making the process take longer. This concept of overhead costs will become clearer when we compare the three programs in this section.

In the description of the three examples there will be an introductory section explaining the basic math involved. Following this will be the discussion of what particular methods of parallelization are used for each program. Finally we analyze the effectiveness of the parallelization process.

3.1 Calculating π

The first example is a program that approximates π to 16 digits. This program is included with the MPICH package to test if the library is properly installed.[2] The method used to calculate π in this program is said to be credited to Gottfried Leibnitz sometime in the 1680s. The basic idea uses the fact that $\arctan 1 = \frac{\pi}{4}$. Thus, we will calculate π by numerically integrating $\tan^{-1}$ which is $\frac{1}{1+x^2}$, then multiply this result by 4 to get an approximation of π. The integrand is defined as $f(x) = \frac{4}{1+x^2}$. The function evaluations are distributed across different computers using the scheme where i is the rank of the processor and $x = 1 + (i - 0.5))$. The function f is summed over a specific number of intervals chosen by the user at run time.

The distribution of the work load is dependent on the number of intervals chosen by the user. Each computer does a particular set of calculations for a certain value or values of x, determined by rank of the computer. Then, after each computer has calculated its own values of the integral, their results are summed. The result is rounded to the first 16 decimal places and compared to the actual value of π. The output of the program indicates to the user the accuracy of the approximation and the time it took to calculate the result.

There are two main parallelization commands used in this program. The

[2]Instructions on how to run this program can be found on the website, `http://www.-unix.mcs.anl.gov/mpi/mpich1/docs/mpichntman.pdf`, under the heading *Making and running an example.*

first one is the broadcast command, which instructs each computer how many intervals of the summation it is responsible for computing. Broadcasting information is a method that uses a tree-based communication scheme to send out data. In Figure 6 the path of how the data is distributed is laid out. This is faster than having the root processor (the computer with rank 0), communicate individually with the other machines. Each computer goes through its calculations at which point the second parallelization command comes into play. This next command is called the reduce command, which also uses a tree-based communication scheme to recollect the data from all the computers. The reduction tree operates the same as the broadcasting tree, except the diagram would be a 180 degree rotation of Figure 6. There is an entry requirement which tells the computer what to do with the data, which in this case is to sum all the results together on the root processor.

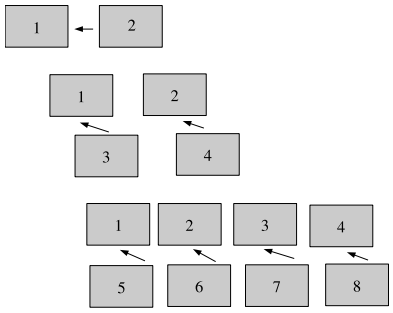

Figure 6: Broadcasting method using a tree-based communication scheme. Each box represents a machine with the number indicating its rank.

This parallelization scheme is not effective in an MIMD environment since the program runs slower with each addition of another computer in the computation process. We conducted several test runs using constant boundary values for the integral but varied the number of processors running the program from 1 to 15 computers. Each time the same trend was apparent, the program is far more efficient on a single computer. In Figure 7 the graph shows the total run time increase as the number of processors is increased.

The reason for this relies on the fact that communication between machines takes time. Thus, to divide up and recollect the data takes longer than merely running the program on a single computer. This would be comparable to a task that takes a large level of explanation. Thus, having a knowlegeabel person

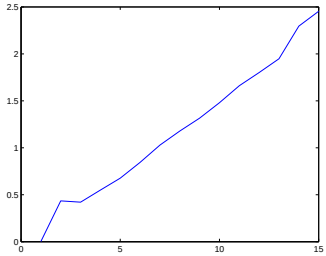

Figure 7: Graph of the efficiency for the calculation of π. The x-axis indicates the base ten logarithm of the number of processors and the y-axis indicates the base ten logarithm of the runtime of the machines in seconds.

complete it themselves would be faster than trying explain what needs to be done to other people.

3.2 The Trapezoid Problem

This program is taken from the Pacheco book, which includes a detailed description of how each of the MPI commands operate (Pacheco, 53-76).[3] The trapezoid rule is a method of approximating the area under a curve by dividing the area into trapezoids and summing their areas. The approximation gets better as the base of the trapezoids gets smaller; thus, the more divisions, the better the result. This notion of dividing up the region into many smaller areas lends itself easily to parallelization. The formula for the Trapezoid Rule is as follows, where in this particular example we set the function $f(x)$ to be x^2,

$$A(x) = \frac{h}{2}[f(x_0) + 2f(x_1) + 2f(x_2) + ... + f(x_n)].$$

The n trapezoids are all assumed to have the same base length h. The length of h is defined by the starting value of the function a, the end value of the function b, and the number of intervals, which yields the definition $h = \frac{b-a}{n}$. The values of $f(x)$ are the actual points on the curve which correlate to the height of the sides of the trapezoids.

[3]The source code for this program can be obtained at `http://www.usfca.edu.mpi`.

The first step, again, is creating a program that will do this calculation on one computer before moving on to the parallelization. The program allows the user to indicate the number trapezoids used in the calculation as well as the a and b boundary values for the function.

We ultimately created two different parallelized versions of the trapezoid program. Both of these programs divide the work for the computers in the same fashion. There are several methods that are standard when dividing up data, among which are block distribution and panel distribution. Block distribution takes chunks of the data and assigns each to machine a chunk, while panel distribution takes each piece of data giving each machine a single packet of information, and continues dispersing the data until all the data is assigned.

The second program uses panel distribution to divide the data. The work is distributed by subinterval, namely each machine is responsible for one or more trapezoids determined by a local a and b value. These local endpoints are the endpoints of the specific region each computer is calculating. This depends on the number of intervals the user chooses to use, so the more intervals the more trapezoids each computer calculates. Then each computer sums up its local area, the data is collected, the data is summed together, and the result is the approximated area under the curve.

The first method of parallelization uses basic communication commands. This is where the root processor sends out the data to each computer one by one and recollects the data one by one. Due to the inefficient manner by which this distribution and recollection of data is done, this program has the same problem as the one which calculates π: where the serial version ends up being faster than the multi-computer version.

The second method uses the broadcasting and reducing commands, which were discussed in the previous example. Instead of each computer having its own individual communications with the root processors, these commands have simultaneous communications occurring on many different computers. This, in turn, optimizes the program further and allows for the program to run faster under parallelization.

3.3 The Pendulum Experiment

The idea for the pendulum experiment came from the book *Chaos and Fractals: New Frontiers of Science* which was a collaborative work by Peitgen, Jurgens, and Saupe. The pendulum experiment is discussed in chapter 12, titled "Strange Attractors and Fractal Basin Boundaries", which covers theories based in chaos and dynamics. In this problem, there is a pendulum with a metal head suspended on a string above a flat surface where three strong magnets are placed in an equilateral triangle. The pendulum is released at different points in the plane on which the magnets rest. Each magnet is associated with a particular color. Each point is colored according to the magnet on which it comes to rest. This problem is an example of the phenomenon called basins of attraction. The set up of the problem is shown in Figure 8.

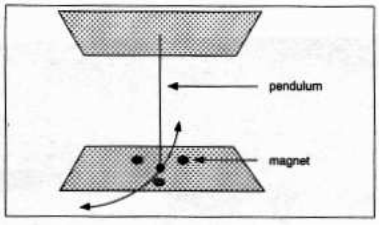

Figure 8: The Pendulum Experiment: Figure copied from *Chaos and Fractals: New Frontiers of Science* (Peitgen, 759).

3.4 Model Assumptions

There are a few assumptions that are made for the set up of this problem. First, the length of the string for the pendulum is considered *long* in relation to the spacing of the magnets. This allows us to assume that the ball moves about on the plane, rather than lifting at the edges as with a sphere of large radius. Secondly, the magnets are positioned below the metal ball head at the vertexes of an equilateral triangle, as mentioned above. Finally, the force of attraction from the respective magnets is proportional to the inverse of the squared distance from the magnet in question, as dictated by Coulomb's Law. This law states that *the magnitude of the electrostatic force between two points charge is directly proportional to the magnitudes of each charge and inversely*

proportional to the square of the distance between the charges.

3.4.1 The Parameters

This model results in a second order non-linear differential equations with two variables. The parameters of the model consist of the friction (due to the air or water or whatever the user decides), the spring constant, and gravity. The pendulum's mass moves parallel to the xy-plane where the magnets are positioned at a distance d from the plane. Thus, if we assume that the pendulum is at position $(x, y, 0)$ and the magnet is at $(x_1, y_1, -d)$. Using Coulomb's Law we will assume that the force of the magnet on the pendulum is inversely proportional to the square of the distance between the two points:

$$\frac{1}{(x_1 - x)^2 + (y_1 - y)^2 + d}. \tag{1}$$

The pendulum motion is limited to the xy-plane, so we must use the component of the force parallel to the place, namely the cosine of the angle α, as shown in the Figure 9. After applying some trigonometric transformations, we can derive the differential equations that apply Newton's law relating total force to the acceleration of the mass.

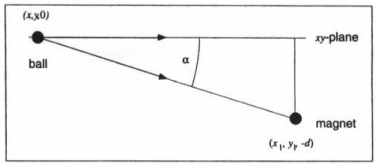

Figure 9: The division of the force of a magnet on the pendulum split into its component parts (Peitgen, 761).

In the resulting equations we define (x_1, y_1),(x_2, y_2),and (x_3, y_3) as the positions of the three magnets. The constant R is the friction and the constant C is the spring in the pendulum string.

$$x'' + Rx' - \sum_{i=1}^{3} \frac{x_i - x}{(\sqrt{(x_i - x)^2 + (y_i - y)^2 + d^2})^3} + Cx = 0 \tag{2}$$

$$y'' + Ry' - \sum_{i=1}^{3} \frac{y_i - y}{(\sqrt{(x_i - x)^2 + (y_i - y)^2 + d^2})^3} + Cx = 0 \tag{3}$$

These equations require the specification of the position (x_0, y_0) and the velocity of (x_0', y_0'). We will assume that the initial velocity will always be 0.

3.4.2 Programming The Experiment

The *Chaos and Fractals* book explicitly stated that this problem would be very tedious to do without the use of a computer simulation and gave reference to the book, *Numerical Recipes In C*, to help with creating one's own computer model [5]. The *Numerical Recipes* book is a reference of algorithms that programmers can use without having to resolve how to code specific computations. The book offers a method to solve the differential equations in this problem called the Runge-Kutta four method.

To solve equations (2) and (3), we use the Runge-Kutta four method, which is a way of approximating the next step of a differential equation. It is similar to Newton's method, in that it requires initial conditions and a predetermined step size. This method acts like a weighted average, taking into account the step size and the incremented time while moving forward in the equation. The equations below demonstrate how the four steps are calculated, where h is the step size:

$$\begin{aligned} k_1 &= hf(x_n, y_n), \\ k_2 &= hf(x_n + 0.5h, y_n + 0.5k_1), \\ k_3 &= hf(x_n + 0.5h, y_n + 0.5k_2), \\ k_4 &= hf(x_n + h, y_n + k_3). \end{aligned}$$

This method generates a substitution allowing us to change the problem to an easier set of first order equations to solve. The code in *Numerical Recipes* is modified for this problem, we need a second substitution to take fourth derivatives.[4] This is an example of how we can use this reference to supply models that we can tweak to suit our needs.

[4]To better understand the modifications necessary for this program, it may be helpful to look at the Runge-Kutta chapter in Cheney and Kincaid's numerical analysis book [2].

Now that we can solve for a specific point in time, the next thing to consider is at what time the calculations should terminate. For this problem we are looking at what time and over which magnet the pendulum reaches equilibrium. There are several factors that we need to consider, one of which is to look at when the velocity gets close to zero and when the change in position also gets very close to zero. Thus, the program will mark the initial point with the appropriate magnet when the position and velocity drop below a certain tolerance. In the program, we must determine how accurately or how quickly we want to arrive at a solution. This will affect how small our error value (ϵ) should be. For this particular application, we let $\epsilon = 0.001$.

Our final step is to discretize the region. We do this to test enough points to get an interesting picture, and have small enough boundaries so that the iterations for any given initial conditions will not take too long. It has been decided that a image generated by 500 by 500 pixels will generate an adequate picture, since this would be testing $250,000$ initial points. We will see that this is true when comparing various resolutions regions. Each pixel corresponds to an initial x and y starting point of the pendulum, and the color represents the magnet over which the pendulum ultimately rests. Without scaling, testing a region where the pixel numbers correspond to actual xy-coordinates would be far too large a region to test with the given positions of the magnets.

Since the magnets are placed in an equilateral triangle, we will assign xy-coordinates $(0, 0)$, $(0, 1)$, and $(0.5, \frac{\sqrt{(3)}}{2})$. It makes sense to test a region centered around the magnets. It is also helpful to test a rectangular region since the final picture will ultimately be a rectangle. The region chosen for this particular setup is the square with the vertexes $(-1, -1)$, $(2, -1)$, $(2, 2)$, and $(-1, 2)$. To have the code properly scale the region, we want the program to take the given xy-coordinates from the 500 by 500 region and map them to the actual region of the experiment.

To do this, we use the following conversions to get the initial xy-coordinates of the pendulum that will be used in the actual calculations. We are given a value from 0 to 499 along the x-axis; we begin with zero because the array starts at 0. We then multiply this number by the length of the x region in the test region, which is 3, over the scaled region minus one. Since the x values start at -1 the equation must take this into account by adding it to the value described in the previous sentence. Then the same method is applied to the y values. To

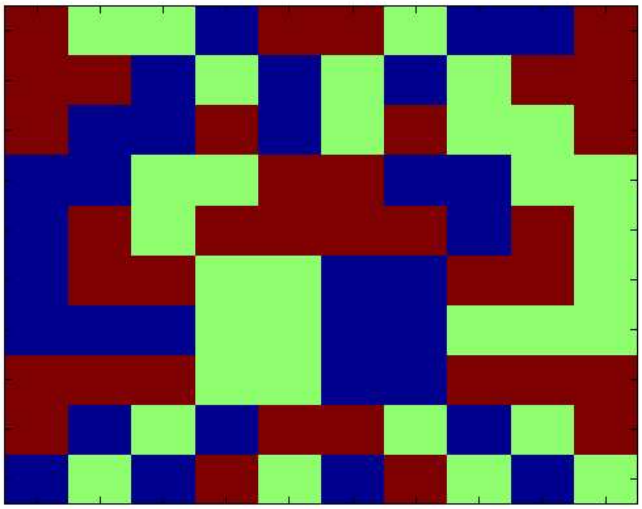

Figure 10: The final test region scaled with a 10 by 10 pixel region.The different colors indicate the magent the pendulum rests over at that intial position.

make the equations simpler to read let X and Y denote the x or y coordinate in the 500 by 500 region and the x and y by the x or y coordinate in the scaled region. Thus, the resulting equations are:

$$x = -1 + \frac{3}{500-1}(X),$$

$$y = -1 + \frac{3}{500-1}(Y).$$

The program loops through all the combinations of the xy values from 0-499 and converts each of these array entries to xy-coordinates in the test region and generates all of the scaled initial positions of the pendulum.

We can see from Figure 10 that a region discritized to a square of 10 by 10 pixels does not use enough initial points to show basins of attraction. Figure 11 uses a scaled region of 50 by 50 pixels generates a better image, which begins to resemble the desired picture. Finally we can see from Figure 12 that the scaled 500 by 500 region does generate a desirable image.

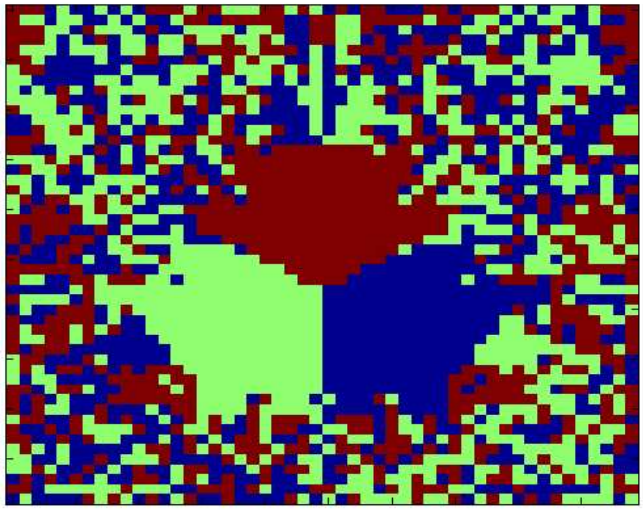

Figure 11: The final test region generated using a 50 by 50 pixel region.

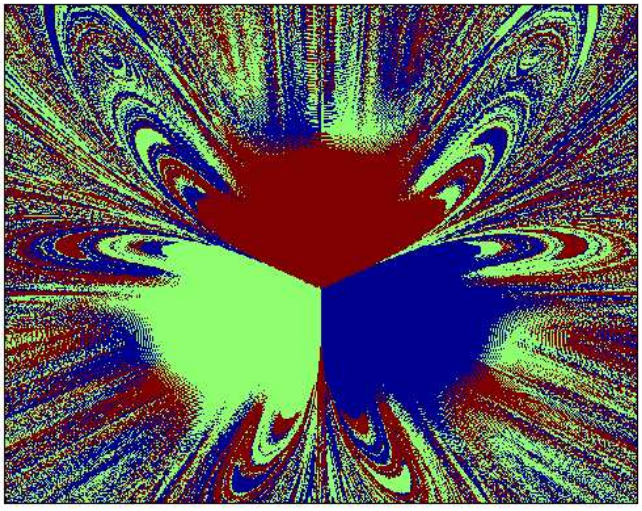

Figure 12: The final test region using the desired region of 500 by 500 pixels.

3.4.3 Parallelization

The pendulum problem running on a single processor takes over 2 hours when calculating a 500 by 500 region, making this program a good candidate for parallelization. The work is divided up so that each computer solves the system that corresponds to a particular block of initial positions in the xy-plane. Thinking of the described xy-plane as a matrix containing these initial points, we divide the data giving each computer a row of xy-coordinates. Each machine is responsible for all the rows whose identifying number is zero modulo the rank of the machine. Thus, each machine will go through the appropriate calculations of its respective rows and place them in the appropriate entry in its local solution matrix. Then, after each machine finishes its allotted work, it will send the information back to the root processor. The root processor then augments its solution matrix with the valid rows coming from each machine until it has data from all of the computers that were active in the system. The root processor should have the completed solution matrix and, using ImageMagick (a graphics library), will generate the final picture.

This parallelization is done using simple send and receive commands, where the root processor communicates individually with each machine. A technique which might have lessened the run time would be to implement a tree to collect the data, namely using the reduce command. This particular tree would be the reverse of the diagram in Figure 6. The problem with using the reduce command is that it requires an operation definition. This operation indicates what the root processor should do with the data. There are some set operations such as addition, subtraction,and multiplication, which lend themselves easily to this function. However, this particular program requires modular addition, which is problematic because the instructions on how to define a new operation are rather ambiguous. MPI has the capacity for the user to define their own kind of operations for the reduction command, but the proper implementation of this tool was not fully defined in the MPI references.

When the program is ultimately tested on different numbers of machines, the total run time of the program is lessened with each additional processor. The problem seems to scale linearly, which essentially means that this method of send and receive is very close to the maximum speed-up possible.

3.5 Runtime Analysis of the Pendulum Problem

In this section we will examine the effectiveness of the parallelization of the pendulum program. As a side note, for conducting multiple test runs of a program on different numbers of processors, it is very helpful to set up a system so that the programs will run, one after another, during a time when the computers are not going to be used by other people. If other users are utilizing the network when the parallel program is being executed, the runtime has a high probability of being affected, giving misleading timing results. Refer to Appendix B for instruction on how to set up this sort of queueing system.

Included in the MPI package is the `Wtime` command, which allows the programmer to print out the runtime of the program within the region that the MPI library has been initialized up until the finalization of the MPI commands. This command is helpful in testing if the parallelized portion is working as expected. However, to test the total runtime of the program, it is necessary to use the C command `ftime`. Each machine has a slightly different runtime. The root processor's runtime indicates the most accurate duration of time, so for the purposes of this paper, the runtime indicated for a given number a processors is that of the root processor.

We conducted three different test sets, each using a different sized region. During each test set, the region size was held constant while the program was tested on 1 machine, then incremented by one up to a network using 15 machines. The test regions were scaled to 10 by 10, 200 by 200, and 500 by 500 pixel regions. The purpose of looking at the timing results on different sized regions yields insight on how the amount of calculations connects to the effectiveness of the parallelization.

The 10 by 10 scaled pixel region showed only moderate improvements as the number of processors increased. In reference to Table 1 we can see that there is an obvious advantage between running the program on 1 processor versus 15. However, note that the time for 7 processors is more than for 6 as is the time for 12 more than for 11 processors. This indicates that more machines does not necessarily guarantee a faster execution time. This may be a result of the Runge-Kutta method which can take substantially more iterations if the initial positions are near the outer boundaries. These longer runs may be a result of the particular division of calculations such that one machine has a high proportion of systems of equations that require more iterations to arrive at a solution.

Table 1: A 10 by 10 scaled region timing results for the root processor.

Number of Processors	Seconds
1	3.452201
2	1.723234
3	1.305459
4	1.053806
5	0.759871
6	0.769884
7	0.780218
8	0.745822
9	0.702356
10	0.487811
11	0.396756
12	0.555738
13	0.399128
14	0.386520
15	0.388152

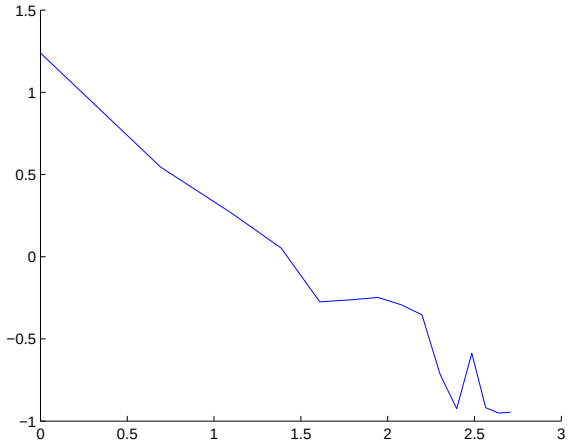

Figure 13: This figure shows the logarithmic plot of the timing results in seconds versus the number or processors for a 10 by 10 pixel region.

To better visualize the table of timing results, it is standard to plot the logarithms of the number of processors versus the logarithms of the timings. In Figure 13 we can see that the plot is fairly linear with discrepancies that correspond with the jumps in the timing results of the previously mentioned table.

The set of test runs which used a 200 by 200 scaled pixel region again indicates that the program is more efficient when run on multiple processors. Due to the increased number of initial positions for the pendulum, the work appears to be consistently distributed. Here, we do not see the problem with the smaller region, where some results were slightly slower with an additional processor. Table 2 shows the specific results of the test runs.

Again, we can refer to the plot of the logarithms of the timing results compared to the number of processors and analyze if the program scales linearly. Figure 14 appears to be linear in nature and substantially smoother than that of the smaller region. This indicates that this size region is a better candidate for parallelization.

To see if scaling the problem to an even larger region increases the effectiveness of parallelization, we tested the 500 by 500 pixel region. As expected, this region scaled even better than the two previously tested regions. Table 2

Table 2: A 200 by 200 scaled region timing results for the root processor.

Number of Processors	Seconds
1	1375.817739
2	746.746972
3	470.685654
4	357.068953
5	275.911337
6	236.557786
7	199.380026
8	177.240036
9	159.956170
10	139.583577
11	136.681368
12	124.087476
13	116.908724
14	111.135736
15	96.719956

Table 3: A 500 by 500 scaled region timing results for the root processor.

Number of Processors	Seconds
1	8636.790243
2	4328.266205
3	2929.253599
4	2178.425224
5	1779.152881
6	1529.094241
7	1299.484267
8	1144.925402
9	1036.194001
10	937.629003
11	828.577152
12	774.330686
13	728.553265
14	669.415217
15	622.937438

corroborates this assumption.

Figure 15 has a very smooth line which indicates that the program scaled linearly. The program when run on 15 processors is approximately 14 times faster than the program run on only one machine. The next section on Amdahl's Law will give an in-depth analysis of the effectiveness of the parallelization as well as specific run time data and timing results.

4 Amdahl's Law

Gene Amdahl presented his theory on predicting optimal parallelization of programs in 1967. Amdahl was a computer programmer for IBM, whose theory states that there are inherent limits to the maximum speed that any parallelized program can achieve. The main idea behind his theory is that there are a limited number of commands in any given program that can be parallelized. Thus, there are some commands that must be run on a single machine and will

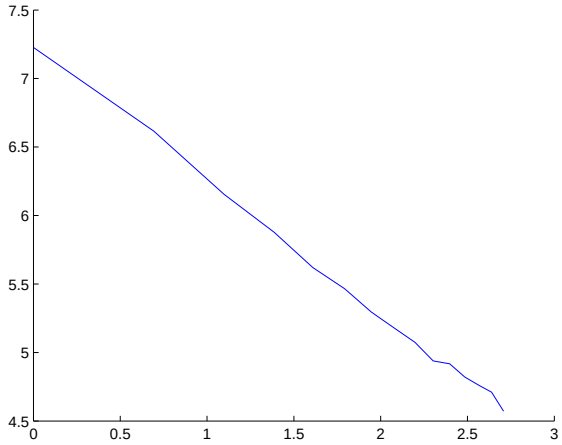

Figure 14: This figure shows the logarithmic plot of the timing results in seconds versus the number of processors for a 200 by 200 pixel region.

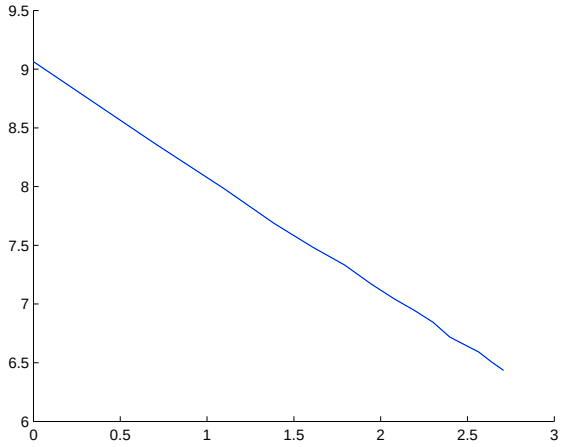

Figure 15: This figure shows the logarithmic plot of the timing results in seconds versus the number of processors for a 500 by 500 pixel region.

take a specific, unchangeable amount of time. Amdahl's Law states that

$$\text{Speedup} = \frac{t_s + t_p(1)}{t_s + \frac{t_p(1)}{P}}, \tag{4}$$

where t_s is the time of one processor to run the serial portion of the program, $t_p(1)$ is the time for one machine to run the parallelizable portion of the program, and P is the number of processors (Amdahl, 483-485). This yields the most efficient speed up factor for the given program. This formula tells us that a program can have a maximum speed of factor of P if the entire program can be done in parallel. However, few programs are entirely parallelizable, thus the speed up factor tends to be less than the number of processors. In the case of pendulum program for a 500 by 500 region, t_s was 3.304753 seconds and $t_p(1)$ was 8633.48549 seconds. Solving this for 15 processors yields a speed up factor of 14.920074445. The observed speed up factor when running the program was about 13.86, which is in the ballpark of the theoretical speed up factor. Calculating the speed up factor for more machines appears to show that there is a point when using more machines seems to be noticeably distant from the most efficient speed. The graph of the perfect speed up factor versus Amdahl's predicted rate from 1 to 100 processors is show in Figure 16. We can see in the figure, that at about 30 processors, Amdahl's predicted speed becomes farther away from the perfectly optimized line.

Using Amdahl's Law we can compare the graph of the speed up factor for 1 through 15 processors of the observed and the predicted optimization of the pendulum program, shown in Figure 17. To see how the observed, the predicted, and perfect rate compare on 15 processors refer to Figure 18. In this figure the upper line is the perfect speed up factor of 15, the lower line is Amdahl's predicted speed up rate, and the star is the actual speed up factor. This shows that there seems to be room for improvement in the code. Attempts to implement changes to make the program more efficient, proved problematic. Another important consideration: if the time it would take to create the parallelized program is substantially longer than simply using the existing program, maybe the effort to parallelize the program is not warranted.

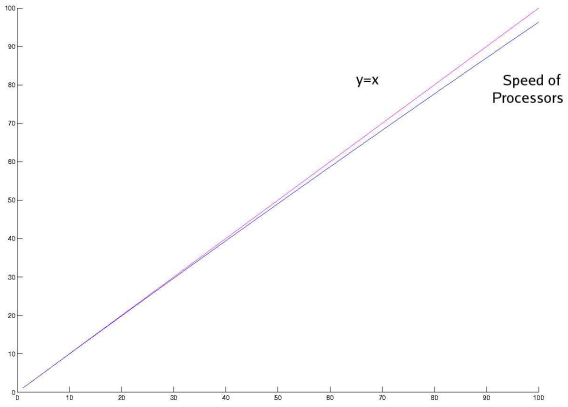

Figure 16: This shows the perfect speed up factor compared to Amdahl's predicted speed up factor up to 100 processors versus the line $y = x$. We can see that as the number of processors increases the efficiency falls further away from the line $y = x$.

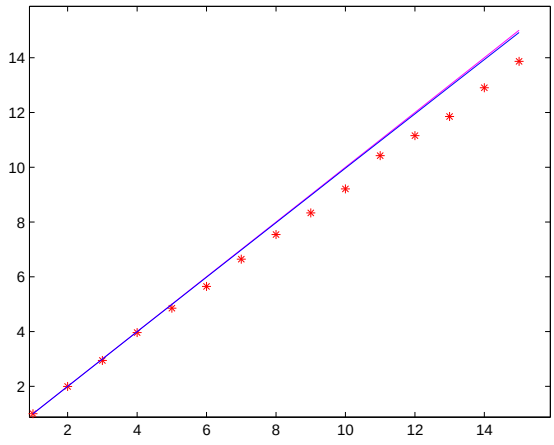

Figure 17: Here the speed up factors are plotted against the number of processors. The stars represent the observed speed up factors. The upper line is the perfect speed up factor, and the lower line is Amdahl's predicted speed up factors.

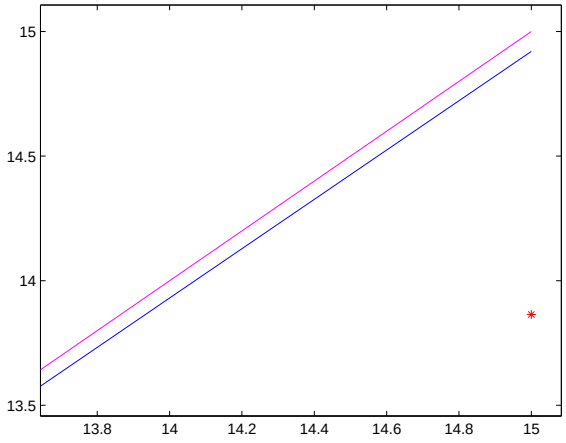

Figure 18: This shows the same graph as Figure 17 zoomed in close to the rates near 15 processors. It is easier to see that Amdahl's prediction is slightly less than the perfect speed up factor and greater than the observed speed up factor.

5 Conclusion

The example problems examined in this paper show that not every problem benefits from parallelization. Issues such as relatively small calculations and costly communication make the problems such as calculating π and $f(x) = x^2$ poor candidates for parallel programming, as observed in Section 3. The pendulum problem is a good candidate for parallelization due to the large number of calculations which benefit from simultaneous usage of different processors. As stated before, the pendulum problem could possibly benefit from the use of the reduction tree, which would require the creation of a new operand command.

All these problems have a similar structure, in that each calculation uses the same equation with a different variable depending on the rank of the respective processor. These kinds of problems are considered *embarrassingly parallel problems* because the segmentation of the problem is obvious. Also, each of these example problems were well-balanced in terms of the load of work each processor was responsible for completing. Thus, an interesting extension of this research would be to examine problems with conditional statements which would leave certain processors temporarily idle during the execution of the program. To

maximize efficiency it might be necessary to distribute certain calculations to some processors and other calculations to other machines to avoid long periods of idle machines. Also, looking at problems where certain calculations are necessary before starting the next set of calculations could be a challenging task. This might allow the programmer to utilize the MPI functions which tag certain information as more important than others, which would create hierarchical queues within the program. These are just a few examples of programs that require more advanced applications of parallelization.

Appendix A: Remote Login Issues

One particular issue that was encountered while running programs on the Whitman MathLab concerns remote logins. This problem common to parallel programs running on secure networks. In order to access a machine, it is necessary to enter a password. To avoid having to enter a password into every computer every time the program was run, we used the command `ssh add` so that for each login, the passwords for each computer needed to be entered only once. However, when we ran larger problems it became helpful to run the programs over night, so we needed a new way to get around the password problem.

The following portion of this sub-section shows how to use the Secure Shell protocol to enable password-less logins between machines in the MathLab. Once you have completed these instructions, if you are successfully logged into one machine, you can login to any other MathLab machine without having to type a password. You may also use ssh to start programs automatically on other machines. This is useful when running parallelized programs written with the MPI parallelization library.

All of the commands mentioned in what follows are to be executed from a command-line terminal. The `(math)∼>` represents a command prompt.

1. Create a password-less ssh keypair, with the command:

   ```
   (math)∼> ssh-keygen -t dsa
   ```

   The following output will be generated:

   ```
   Generating public/private dsa key pair.
   ```

   Simply hit enter at the following prompt:

`Enter file in which to save the key (/home/schuelaw/.ssh/id_dsa):`

Hit enter here:
`Enter passphrase (empty for no passphrase):`

and here:
`Enter same passphrase again:`

The following output will be generated:
```
Your identification has been saved in /home/schuelaw/.ssh/id_dsa.
Your public key has been saved in /home/schuelaw/.ssh/id_dsa.pub.
The key fingerprint is:
cf:5b:bc:97:9c:4e:b6:ad:b9:7e:0e:7e:7f:8e:ab:a8 schuelaw@math.whitman.edu
```

2. Add the contents of the file ∼`/.ssh/id_dsa.pub`, to the file ∼`/.ssh/authorized_keys2` with the following commands:

   `(math)∼> mkdir .ssh` (ignore errors, if `.ssh` already exists)
   `(math)∼> cd .ssh`
   `(math)∼/.ssh> cat id_dsa.pub >> authorized_keys2`
   `(math)∼/.ssh> cd`

3. Test the capability by attempting to login to a different machine:

   `(math)∼> ssh sloth`

   You will know if you are successful if no password is requested and the prompt changes to:

   `(sloth)∼>` (note "`sloth`" instead of "`math`")

This method of avoiding the entering of a password each time we want to access another computer allows us to be able to run long strings of programs without having to be present when the programs are executing. Also when we are concerned with the runtime of a program, we do not want to consider the speed at which the user can input passwords.

Appendix B: Queuing

In order to execute multiple runs of a program at a set time it is necessary to set up a queue. This is done by first writing a script file entitled `batch.sh`, the first line of which should read: `!/bin/sh`. Each subsequent line is a command that the program would type at the prompt of a terminal when executing the program. To indicate what time the programs will begin execution, in the terminal, type `at -f batch.sh HH:MM` where `HH` and `MM` indicate the hours and minutes respectively of a 24-hour clock system. To check to see what jobs are set up, type `atq` in the prompt, and to delete jobs, type `atrm` followed by the number associated with that job.

References

[1] Amdahl, G.M. Validity of single-processor approach to achieving large-scale computing capability, *Proceedings of the AFIPS Conference*. Reston, VA. 1967. pp. 483-485.

[2] Cheney, Ward and Kincaid, David. Numerical Analysis: mathematics of scientific computing. Pacific Grove, CA: Brooks/Cole Publishing Company, 1991.

[3] Gustafson, J.L. Reevaluating Amdahl's Law, CACM, 31(5), 1998. pp. 532-533.

[4] Pacheco, Peter S. Parallel Programming with MPI. San Francisco, C: Morgan Kaufmann Publishers, Inc, 1997.

[5] Peitgen, Heinz-Otto. Chaos and fractals: new frontiers of science. New York: Springer-Verlag, 1992.

[6] Press, William H. Numberical Recipes in C: the art of scientific computing. New York: Cambridge University Press, 1998.