IMPLEMENTATION OF SOME PARALLEL ALGORITHMS ARISING IN SPARSE MATRIX AND OTHER APPLICATIONS

by

DAVID LASS

ROGER B. SIDJE, COMMITTEE CHAIR
DAVID HALPERN
BRENDAN AMES
BRANDON DIXON

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Arts
in the Department of Mathematics
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2017

ABSTRACT

Generally the processing time of a program can be improved by splitting the program into several portions and executing each portion on its own computing core. This process allows us to take advantage of as much of the computer system hardware as we can. However, this also can cause some complexities to arise such as dependency issues.

Sparse matrices are, in the most general form, matrices with a large number of zero entries relative to their number of nonzero entries. These matrices are commonly occurring in applications like in a Partial Differential Equation (PDE). Naturally, systems of linear equations involving these sparse matrices are common and there are several methods for solving these systems. As most of these matrices are relatively large, it can be inefficient to solve them directly. Instead many ways to solve these systems involve the use of iterative methods.

Combining parallel processing with these iterative methods, we can quickly and efficiently solve large sparse systems.

The following thesis contains the use of two types of parallel processing, provides applications of both, shows the value of using iterative methods with sparse matrices, and examines solving partial differential equations using several iterative methods with parallel processing.

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Roger Sidje, for helping me throughout the research. Without his guidance this paper would not have been possible. Dr. David Young at the Alabama Supercomputer helped me set up programs on the parallel architectures. This saved hours of time on learning how to compile and link. A good portion of credit should go to all of my previous mathematical teachers. I would not be the person I am today without their teachings.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Introduction to Parallel Computing

Historically, computer cores and memory were expensive so there was a benefit of designing algorithms to minimize their usage. Nowadays, thanks to advancements in technology, the costs have fallen dramatically. This has made parallel computing feasible. Parallelism in the most basic sense is dividing a problem into several parts, executing each of these parts independently at the same time, and merging the results to get a solution. This can greatly reduce the time required to run a program and thus can solve larger, complex problems more efficiently.

Take for example a simple array addition:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix} + \begin{bmatrix} 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

On a single processor, this would involve performing eight additions in sequence. If we were to have four processors then the array could be split into four groups of two and each processor could do two additions. The benefit is that each of the four processors can be doing additions at the same time thus reducing the total time of the computation. In this example, the speedup would be slightly less than four times because of the overhead of splitting the array and merging the pieces after the additions.

Though this example may seem trivial, this thesis will examine how parallelism can be used in a wide variety of algorithms to improve overall computation time. Specifically,

the use of iterative methods for solving linear systems with sparse matrices is considered. But first, the systems being used and a short background of parallel computing will be discussed.

1.1.1 Supercomputer and Software Used

For the purpose of this thesis, the Alabama Supercomputer (ASC) is used. This supercomputer can be thought of as a collection of several individual computers used as one. This is known as a cluster. The computers that form the cluster are sometimes called servers or nodes in the cluster. When programming for the ASC, the user has several options such as how many cores to use, how much memory to use, or which system of the supercomputer to use. For the purpose of this thesis, a majority of the programs will be executed on the Dense Memory Cluster (DMC) system of the ASC, with the exception that smaller programs can be executed directly on the login node.

The ASC can be accessed remotely via Secure Shell (SSH). This allows users to access all of the supercomputer's capabilities from any university campus across the state of Alabama. In this thesis, the PuTTY terminal is used for this SSH process. In Figure 1.1, the simple login process through PuTTY is shown.

When discussing parallel computing, there are two main types of multiprocessor computer systems: shared memory and distributed memory. They both can perform calculations in parallel but differ in hardware requirements and implementation. Both of these systems will be examined and coded in this thesis.

For all parallel programming, the FORTRAN 90 programming language will be used. This language is compatible with implementations of both shared and distributed memory.

1.1.2 Shared Memory

With shared memory, as the name implies, each of the processors can access the same block of RAM simultaneously. In this form, every loop of a do loop can be split to its own core if desired.

OpenMP is an interface that is commonly used for shared memory programming. The most basic Hello World program using OpenMP is as follows:

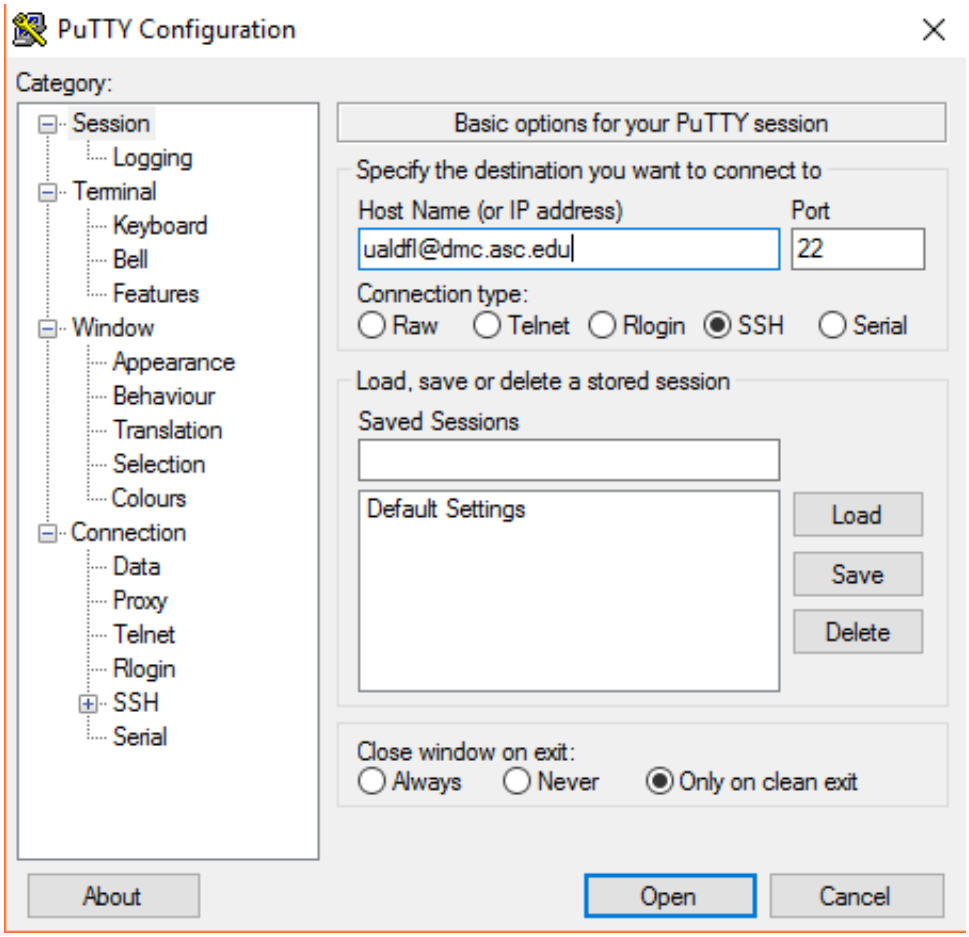

Figure 1.1: Putty Login Screen

```
program hello
    integer :: threads, id
    integer :: omp_get_num_procs, omp_get_thread_num
    threads = omp_get_num_procs()
!$OMP PARALLEL
    write(*,*) "hello from thread:", omp_get_thread_num(), "out of", threads
!$OMP END PARALLEL
    write(*,*) "Number of Threads", threads
end
```

Note the use of the directives !$OMP PARALLEL and !$OMP END PARALLEL to select the lines to run in parallel. This code must be compiled and sent to the DMC. The compilation process is done frequently in coding with a single statement on the command

line. The execution process is generally done via a script to select specific parameters. Here we show the full process required to execute a program by sending a script file through the queue. The shell script is as follows:

```
#!/bin/sh
source /opt/asn/etc/asn-bash-profiles-special/modules.sh
module load intel
module list
ifort -fopenmp -o compHi hello.f90
./compHi
```

Consider the lines of this script individually. The first line tells which interpreter to use. The following source line includes the supercomputer's path to modules. These modules are basically packages of routines and functions. The third line loads the Intel module. Intel provides the ifort compiler and the OpenMP capabilities. Though not required, the module list line shows which modules have been loaded and can be beneficial for error analysis. Lastly, the program is compiled with the -fopenmp linker which is required for ifort to process OpenMP commands, and the compiled program is run.

The first time using this file, it must be made executable with a simple chmod +x command. The executable is then sent to the queue with a run_script command. The options of queue type, number of processor cores, and memory allotment are prompted. After selecting a small queue with eight processor cores and standard memory we obtain the output in Figure 1.2:

Each of the cores says, "Hello" and then tells their processor number. Eight processors are used when running the program, and thus the processors' numbers vary from zero to seven. Note that the processors do not write to the screen in order. This is because the write statements for each of the eight processors run in parallel. Thus, in this instance, each of the processors attempt to write to the screen at once, causing a race condition. Race conditions are situations when we attempt to complete two operations at the same time but the computer hardware only allows for sequential operations. These are things we must consider when programing code to run in parallel.

```
============================================================
=====         Summary of your script job               =====
============================================================
  The script file is: openMp.sh
  The time limit is 60:00:00 HH:MM:SS.
  The target directory is: /home/ualdfl/helloTest/openMpVer
  The working directory is:  /scratch-local/ualdfl.z.67210
  The memory limit is: 4gb
  The job will start running after: 2017-01-24T15:11:23
  Job Name: z
  Virtual queue: small
  QOS: -p dmc,uv,knl --qos=small
  Constraints: --constraint=dmc
  Using  8  cores on master node  dmc49
  Node list:  dmc49
  Nodes:  dmc49 dmc49 dmc49 dmc49 dmc49 dmc49 dmc49 dmc49
  Queue submit command:
sbatch -p dmc,uv,knl --qos=small -J z --begin=2017-01-24T15:11:23 --requeue --mail-user=dflass@c
rimson.ua.edu -o z.o67210 -t 60:00:00 -N 1-1 -n 8 --mem-per-cpu=500mb --constraint=dmc
 hello from thread:           6 out of           8
 hello from thread:           5 out of           8
 hello from thread:           7 out of           8
 hello from thread:           4 out of           8
 hello from thread:           0 out of           8
 hello from thread:           3 out of           8
 hello from thread:           1 out of           8
 hello from thread:           2 out of           8
 Number of Threads          8
```

Figure 1.2: Hello World Output with OpenMP using eight cores

Now let us consider distributed memory.

1.1.3 Distributed Memory

In distributed memory, each processor has its own memory. That is, while each of the processors are connected to a hub, the memories of each processor are not connected.

Message Passing Interface (MPI) is generally the standard interface for these distributed systems and will be used in this thesis. The code for a Hello World program is different than that of the OpenMP shared memory example, despite the similar names.

```
program hello
    integer rank, id, ierror, numProc
    call MPI_INIT(ierror)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, numProc, ierror)
    call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierror)
    write(*,*) "node", id, "says Hello World"
    if(id==0) write(*,*) "There are ", numProc, "nodes"
end
```

Note that there are no directives specifying which steps are to be executed in parallel because the entire program will be essentially run in parallel. That is, the variable like ID and numProc are local to each memory. More functions are required to determine the

number of processors and give each processor an ID. The final if statement shows how we can get only one processor to do something. Each processor has a unique ID, so only one processor will execute the contents of the if statement. Again, an executable script is built to send this program to the queue.

```
#!/bin/sh
source /opt/asn/etc/asn-bash-profiles-special/modules.sh
module load openmpi
module list
mpifort -o compMpi projMpi.f90
srun compMpi
```

This script is largely the same as the script for the OpenMP version. One difference is that the module loaded is the OpenMPI module rather than the Intel module, as the program is compiled with mpifort rather than ifort. Rather than running with ./, the srun statement is used since MPI executables need a launcher to run instead of running directly.

This MPI output, seen in Figure 1.3, is similar to the output from the OpenMP version:

```
============================================================
=====         Summary of your script job               =====
============================================================
  The script file is: mpi.sh
  The time limit is 60:00:00 HH:MM:SS.
  The target directory is: /home/ualdfl/helloTest/mpiVer
  The working directory is:  /scratch-local/ualdfl.8core.66828
  The memory limit is: 4gb
  The job will start running after: 2017-01-22T15:53:21
  Job Name: 8core
  Virtual queue: small
  QOS: -p dmc,uv,knl --qos=small
  Constraints: --constraint=dmc
  Using  8  cores on master node  dmc49
  Node list:  dmc49
  Nodes:  dmc49 dmc49 dmc49 dmc49 dmc49 dmc49 dmc49 dmc49
  Queue submit command:
sbatch -p dmc,uv,knl --qos=small -J 8core --begin=2017-01-22T15:53:21 --requeue --mail-user=dflass@c
rimson.ua.edu -o 8core.o66828 -t 60:00:00 -N 1-1 -n 8 --mem-per-cpu=500mb --constraint=dmc

Currently Loaded Modules:
  1) autogen/5.11.1   3) ofed/3.4-1.0.0
  2) gcc/6.1.0        4) openmpi/1.10.2-gnu-cent-pmi2

 node           1 says Hello World
 node           7 says Hello World
 node           0 says Hello World
 There are            8  nodes
 node           6 says Hello World
 node           2 says Hello World
 node           3 says Hello World
 node           4 says Hello World
 node           5 says Hello World
```

Figure 1.3: Hello World Output with MPI using eight cores

The main difference to be seen here is that even though the final write statement telling how many processors are used is at the end, the actual write is in the middle. This is due to race conditions since we cannot specify which statements should be in parallel and which should be in series. If this is executed multiple times, the orders will change seemingly at random because of the strange behavior of the race conditions.

Both MPI and OpenMP will be used over the course of this thesis. In practice, many factors like the type of applications or hardware of the computer system would play into whether OpenMP or MPI is used.

But in order to use this parallel processing, there must be a program to run. This thesis shows several minor applications, but has a focus on solving sparse matrix systems, so an introduction to the sparse matrix formats will be discussed in the following section.

1.2 Introduction to Sparse Matrix Systems

The most basic system of linear equations can be constructed in the matrix form $Ax = b$ where A is an $m \times m$ matrix, b is an $m \times 1$ vector, and x is the $m \times 1$ solution vector.

When these systems occur naturally, often the A matrix is full of mostly zero entries. Thus the standard dense data structure for matrices consumes more space and requires more computations than necessary.

Different sparse matrix formats can minimize storage by describing the structure of the matrix while also not explicitly holding the position of the zero entries. There is not a single format that is objectively the best at minimizing storage requirement and computational rigor. In fact, the format that is best to use often depends upon the problem at hand.

The following section examines the standard matrix format next to several different sparse matrix formats.

1.2.1 Matrix Formats

Consider the standard form of a sparse matrix A:

$$\begin{bmatrix} 55 & 0 & 0 & 49 & 0 & 89 & 80 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 73 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 80 & 94 & 0 & 0 \\ 68 & 0 & 13 & 0 & 72 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \end{bmatrix}.$$

It should be noted that this matrix has 64 total entries, 50 of which are zero, and 14 of which are nonzero. Obviously the number of zeroes far exceeds the number of nonzero entries, and it is more economical to avoid storing the zeros, especially as the matrices get larger.

There are several major data structures for storing sparse matrices. Four of these sparse matrix formats will be examined in the following section.

Coordinate Form

The goal is to minimize storage space used and a simple way to do that is to remove all of the zero entries. Of course, the structure of the matrix still needs to be preserved. Thus, we consider the nonzero entries by their row and column in the the matrix. This way, we can reconstruct the original matrix if desired by starting with a matrix of all zeroes and replacing the nonzero positions by their corresponding entry. The above matrix A can be reformatted in Coordinate form (COO) as:

$$\begin{bmatrix} 7 & 11 & 5 & 55 & 80 & 68 & 72 & 13 & 73 & 49 & 89 & 80 & 9 & 94 \\ 5 & 8 & 3 & 1 & 1 & 7 & 7 & 7 & 3 & 1 & 1 & 6 & 6 & 6 \\ 5 & 6 & 7 & 1 & 7 & 1 & 5 & 3 & 3 & 4 & 6 & 5 & 2 & 6 \end{bmatrix}.$$

The first row of this matrix, aA, corresponds to the value of the entry in the dense matrix. The second row, iA, and third row, jA, correspond to the row and column

coordinates of the nonzero entry, respectively.

Note that the dense form of the matrix A had 64 entries. The Coordinate format of A had just 42 entries. In general, for a matrix B, let n be the dimension of the matrix and let nz be the number of nonzero entries. The standard storage of B requires storage for n^2 elements. The COO format requires $3 * nz$ elements of storage. Thus, in terms of storage, it is beneficial to use the Coordinate format when $n^2 > 3 * nz$.

However, the elements in Coordinate format are not necessarily ordered. This makes it more difficult to use forms of parallelism. Some other formats provide a more ordered approach while also holding the reduced storage.

Compact Row Storage/Compact Column Storage

Instead of focusing on both row and column coordinates, only one needs be stored to maintain order of a sparse matrix. Then the number of entries in each row or column must be tracked. As an example, the compact row storage (CRS) of the above matrix A is as follows:

$$\begin{bmatrix} 55 & 49 & 89 & 80 & 73 & 5 & 7 & 9 & 80 & 94 & 68 & 13 & 72 & 11 \\ 1 & 4 & 6 & 7 & 3 & 7 & 5 & 2 & 5 & 6 & 1 & 3 & 5 & 6 \\ 1 & 5 & 5 & 7 & 7 & 8 & 11 & 14 & 15 & & & & & \end{bmatrix}.$$

The entries of the standard form matrix are read left-to-right then top-to-bottom. The first row aA is composed of the ordered entries. The following row contains the column coordinate of the entry. The final row is the number of the first nonzero entry to begin the row. That is, the fifth entry would be the number of nonzero entries in the first four rows plus one. Thus, there are $n + 1$ entries of the third row, where n is the number of rows in the original matrix.

Compact Column Storage is essentially the same format in reverse. That is, the entries will be read top-to-bottom then left-to right. The row coordinate is held and the number of entries with respect to the column is the final row. Thus, the Compact Column Storage (CCS) format is as follows:

$$\begin{bmatrix} 55 & 68 & 9 & 73 & 13 & 49 & 7 & 80 & 72 & 89 & 94 & 11 & 80 & 5 \\ 1 & 7 & 6 & 3 & 7 & 1 & 5 & 6 & 7 & 1 & 6 & 8 & 1 & 3 \\ 1 & 3 & 4 & 6 & 7 & 10 & 13 & 15 & 15 & & & & & \end{bmatrix}.$$

Notice that both of these formats have fewer total entries than the Coordinate format. Also there is a level of order as CRS/CCS has all of the entries ordered by column or row. This allows some computations to be performed in parallel, a topic that will be discussed in a later section.

Jagged Diagonal

The previous three formats are three of the most common sparse matrix storage formats; however, they are not exclusive. There are several other formats. One of these is the Jagged Diagonal (JAD) form.

This form is a bit more complicated than the previous ones. Initially, the standard form matrix is permuted so that the row with most nonzero entries is on the top and the number of nonzero entries decreases in the subsequent rows. Thus with the above matrix we need to convert from

$$\begin{bmatrix} 55 & 0 & 0 & 49 & 0 & 89 & 80 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 73 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 80 & 94 & 0 & 0 \\ 68 & 0 & 13 & 0 & 72 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \end{bmatrix}$$

to

$$\begin{bmatrix} 55 & 0 & 0 & 49 & 0 & 89 & 80 & 0 \\ 0 & 9 & 0 & 0 & 80 & 94 & 0 & 0 \\ 68 & 0 & 13 & 0 & 72 & 0 & 0 & 0 \\ 0 & 0 & 73 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Then the first nonzero entry of each row is taken, followed by the subsequent entries. The column that each entry is stored in is held as well as the number of non-zeros in each diagonal. This gives us the following JAD:

$$\left[\begin{array}{cccccc|cccc|ccc|c} 55 & 9 & 68 & 73 & 7 & 11 & 49 & 80 & 13 & 5 & 89 & 94 & 72 & 80 \\ 1 & 2 & 1 & 3 & 5 & 6 & 4 & 5 & 3 & 7 & 6 & 6 & 5 & 7 \\ 1 & 7 & 11 & 14 & 15 & & & & & & & & & \end{array}\right].$$

There is a minor benefit that this method would take less storage than the other formats. This is at the expense that some computations are more complex and would be more time intensive.

This subsection has focused solely on storing the large sparse matrices in a data structure. In order for these matrices to be useful, certain computations must be made. After all, the goal is to solve systems $Ax = b$, and in order to solve this equation there must be ways to perform a variety of matrix computations. The following subsection examines some of these calculations in detail.

1.2.2 Matrix Computations with Sparse Formats

The most basic of matrix computations is a simple matrix vector multiplication. For the standard format of a matrix, the product Ax is quite simple to compute. Take the above matrix A:

$$\begin{bmatrix} 55 & 0 & 0 & 49 & 0 & 89 & 80 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 73 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 80 & 94 & 0 & 0 \\ 68 & 0 & 13 & 0 & 72 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \end{bmatrix},$$

with a simple vector x:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}.$$

The multiplication of Ax is computed by taking the entries of each row of the matrix A and multiplying each entry by the associated entry of x. This can be implemented as follows:

```
b(:) = 0
do i = 1, m
  do j = 1, n
    b(i) = b(i) + A(i, j) * x(j)
  end do
end do
```

Running through this code we get the following matrix-vector product.

$$\begin{bmatrix} 55 & 0 & 0 & 49 & 0 & 89 & 80 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 73 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 80 & 94 & 0 & 0 \\ 68 & 0 & 13 & 0 & 72 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 11 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{bmatrix} =$$

$$\begin{bmatrix} 55*1+0*2+0*3+49*4+0*5+89*6+80*7+0*8 \\ 0*1+0*2+0*3+0*4+0*5+0*6+0*7+0*8 \\ 0*1+0*2+73*3+0*4+0*5+0*6+5*7+0*8 \\ 0*1+0*2+0*3+0*4+0*5+0*6+0*7+0*8 \\ 0*1+0*2+0*3+0*4+7*5+0*6+0*7+0*8 \\ 0*1+9*2+0*3+0*4+80*5+94*6+0*7+0*8 \\ 68*1+0*2+13*3+0*4+72*5+0*6+0*7+0*8 \\ 0*1+0*2+0*3+0*4+0*5+11*6+0*7+0*8 \end{bmatrix} = \begin{bmatrix} 1345 \\ 0 \\ 254 \\ 0 \\ 35 \\ 982 \\ 467 \\ 66 \end{bmatrix}$$

This shows that in the standard form, matrix vector multiplication is a fairly simple process. The double loop in the code shows that the multiplication can be completed in order $O(n * m)$ time where n and m are the dimensions of the matrix A.

This does require a large number of unnecessary computations as the large number of zero entries is still being considered. The sparse matrix format solutions are roughly the same difficulty in terms of implementation at a lower computational complexity.

Take for example the Compact Row Storage and Compact Column Storage formats

of the above matrix A:

$$\begin{bmatrix} 55 & 49 & 89 & 80 & 73 & 5 & 7 & 9 & 80 & 94 & 68 & 13 & 72 & 11 \\ 1 & 4 & 6 & 7 & 3 & 7 & 5 & 2 & 5 & 6 & 1 & 3 & 5 & 6 \\ 1 & 5 & 5 & 7 & 7 & 8 & 11 & 14 & 15 & & & & & \end{bmatrix}$$

$$\begin{bmatrix} 55 & 68 & 9 & 73 & 13 & 49 & 7 & 80 & 72 & 89 & 94 & 11 & 80 & 5 \\ 1 & 7 & 6 & 3 & 7 & 1 & 5 & 6 & 7 & 1 & 6 & 8 & 1 & 3 \\ 1 & 3 & 4 & 6 & 7 & 10 & 13 & 15 & 15 & & & & & \end{bmatrix}$$

For CRS, we know that the second row corresponds to the specific column and the third row allows us to determine the number of entries per row. From these two things we can construct an algorithm to multiply by the x vector from above:

```
b(:) = 0
do i = 1, n
    do j = jA(i), jA(i+1)-1
        b(i) = b(i) + aA(j) * x(iA(j))
    end do
end do
```

This algorithm provides us with the following computation:

$$\begin{bmatrix} 55*1+49*4+89*6+80*7 \\ 0 \\ 73*3+5*7 \\ 0 \\ 7*5 \\ 9*2+80*5+94*6 \\ 68*1+13*3+72*5 \\ 11*6 \end{bmatrix} = \begin{bmatrix} 1345 \\ 0 \\ 254 \\ 0 \\ 35 \\ 982 \\ 467 \\ 66 \end{bmatrix}.$$

which indeed results in the same solution as the dense format implementation but

does not require the unnecessary multiplications by zero. Here, the complexity is of order $O(nz)$ where nz is the number of nonzero entries. This is because the outer i loop loops over all of the rows of the matrix and the inner loop j loops over only the non-zero entries of the specific row i.

Consider the CCS, we can easily find an algorithm that does the same multiplication:

```
b(:) = 0
do i = 1, n
    do j = iA(i), iA(i+1)-1
        b(jA(j)) = b(jA(j)) + aA(j) * x(i)
    end do
end do
```

This algorithm can be easily verified with the above to obtain the same answer, also with $O(nz)$ complexity. It should be noted that in the coding of these algorithms for the Coordinate Storage (COO), Compressed Row Storage (CRS), and Compressed Column Storage (CCS) formats, the aA portion of the matrix is a double precision real vector, while the iA and jA portions are integer vectors.

Thus, with relatively simple coding, the matrix vector multiplication can hold for the sparse matrix formats. Also using such formats reduces the number of computations required for this multiplication, which shows the value of using one of these formats. In a following chapter, we will use these storage formats while introducing our iterative methods of solving linear systems. Parallelism will show how the speed of some of these calculations can be improved without changing the underlying complexity of the algorithm at hand. Each of the above algorithms for matrix-vector multiplication in the sparse formats can be parallelized.

1.3 Partial Differential Equation

One common application of a sparse system of linear equations that will be considered throughout this thesis is the solution to PDEs. In this thesis, the solutions to two PDEs will be approximated through similar methods. In a later section, a system of linear

equations will be constructed for each equation and solved with a few different iterative methods. For now, the two equations will be introduced.

1.3.1 Equation 1

The first equation to be tested is a two dimensional convection-diffusion PDE. Solve for $u(x, y)$ in the following equation

$$-\epsilon(u_{xx} + u_{yy}) + au_x = x(1 - p^2 x)\sin(ly), 0 \leq x \leq 1, 0 \leq y \leq 1.$$

The solution on the boundary is defined as:

$$u(x, y) = (Ax^2 + Bx + C)\sin(ly)$$

where the constants are as follows:

$$A = -p, B = (1 - 2aA)/p, C = (2\epsilon A - aB)/p, p = \epsilon l^2.$$

In order to determine that our methods of solving this equation are successful, we do need to have the true solution to the PDE. This allows the comparison of the derived approximation to the true solution matrix. This equation has the actual solution:

$$u(x, y) = (Ax^2 + Bx + C)\sin(ly)$$

1.3.2 Equation 2

Next, consider creating our own PDE. Take a function

$$u(x, y) = 4x^2 y + 2xe^{2y}$$

Then find a few partial derivatives:

$$u_x = 8xy + 2e^{2y}; u_{xx} = 8y$$

$$u_y = 4x^2 + 4xe^{2y}; u_{yy} = 8xe2y$$

From these, we can build a PDE quite easily. Consider from $0 \leq x \leq 1; 0 \leq y \leq 1$:

$$u_{xx} + 3u_y - u_{yy} = 8y + 4x(3x + e^{2y})$$

With the solution on the boundary:

$$u(0, y) = 0; u(1, y) = 4y + 2e^{2y}; u(x, 0) = 2x; u(x, 1) = 4x^2 + 2xe^2$$

The true solution is $u(x, y)$ above. So, we would like our methods to have approximations near that equation.

Over the course of this thesis, it will be shown how to set these PDEs up in the form of a sparse matrix system of linear equations and how to solve those equations iteratively using parallel processing.

2 PARALLEL PROCESSING BASIC APPLICATIONS

Before considering the more complex PDE system, some simple applications can be considered to further examine parallel processing. In this chapter, two algorithms will be shown and programmed, one with MPI and the other with OpenMP. These examples will demonstrate the effectiveness of parallel processing and provide more background.

2.1 Sieve of Eratosthenes Algorithm

The first program to be considered is the Sieve of Eratosthenes Algorithm. The purpose of this sieve is to find a list of prime numbers below a certain limit. For the purpose of this exercise, we will determine the number of prime numbers under a certain value. For an example use the number twenty. Below twenty, we have the prime numbers 2, 3, 5, 7, 11, 13, 17, and 19. So for an input of twenty, we would like for the program to return 8, corresponding to the eight primes below twenty. Here, we will focus specifically on using distributed memory, MPI. First, let us consider the various approaches to determine the number of primes.

2.1.1 Approaches

Consider the most naive algorithm to determine the number of primes. We can construct an array of every number from two up to the desired limit. Then, starting with three, we can test if the number is divisible by any of the numbers below it. That is for three, we would check if it is divisible by two, for four we would check if it is divisible by two or three, and so on in that manner. If a number is not divisible by any number below it, then it is a prime number. Though very simple to code, this algorithm would have a dreadful complexity in time. For each number up to the limit, we would need to loop through all of the numbers below it.

The algorithm can be improved by holding a list of prime numbers during the computation. The tested number will only check divisibility against this list. Also, only the primes up to the square root of the tested number are considered. Consider the number 53. If none of the prime numbers below or equal to the square root of 53 are factors of 53, then it must be prime. This can be easily proven by contradiction. Now, the number 53 is only tested against 2, 3, 5, and 7 instead of being tested against every number from 2 to 52. Since 53 is not divisible by any element of this shorter list, and the next prime 11 is above the square root of 53, 53 must be a prime number. In this improvement, since we tested that 53 was not divisible by 2, we know that it will also not be divisible by any multiples of 2, and thus we will not test against those. By only considering the primes up to the square root of the target number, we avoid unnecessary computations.

With that improvement, we do fewer operations with each number; however, we still have to test each number from three up to the limit against the list of primes to determine if it is prime itself. For example, despite 121 obviously not being prime as the square of 11, this method would still test 121 against all of the primes under 11. Another algorithm completely changes the approach to avoid having to test each number individually. This algorithm involves the use of a sieve. A sieve involves taking a list and marking or removing elements systematically. Here, we consider a list from two up to our limit number. Then, starting at the beginning, each of the multiples of a prime number are crossed off or removed from the list. Consider the list of numbers from two to thirty-three:

2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33

First, we take the number 2 and remove all of the multiples of 2 from the list:

2,3,~~4~~,5,~~6~~,7,~~8~~,9,~~10~~,11,~~12~~,13,~~14~~,15,~~16~~,17,~~18~~,19,~~20~~,21,~~22~~,23,~~24~~,25,~~26~~,27,~~28~~,29,~~30~~,31,~~32~~,33

The next number of the list is 3. We remove all of the multiples of 3.

2,3,~~4~~,5,~~6~~,7,~~8~~,~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19,~~20~~,~~21~~,~~22~~,23,~~24~~,25,~~26~~,~~27~~,~~28~~,29,~~30~~,31,~~32~~,~~33~~

And next, we remove the multiples of the next number 5.

2,3,~~4~~,5,~~6~~,7,~~8~~,~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~,~~16~~,17,~~18~~,19,~~20~~,~~21~~,~~22~~,23,~~24~~,~~25~~,~~26~~,~~27~~,~~28~~,29,~~30~~,31,~~32~~,~~33~~

This process is continued until the multiples of each prime number up to the square

root of the largest number of the list are removed.

2,3,5,7,11,13,17,19,23,29,31

All that remains is a list of primes. In this situation, we only have to consider the prime numbers and remove their multiples. We do not have to individually consider every number. This decreases the time complexity.

Since this implementation (known as the Sieve of Eratosthenes) is the most efficient, it will be the method parallelized. Code was adapted from [6]. The following section shows and explains the parallel coding of this algorithm in detail.

2.1.2 Coding

The unabridged code for the parallel sieve algorithm can be seen in the appendix in section A.1. The following is a description of what is happening in each section of the program.

This program is split into several steps for clarity. The steps are split by a row of exclamation points. Consider the first portion. The program is defined, the MPI module is loaded, and some variables and parameters are initialized. These variables include the processor ID number, the number of processors used, block controlling variables, and an array of numbers. For some variable N, N is the limit to determine how many primes are less than or equal to N. Here it is set to ten million.

The next section does basic MPI initialization. It provides each processor with an ID and defines a master node. The number of processors and range are output merely for the benefit of the program user. This brings us to splitting by processor blocks.

First, since only the number two and odd numbers can be prime, we automatically can remove all of the even numbers. Next, it benefits us to split the processor blocks in two and hold the top and bottom in the same processor. If all of the lowest numbers are on a single processor, then once all of those numbers are processed the processor would be sitting completely idle. By splitting the top and bottom numbers on the same processor, we prevent idle processors. Consider the limit number of 105, the prime array processor will be split as follows:

Block 0: 1 3 5 7 9 11 13 — 95 97 99 101 103 105

Block 1: 15 17 19 21 23 25 27 — 83 85 87 89 91 93

Block 2: 29 31 33 35 37 39 41 — 71 73 75 77 79 81

Block 3: 43 45 47 49 51 53 55 — 57 59 61 63 65 67 69

Of course, we don't actually need to store these numbers, as they can be determined by the indexing of the array. So, arrays of this length are constructed and filled with ones indicating prime numbers. Immediately, the index corresponding to the number one is zeroed, as one is itself not a prime number.

Then we have a looping process. Starting with the first processor, we find the lowest unused prime number and broadcast it to the other processors. The processors calculate the indexes of the multiples of that prime number and zero the value of the prime array at those indexes.

Take for example the first loop using the above split. The number 3 is the first unused prime, so the indexes 5, 1, 3, and 2 are calculated to have multiples of 3 on processors 0, 1, 2, and 3 respectively. These indexes are correct as they correspond to the numbers 9, 15, 33, and 45, all of which are divisible by 3. Then a loop will zero the prime array indexes for all multiples of three on each processor. A similar indexing routine is used for the top half of the array. So after the first iteration of the do loop, the arrays will look as follows:

Block 0: 0 1 1 1 0 1 1 — 1 1 0 1 1 0

Block 1: 0 1 1 0 1 1 0 — 1 1 0 1 1 0

Block 2: 1 1 0 1 1 0 1 — 1 1 0 1 1 0

Block 3: 1 0 1 1 0 1 1 — 0 1 1 0 1 1 0

Corresponding to the numbers:

Block 0: ~~1~~ 3 5 7 ~~9~~ 11 13 — 95 97 ~~99~~ 101 103 ~~105~~

Block 1: ~~15~~ 17 19 ~~21~~ 23 25 ~~27~~ — 83 85 ~~87~~ 89 91 ~~93~~

Block 2: 29 31 ~~33~~ 35 37 ~~39~~ 41 — 71 73 ~~75~~ 77 79 ~~81~~

Block 3: 43 ~~45~~ 47 49 ~~51~~ 53 55 — ~~57~~ 59 61 ~~63~~ 65 67 ~~69~~

Then the subsequent primes of the master node are considered, until all the primes are looped, then the next node is declared master and the process is repeated.

Once all of the non-prime numbers are zeroed from the array, we just need to determine how many ones remain. A quick loop counts up all of the ones on each processor and an MPI_Reduce call adds all of the primes from each processor. Then that number is incremented by one (to consider the prime number 2) and the result is displayed.

The effect of multiple processors with this code is examined in the following section.

2.1.3 Speed Benefit of Parallelism

In order to aptly compare the speeds, we need to consider a wide variety of inputs. We need to not only consider different number of cores, we should also consider larger and smaller problems. For example, we can consider the timing of this program on 1, 2, 4, or 8 cores with primes under one million, ten million, or one hundred million. Table 2.1 shows the impact of size or program and number of cores on the time required to run the Sieve of Eratosthenes algorithm.

	Sieve Algorithm Timing			
Prime Limit	1 core	2 core	4 core	8 core
1,000,000	1.16E-2	5.49E-3	3.44E-3	2.21E-3
10,000,000	0.39841	0.256503	0.110714	6.63E-2
100,000,000	6.92004	2.609132	1.291691	0.767671

Table 2.1: Sieve Program Timing by Number of Cores

Obviously increasing the prime limit does increase the time required to run the program. In fact, it does seem to take about ten times longer when the limit is multiplied by a factor of ten. This tells us that the scalability of the problem is not great. It would be fairly unrealistic to use this program to determine the number of primes under a very large (think 1E20) number, even with a large number of cores.

The average speed does seem to roughly double when the number of cores is doubled. This shows the power of parallelism. The change from one core to two cores is generally greater than the difference from increasing cores above two. This makes sense, as we would expect each additional core to add additional overhead.

2.2 Merge Sort Parallelized

The second program to be considered is the Mergesort Algorithm. Sorting an array of numbers is one of the most common routines in existence. It improves the ease of searching for a specific element, allows for more ease in pattern discovery or statistics, and improves our ability to compare two distinct sets. As such, it is a pretty heavily studied practice. The basic problem is to start with some unordered list and derive a list in proper numeric order. For example take the list (24, 39, 85, 44, 73, 63, 19, 61). The goal would be to reorder the numbers as follows (19, 24, 39, 44, 61, 63, 73, 85). For this example, we will program specifically using shared memory, OpenMP, and contrast this with MPI. First we consider a variety of approaches to sort an unordered list.

2.2.1 Approach

The most naive and basic approach is to scan the array for the minimum element and move it to the beginning then repeat the process on the remainder of the array. This process known as Selection Sort is very slow because each element is compared to every element after it in the array. The first step would be to loop through the array searching for the lowest number. Once that number is found it is swapped with the first element of the array and the process is repeated on the remainder of the array. Take for example the array above. After looping the array once, the number 19 is determined to be the lowest and is swapped with 24 to get:
(19, 39, 85, 44, 73, 63, 24, 61)
Next, the last 7 elements are looped and 24 is the lowest so it is swapped with 39 to get:
(19, 24, 85, 44, 73, 63, 39, 61)
Obviously, this sort could be improved by not having to repeatedly loop an array to find the minimum.

Consider a different algorithm Mergesort. This involves taking the array and successively splitting it until there is a single element in each portion. Then two portions will be merged several times in succession. First each portion will have one element and only a simple swapping procedure needs to take place if the elements are not ordered. When the portions have more than one element apiece, the lower of each portion is compared

and the lesser of those two is placed first. Then, more comparisons place each subsequent element.

Take the above array. It can be divided into eight sets of one element apiece:
(24), (39), (85), (44), (73), (63), (19), (61)
The first merge requires just four comparisons to order these sections:
(24, 39), (44, 85), (63, 73), (19, 61)
Next, each section is merged with the adjacent section by comparing the smallest value of each.
(24, 39, 44, 85), (19, 61, 63, 73)
The last merge requires a maximum of seven comparisons. This shows that the whole array requires only a small fraction of the number of comparisons of Selection Sort to properly sort.

Mergesort is asymptotically the fastest of the comparison-based sorting algorithms. [3] There are some algorithms (like Counting Sort or Radix Sort) that work more efficiently than Mergesort. These algorithms are not comparison-based and require some assumptions about the data.

Bucket Sort for example is a sorting algorithm that places each number into a specific bin, but requires that the keys to be sorted are uniformly distributed. Radix sort on the other hand bases its sort on the digits in a number. When sorting with fixed length integers, this is fine, but real numbers or large number disparities can hurt its operation.

Quicksort is another comparison-based algorithm that can have the same time complexity of Mergesort. The idea of Quicksort is that a pivot is selected and the values less than the pivot are moved left, while those greater are moved right. Then a pivot is selected from each sub-array recursively. However, for traditional Quicksort, there are pathologically bad arrays for which the time complexity is worse than Mergesort. To overcome this a medians of medians process is required, which could potentially detract from the advantages of Quicksort.

Since Mergesort does not require any of these assumptions or considerations, it will be the sorting algorithm used in this example.

2.2.2 Coding

The OpenMP implementation coding of this algorithm can be found in the appendix in Section A.2. If there are a power of two processors, we can maximize the effectiveness of the parallelism. Therefore, for the purpose of this thesis, a power of two processors will be assumed.

First, after all of the variable declarations and initializations, a large random array is constructed with integer keys with values from zero to one thousand. This will be the array to be sorted. Since we are dealing with shared memory in this example, there is a single copy of this array that is accessible from every core. This contrasts the above example where each processor had a unique array that could not be accessed by the other processors.

The number of processors is obtained with an OpenMP function, and then the random array is split into one equally sized portion per core. Each of these portions call Mergesort in parallel. Let us examine the recursive Mergesort routine.

This routine takes in an array and two indexes. If the indexes are separated by less than two, then the routine is terminated, otherwise the routine is called recursively on the first half and second half of the array before calling the Merge function. Thus, this function will be recursively called until each portion of the array has a single element, then merges the elements correctly. Consider the Merge function.

The Merge function has a dummy array to hold and swap elements of the initial random array. It requires three input indexes, the starting index of the first portion, the starting index of the second portion, and the ending index of the second portion. It loops until both portions of the array are sorted through. If one portion is empty, the other portion is dumped. Otherwise the lesser of the two entries is removed and placed in sorted position. That is, consider the portions (3, 5 ,6) and (4, 9, 11). First the least of the two portions are compared. Since 3 is smaller, we can move the iterator of the first to 5, and have the following:

Sorted (3)

Portions (5, 6), (4, 9, 11)

Next the 4 and 5 are compared and the lesser is added to the sorted list. In the following two loops, 5 and 6 are less than nine, so they are added to the sorted portion. This leaves the first list empty and we have:

Sorted (3, 4, 5, 6)

Portions (), (9, 11)

Since one of the portions is empty, we know that the other portion is already fully sorted and can directly dump it onto the sorted list.

Once all of the processors successfully Mergesort their own portion of the array, we can simply call the Merge routine some number of times to sort the array between the processors.

If there are only two processors, we have two distinct sorted sets and only need to merge once. If there are four processors, we have four sets and need to merge twice to get to two sorted sets and once more for the final set. The do-while loop handles this process for all powers of two.

Then once the proper number of merges takes place, the array is fully sorted. The value of using parallel processing on array sorting is examined in the next section.

2.2.3 Programming Timing

As with the timing of the sieve algorithm, it is beneficial to test Mergesort against a variety of problems. The following table 2.2 shows the functionality of the Mergesort algorithm with one, two, four, and eight cores on unsorted arrays of size 2,048, 32,768, and 131,072. As expected, increasing the number of cores improves the computing time

	Mergesort Algorithm Timing			
Array Size	1 core	2 core	4 core	8 core
131,072	0.021878	0.013304	0.007587	0.007165
524,288	0.137901	0.095384	0.049597	0.034344
1,048,576	0.32725	0.15572	0.09464	0.06328

Table 2.2: Mergesort Timing by Number of Cores

significantly. The array size has an expected result as well. An increase in array size causes an increase in timing. Unlike the Sieve algorithm from the previous section, the Mergesort algorithm seems to scale fairly well. Larger problems do increase time, but

doubling the array size less than doubles the computation time.

3 ITERATIVE SYSTEMS TO SOLVE SPARSE MATRIX EQUATIONS

Now that we have seen the uses of sparse matrices and have a general understanding of the value and implementation of parallel processing, we can consider applications of parallel processing on solving linear systems with sparse matrices. We have already shown that the matrix vector product Ax is fairly simple to calculate when A is sparse. Now let's, consider a system of linear equations $Ax = b$ where A and b are known and the goal is to solve for x. In basic linear algebra classes, the approach would be to find the inverse of A, A^{-1}, and solve for x directly with $x = A^{-1}b$. While this process is fine for small matrices, it is a tedious and computationally heavy process for mid-sized matrices. For the large sparse matrices present in applications like PDEs, it is completely unreasonable.

This leads us to other approaches. There are some direct approaches to solve the system, notably using matrix decomposition. The inverse of a matrix is virtually never found in practice. Rather, a process such as Gaussian Elimination is used to directly compute the action of the inverse simply. Instead, in this thesis we consider solving sparse systems with the use of iterative methods. An iterative method is a method that takes an initial guess of the solution and goes through a process to improve that guess. While the first few guesses may be wildly inaccurate, over time they should improve and hopefully converge to the true solution. There are several types of these iterative methods. For the purpose of this thesis we will focus on a select few.

3.1 Splitting methods

There are several classes of iterative methods, perhaps the most simple being splitting methods. We discuss three fundamental methods in this class, the Jacobi method, the Gauss-Seidel Method, and the SOR. All of these are related in the method setup.

Consider the system $Ax = b$. We can split the matrix A into two matrices M and N

such that $A = M - N$. Substituting this into the system, we have

$$Ax = (M - N)x = Mx - Nx = b$$

Consider an initial guess $x^{(0)}$. By moving the Nx term to the right hand side and left multiplying by M^{-1}, we can derive an iterative system:

$$x^{(i)} = M^{-1}[Nx^{(i-1)} + b]$$

Note that we are willing to take the inverse M^{-1} but not A^{-1} is because we can choose M and N to make operating with the inverse M^{-1} very simple. In practice, the inverse isn't even taken for most choices of M. The matrix M is merely chosen in a way that a simple method like back substitution can be used. The difference in the three splitting methods are in how M and N are selected. Before looking at the specific method, let's consider the convergence of this scheme.

3.1.1 Convergence of Splitting Method

First, we know that for the actual solution x, we have $Ax = (M-N)x = Mx-Nx = b$. This gives us that $x = M^{-1}Nx + M^{-1}b$. Subtracting the iterative step from this gives us:

$$x = M^{-1}Nx + M^{-1}b$$

$$(-)x^{k+1} = M^{-1}Nx^{(k)} + M^{-1}b$$

$$x - x^{k+1} = M^{-1}N(x - x^{(k)})$$

Let $e^{(k+1)}$ be the error at iteration $k + 1$.

$$e^{(k+1)} = M^{-1}N(e^{(k)})$$

$$= (M^{-1}N)(M^{-1}N)(e^{(k-1)})$$

$$= (M^{-1}N)^{k+1}(e^{(0)})$$

In order for the limit of the error to approach zero, the norm $||M^{-1}N||$ must be less than 1. That is, when the spectral radius $\rho(M^{-1}N) < 1$ we have convergence of the splitting method. Now, with this in mind, we can check out the common choices for the M/N split.

3.1.2 Jacobi Method

With the goal of finding an easy M to operate with the inverse, the most obvious selection is a diagonal matrix. For the Jacobi method, the matrix A is split into $A = M - N$ where M is the diagonal and N is the negative of the off-diagonal.

The steps of the Jacobi Method are fairly simple after this split. First take the initial estimate vector

$$x^{(0)} = guess.$$

Then loop over the size of the matrix $i = 1 : n$

$$x^{(k+1)}(i) = \left(b - \sum_{j=1, j \neq i}^{n} a_{ij} x(j)^{(k)} \right) / a_{ii}$$

With sparse matrices, this code is fairly simple to implement and parallelize. Let's use the CRS:

```
x(0) = initGuess
do p = 1, numIter
    !$OMP PARALLEL DO
    do i=1, n
        t=b(i)
        do j = IA(i), IA(i+1) - 1
            if j=i
                d=A(j)
            else
                t = t - A(j) * x(jA(j))
            end if
        end do
```

```
        x(i) = t/d
    end do
    !$OMP END PARALLEL DO
end do
```

First, a guess of the true solution is taken. Generally if the values are completely unknown, the guess is a zero vector. Then comes the iteration loop. As with all of the iterative processes discussed here, the estimation is improved with more and more iterations as long as the method converges. Looping over the dimension of the solution vector, first the value of the right hand side vector is taken. Then for each element in the correct row of A, either the diagonal is stored or the proper value is subtracted from the right hand side value. This matches with the idea from above. Once completed with this loop, the new x vector is constructed one element at a time. This new x would be taken into the next iteration.

Consider where parallel directives were placed. Obviously the iterative steps cannot be done in parallel, by the nature of iteration. As for attempting to parallelize the calculations of the solution vector, there are some things to think about. Each calculation of $x^{(k+1)}(i)$ involves using calculations on other elements of $x^{(k)}$. If we parallelize the loop as above, there may be race conditions as to whether the new $x^{(k+1)}(j)$ term in the calculation is used or whether the old $x^{(k)}(j)$ term is used as the algorithm is designed for. Either way, the $x^{(k+1)}(i)$ term should improve from the previous iteration, so it does not matter in this particular context.

Otherwise the parallelism would be in the innermost loop. We will see later that for the set up of some matrices, there are a maximum number of elements in each row which would cause the center loop to not benefit much from parallel processing. If we do choose to parallelize the center loop, we would need to add a $OMP REDUCTION(+:t) statement, to make sure that the additions of t are properly combined.

Take an example $Ax = b$ as follows:

$$A = \begin{bmatrix} 3 & 0 & 0 & 1 \\ 1 & 4 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 2 & 0 & 3 & 6 \end{bmatrix} \equiv \begin{bmatrix} 3 & 1 & 1 & 4 & 2 & 2 & 2 & 3 & 6 \\ 1 & 4 & 1 & 2 & 4 & 3 & 1 & 3 & 4 \\ 1 & 3 & 6 & 7 & 10 & & & & \end{bmatrix}$$

$$b = \begin{bmatrix} 6 \\ -2 \\ 6 \\ 13 \end{bmatrix}$$

It is quite simple to show that the true solution of this system would be:

$$x = \begin{bmatrix} 2 \\ -1 \\ 3 \\ 0 \end{bmatrix}$$

Let's consider the first few iterations of the formula. First, allow the guess of x to be a vector of 1s. On the first iteration, we obtain:

$$x^{(1)} = \begin{bmatrix} 5/3 \\ 17/12 \\ 3 \\ 1/9 \end{bmatrix}$$

And after just the second iteration, we get within five hundredths:

$$x^{(2)} = \begin{bmatrix} 1.9629 \\ -1.04629 \\ 3 \\ 1.234 * 10^{-2} \end{bmatrix}$$

It is fairly evident that in this scenario, the Jacobi Method converges. In fact, for every diagonally dominant matrix, the Jacobi method will converge. The proof of this is provided in the following subsection.

Diagonally Dominant Jacobi Proof

The Jacobi can be proven to converge for diagonally dominant A. The proof is as follows:

Consider a matrix A

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1n} \\ a_{21} & a_{22} & a_{23} & ... & a_{2n} \\ a_{31} & a_{32} & a_{33} & ... & a_{3n} \\ ... & ... & ... & ... & ... \\ a_{n1} & a_{n2} & a_{n3} & ... & a_{nn} \end{bmatrix}$$

In order to be diagonally dominant, the diagonal entry must be greater in magnitude than the sum of the remaining entries' magnitudes on its containing row (or column). That is, for all i:

$$|a_{ii}| > \sum_{j=1 j \neq i}^{n} |a_{ij}|$$

With that in mind, A can be split into the Jacobi.

$$M = \begin{bmatrix} a_{11} & 0 & 0 & ... & 0 \\ 0 & a_{22} & 0 & ... & 0 \\ 0 & 0 & a_{33} & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & a_{nn} \end{bmatrix} \quad N = - \begin{bmatrix} 0 & a_{12} & a_{13} & ... & a_{1n} \\ a_{21} & 0 & a_{23} & ... & a_{2n} \\ a_{31} & a_{32} & 0 & ... & a_{3n} \\ ... & ... & ... & ... & ... \\ a_{n1} & a_{n2} & a_{n3} & ... & 0 \end{bmatrix}$$

The inverse of a diagonal matrix is another diagonal matrix with the entries all inverted. This can be easily verified as the two multiply to the identity matrix.

The convergence analysis from above determined that the iteration would converge if the matrix product $M^{-1}N$ has a spectral radius less than one. This means that if any norm can be shown to be less than one, then the iteration would converge.

So, when M^{-1} and N are multiplied:

$$M^{-1}N = - \begin{bmatrix} 0 & a_{12}/a_{11} & a_{13}/a_{11} & ... & a_{1n}/a_{11} \\ a_{21}/a_{22} & 0 & a_{23}/a_{22} & ... & a_{2n}/a_{22} \\ a_{31}//a_{33} & a_{32}/a_{33} & 0 & ... & a_{3n}/a_{33} \\ ... & ... & ... & ... & ... \\ a_{n1}/a_{nn} & a_{n2}/a_{nn} & a_{n3}/a_{nn} & ... & 0 \end{bmatrix}$$

So the infinity norm would be the easiest calculation. Let p be the maximum row sum in magnitude. The infinity norm would be:

$$\left| \frac{a_{p1} + a_{p2} + a_{p3} + ... + a_{p(p-1)} + a_{p(p+1)} + a_{p(p+2)} + ... + a_{p(n)}}{a_{pp}} \right|$$

Using the fact that

$$|a_{ii}| > \sum_{j=1 j \neq i}^{n} |a_{ij}|$$

In a diagonally dominant matrix, it can be shown that the above fraction is less than

$$\frac{|a_{pp}|}{|a_{pp}|}$$

Which is equal to one. Therefore, if a matrix is diagonally dominant the spectral radius is less than one and the Jacobi method is a convergent iterative algorithm.

We will see later that this is an important fact. Often numerical methods to solve partial differential equations provide a strictly diagonally dominant matrix. Note that this is not required for the Jacobi to converge. However, if the Jacobi method is ran and the solution is significantly outside of what is expected, it would be worth checking to see if the matrix is diagonally dominant before proceeding. If not, a different method may be required.

3.1.3 Gauss-Seidel and SOR

As previously mentioned, there are three major splitting methods, the Jacobi method, the Gauss-Seidel method, and the SOR method. Though the primary focus here is on the Jacobi method, the other two will be briefly considered. We shall see later in this section that the guaranteed convergence is not the same for these methods as the Jacobi. Even without diagonally dominant matrices, it is sometimes possible to assure that the iterative system will reach the true solution. Also, often the solutions to different methods can converge faster than others so in applications it is worth considering several different methods.

Gauss-Seidel

For each of the three splitting methods, to solve $Ax = b$, A is split into $A = M - N$ and x is iterated with $x^{(i+1)} = M^{-1}[b + Nx^{(i)}]$. The difference is in what the M and N matrices are. In Gauss-Seidel, the M is the diagonal plus the lower triangular portion of A, while N is the negative of the strictly upper triangular portion of A. It is also possible to take M as the diagonal and upper triangular portion, or even to alternate between iterations. We describe just the first lower triangular choice since the principle is the same.

It is slightly more complex than Jacobi because now we need to work with a lower triangular matrix, rather than just inverting a diagonal matrix. This involves the use of forward substitution. Consider forward substitution for a compressed matrix A. The ideal format would be CRS.

SOR

Last of the three splitting methods is Successive Over Relaxation (SOR). SOR alters the Gauss-Seidel method to improve convergence. Again, let us start with the system $Ax = b$. Let D be the diagonal of A, L be the strictly lower triangular portion of A and U be the strictly upper triangular portion of A. We can confirm that A can be split as follows:

$$A = M - N; M = D/\omega + L; N = (1/\omega - 1)D - U$$

And the same iteration can be used:

$$x^{(i+1)} = M^{-1}Nx^{(i)} + M^{-1}b$$

Here, the ω is called the relaxation factor. Again, since the matrix M is lower triangular, we can use forward substitution.

We have shown that the Jacobi method converges for strictly diagonally dominant matrices. Obviously if the SOR method is more computationally intensive, we would want some benefit. We can show that for any Hermitian positive definite matrix with $0 < \omega < 2$ the method of Successive Over Relaxation will converge for any initial guess. [5] This is a much stronger statement than that from the Jacobi, hence the value of this more complex method.

3.2 Descent methods

Above, the Jacobi, Gauss Seidel, and SOR method were discussed. These splitting methods aren't the only iterative systems of solving sparse matrix problems. There are also descent methods.

3.2.1 Conjugate Gradient Method

Again, the goal is to solve $Ax = b$ this time for the special case of A being a symmetric positive definite matrix. If we consider the quadratic form

$$Q(x) = \frac{1}{2}x^T Ax - x^T b$$

We can show that setting the gradient of this equal to zero is equivalent to finding the point that $Ax = b$. We can observe that:

$$\nabla Q = \begin{bmatrix} \frac{\partial Q}{\partial x_1} \\ \frac{\partial Q}{\partial x_2} \\ \cdots \\ \frac{\partial Q}{\partial x_n} \end{bmatrix}$$

and

$$Ax = \begin{bmatrix} \sum a_{1j}x_j \\ \sum a_{2j}x_j \\ \cdots \\ \sum a_{nj}x_j \end{bmatrix}$$

Then, considering each of the right hand terms from the Q equation above:

$$x^T Ax = \sum_i \sum_j a_{ij}x_i x_j$$

$$x^T b = \sum_i x_i b_i$$

Thus, the partial derivative with respect to x_1 of Q is:

$$\frac{\partial}{\partial x_i}Q = \frac{\partial}{\partial x_i}\left[1/2\sum_i \sum_j a_{ij}x_i x_j - \sum_i x_i b_i\right]$$

Consider the first term of this partial derivative. When i and j are both equal to one, we get $1/2a_{11}x_1^2$, of which the partial derivative is $a_{11}x_1$. When i and j are not equal, we

would have both $1/2a_{12}x_1x_2$ and $1/2a_{21}x_2x_1$. The partial derivatives of these are $1/2a_{12}x_2$ and $1/2a_{21}x_2$ respectively, and since this method requires a symmetric positive definite matrix, these sum to $a_{12}x_2$. Thus we can arrive at the partial derivatives with respect to x_i of Q are:

$$\nabla Q = \begin{bmatrix} \sum a_{1j}x_j - b_1 \\ \sum a_{2j}x_j - b_2 \\ \cdots \\ \sum a_{nj}x_j - b_n \end{bmatrix}$$

From the above we know that this is equal to $Ax - b$.

Thus this shows that solving $\nabla Q = 0$ is equivalent to solving $Ax - b = 0$. $\nabla Q = 0$ means that there is an extremum of Q. Specifically, solving $Ax - b = 0$ amounts to an optimization problem of finding the global minimum of the function:

$$Q(x) = \frac{1}{2}x^T Ax - x^T b$$

CG Iteration

The idea of the Conjugate Gradient method is to move from guesses x_i to x_{i+1} with the updating formula:

$$x_{i+1} = x_i + \alpha_i p_i$$

Where p_i is a direction vector and α_i is a scaling factor. Both of these are chosen such that $E(x_{i+1}) \leq E(x_i)$ where the error function is defined as:

$$E(x) = e(x)^T Ae(x) = (\hat{x} - x)^T (A\hat{x} - Ax).$$

The first portion to be considered is the scaling factor α_i. When the directional vector p_i is already constructed, we can determine the α_i to most improve the error of the iteration. That is we want α_i such that:

$$\alpha_i = \arg\min_{\alpha} E(x_{i+1}) = \arg\min_{\alpha} [x_i + \alpha_i p_i]$$

First we can take the $E(x_i + \alpha_i p_i)$ definition and determine the ideal α values.

$$E(x_i + \alpha_i p_i) = [\hat{x} - (x_i + \alpha_i p_i)]^T A[\hat{x} - (x_i + \alpha_i p_i)]$$

We know that $A\hat{x}$ is equal to the true solution b, and that α_i is a scalar to distribute the second portion of the above equation:

$$= [\hat{x} - (x_i + \alpha_i p_i)]^T [b - Ax_i - \alpha_i Ap_i)]$$

$$= [\hat{x} - (x_i + \alpha_i p_i)]^T [r_i - \alpha_i Ap_i)]$$

$$= \hat{x}^T r_i - \alpha_i \hat{x}^T Ap_i - (x_i + \alpha_i p_i)^T r_i + \alpha_i (x_i + \alpha_i p_i)^T Ap_i$$

$$= e_i^T r_i - 2\alpha_i r_i^T p_i + \alpha_i^2 p_i^T Ap_i.$$

In order to find the optimum α_i, we must determine the value of α_i that will minimize the above function. This brings us to the partial derivative with respect to α_i:

$$\partial E(x_i + \alpha_i p_i)/\partial \alpha_i = -2r_i^T p_i + 2\alpha_i p_i^T Ap_i$$

$$\alpha_i = (r_i^T p_i)/(p_i^T Ap_i).$$

So now, with the α being determined, we can find the p_i to improve the error the most. Again, we can start with $E(x) = e(x)^T Ae(x)$ Thus the error for the second iteration is:

$$E(x_i + \alpha_i p_i) = (\hat{x} - (x_i + \alpha_i p_i))^T A(\hat{x} - (x_i + \alpha_i p_i)).$$

We have already determined that $\alpha_i = (r_i^T p_i)/(p_i^T Ap_i)$. So with the rearrangement of the error equation and the use of α_i above, we have:

$$E(x_i + \alpha_i p_i) = [(\hat{x} - x_i) - \alpha_i p_i]^T A[(\hat{x} - x_i) - \alpha_i p_i]$$

$$= [e_i - \alpha_i p_i]^T A[e_i - \alpha_i p_i]$$

$$= e_i^T A e_i - 2\alpha_i e_i^T A p_i + \alpha_i^2 p_i^T A p_i$$

$$= e_i^T A e_i - 2(r_i^T p_i)^2/(p_i^T A p_i) + (r_i^T p_i)^2/(p_i^T A p_i)$$

$$= e_i^T A e_i - (r_i^T p_i)^2/(p_i^T A p_i)$$

$$= E(x_i) - (r_i^T p_i)^2/(p_i^T A p_i).$$

Using the definition that $e_i^T A e_i$ is equal to the error function for the initial x_i. Thus we have:

$$E(x_{i+1}) = E(x_i) - (r_i^T p_i)^2/(p_i^T A p_i).$$

We know that $E(x_i) = r_i^T A^{-1} r_i$, thus we can rewrite the above as:

$$E(x_{i+1}) = E(x_i)[1 - (1/(r_i^T A^{-1} r_i))(r_i^T p_i)^2/(p_i^T A p_i).$$

This brings us to use the Rayleigh Quotient. This tells us that

$$\lambda_{\min}(A) <= (x^T A x)/(x^T x) <= \lambda_{\max}(A),$$

which when substituted in the Big E function above gives us:

$$E(x_{i+1}) = E(x_i)[1 - (r_i^T p_i)^2/[(1/\lambda_{\min}) * r_i^T r_i \lambda_{\max} p_i^T p_i].$$

Now we can use the definition of the condition number of a matrix. The condition number K is equal to the maximum eigenvalue over the minimum eigenvalue. This can be substituted into the above to get:

$$E(x_{i+1}) = E(x_i)[1 - 1/(K_2(A))[(r_i/||r_i||)^T (p_i/||p_i)]^2].$$

Thus, the convergence is assured if p_i are chosen such that:

$$[(r_i/||r_i||)^T (p_i/||p_i)]^2 \geq w > 0$$

. Then note that for any choice of p_i we have:

$$p_i^T r_{i+1} = p_i^T (b - Ax_{i+1}) = p_i^T (b - A(x_i + \alpha_i p_i))$$

$$= p_i^T (b - Ax_i - \alpha_i Ap_i) = p_i^T r_i - \alpha_i p_i^T Ap_i = 0.$$

So, for a choice of p_i that is not directly perpendicular to the residual would eventually lead to convergence. So we have several choices of p_i. Three common ones are as follows:

1. Univariate relaxation: $p_0 = e_1, p_1 = e_2, ...$ where e_i is the standard basis vector.
2. Steepest Descent $p_i = r_i$.
3. Conjugate Directions. Picking p_i such that $p_i^T Ap_j = 0, i \neq j$.

CG overview

So, to reiterate the steps of the Conjugate Gradient method. First an initial guess of x_0 is chosen. Next the residual of that guess is calculated. A p_0 is chosen, the α_0 is computed, and the next iteration of x, x_i is constructed with $x_1 = x_0 + \alpha_0 p_0$. Here, the p_i directional vectors can be picked in a variety of methods.

CG Convergence analysis

We saw above that the splitting methods would converge when $||M^{-1}N|| < 1$. We know that the conjugate gradient method will converge as long as a p is chosen such that p is non orthogonal to the residual. There is a stronger conclusion for the convergence as follows:

Theorem 1 *The iterative scheme $x_{i+1} = x_i + \alpha p_i$ leads to the exact solution in at most n iterations if α_i are chosen according to the optimality criteria and the p_i are chosen as conjugate directions.*

Proof: First we can start with the dot product of the p_j vector with the residual r_{i+1}:

$$p_j^T r_{i+1} = p_j^T (r_i - \alpha_i Ap_i)$$

For $j < i$ this gives us:

$$p_j^T r_{i+1} = p_j^T r_i.$$

For $j = i$, the quantity equals zero. Thus this gives us:

$$p_j^T r_n = p_j^T r_{n-1} = ... = p_j^T r_{j+1} = 0$$

And r_n is perpendicular to each of the p_0 to p_{n-1} vectors, and is equal to zero. If the final residual is equal to zero then the exact solution is found.

With this in mind, the best approach would seem to be finding conjugate direction p vectors. There are many approaches to finding these. Also, realize that this method is only for the special case where A is symmetric positive definite. If not in this format, we would not have this incredibly powerful theorem.

3.2.2 CG Code

Conjugate Gradient Method. The pseudocode for the method is as follows:

Choose x_0

set $p_0 = r_0 = b - Ax_0$

for i = 0,1,2,...

— $\alpha_i = (p_i^T r_i)/(p_i^T A p_i)$

— $x_{i+1} = x_i + \alpha_i p_i$

— $r_{i+1} = r_i - \alpha_i A p_i$

— $\beta_i = -(r_{i+1}^T A p_i)/(p_i^T A p_i)$

— $p_{i+1} = r_{i+1} + \beta_i p_i$

end

This can be simplified as

$$p_i^T r_i = (r_i + \beta_{i-1} p_{i-1})^T r_i$$

$$= r_i^T r_i + (\beta_{i-1} p_{i-1})^T r_i$$

Since p_{i-1} is orthogonal to r_i we have $p_i^T r_i = r_i^T r_i$. Also we have

$$Ap_i = 1/\alpha_i[r_i - r_{i+1}]$$

and

$$r_{i+1}^T r_i = [r_i - \alpha_i A p_i]^T r_i$$

$$= r_i^T r_i - \alpha_i p_i^T A r_i$$

$$= r_i^T r_i - \alpha_i p_i^T A[p_i - \beta_{i-1} p_{i-1}]$$

$$= r_i^T r_i - \alpha_i p_i^T A p_i + \alpha_i \beta_{i-1} p_i^T A p_{i-1} = 0.$$

Thus we can simplify β_{i+1} to:

$$\beta_{i+1} = -(r_{i+1}^T A p_i)/(p_i^T A p_i)$$

$$= -(r_{i+1}^T [1/\alpha_i (r_i - r_{i+1})])/(p_i^T A p_i)$$

$$= (r_{i+1}^T r_{i+1})/(r_i^T r_i)$$

This brings us to the classical version of the Conjugate Gradient method:

Choose x_0

x(0)

p(0)=r(0)=b - Ax(0)

for i = 0,1,...

— $\alpha_i = (r_i^T r_i)/(p_i^T A p_i)$

— $x_{i+1} = x_i + \alpha_i p_i$

— $r_{i+1} = r_i - \alpha_i A p_i$

— $\beta_i = (r_{i+1}^T r_{i+1})/(r_i^T r_i)$

— $p_{i+1} = r_{i+1} + \beta_i p_i$

end

Unfortunately in this form, there is little that can be done in terms of parallelizing,

unless deeper considerations are made (such as in the matrix vector products). While this method is powerful and important enough to examine in this thesis, the benefits due to parallelism would require more developments beyond the scope of the thesis and thus will not be implemented.

3.2.3 Preconditioned CG

Sometimes it benefits the convergence to alter the problem to improve condition number. That is, we know solving $Ax = b$ is equivalent to solving $(M_1AM_2)M_2^{-1}x = M_1b$. So, we can adjust the initial problem to solving:

$$\hat{A}\hat{x} = \hat{b}$$

Where $\hat{A} = M_1AM_2, \hat{b} = M_1b$, and $x = M_2\hat{x}$. Solving this system can, with the correct choices of M_1 and M_2 improve the convergence of the Conjugate Gradient method without significantly complicating the problem.

One possible choice of the preconditional is the Incomplete Choleski. Here, $\hat{A}$ would be defined as

$$\hat{A} = L^TAL,$$

where L is from the Choleski decomposition.

3.2.4 Non-symmetric case

All of this work has been assuming a symmetric A. The conjugate gradient method can be expanded into a non-symmetric case though. Taking non-symmetric A and the system $Ax = b$, we can left multiply both sides by A^T to get $\hat{A}x = \hat{b}$, where $\hat{A} = A^TA, \hat{b} = A^Tb$ The negative from this approach would be that the condition number of the new system would be the square of the condition number from the initial system.

3.3 Krylov Subspace Methods

In the basic sense, a Krylov subspace is the space spanned by the reflections of a vector on powers of a matrix. That is if $x \in K_m(A, v)$, then

$$x = a_0 v + a_1 Av + a_2 A^2 v + ... + a_{m-1} A^{m-1} v = p_{m-1}(A)v.$$

Take the example matrix A:

$$A = \begin{bmatrix} 5 & 6 & 2 \\ 0 & -1 & -8 \\ 1 & 0 & 2 \end{bmatrix}.$$

The characteristic polynomial of A is

$$p(\lambda) = \lambda^3 - 6\lambda^2 + 1\lambda + 56.$$

This polynomial can be determined by finding the determinant of $A - I\lambda$. Thus, we also get:

$$A^3 - 6A^2 + A + 56I = 0.$$

Which, with some manipulation provides:

$$A^3 - 6A^2 + A = -56I$$

$$-\frac{1}{56}(A^3 - 6A^2 + A) = I.$$

Left multiplying both sides by A^{-1}:

$$-\frac{1}{56}A^{-1}(A^3 - 6A^2 + A) = A^{-1}.$$

$$-\frac{1}{56}(A^2 - 6A + I) = A^{-1}.$$

So, by this, we can see that the inverse of a matrix can be written as a polynomial of the powers of that matrix. The catch to this is that the coefficients of the polynomial require the knowledge of the characteristic polynomial, which is computationally intensive to determine. Regardless, the implication for this is that:

$$A^{-1}v \in \{v, Av, ..., A^{n-1}v\}$$

where n is the dimension of A.

3.3.1 Krylov Subspace Method/Arnoldi

With this in knowledge, we can try to solve linear system $Ax = b$. We can seek an approximation of $A^{-1}b$ in the Krylov Subspace $K_m(A, b)$ where $m \ll n$.

In order to use the Krylov subspaces, we need a basis that allows numerical operations to be stable. The Arnoldi algorithm is a procedure that constructs orthonormal basis of the Krylov subspace. The input of the Arnoldi procedure is the matrix A, the vector v and the approximation number m which is less than the dimension of the matrix. The output is the basis V of the Krylov subspace and the Upper Hessenberg matrix $\hat{H}$.

The code of the Arnoldi procedure is as follows:

$v_1 = v/||v||$

for j = 1:m

. $w = Av_j$

. for i = 1:j

. . $h_{i,j} = v_i'w$

. . $w = w - h_{i,j}v_i$

. end

. $h_{j+1,j} = ||w||$

. $v_{j+1} = w/h_{j+1,j}$

end

So, using this method to solve $Ax = b$, we first select an initial x_0. Then $x_m =$

$x_0 + V_m y$. This gives a residual of:

$$r_m = b - Ax_m = b - A(x_0 + V_m y_m)$$

$$= b - Ax_0 + AV_m y_m = r_0 - AV_m y_m.$$

3.3.2 Two Types

There are two different popular options for the Krylov Subspace calculations, the first of these being the Full Orthogonalization Method (FOM). The second method is the Generalized Minimal Residual Method, (GMRES). As we saw, they both have: $r_m = r_0 - AV_m y_m$. They differ in their selection of y_m.

FOM

In the Full Orthogonalization Method, we take the above equation and left multiply both sides by V_m^T. This gives us:

$$V_m^T r_m = V_m^T (r_0 AV_m y_m).$$

With this method, we set $V_m^T r_m = 0$. This constraint is known as the Petrov-Galerkin condition. With this constraint and letting $V_m^T AV_m = H_m$ we get:

$$H_m y_m = V_m^T r_0 = V_m^T (V_m(\beta e_1))$$

and instead of solving the larger m by m system, we can solve the smaller:

$$y_m = H_m^{-1}(\beta e_1).$$

GMRES

In the Generalize Minimal Residual Method, we do not use the same Petrov-Galerkin constraint as FOM. Instead we select the value such that the norm of r_m is minimized.

This is equivalent to minimizing:

$$||b - A(x_0 + V_m y_n)||$$

$$= ||b - Ax_0 - AV_m y_n||$$

$$= ||r_0 - AV_m y_n||$$

and from the Arnoldi Process, we have $AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^T$. So we have

$$V_m^T A V_m = H_m.$$

Thus, the important part of GMRES is to find the vector y_n which minimizes

$$||\beta e_1 - \hat{H}_m y_n||,$$

where $\hat{H}_m$ is H_m augmented with $h_{m+1,m} e_m^T$ under its last row.

4 PARTIAL DIFFERENTIAL EQUATIONS

4.1 Setting Up the System: Equation 1

Since this paper has a focus on solving large sparse systems with parallelism, it is important to understand the applications of these systems. In the scenario of partial differential equations, this involves setting up the system of equations.

Using numerical analysis, the approximations of the partial derivatives of an equation can be easily obtained with a simple five-point stencil. Figure 4.1 demonstrates the stencil.

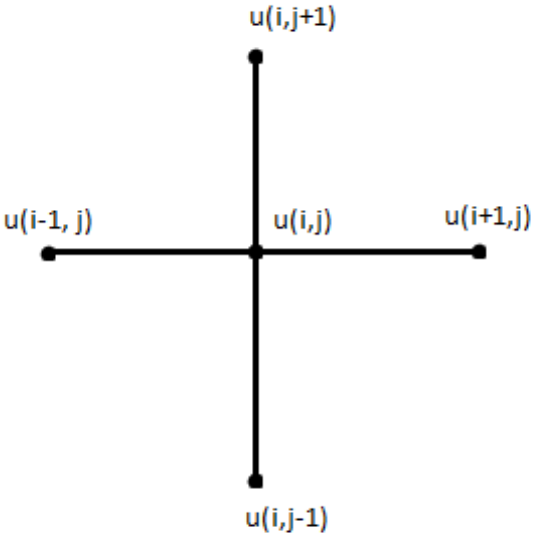

Figure 4.1: Five-Point Stencil

Each point $u_{i,j}$ can be approximated by replacing the partial derivative terms in the equation by their five-point stencil equivalent. First consider the partial derivative u_x

with the center step selection:

$$u_x|_{x_i,y_i} = \frac{\partial u}{\partial x}(x_i, y_j) \approx \frac{u_{i+1,j} - u_{i-1,j}}{2h}.$$

Then, consider the u_{xx} approximations:

$$u_{xx} \approx \frac{\partial/\partial x(u_{i+1,j}) - \partial/\partial x(u_{i,j})}{h}$$

$$\approx \frac{\frac{u_{i+1,j}-u_{i,j}}{h} - \frac{u_{i,j}-u_{i-1,j}}{h}}{h}$$

$$= \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}.$$

From this, we can easily get the u_{yy} approximation:

$$u_{yy} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}.$$

We can substitute these into Equation #1:

$$-\epsilon(u_{xx} + u_{yy}) + au_x = x(1 - p^2 x)\sin(ly).$$

To obtain:

$$-\epsilon\left(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}\right)$$

$$+a\frac{u_{i+1,j} - u_{i-1,j}}{2h} = ih(1 - p^2 ih)\sin(ljh).$$

This can be grouped by the specific element of the grid:

$$(-\epsilon/h^2 + a/(2h))u_{i+1,j} + (-\epsilon/h^2 + a/(2h))u_{i-1,j} + 4\epsilon/h^2 u_{i,j} - \epsilon/h^2 u_{i,j+1} - \epsilon/h^2 u_{i,j-1}.$$

The above quantity is equal to the right hand side of the equation:

$$ih(1 - p^2 ih)\sin(ljh).$$

These coefficients are constants, so we can define the constants:

$$\beta = -\epsilon/(h^2) + a/(2h),$$

$$\delta = -\epsilon/(h^2) - a/(2h),$$

$$\alpha = 4\epsilon/(h^2),$$

$$\gamma = -\epsilon/(h^2).$$

This simplifies the grid equations to

$$\beta u_{i+1,j} + \delta u_{i-1,j} + \alpha u_{i,j} + \gamma u_{i,j+1} + \gamma u_{i,j-1}.$$

Now, the matrix must be constructed. All of the boundary constraints are known, so only the interior nodes will be in the vector being solved for. This vector is as follows:

$$\begin{bmatrix} u_{11} \\ u_{21} \\ ... \\ u_{n1} \\ u_{12} \\ ... \\ u_{nn} \end{bmatrix}.$$

The matrix and solution vector can be methodically constructed from the above equations.

$$\begin{bmatrix} Q & G & 0 & 0 & 0 & ... & 0 & 0 & 0 \\ G & Q & G & 0 & 0 & ... & 0 & 0 & 0 \\ 0 & G & Q & G & 0 & ... & 0 & 0 & 0 \\ .. & ... & ... & ... & ... & ... & .. & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & ... & G & Q & G \\ 0 & 0 & 0 & 0 & 0 & ... & 0 & G & Q \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ ... \\ U_{N-1} \\ U_N \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ ... \\ F_{N-1} \\ F_N \end{bmatrix},$$

Where

$$Q = \begin{bmatrix} \alpha & \beta & 0 \\ \delta & \alpha & \beta \\ 0 & \delta & \alpha \end{bmatrix}; G = \begin{bmatrix} \gamma & 0 & 0 \\ 0 & \gamma & 0 \\ 0 & 0 & \gamma \end{bmatrix}; U_i = \begin{bmatrix} u_{1i} \\ u_{2i} \\ ... \\ u_{ni} \end{bmatrix}; F_i = \begin{bmatrix} \hat{f}_{1i} \\ \hat{f}_{2i} \\ ... \\ \hat{f}_{ni} \end{bmatrix};$$

and each of the $\hat{f}$ are equal to the evaluation of the function $ih(1-p^2ih)sin(ljh)$ subtracted by the known portions of the right hand side. For example $\hat{f}_{11} = f_{11} - \delta u_{01} - \gamma u_{10}$

This sets up the sparse matrix. This process is shown with both MATLAB and FORTRAN code later in the text. Once this matrix system is constructed, the iterative methods discussed in the previous chapter can be used to determine the approximate solution of the PDE at each point on the grid.

4.2 Setting Up the System Equation 2

Consider the second equation:

$$u_{xx} + 3u_y - u_{yy} = 8y + 4x(3x + e^{2y}),$$

$$u(0,y) = 0; u(1,y) = 4y + 2e^{2y}; u(x,0) = 2x; u(x,1) = 4x^2 + 2xe^2.$$

Again, we can consider the finite difference methods as above to get the five point stencil scheme.

$$(\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}) + 3(\frac{u_{i,j+1} - u_{i,j-1}}{2h}) = 8jh + 4ih(3ih + e^{2jh}).$$

Splitting this provides us with:

$$(1/h^2)u_{i+1} + (-2/h^2)u_{i,j} + (1/h^2)u_{i-1,j} + (3/2h)u_{i,j+1} + (-3/2h)u_{i,j-1} = 8jh + 4ih(3ih + e^{2jh}).$$

Though the equation is slightly different, the same process can be done setting the coefficients to constants:

$$\omega = (-2/h^2)$$

$$\zeta = (1/h^2)$$

$$\eta = (3/2h).$$

And the matrix can be setup similar to above only with:

$$Q = \begin{bmatrix} \omega & \zeta & 0 \\ \zeta & \omega & \zeta \\ 0 & \zeta & \omega \end{bmatrix}; G = \begin{bmatrix} \eta & 0 & 0 \\ 0 & \eta & 0 \\ 0 & 0 & \eta \end{bmatrix};$$

And with the subdiagonal be block $-G$ rather than block G.

4.3 Solving the System Equation 1

4.3.1 MATLAB Sequential solution

In the Section A.3 of the appendix, the MATLAB code for solving the PDE from Equation #1 is shown. There are four inputs for the equation parameters and the dimension of the grid. First, the step size, x dimensions, y dimensions, and equation constants were calculated. From that the true solution is determined.

Then the sparse matrix system is constructed. First the coefficients of the matrix are calculated, then the right hand side F vector was determined. The toeplitz and kronecker functions are used to form the sparse matrix with the calculated coefficients. Then a guess vector of all ones is created. A Jacobi method function is called with the matrix and both of these vectors. [2] Lastly as a simple verification that the program was working properly, the derived solution was reshaped and compared in a meshgrid against the true solution.

The effectiveness of this program is quickly tested. Take the inputs $\epsilon = 1; \alpha = 1; N = 128; l = 16$. When we graph the true solution in Figure 4.2, we get the following:

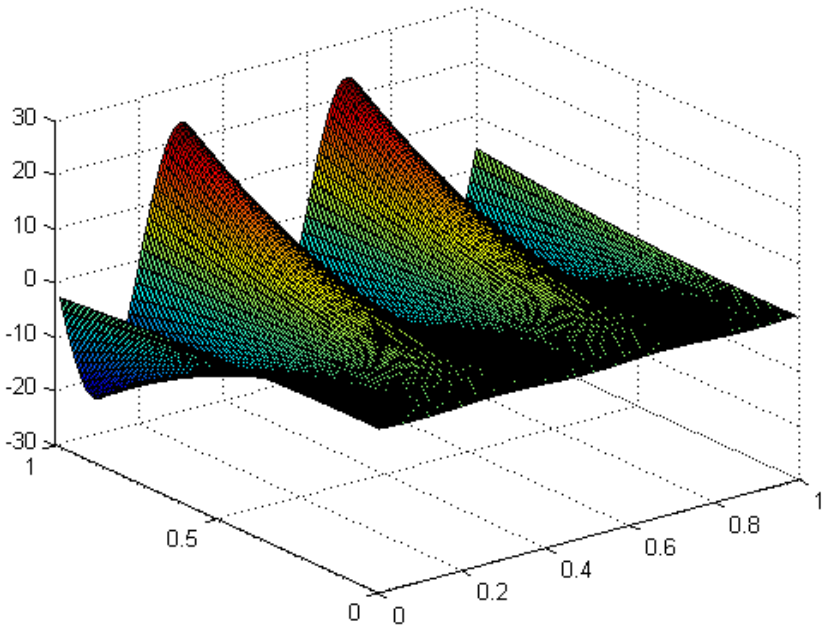

Figure 4.2: Solution of PDE with e=1, a=1, N=128, l=16

This solution can be compared to the approximation obtained with just ten iterations of the Jacobi method. This can be seen in the following Figure 4.3.

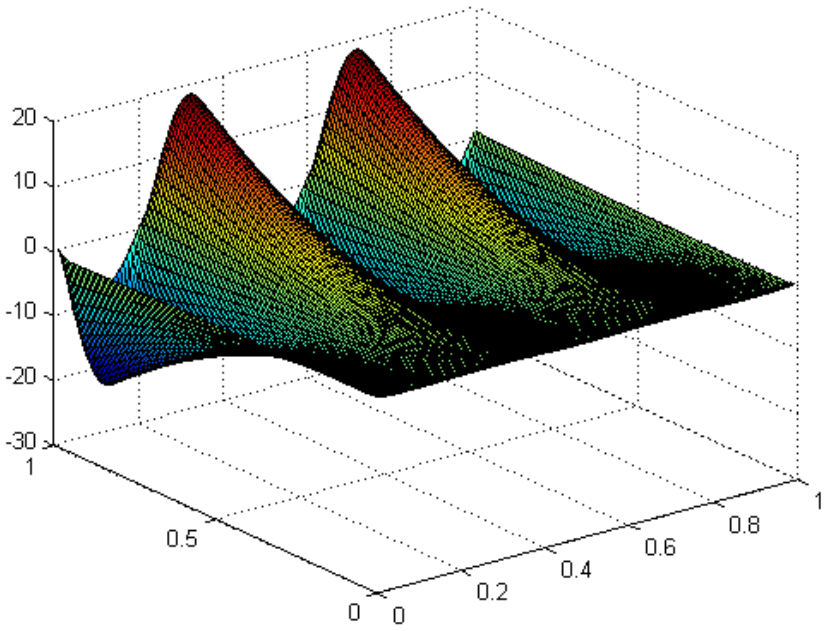

Figure 4.3: Jacobi Method Approximation, 10 iterations

Obviously, the approximation is converging to the true solution fairly quickly. By looking at the formed matrix, we can see that the true solution and derived solution match our expectations. The α term that is on the diagonal largely outnumbers the off-diagonal terms and we have already proven that a diagonally dominant system converges with the Jacobi Method, so the system must converge.

4.3.2 FORTRAN Parallel solution

The next step was to construct a parallel version of this code. The code for this process can be seen in the appendix in section A.4. A module is constructed containing the sparse matrix arrays and the above sparse parallel Jacobi method. This module is used, and all other required variables are declared and allocated. Unlike the MATLAB program, when submitting a program to the queue of the ASC no user input is allowed, thus the inputs were hardcoded (they could also have been read from a file). Again, the all of the constants for the equation and five point stencil were calculated and a true solution matrix was constructed. The left hand side matrix was methodically constructed in sparse form. First the aA and jA vectors were filled since they were the same length. Some loops of that process were able to be parallelized. Next the iA vector was constructed. Finally the parallel Jacobi was called and the solved outputs were shown to verify that the solution matched that from the MATLAB implementation. Figures 4.4 and 4.5 show that the outputs of similar inputs on MATLAB and FORTRAN match.

With inputs of $\epsilon = .1, \alpha = 1, N = 128, l = 3$ and 500 iterations of the Jacobi method, the FORTRAN code for the Jacobi method ran roughly three to four times faster than the MATLAB equivalent with a single core usage. However, this could have been a byproduct of the MATLAB code being ran on a personal laptop where the FORTRAN code was executed on the ASC. That being mentioned, the computed results of the FORTRAN code executed on a single core was at least comparable to MATLAB. So with that in consideration, we look to the speed difference of increasing the number of cores.

```
>> pde(0.1,1,3,3)

A =

    3.6000    0.6000         0   -0.9000         0         0         0         0         0
   -2.4000    3.6000    0.6000         0   -0.9000         0         0         0         0
         0   -2.4000    3.6000         0         0   -0.9000         0         0         0
   -0.9000         0         0    3.6000    0.6000         0   -0.9000         0         0
         0   -0.9000         0   -2.4000    3.6000    0.6000         0   -0.9000         0
         0         0   -0.9000         0   -2.4000    3.6000         0         0   -0.9000
         0         0         0   -0.9000         0         0    3.6000    0.6000         0
         0         0         0         0   -0.9000         0   -2.4000    3.6000    0.6000
         0         0         0         0         0   -0.9000         0   -2.4000    3.6000

b =                         x =

   -6.4667                     -2.1150
   -1.2243                     -2.0153
   -0.1735                     -1.9373
   -6.8281                     -2.6184
    0.2968                     -2.3516
    1.3887                     -2.1819
   -7.3816                     -2.3396
   -1.3975                     -2.1923
   -0.1980                     -2.0621
```

Figure 4.4: MATLAB Output N=3

```
ualdfl@dmcvlogin1:mpiVer> ./test
 A
   3.60000000000000       0.600000000000000      -0.900000000000000
  -2.40000000000000       3.60000000000000       0.600000000000000
  -0.900000000000000     -2.40000000000000       3.60000000000000
  -0.900000000000000     -0.900000000000000      3.60000000000000
   0.600000000000000     -0.900000000000000     -0.900000000000000
  -2.40000000000000       3.60000000000000       0.600000000000000
  -0.900000000000000     -0.900000000000000     -2.40000000000000
   3.60000000000000      -0.900000000000000     -0.900000000000000
   3.60000000000000       0.600000000000000     -0.900000000000000
  -2.40000000000000       3.60000000000000       0.600000000000000
  -0.900000000000000     -2.40000000000000       3.60000000000000
           1           2           4           1           2           3
           5           2           3           6           1           4
           5           7           2           4           5           6
           8           3           5           6           9           4
           7           8           5           7           8           9
           6           8           9
           1           4           8          11          15          20
          24          27          31          34
 b
  -6.46670060487137      -1.22429895064191      -0.173502310305940
  -6.82810717508375      0.296754758514706       1.38871159674370
  -7.38157321449426      -1.39750591418814      -0.198048446133787
 x
  -2.11506600205096      -2.01515205384949      -1.93703322721973
  -2.61847581646863      -2.35145552736199      -2.18161375382925
  -2.33968853197183      -2.19220455702907      -2.06188660040275
```

Figure 4.5: FORTRAN Output N=3

4.3.3 Parallel Times

In order to get an accurate overview of the timings, larger problems were used as they provided a clearer difference in time average. Different sized grids were examined, from 128 splits, to 512 splits, to 1024 splits. Also different number of iterations of the Jacobi

method were examined from 500 to 5000 were done. The tables below show the average timings of each of these options on 1, 2, 4, or 8 processors.

The tables of these splits are as follows:

	Jacobi Method 500 Iterations Timing			
Grid Size	1 core	2 core	4 core	8 core
128	0.13206	0.074704	0.058224	0.063723
512	1.87326	1.29797	0.68935	0.68707
1024	9.103253	4.69777	2.62513	2.37764

Table 4.1: PDE #1 Jacobi Timing 500 Iterations

	Jacobi Method 5000 Iterations Timing			
Grid Size	1 core	2 core	4 core	8 core
128	1.521985	0.75577	0.52349	0.48198
512	25.14018	9.75753	5.90145	5.39222
1024	90.86935	52.48527	24.05471	21.40481

Table 4.2: PDE #1 Jacobi Timing 5000 Iterations

As expected, more iterations requires more time, but the error norm decreased. To compare errors, the difference in the approximate solution from the program and the true solution was taken and the norm2 of that vector was taken. Regardless of the number of iterations, increasing grid divisions seems to drastically increase the amount of time to run the program. This makes sense because doubling the size of one dimension of the matrix is equivalent to quadrupling the overall number of splits.

As we have already seen in previous sections, increasing the number of cores does improve timing, but the improvement is diminishing. When comparing very small grid sizes the parallel benefit was less presumably due to overhead. In fact, for the smallest problem tested, the four core runs actually were quicker on average relative to the eight core runs. With larger problems, more cores gave a larger benefit.

As expected, adding more cores and properly parallelizing code can drastically reduce the times to run a mathematical algorithm. This is not the only way to use parallelism to improve timings though. Changing the set up of a system can also have a positive benefit on computation time. In this chapter we will consider a Boundary Value Problem (BVP) and set up the problem in various methods to approximate the solution.

5.1 Boundary Value Problem

Take a boundary value problem:

$$y'' - y' - 2y = \cos(x), 0 \leq x \leq \pi/2, y(0) = -0.3, y(\pi/2) = -0.1$$

This problem has the solution $y(x) = -1/10(\sin(x) + 3\cos(x))$. [1] There are several methods to try to solve this problem numerically. Linear shooting poses a viable approach, as well as domain decomposition. First we can consider linear shooting.

5.2 Linear Shooting

Linear shooting is for problems that can be phrased in the form:

$$y'' = p(x)y' + q(x)y + r(x), a \leq x \leq b, y(a) = \alpha, y(b) = \beta$$

Where $p(x), q(x)$, and $r(x)$ are continuous on $[a, b]$ and $q(x) > 0$ on $[a, b]$. The basic approach is to take the BVP and replacing it with two initial-value problems.

$$y'' = p(x)y' + q(x)y + r(x), a \leq x \leq b, y(a) = \alpha, y'(a) = 0$$

and

$$y'' = p(x)y' + q(x)y, a \leq x \leq b, y(a) = 0, y'(a) = 1.$$

We can then combine the two solutions of these equations to approximate the solution of the boundary value problem as:

$$y(x) = y_1(x) + [\beta - y_1(b)]/[y_2(b)]y_2(x).$$

This can be verified as the true solution of the BVP by taking derivatives and plugging into both sides of the equation.

Considering the above boundary value problem, the two initial value problems to be solved are:

$$y''(x) = y' + 2y + cos(x), y(0) = -0.3, y'(0) = 0$$

$$y''(x) = y' + 2y, y(0) = 0, y'(0) = 1.$$

These equations can be solved directly; however, the general approach would be to use an iterative method to approximate the solution. The Runga-Kutta 4th Order (RK4) method is commonly chosen due to its high accuracy relative to other methods like Euler's Method. [4].

5.2.1 RK4 to Solve Derived IVPs

Often the initial value problems that arise from the linear shooting method are not directly solvable. In these cases it is often beneficial to use the RK4 method. An initial value problem is in the following form:

$$y''(x) = f(y', y, x) \quad y(a) = \alpha \quad y(b) = \beta$$

To solve this problem with RK4, we start with the known $\omega_0 = \alpha$. From there four k terms are calculated as follows:

$$k_1 = hf(t_i, \omega_i); k_2 = hf(t_i + h/2, \omega_i + k_1/2)$$

$$k_3 = hf(t_i + h/2, \omega_i + k_2/2), k_4 = hf(t_{i+1}, \omega_i + k_3)$$

And with these four values, we can approximate the value of the differential equation at the next position.

$$\omega_{i+1} = \omega_i + 1/6(k_1 + 2k_2 + 2k_3 + k_4)$$

The process would be iterated until every ω_i up to ω_b is determined.

5.3 Domain Decomposition

Instead of going through this process of building two initial-value problems, approximating the solution of those, and combining their solutions to obtain the solution of the boundary value problem, we can consider using the same derivative approximations that were used for the PDEs.

$$y'' = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

$$y' = \frac{y_i - y_{i-1}}{h}$$

Combining these with the above BVP, we get:

$$(1/h^2)y_{i+1} + (-2/h^2 + 1/h + 1)y_i + (1/h^2 - 1/h)y_{i-1} = cos(x)$$

Which, as we have seen, allows for the systematic construction of a matrix system:

$$\begin{bmatrix} \alpha & \beta & 0 & 0 & 0 & ... & 0 & 0 & 0 & 0 & \\ \gamma & \alpha & \beta & 0 & 0 & ... & 0 & 0 & 0 & 0 & \\ 0 & \gamma & \alpha & \beta & 0 & ... & 0 & 0 & 0 & 0 & \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & 0 & ... & \gamma & \alpha & \beta & 0 & \\ 0 & 0 & 0 & 0 & 0 & ... & 0 & \gamma & \alpha & \beta & \\ 0 & 0 & 0 & 0 & 0 & ... & 0 & 0 & \gamma & \alpha & \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ ... \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} f_0 - \gamma u_0 \\ f_1 \\ ... \\ f_{n-1} \\ f_n - \beta u_{N+1} \end{bmatrix},$$

where $\alpha = (-2/h^2 + 1/h + 1)$, $\beta = (1/h^2)$, $\gamma = (1/h^2 - 1/h)$, and $f_i = \cos(x_i)$.

While this system itself is fairly simple to solve, we can reevaluate the setup to allow for parallel processing improvements. Consider the line of x as follows:

Figure 5.1: Boundary Value Problem Standard Grid

With a boundary value problem, we know the value of u_0 and u_{N+1}, the first and last elements of the line. With the above system, we solve for each u_i in order. We do not rearrange the vector in any manner. Consider instead the line as follows:

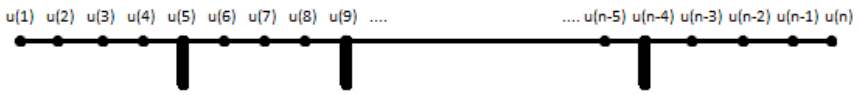

Figure 5.2: Boundary Value Problem Domain Decomposition Grid

Still, the first and last element are known, but now we specify evenly spaced separator points. This will split the domain

$$D_1 = [x_j]_{j=1,q}$$

$$D_2 = [x_j]_{j=q+2,2q+1}$$

$$...$$

$$D_p = [x_j]_{j=(p-1)q+p,pq+(p-1)},$$

for the interior point sections and

$$D_s = x_{q+1}, x_{2(q+1)}, ..., x_{(p-1)(q+1}$$

For the separator points.

Now with this arrangement, we can adjust the system. For example, if we were to have eleven interior points before domain decomposition, we get:

$$\begin{bmatrix} \alpha & \beta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \gamma & \alpha & \beta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \gamma & \alpha & \beta & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \gamma & \alpha & \beta & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \gamma & \alpha & \beta & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \gamma & \alpha & \beta & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \gamma & \alpha & \beta & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \alpha & \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \alpha & \beta & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \alpha & \beta \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \alpha \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \end{bmatrix} = \begin{bmatrix} f_1 - \gamma u_0 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \\ f_{10} \\ f_{11} - \beta u_{N+1} \end{bmatrix},$$

whereas, with domain decomposition, we can split the domain into three sets of three interior points and two separation points. This give us the form:

$$\left[\begin{array}{ccc|ccc|ccc|cc} \alpha & \beta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \gamma & \alpha & \beta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \gamma & \alpha & 0 & 0 & 0 & 0 & 0 & 0 & \beta & 0 \\ \hline 0 & 0 & 0 & \alpha & \beta & 0 & 0 & 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & \gamma & \alpha & \beta & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \gamma & \alpha & 0 & 0 & 0 & 0 & \beta \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & \alpha & \beta & 0 & 0 & \gamma \\ 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \alpha & \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \alpha & 0 & 0 \\ \hline 0 & 0 & \gamma & \beta & 0 & 0 & 0 & 0 & 0 & \alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & \gamma & \beta & 0 & 0 & 0 & \alpha \end{array}\right] \left[\begin{array}{c} u_1 \\ u_2 \\ u_3 \\ \hline u_5 \\ u_6 \\ u_7 \\ \hline u_9 \\ u_{10} \\ u_{11} \\ \hline u_4 \\ u_8 \end{array}\right] = \left[\begin{array}{c} f_1 - \gamma u_0 \\ f_2 \\ f_3 \\ \hline f_5 \\ f_6 \\ f_7 \\ \hline f_9 \\ f_{10} \\ f_{11} - \beta u_{N+1} \\ \hline f_4 \\ f_8 \end{array}\right]$$

Though the tri-diagonal matrix system may seem more simple to solve, this organization provides for a more parallelizable solution. This can be illustrated better in block form. The domain decomposition step provides a block matrix in the form:

$$\begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & ... & & \\ & & & A_p & B_p \\ \hat{B}_1 & \hat{B}_1 & ... & \hat{B}_p & A_s \end{bmatrix}.$$

We can consider the inverse of a block matrix Q:

$$Q = \begin{bmatrix} A & B \\ C & D \end{bmatrix}; \quad Q^{-1} = \begin{bmatrix} (A - BD^{-1}C)^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -D^{-1}C(A - BD^{-1}C)^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

with the corresponding unknown x_I and right hand side c_I in block form where $[u_1, u_2, u_3]^T = x_1$ and so on. It isn't difficult to show that

$$x_I = A_I^{-1}(c_I - Bx_S)$$

The process that can easily be parallelized since $A_I^{-1} = \text{diag}(A_1^{-1}, A_2^{-1}, ..., A_p^{-1})$, and

$$(A_S - \hat{B}A^{-1}B)^{-1}x_S = c_S - \hat{B}A_I^{-1}c_I$$

The appendix Section A.6 shows the parallel program to solve this type of system in the symmetric positive definite case.

6 CONCLUSION

As we have examined over the course of this thesis, parallelism is valuable to many algorithms in mathematics. For larger sparse problems, it benefits the user to use parallelism concurrently with sparse data structures to reduce storage requirement, computational complexity, and time to run the program. In these instances iterative methods to solve sparse systems are beneficial and should be considered. We have seen that both shared memory and distributed memory can have an impact on the design and run time of an algorithm. It was examined how the arrangement of a matrix in domain decomposition can allow for some of the parallelism benefits discussed earlier. Also, it was tested that the results computed with MATLAB were in agreement with the parallel codes

It would be beneficial in the future to test these parallel iterative systems against each other. Here we could see how the timing of both Jacobi and Domain Decomposition methods would match, how the errors from the true solution behave, and how much space or storage is required. Also, it would be worth examining more of the iterative systems timings for various sizes and types of problems. It is easy to show that some systems will converge in situations that others may not.

REFERENCES

[1] Richard L. Burden. Boundary value problem. In *Numerical Analysis*, page 677, January 2011.

[2] John Burkardt. Jacobi.f90. In *The Jacobi Iteration for Linear Systems*, 2017. Accessed: 2017-02-18.

[3] Roberto Hibbler. Merge sort. In *Evaluation of the Merge Sort algorithm*, pages 1–3, 2017. Accessed: 2017-02-18.

[4] Max Lotkin. On the accuracy of runge-kutta's method. In *Mathematical tables and other aids to computation*, pages 128–133, 1951.

[5] Joseph E. Pasciak. More on the successive over relaxation method. In *MATH 693*, pages 1–3, 2017. Accessed: 2017-02-18.

[6] Roger B. Sidje. Sieve of eratosthenes. In *A Self-Starting Guide for the IBM SP2*, pages 26–28, January 1995.

A SOURCE CODES

A.1 Sieve MPI code implementation

```
program main
use mpi
parameter( BBMAX=2*1000000, NMAX=2*20*BBMAX)
integer :: id,master, numProc, ierr
integer :: N,M
integer :: N2, BB, B1, B2, L1, L2, counter, globalsum
integer,dimension(:) :: prime(BBMAX)
real :: start, finish
N=10000000
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! Initialize mpi
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, id, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, numProc, ierr)
  ! Display number of processors and range
  master = 0
  if(id.eq.master) then
    write(*,*) "numproc="
    write(*,*) numProc
    write(*,*) "range ="
    write(*,*) N
  end if
  ! Sends range to all processors
  call MPI_Bcast(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
  M=int(sqrt(dble(N)))
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! Splitting into sections
  N2 = N/2
  if(N2*2.ne.N) N2 = N2 + 1
  BB = mod(N2, 2*numProc)
  B1 = (N2-BB)/(2*numProc)
  B2 = B1
  if(id.lt.BB) then
    B1 = B1 + 1
    L1 = B1*id
```

```
  else
    L1 = B1*id +BB
  end if
  if(2*numProc-id-1.lt.BB) then
    B2 = B2 + 1
    L2 = B2*(2*numProc-id-1)
  else
    L2 = B2*(2*numProc-id-1) + BB
  end if
  BB = B1 + B2
  if(BB.gt.BBMAX) then
    write(*,*) "insuff space"
  end if
  do i = 1,BB
    prime(i) = 1
  enddo
  if(id.eq.0) prime(1) = 0
  ind = 1

  if(id.eq.master) then
    call cpu_time(start)
  end if
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ! Algorithm
  do
    if(id.eq.master) then
      actor = 0
      ind = ind + 1
      do while(ind.le.B1 .and. 2*(L1+ind)-1.le.M)
        if(prime(ind).ne.0) then
          actor = 2*(L1+ind)-1
          !write(*,*) "actor="
          !write(*,*) actor
          exit
        endif
        ind = ind + 1
      enddo
      if(actor.eq.0 .and. 2*(L1+ind)-1.le.M) then
        if(master.ne.lastp) actor = -(master+1)
      endif
    endif

    call MPI_Bcast(actor, 1, MPI_INTEGER, master, MPI_COMM_WORLD, ierr)

    if(actor.eq.0) exit
    if(actor.lt.0) then
```

```
      master = -actor
    else
      istart = (actor*actor+1)/2 - L1
      if(istart.le.0) then
        i = -istart/actor + 1
        istart = istart + i*actor
      endif
      do i = istart, B1, actor
        prime(i) = 0
      enddo
      istart = (actor*actor+1)/2 - L2
      if(istart.le.0) then
        i = -istart/actor+1
        istart = istart + i*actor
      endif
      do i = B1+istart,BB,actor
        prime(i) = 0
      enddo
    endif
  enddo

  counter = 0
  do i=1,BB
    if(prime(i).ne.0) then
      counter = counter + 1
!      if(i.le.B1) write(*,*) 2*(L1+1)-1
!      if(i.gt.B1) write(*,*) 2*(L2+i-B1)-1
    endif
  enddo

  call MPI_Reduce(counter, globalsum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD,ierr)

  if(id==master) then
    call cpu_time(finish)
    write(*,*) "Time elapsed = "
    write(*,*) finish-start
    write(*,*) "Time/numCores = "
    write(*,*) (finish-start)/numProc
    write(*,*) "With"
    write(*,*) numProc
    write(*,*) "processors"
  endif

  if(id.eq.0) then
    write(*,*) "Number of primes:"
    write(*,*) globalsum + 1
```

```
  endif
end
```

A.2 Merge Sort OpenMP code implementation

```
program sort
  integer i,j, error, n
  real, dimension(:), allocatable :: a
  integer, dimension(:), allocatable :: a_int
  integer :: arraySize, omp_get_num_procs
  double precision :: start, finish, omp_get_wtime
  integer :: numThread
  arraySize = 32768
  if(.not. allocated(a)) allocate(a(arraySize))
  if(.not. allocated(a_int)) allocate(a_int(arraySize))
  call RANDOM_NUMBER(a)
  a_int = 1000*a
  !write(*,*) a_int
  start = omp_get_wtime()
  numThread = omp_get_num_procs()
!$OMP PARALLEL DO
  do i=1,numThread
    call ms(a_int, (i-1)*arraySize/numThread, i*arraySize/numThread)
  end do
!$OMP END PARALLEL DO
  j=2
  do while(numThread/j.gt.0)
!$OMP PARALLEL DO
    do i=1,numThread/j
      call merg(a_int,(j*i-j)*arraySize/numThread,(j*i-j/2)*arraySize/numThread,(j*
    end do
!$OMP END PARALLEL DO
    j=j*2
  end do
  finish = omp_get_wtime()
  write(*,*) finish - start
  write(*,*) "list"
  write(*,*) numThread
  !write(*,*) a_int
end

recursive subroutine ms(a_int, first, last)
  integer,intent(inout) :: a_int(32768)
  integer :: first, last
```

```
  integer sub
  sub = last-first
  if(sub .lt. 2) then
    return
  else
    sub = sub/2
    call ms(a_int, first, first+sub)
    call ms(a_int, first+sub, last)
    call merg(a_int, first, first+sub, last)
  end if
end

subroutine merg(a_int, f, m, l)
   integer,intent(inout) :: a_int(32768)
   integer :: f, m, l
   integer tmp(32768)
   integer ftemp, mtemp, ltemp, x
   ftemp = f
   mtemp = m
   ltemp = f
   do while((ftemp.lt.m) .or. (mtemp .lt.l))
     if(ftemp .eq. m) then
        tmp(ltemp+1) = a_int(mtemp+1)
        mtemp=mtemp+1
     else if(mtemp .eq. l) then
       tmp(ltemp+1) = a_int(ftemp+1)
       ftemp=ftemp+1
     else if(a_int(ftemp+1) .lt. a_int(mtemp+1)) then
       tmp(ltemp+1) = a_int(ftemp+1)
       ftemp=ftemp+1
     else
       tmp(ltemp+1) = a_int(mtemp+1)
       mtemp = mtemp+1
     end if
     ltemp = ltemp+1
   end do
  do x=f,l-1
     a_int(x+1) = tmp(x+1)
   end do
end
```

A.3 PDE 1 matlab solution

```
function pde(eps, a, N, l)
```

```
h = 1/(N+1);
x = [h:h:(1-h)]';
y = x;
p = eps*l^2; A = -p; B = (1 - 2*a*A)/p; C = (2*eps*A - a*B)/p;
uexact2 = (A*x.^2 + B*x + C)*sin(l*y');

beta = -eps*N^2+a/2*N;
delta = -eps*N^2-a/2*N;
alpha = 4*eps*N^2;
gamma = -eps*N^2;

f = x.*(1 - p^2*x)*sin(l*y');
f(:,1) = f(:,1) - gamma*uexact2(:,1);
f(1,:) = f(1,:) - delta*uexact2(1,:);
f(:,N) = f(:,N) - gamma*uexact2(:,N);
f(N,:) = f(N,:) - beta*uexact2(N,:);

I = speye(N);
Tdiag = sparse(toeplitz([alpha, delta, zeros(1,N-2)], [alpha, beta, zeros(1,N-2)]))
Toff = sparse(toeplitz([0, gamma, zeros(1,N-2)]));
T = kron(I,Tdiag) + kron(Toff,I);
guess = ones(N^2,1);
tic;
[uap, ~, ~, ~] = jacobi(T,guess,f(:),500,0.0001);
%[uap, ~, ~, ~] = gs(T,guess,f(:),5000,0.0001);
%[uap, ~, ~, ~] = sor(T,guess,f(:),1.5,5000,0.0001);
%[uap, ~, ~, ~] = cg(T,guess,f(:),.01,5000,0.0001);
%[uap, ~, ~, ~] = gmres(T,guess,f(:),.01,10,1000,0.0001,N^2);
%uap = T\f(:);
toc;
size(f(:))
uap = reshape(uap,N,N);
dif = uexact2 - uap;

[X,Y] = meshgrid(x,y);
figure(1), surf(X,Y,uexact2);
figure(2), surf(X,Y,uap);
figure(3), surf(X,Y,dif);
```

A.4 PDE 1 FORTRAN Solution

```
program pde
  use mat
  implicit none
  !!!! DECLARING VARIABLES
```

```
!Input arguements
integer :: N, l
double precision :: eps, alp
!derived constants
double precision :: p, A, B, C
!5-point stencil coefficients
double precision :: beta, delta, alpha, gamm
!matrix handling
integer :: ierror, numProc, id
double precision :: h
double precision, dimension(:), allocatable :: x, y, xt, yt
integer :: i,j,k,q,st,st2,st3
!timing
double precision :: start, finish, JacStart, JacFin, omp_get_wtime
!initialize inputs
N = 3
l = 3
eps = 0.1d0
alp = 1d0
start = omp_get_wtime()
!allocate arrays or matrices
if(.not. allocated(x)) allocate(x(N))
if(.not. allocated(y)) allocate(y(N))
if(.not. allocated(xt)) allocate(xt(N))
if(.not. allocated(yt)) allocate(yt(N))
if(.not. allocated(true)) allocate(true(N,N))
if(.not. allocated(f)) allocate(f(N,N))
if(.not. allocated(bSol)) allocate(bSol(N*N))
if(.not. allocated(aA)) allocate(aA(5*N*N-4*N))
if(.not. allocated(jA)) allocate(jA(5*N*N-4*N))
if(.not. allocated(iA)) allocate(iA(N*N+1))
!Starting set up dimensions and PDE coefficients
h = 1.0d0/(N+1)
x = h*real([(i,i=1,size(x))], kind(h))
y = x
p = eps*l**2
A = -p
B = (1 - 2*alp*A)/p
C = (2*eps*A - alp*B)/p
!outer product to get true solution
xt = A*x*x + B*x + C
yt = sin(l*y)
true = spread(xt(1:N),dim=2,ncopies=N)*spread(yt(1:N),dim=1,ncopies=N)
!five point stencil coefficients
beta = -eps*N*N+alp/2*N
delta = -eps*N*N-alp/2*N
```

```
alpha = 4*eps*N*N
gamm = -eps*N*N
!right hand side vector creation
xt = x * (1 - p*p*x)
f = spread(xt(1:N),dim=2,ncopies=N)*spread(yt(1:N),dim=1,ncopies=N)
f(:,1) = f(:,1) - gamm*true(:,1)
f(1,:) = f(1,:) - delta*true(1,:)
f(:,N) = f(:,N) - gamm*true(:,N)
f(N,:) = f(N,:) - beta*true(N,:)
bSol = reshape(f,(/N*N/))
!LHS matrix creation
!first submatrix
aA(1:3) = (/alpha,beta,gamm/)
jA(1:3) = (/1,2,N+1/)
!$OMP PARALLEL DO
do i=2,N-1
  aA(4*i-4:4*i-1) = (/delta,alpha,beta,gamm/)
  jA(4*i-4:4*i-1) = (/i-1,i,i+1,N+i/)
end do
!$OMP END PARALLEL DO
aA(4*N-4:4*N-2) = (/delta,alpha,gamm/)
jA(4*N-4:4*N-2) = (/N-1,N,2*N/)
k=4*N-2
q=5*N-2
!middle
!$OMP PARALLEL DO
do j=2,N-1
  st = (j-2)*q+k+1
  aA(st:st+3) = (/gamm,alpha,beta,gamm/)
  jA(st:st+3) = (/(j-2)*N+1,(j-1)*N+1,(j-1)*N+2,j*N+1/)
  do i=2,N-1
    st2 = st+3+(i-2)*5+1
    aA(st2:st2+4) = (/gamm,delta,alpha,beta,gamm/)
    jA(st2:st2+4) = (/(j-2)*N+i,(j-1)*N+i-1,(j-1)*N+i,(j-1)*N+i+1,j*N+i/)
  end do
  st3 = k + q*(j-1) - 3
  aA(st3:st3+3) = (/gamm,delta,alpha,gamm/)
  jA(st3:st3+3) = (/(j-1)*N, j*N-1, j*N, (j+1)*N/)
end do
!$OMP END PARALLEL DO
! Bottom
st = st3+4
aA(st:st+3) = (/gamm,alpha,beta/)
st3 = N*(N-1)
jA(st:st+3) = (/st3-N+1,st3+1,st3+2/)
!$OMP PARALLEL DO
```

```
do i=2,N-1
  st2 = st+3+(i-2)*4
  aA(st2:st2+3) = (/gamm,delta,alpha,beta/)
  jA(st2:st2+3) = (/st3-N+i,st3+i-1,st3+i,st3+i+1/)
end do
!$OMP END PARALLEL DO
aA(5*N*N-4*N-2:5*N*N-4*N) = (/gamm,delta,alpha/)
jA(5*N*N-4*N-2:5*N*N-4*N) = (/st3,st3+N-1,st3+N/)
! iA portion construction
j=1
iA(1) = j
do i=2,N*N
  if(i-1.le.N .or. i-1.ge.N*(N-1)+1) then
    if(mod(i-1,N).le.1) then
      j=j+3
    else
      j=j+4
    end if
  else
    if(mod(i-1,N).le.1) then
      j=j+4
    else
      j=j+5
    end if
  end if
  iA(i) = j
end do
iA(N*N+1) = 5*N*N - 4*N +1
finish = omp_get_wtime()
write(*,*) "pre Jacobi"
write(*,*) finish-start
!Call parallel Jacobi
start = omp_get_wtime()
call jacobiCRS
finish = omp_get_wtime()
write(*,*) finish-start
!write Solution
write(*,*) "A"
write(*,*) aA
write(*,*) jA
write(*,*) iA
write(*,*) "b"
write(*,*) bSol
write(*,*) "x"
write(*,*) xSol
!determine error
```

```
  f = reshape(xSol, (/N,N/))
  f = abs(f-true)
  !write(*,*) f
  write(*,*) norm2(f)
  !write(*,*) "true"
  !write(*,*) true
end program
```

A.5 Linear Shooting

```
program linshoot
implicit none
  integer :: i,j,N
  double precision :: a, b, alph, beta, h, t, p1, p2, temp, x
  double precision :: funct, functy, functyp
  double precision, dimension(:,:) :: k(4,2), khat(4,2)
  double precision, dimension(:), allocatable :: w1, w2, u1, u2
    a = 1
    b = 2
    alph = 1
    beta = 2
    N = 10
    h = (b-a)/N
    t = a

    if(.not. allocated(w1)) allocate(w1(N+1))
    if(.not. allocated(w2)) allocate(w2(N+1))
    if(.not. allocated(u1)) allocate(u1(N+1))
    if(.not. allocated(u2)) allocate(u2(N+1))
    w1(1) = alph
    w2(1) = 0
    u1(1) = 0
    u2(1) = 1

        do i=2,N+1
          x=a+(i-2)*h
          k(1,1)=h*w2(i-1)
          k(1,2)=h*funct(x,w1(i-1),w2(i-1))
          k(2,1)=h*(w2(i-1)+1/2.*k(1,2))
          k(2,2)=h*funct(x+h/2,w1(i-1)+1/2.*k(1,1),w2(i-1)+1/2.*k(1,2))
          k(3,1)=h*(w2(i-1)+1/2.*k(2,2))
          k(3,2)=h*funct(x+h/2,w1(i-1)+1/2.*k(2,1),w2(i-1)+1/2.*k(2,2))
          k(4,1)=h*(w2(i-1)+k(3,2))
          k(4,2)=h*funct(x+h,w1(i-1)+k(3,1),w2(i-1)+k(3,2))
```

```
          w1(i)=w1(i-1)+(k(1,1)+2*k(2,1)+2*k(3,1)+k(4,1))/6
          w2(i)=w2(i-1)+(k(1,2)+2*k(2,2)+2*k(3,2)+k(4,2))/6

          khat(1,1)=h*u2(i-1)
          khat(1,2)=h*(functy(x,w1(i-1),w2(i-1))*u1(i-1)&
            +functyp(x,w1(i-1),w2(i-1))*u2(i-1))
          khat(2,1)=h*(u2(i-1)+1/2.*khat(1,2))
          khat(2,2)=h*(functy(x+h/2,w1(i-1),w2(i-1))*(u1(i-1)+1/2.*khat(1,1))&
            +functyp(x+h/2,w1(i-1),w2(i-1))*(u2(i-1)+1/2.*khat(1,2)))
          khat(3,1)=h*(u2(i-1)+1/2.*khat(2,2))
          khat(3,2)=h*(functy(x+h/2,w1(i-1),w2(i-1))*(u1(i-1)+1/2.*khat(2,1))&
            +functyp(x+h/2,w1(i-1),w2(i-1))*(u2(i-1)+1/2.*khat(2,2)))
          khat(4,1)=h*(u2(i-1)+khat(3,2))
          khat(4,2)=h*(functy(x+h,w1(i-1),w2(i-1))*(u1(i-1)+khat(3,1))&
            +functyp(x+h,w1(i-1),w2(i-1))*(u2(i-1)+khat(3,2)))
          u1(i)=u1(i-1)+(khat(1,1)+2*khat(2,1)+2*khat(3,1)+khat(4,1))/6
          u2(i)=u2(i-1)+(khat(1,2)+2*khat(2,2)+2*khat(3,2)+khat(4,2))/6
        end do
    p1 = alph
    p2 = (beta - w1(N+1))/(u1(N+1))
    do i=1,N
      write(*,*) a + (i-1)*h
      write(*,*) w1(i) + p2*u1(i)
      write(*,*) w2(i) + p2*u2(i)
    end do
end


real function funct(x,y,yp)
  real, intent(in) :: x,y,yp
  funct=yp + 2*y + cos(x)
end function

real function functy(x,y,yp)
  real, intent(in) :: x,y,yp
  functy=-1/8.*(yp)
end function

real function functyp(x,y,yp)
  real, intent(in) :: x,y,yp
  functyp=-1/8.*y
end function
```

A.6 Domain Decomposition

```
program DDSYS
implicit none
double precision, dimension(:,:), allocatable :: A,B,C
integer i,j,n,m,p,q
interface
  subroutine DDSLV(A,B,C,m,p,q)
  double precision, dimension(:,:) :: A,B,C
  integer m,p,q
  end subroutine DDSLV
end interface

p=10
m=5
q=4
n=m*p+q

ALLOCATE(A(m,n),B(m,q*p),C(m,p+1))
!if(.not. allocated(A)) allocate(A(m,n))
!if(.not. allocated(B)) allocate(B(m,q*p))
!if(.not. allocated(C)) allocate(C(m,p+1))

do i = 1, m-1
  A(i,i+1:n:m)=-1
  A(i,i:n:m)=2.0d0
  A(i+1,i:n:m)=-1.0d0
end do

A(m,i:n:m)=2.0d0
B=1.0d0
C=1.0d0

write(*,*) 'A='
do i=1,m
  write(*,*) (A(i,j),j=1,n)
end do
write(*,*) 'B='
do i=1,m
  write(*,*) (B(i,j), j=1, q*p)
end do

call DDSLV(A,B,C,m,p,q)

do i=1,m
  write(*,*) (C(i,j), j=1,p+1)
```

```
end do

end program DDSYS

!-----------------------------------------------------

subroutine DDSLV(A,B,C,m,p,q)
implicit none
!inputs
double precision, dimension(:,:) :: A,B,C
integer m,p,q

integer n,k,info,ipiv(1:m), i, j
double precision, dimension(m,q+1) :: Y

write(*,*) "starting"

n=m*p+q
do k=1,p
  call DPOTRF('L', m, A, m, info)
  write(*,*) "pot"
  Y(:,1:q)=B(:,q*(k-1)+1:q*k)
  Y(:,q+1)=C(:,k)
  write(*,*) "Y"

  call DTRSM('L','L','N','N',m,q+1,1.,A(1,m*(k-1)+1),m,Y,m)
  Y(1:q,1:q+1)=MATMUL(Transpose(Y(:,1:q)),Y)
  A(1:q,n-q+1:n)=A(1:q,n-q+1:n)-Y(1:q,1:q)
  C(1:q,p+1)=C(1:q,p+1)-Y(1:q,q+1)
end do

do i=1,q
  write(*,*) (A(i,j),j=n-q+1,n),C(i,p+1:p+1)
end do

call DSYSV('L',q,1,A(1,n-q+1),m,ipiv,C(1,p+1),m,Y,m*(q+1),info)

write(*,*) 'C'
do i=1,m
  write(*,*) (C(i,j), j=1,p+1)
end do

do k=1,p
  C(:,k:k) = C(:,k:k)-MATMUL(B(:,q*(k-1)+1:q*k),C(:,p+1:p+1))
  call DPOTRS('L', m,1,A(1,m*(k-1)+1),m,C(1,k),m,info)
end do
```

```
end
```