CS511 - Final Project - Jacobi/Gauss-Seidel Parallelization

Matthew Holman, Paden Rumsey - Department of Computer Science

I. Abstract

***Abstract*—For this project, we implemented two different calculation methods for solving a 'boundary condition' matrix problem. This problem was solved using both the Jacobi and the Gauss-Seidel methods for updating indices the matrix. Paden Rumsey (hereafter P.R.) implemented the Jacobi method, and Matthew Holman (hereafter M.H.) implemented the Gauss-Seidel method. The order in which the matrix indices were updated was dictated in one of two different ways. The first ordering scheme was an 'iterative' method that traversed the matrix in a row-wise, top to bottom fashion. The second ordering scheme was a 'ring' method, which involved solving the matrix calculations in such a way that they traversed a series of concentric 'rings' from the outside border of the matrix to the inside.**

A total of seven programs were created. The Gauss-Seidel and Jacobi schemes both feature serial and parallel 'ring' implementations, as well as serial 'iterative' implementations for both. The Gauss-Seidel method was also implemented as a parallel 'iterative' program as well. The 'parallel' programs were created using OpenMP. We then collected timing data and the number of iterations needed for each implementation's matrix values to converge such that the amount of change (called 'error' in the code) between matrix iterations fell below a certain threshold that was kept constant between all the runs of all the programs. We then plotted the data and compared the performance behaviors of the various programs.

The results we observed suggest that our parallel versions of these programs are less efficient than the iterative versions for matrices smaller than 100×100. However, our observations do indicate that speedup does occur when utilizing more threads when running the parallel solutions.

***Keywords*—Gauss-Seidel method, Jacobi method, matrix, OpenMP, threads**

Introduction

The 'boundary condition' problems we solved featured a matrix consisting of low initial values which was enclosed by a 'ring' of higher values. It was these higher values which we wished to progressively distribute throughout the rest of the matrix. The distribution method could be thought of as a ring of hot material whose heat slowly 'leaches' inward through a plane of colder material. The indices in the matrix to which the 'heat' is distributed is known as the 'relaxation mesh.' Our goal was to have all the values in the relaxation mesh be updated in such a way that the values on the boundary propagated towards the center and all the values in the relaxation mesh were close to or matched the boundary values.

There are likely many different ways to solve this problem. We elected to test two methods of calculating the matrix indices, using two different approaches to ordering those calculations. The calculation methods we used are called the 'Gauss-Seidel' method, and the 'Jacobi' method. The first ordering approach is an 'iterative' method of updating the relaxation mesh. The second ordering approach is one wherein the mesh is updated in an order determined by the placement of concentric 'rings.'

Calculation Methods

The 'Gauss-Seidel' method used in the serial versions of our programs to calculate the matrix indices can be expressed in the following line of code:

```
x[i] = 0.25 * (x[i-1] + x[i+1] + x[i-
    ↪ matrix_size] + x[i+matrix_size
    ↪ ]);
```

Here, `x[i]` refers to the matrix index being updated, while `x[i-1]` refers to the matrix index to the left of the current index and `x[i+1]` refers to the matrix index to the right. `x[i-matrix_size]` refers to the index above the current index, while `x[i+matrix_size]` refers to the index below. These values are summed and then multiplied by 0.25, which has the effect of averaging the four values. This average is then assigned to the matrix index under consideration.

The 'Jacobi' method of matrix index calculation is very similar to the 'Gauss-Seidel' method:

```
x[i] = 0.25 * (last_x[i-1] + last_x[i
    ↪ +1] + last_x[i-matrix_size] +
    ↪ last_x[i+matrix_size]);
```

The primary difference between the 'Gauss-Seidel' method and the 'Jacobi' method, is that the 'Jacobi' method takes advantage of the values in the previous iteration of the matrix, while the 'Gauss-Seidel' method only uses the values in the matrix currently under consideration. This means that the 'Gauss-Seidel' method could potentially return different values when calculating the matrix indices than the 'Jacobi' method would, and vice versa.

Ordering Approaches

To understand the two approaches to calculation ordering, consider the following 'initial' matrix:

```
10      10      10      10      10
10      1       1       1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10
```

Here, the values of '10' surrounding the mesh of '1s' are the 'boundary condition values' that we wish to propagate through the mesh.

In a serialized program with 'iterative' ordering, the calculations used to propagate the boundary condition values through the relaxation mesh occur in the following order:

```
xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      4       5       6       xx
xx      7       8       9       xx
xx      xx      xx      xx      xx
```

Notice, that the order in which the calculations take place is in an 'iterative' fashion from the top left to the bottom right of the relaxation mesh.

Considering the same initial matrix used to demonstrate the 'iterative' approach, a serialized 'ring' approach to calculation ordering. A serialized program with 'ring' ordering would carry out the calculations in the following order:

```
xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      4       9       5       xx
xx      6       7       8       xx
xx      xx      xx      xx      xx
```

Here, that the 'rings' in the 'ring' ordering method consist of sequences of matrix indices inside the mesh which form a two-dimensional 'ring'-like shape. In the case above, the outer 'ring' consists of the indices { **1, 2, 3, 4, 5, 6, 7, 8** }, while the inner 'ring' consists of index { **9** }

We believe these two approaches can be modified and improved with parallel programming techniques due to the underlying structure of the problem and its solution. This report focuses on analyzing the effectiveness of our implementations of parallelization in these algorithms.

SOLUTION

The seven programs we created can be divided into two sets: the Jacobi programs, and the Gauss-Seidel programs. Because each set of programs was created by a different author, the underlying implementations for the programs in each set will be similar. However, the implementations of the programs *between* sets will be different. Therefore, performance comparisons are most appropriately done between programs in the same set.

Both the serial and parallel versions of the Gauss-Seidel programs take advantage of similar implementations between programs. This means that the Gauss-Seidel serial programs are similar to one another, and the Gauss-Seidel parallel programs are also similar. The differences between each parallel or serial program in each Gauss-Seidel subset comes by way of the ordering of matrix indices in the matrix update ordering array, '`o[]`.' The Jacobi serial and parallel programs implemented by P.R. use different strategies to complete similar goals.

The seven program implementations we created are as follows:

1) Gauss-Seidel Serial Version, Iterative Ordering
2) Gauss-Seidel Serial Version, Ring Ordering
3) Gauss-Seidel Parallel Version, Iterative Ordering
4) Gauss-Seidel Parallel Version, Ring Ordering
5) Jacobi Serial Version, Iterative Ordering
6) Jacobi Serial Version, Ring Ordering
7) Jacobi Parallel Version, Ring Ordering

Gauss-Seidel and Jacobi Method - Serial Versions

For all the serial programs listed above, the values in the matrices are changed using the specified calculation method. Further, the order of these calculations is determined by the ordering scheme under consideration. These programs use only a single thread in their implementations, and serve as a baseline for comparison against the parallel versions of these programs.

Gauss-Seidel - Parallel Versions, Both Orderings

II. GSPVBO

Both Gauss-Seidel parallel programs implemented by M.H. both take advantage of a 'cyclic distribution' method of assigning calculation tasks to the threads. The only true difference between the 'iterative' version and the 'ring' version is the ordering of matrix indices inside the 'update ordering array' that the parallelized function accepts as an argument. Therefore any discussion that applies to the working of one program applies to both.

The 'cyclic distribution' method used in the Gauss-Seidel programs is illustrated in the following code which is run in parallel by each thread:

```
int ID = omp_get_thread_num();
int nthrds = omp_get_num_threads();
...
for (m = ID; m < nOA_size; m = m +
    ↪ nthrds){
     i = o[m];
     ...
}
```

Here, the starting index in the loop corresponds to the ID (sometimes called 'rank') of the thread. The variable '`nthrds`' contains the number of threads requested by the program. The variable '`nOA_size`' contains the size of the one-dimensional array that contains the matrix update ordering. Indexing into the matrix update ordering array, '`o[]`,' inside the 'for' loop with the variable '`m`' in this way allows each thread to receive a comparable number of calculations to perform.

For example, a Gauss-Seidel program requesting four threads and using the 'iterative' update approach would update the matrix in the following fashion, where the numbers in the matrix indicate the thread ID:

```
xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      4       1       2       xx
xx      3       4       1       xx
xx      xx      xx      xx      xx
```

A parallelized Gauss-Seidel implementation requesting four threads and using the 'ring' update approach would update the matrix as follows:

```
xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      4       1       1       xx
xx      2       3       4       xx
xx      xx      xx      xx      xx
```

As with the serial versions of our programs, the values in the matrices were updated in an order determined by the ordering scheme being considered. Additionally, although the general idea of the Jacobi vs. Gauss-Seidel methods for updating the matrix indices still hold, there is an additional consideration for the Gauss-Seidel method.

The consideration affecting a parallel Gauss-Seidel method relates to the order in which different threads update the 'current' matrix. Different thread update orderings may lead to some matrix indices receiving different values than they otherwise would in a serial version.

To illustrate this, consider a single update iteration to the following matrix which is shared, and being updated by, two different threads:

```
Iteration 0 - 'current'
    ↪ matrix at calculation
    ↪ start

10      10      10      10      10
10      *1*     *1*     1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10
```

The indices to be updated above are indicated by asterisks. Because the above matrix is a resource shared between both threads, and the order that the threads update the matrix cannot be directly specified, there are three different possible outcomes to this iteration. The first possibility is:

```
Iteration 1 - thread 0
    ↪ updates first

10      10      10      10      10
10      5.5     *1*     1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10

Followed by thread 1

10      10      10      10      10
10      5.5     4.375   1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10
```

The second possibility is:

```
Iteration 1 - thread 1
    ↪ updates first

10      10      10      10      10
10      *1*     3.25    1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10

Followed by thread 0

10      10      10      10      10
10      6.063   3.25    1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10
```

The third possibility occurs when threads 0 and 1 update simultaneously. This assumes that neither '1' in the 'current' matrix is updated before either thread's calculations are made:

```
Iteration 1 - threads 0 & 1
    ↪ update simultaneously

10      10      10      10      10
10      5.5     3.25    1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10
```

M.H. attempted to mitigate this consideration by providing each thread its own unique copy of the 'current' matrix, and then updating a separate copy of the 'current' matrix that is shared between all the threads. This shared 'current' matrix is then copied back into each thread's unique copy of the 'current' matrix for use in the next iteration.

For example, consider two iterations updating the same initial matrix used in the previous example with the 'iterative' approach to updating. This matrix is shown being updated by two threads using the 'cyclic distribution' method of assigning calculations to each thread:

```
Initial matrix

10      10      10      10      10
10      1       1       1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10
```

```
Iteration 0 - thread 0

10      10      10      10      10
10      *1*     1       *1*     10
10      1       *1*     1       10
10      *1*     1       *1*     10
10      10      10      10      10

10      10      10      10      10
10      5.5     1       5.5     10
10      1       1       1       10
10      5.5     1       5.5     10
10      10      10      10      10
```

```
Iteration 0 - thread 1

10      10      10      10      10
10      1       *1*     1       10
10      *1*     1       *1*     10
10      1       *1*     1       10
10      10      10      10      10

10      10      10      10      10
10      1       3.25    1       10
10      3.25    1       3.25    10
10      1       3.25    1       10
10      10      10      10      10
```

```
Iteration 0 - thread 0 and
    ↪ thread 1 combined

10      10      10      10      10
10      5.5     3.25    5.5     10
10      3.25    1       3.25    10
10      5.5     3.25    5.5     10
10      10      10      10      10
```

```
Iteration 1 - thread 0

10      10      10      10      10
10      *5.5*   3.25    *5.5*   10
10      3.25    *1*     3.25    10
10      *5.5*   3.25    *5.5*   10
10      10      10      10      10

10      10      10      10      10
10      6.63    3.25    6.63    10
10      3.25    3.25    3.25    10
10      6.63    3.25    6.63    10
10      10      10      10      10
```

```
Iteration 1 - thread 1

10      10      10      10      10
10      5.5     *3.25*5.5       10
10      *3.25*1         *3.25*10
10      5.5     *3.25*5.5       10
10      10      10      10      10

10      10      10      10      10
10      5.5     5.5     5.5     10
10      5.5     1       5.5     10
10      5.5     5.5     5.5     10
10      10      10      10      10
```

```
Iteration 1 - thread 0 and
    ↪ thread 1 combined

10      10      10      10      10
10      6.63    5.5     6.63    10
10      5.5     3.25    5.5     10
10      6.63    5.5     6.63    10
10      10      10      10      10
```

Please note that although the above strategy somewhat mitigates the possibility of updating the 'current' matrix in an unpredictable way, it still allows for the characteristic manner of updating that the Gauss-Seidel method uses. However, this effect is constrained to matrix indices which would be affected by the individual thread that is assigned to them. For example, consider the same initial matrix used in the previous example being updated by three threads instead of two while using the 'iterative' update ordering approach:

```
Initial matrix

10      10      10      10      10
10      1       1       1       10
10      1       1       1       10
10      1       1       1       10
10      10      10      10      10

Update order for indicated threads:
{6, 7, 8, 11, 12, 13, 16, 17, 18}

xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      1       2       3       xx
xx      1       2       3       xx
xx      xx      xx      xx      xx
```

```
Thread 1 updates:
{*6*, *11*, *16*}

10      10      10      10      10
10      5.5     1       1       10
10      4.375   1       1       10
10      6.344   1       1       10
10      10      10      10      10
```

```
Thread 2 updates:
{*7*, *12*, *17*}

10      10      10      10      10
10      1       3.25    1       10
10      1       1.563   1       10
10      1       3.391   1       10
10      10      10      10      10
```

```
Thread 3 updates:
{*8*, *13*, *18*}

10      10      10      10      10
10      1       1       5.5     10
10      1       1       4.375   10
10      1       1       6.344   10
10      10      10      10      10
```

```
All threads combined

10      10      10      10      10
10      5.5     3.25    5.5     10
10      4.375   1.563   4.375   10
10      6.344   3.391   6.344   10
10      10      10      10      10
```

Although the above strategy does not completely remove the effect of the Gauss-Seidel calculation as it applies to the 'current' matrix, it does lessen the impact of different threads updating the matrix in different orders. A similar protective effect occurs using the 'ring' update approach as well.

Jacobi - Parallel Version, Ring Ordering

The way that P.R. chose to implement his solution was by 'slicing' the Jacobi matrix amongst its rings. The work for each ring was calculated and split amongst the variety of threads. This is done by utilizing the **omp_get_thread_num** function. The amount of work to be done is calculated using the starting and ending indices for each of the rings. If the work in the ring requires fewer than the number of available threads, then

those threads skip the whole process and wait at the next barrier.

```
Example: 4 Threads - 5 Dim. Matrix

In a 5 dimensional matrix the first
ring has 16 values, the second
has 8 and the last has 1

Ring One:    XXXX|XXXX|XXXX|XXXX
               1   2    3    4
Update

Ring Two:    XX|XX|XX|XX
             1  2  3  4
Update

Ring Three:  X
             1
Update

Do it all again until overflow
or max iterations is reached
```

P.R. determined the above strategy to be a logical way to parallelize the algorithm. The interior matrix indices needed to have values propagated to *their* boundaries. As such, P.R. could not simply 'split up' the entire matrix. P.R. needed to proceed ring by ring so he could propagate the information inward. P.R. believed that failing to do so would result in many more iterations needed to converge to the desired matrix values than the serialized version would.

EXPERIMENTAL METHODOLOGY

To collect data on our programs, we conducted two different experiments. The first experiment was a comparison between the seven different programs. These programs were compared in terms of the number of iterations required for the matrix to converge, as well as the execution time times those program runs took. The second experiment was a measure of the speedup between differing number of threads in the parallel versions of the programs with 'ring' ordering.

In the first experiment, the trials were parameterized using different matrix sizes and differing thread counts (for the parallel implementations) for each trial. The thread counts ranged from one to four, and the matrices were in dimensions from 10×10 to 100×100 for each graph. The graph dimensions were incremented at a granularity of one for each trial. Therefore, a total of $4 \times 90 = 360$ trials were conducted for each program.

Figures 1, 3, 5, and 7 show the number of iterations per matrix size each implementation took to converge.

Figures 2, 4, 6, and 8 show the execution time, measured in μs, it took each implementation to converge for each matrix dimension. The following code excerpt demonstrates where the starting and stopping time for each trial is executed in each of the seven programs:

```
gettimeofday(&tvStart, &tzp);
result = JacobiBC(...)
gettimeofday(&tvEnd, &tzp);
```

In the previous code example, the function 'JacobiBC' returns the number of iterations it took for the program to converge, and stores this value in the variable 'result.' This function, being enclosed by the two calls to 'gettimeofday' is therefore the only function whose timing is being considered.

In the second experiment, only the parallel versions of the Gauss-Seidel and Jacobi 'ring' ordering implementations were compared. These comparisons were made using larger matrix dimensions than the first experiment. As with the first experiment, we collected data for the time, measured in μs, for the matrices to converge. This data was collected for 1 to 4 threads, with matrix dimensions ranging between 10×10 to 300×300. As such, a total of $4 \times 290 = 1160$ trials were conducted for each program. Our results as shown in **Figures 9 and 10**.

With the exception of the Gauss-Seidel parallel version program with iterative ordering, all of the experiments were run on a computer with a Linux Operating system, and a quad core processor. All the programs were compiled using the following compiler options:

g++ -03 -fopenmp

To collect the data for the Gauss-Seidel parallel version program with iterative ordering, M.H.'s used a pc with the Windows 10 operating system. M.H. also used the C++ programming language which was compiled with the g++ compiler found in version 8.1.0 distribution of mingw. This compiler features a Windows compatible versions of the omp.h library. M.H.'s programs were compiled using the compiler options noted above.

PROBLEMS

General: Gauss-Seidel Method

The most difficult aspect of this part of the project was programming with OpenMP in general. In order to keep the parallel programs from 'hanging,' it was necessary to ensure that all the threads were synchronized at `#pragma omp barrier` constructs, and that *all* the threads were counted by the program as being ended before leaving the parallelized section. This last requirement necessitated the use of arrays which contained information for each thread, indexed to that thread's ID.

Gauss-Seidel Ring Method: Parallel Version

As this was the first program of to be parallelized by M.H. all of the 'general' problems mentioned above were discovered with this program. These problems were applicable to both parallel programs.

Gauss-Seidel Iterative Method: Parallel Version

Determining the code to generate a proper ordering of the matrix points that takes the enclosing border into account was challenging.

Jacobi Ringed Method: Serial Version

There was a bug in this implementation that would hamper odd sized matrices convergence times. P.R. fixed this bug and applied it to the parallel version as well.

Jacobi Ringed Method: Parallel Version

P.R. assumed that this implementation would be fairly easy. Unfortunately, this did not turn out to be the case. P.R. used OpenMP barriers in a variety of locations. P.R. did so to synchronize the threads threads before they could move on to calculate a future 'ring' in the matrix. This means that the calculations required by the previous ring had to be completed before the threads could proceed. Additionally, the threads could not move on to the next ring if there was an 'overflow' - meaning a matrix value that exceeded a predefined limit.

The aforementioned issues resulted in a number of 'hanging' errors in the program that needed to be tracked down. One such error was when P.R. had some threads 'skip' the implementation because there were fewer indices than threads. This caused some threads to move past the barrier, while the others were blocked at the barrier. Additionally, it was difficult to calculate the amount of work needing to be done for each 'ring,' and then distribute that work amongst the various threads. P.R. believes there is a race condition still present in the Jacobi code. This is because the number of iterations required for the parallel version of the Jacobi 'ring' code to converge can sometimes be less than that of the serial version of the ringed Jacobi program.

RESULTS

For the first experiment, the numbers of iterations it takes for the matrices in our implementations to converge can vary from what we would otherwise expect to see. Using more than one thread, we would expect our parallel implementations to converge in fewer iterations than the serial versions. However, for all but 3 threads, the Jacobi parallel iterations are equal to the Jacobi iterative iterations. For the Gauss-Seidel parallel programs, these implementations seemingly always require *more* iterations to converge than their serialized counterparts. Additionally, the number of iterations required for the Gauss-Seidel 'iterative' program to converge varies widely as the matrix size increases over two or more threads. However, the number of required iterations for the parallel Gauss-Seidel 'iterative' program shows a generally increasing trend, regardless of the number of threads being used.

As for the time analysis in the first experiment, in general the Jacobi parallel 'ring' method code takes the longest time to converge. The parallelized Gauss-Seiedel 'ring' implementaiton also take a significant amount of time to converge. The timing for the Gauss-Seidel 'iterative' implementation has timings that match the number of iterations it takes the matrix to converge depending on the matrix size. As with the data for the iterations required for this program to converge, the timing data generally shows an upward trend as well. The serialized Jacobi 'ring' and 'iterative' programs typically takes the least amount of time to converge.

For the second experiment, the single threaded implementation in the Gauss-Seidel 'ring' program converges the most quickly until the matrix size reaches approximately 180×180 - then it converges the most slowly. The Gauss-Seidel programs using two, three, and four threads appear to converge in more or less the same amount of time, regardless of the matrix size.

As for the Jacobi 'ring' program, the runs with either one thread or four threads are typically the slowest to converge. The trials using two and three threads are faster than those using one or four. Before the matrix size reaches approximately 270×270, the run using three threads converged faster than the run that used two.

DATA

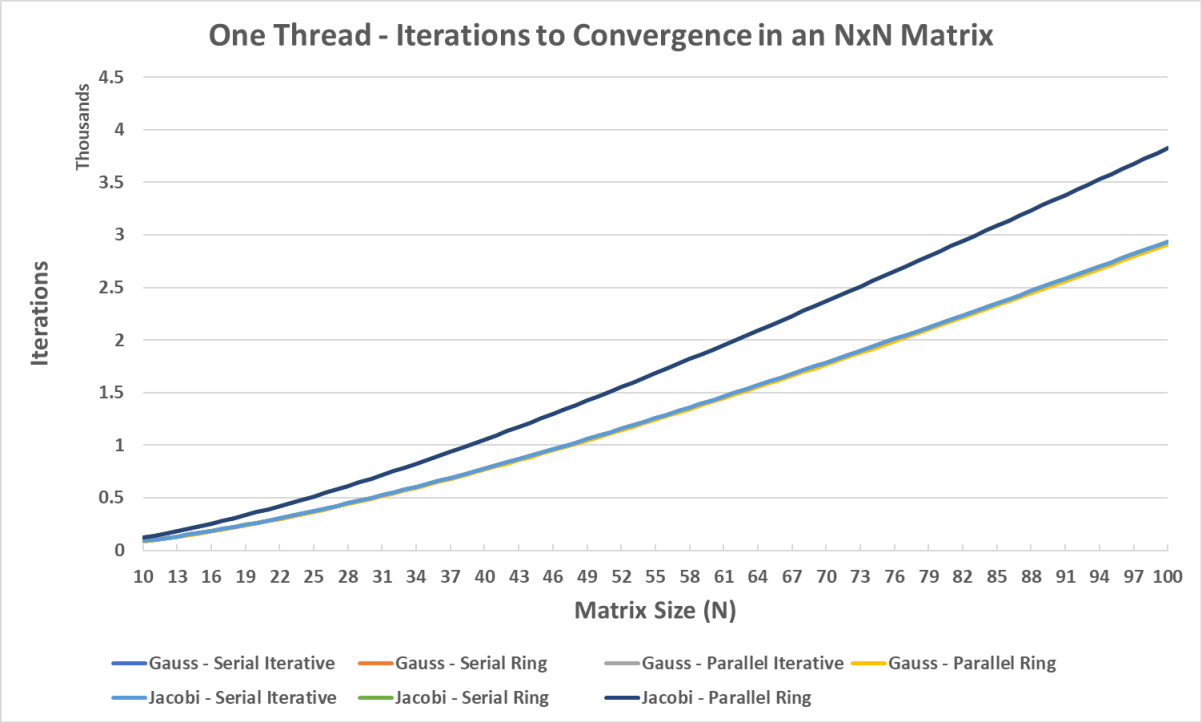

Figure 1: Experiment 1

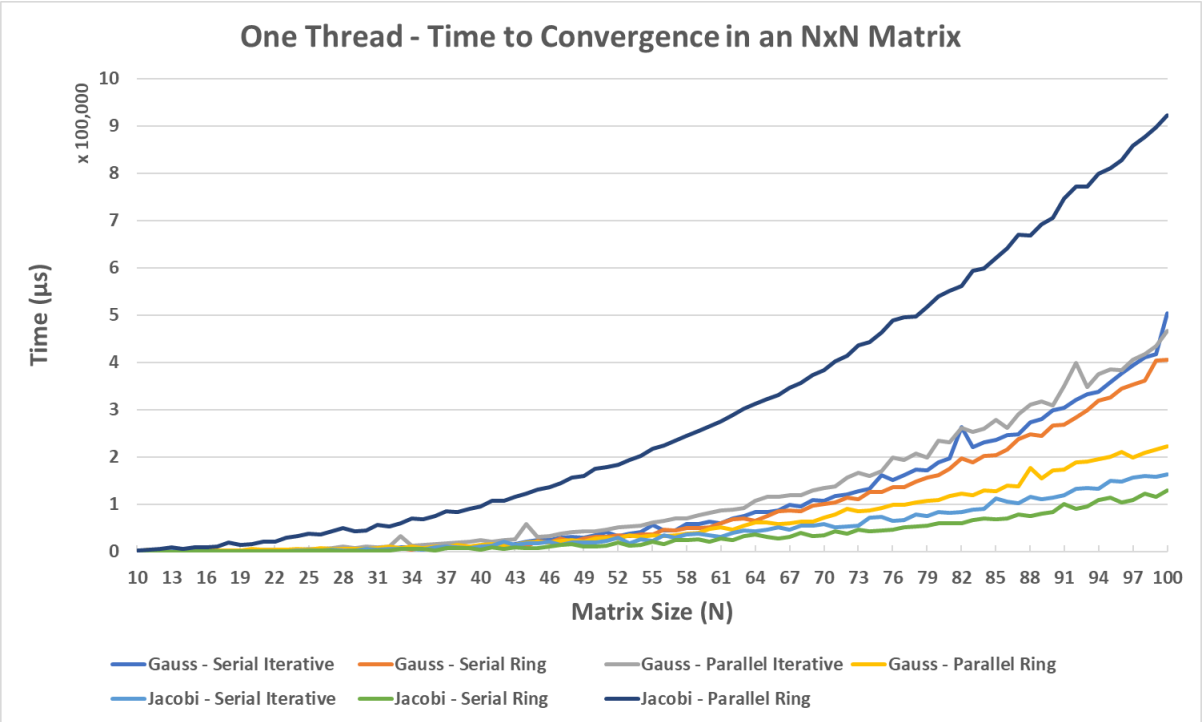

Figure 2: Experiment 1

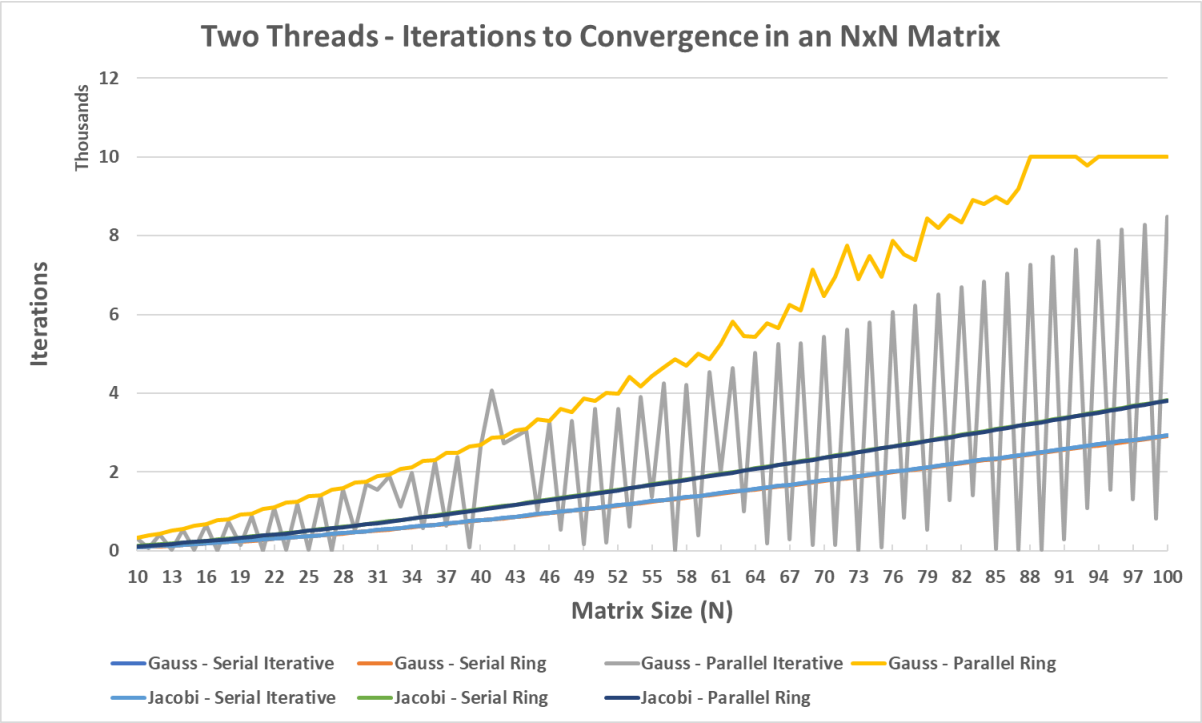

Figure 3: Experiment 1

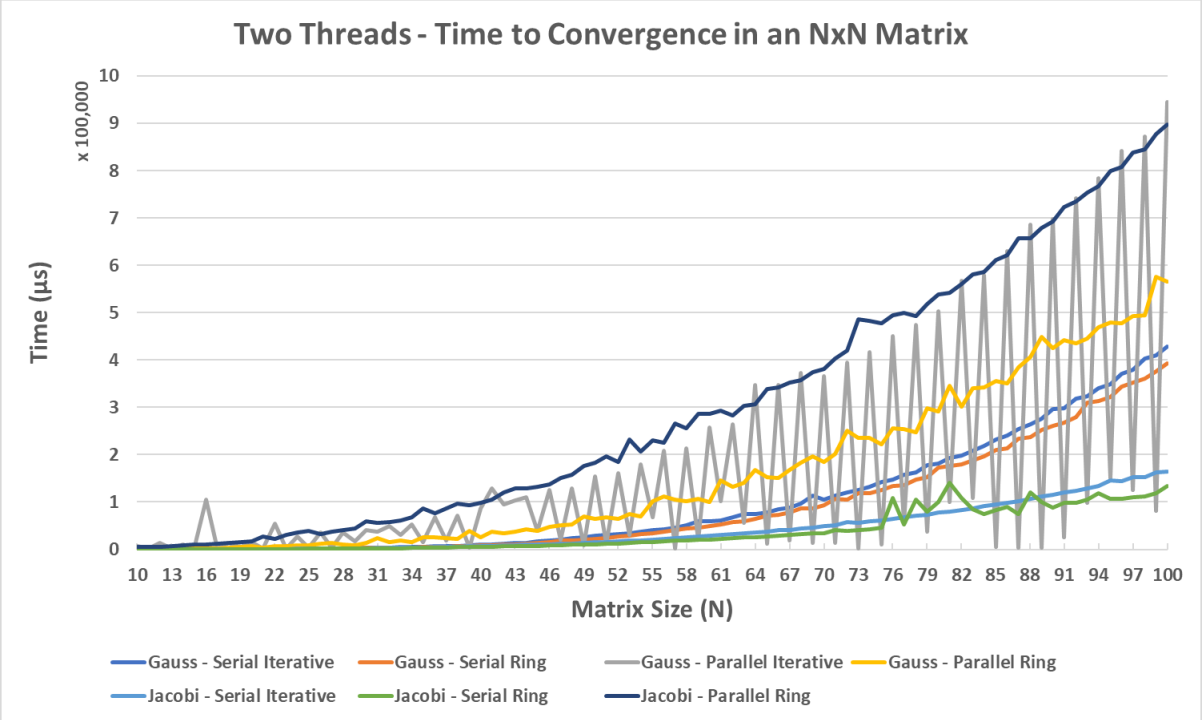

Figure 4: Experiment 1

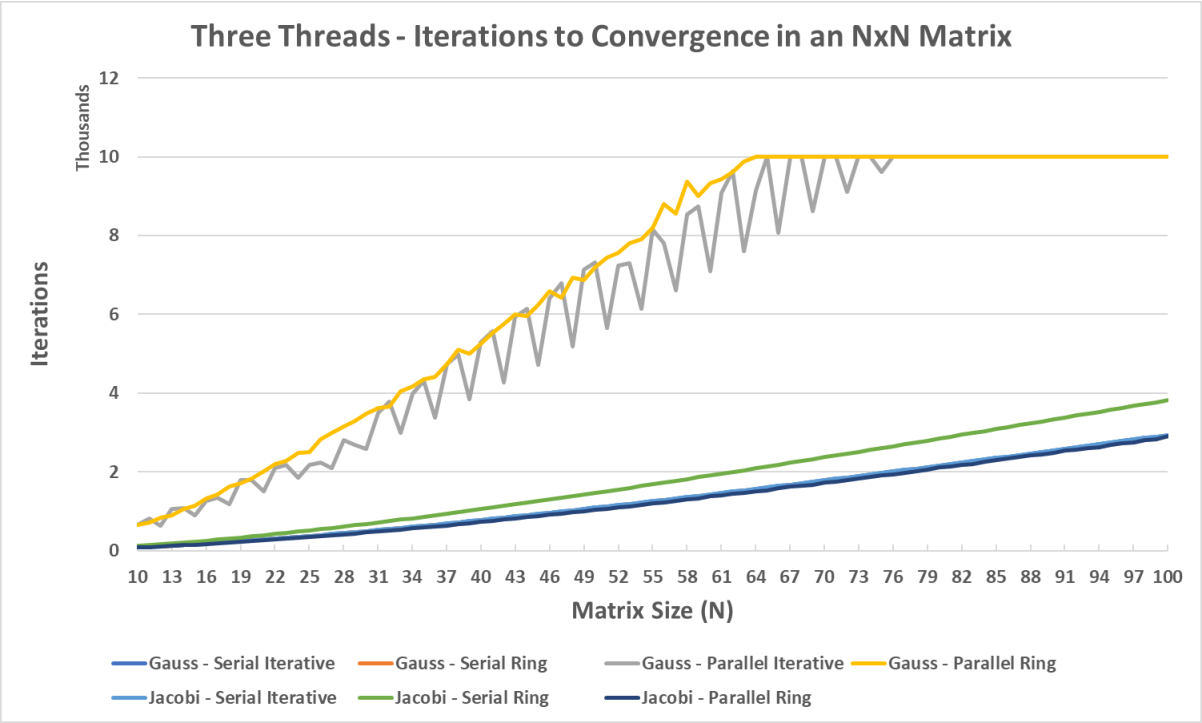

Figure 5: Experiment 1

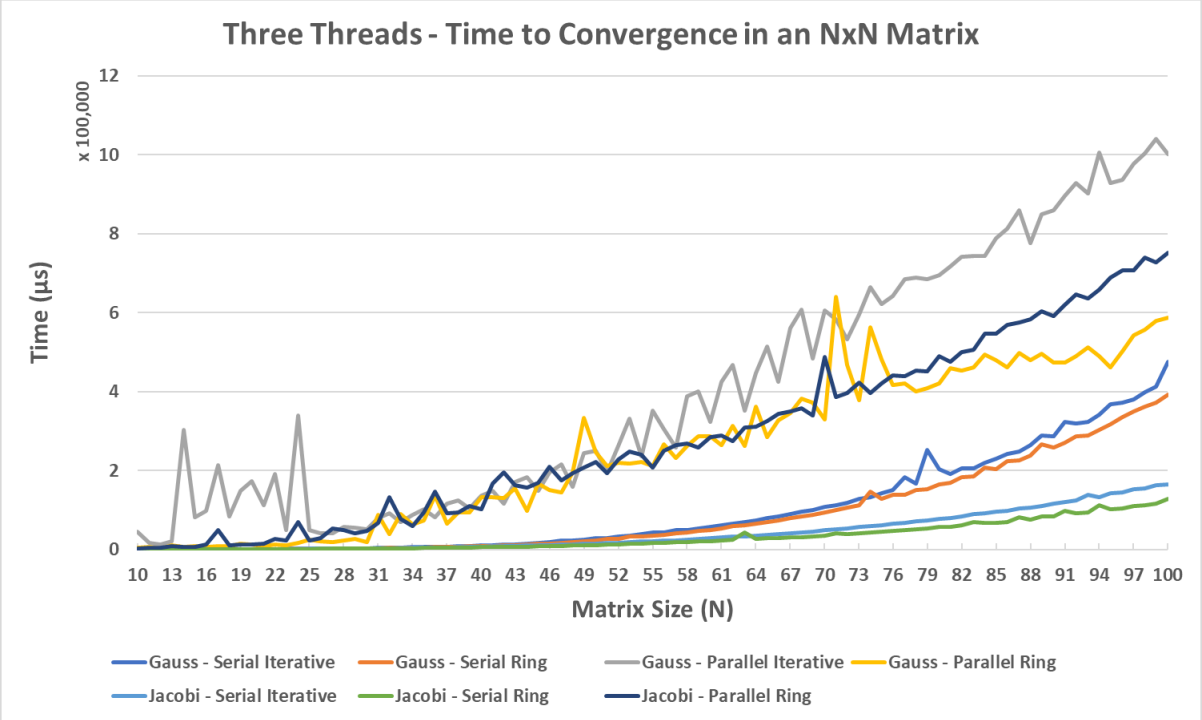

Figure 6: Experiment 1

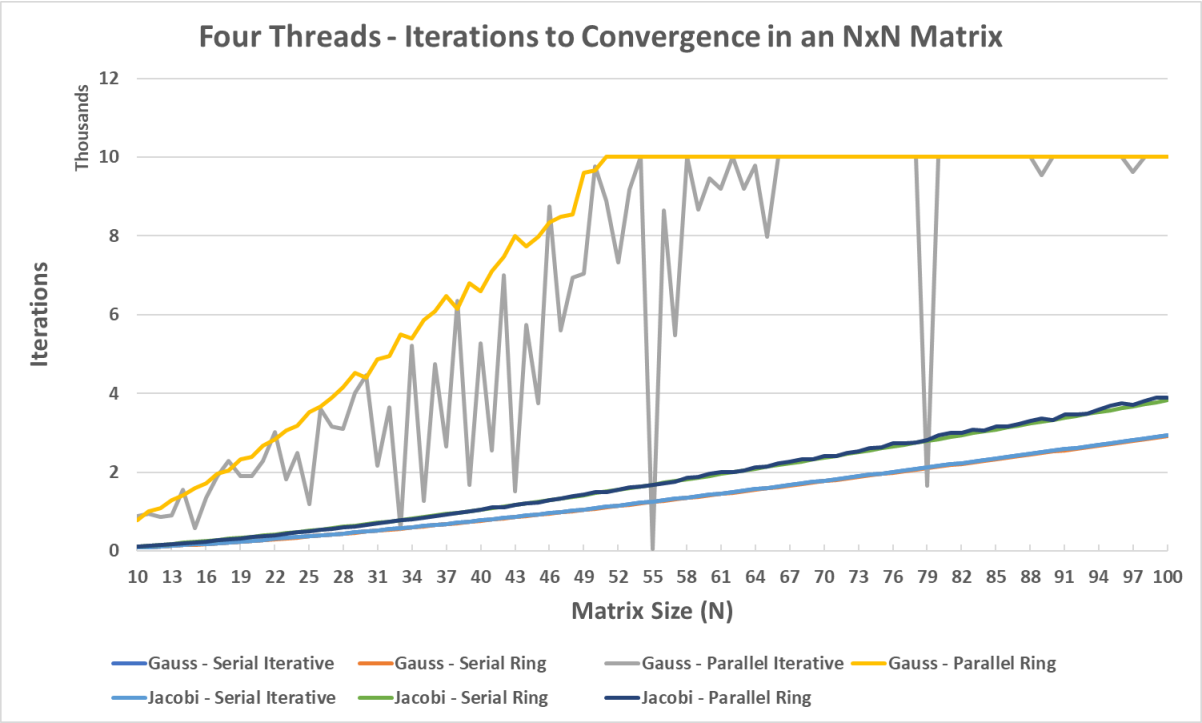

Figure 7: Experiment 1

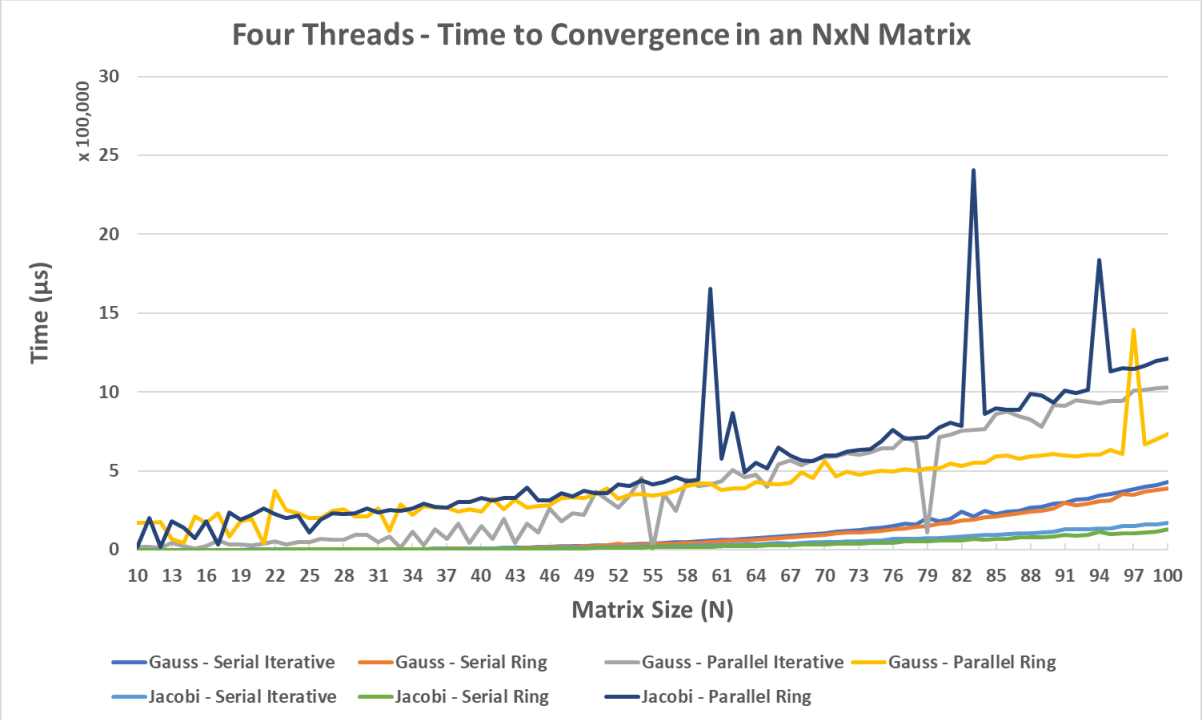

Figure 8: Experiment 1

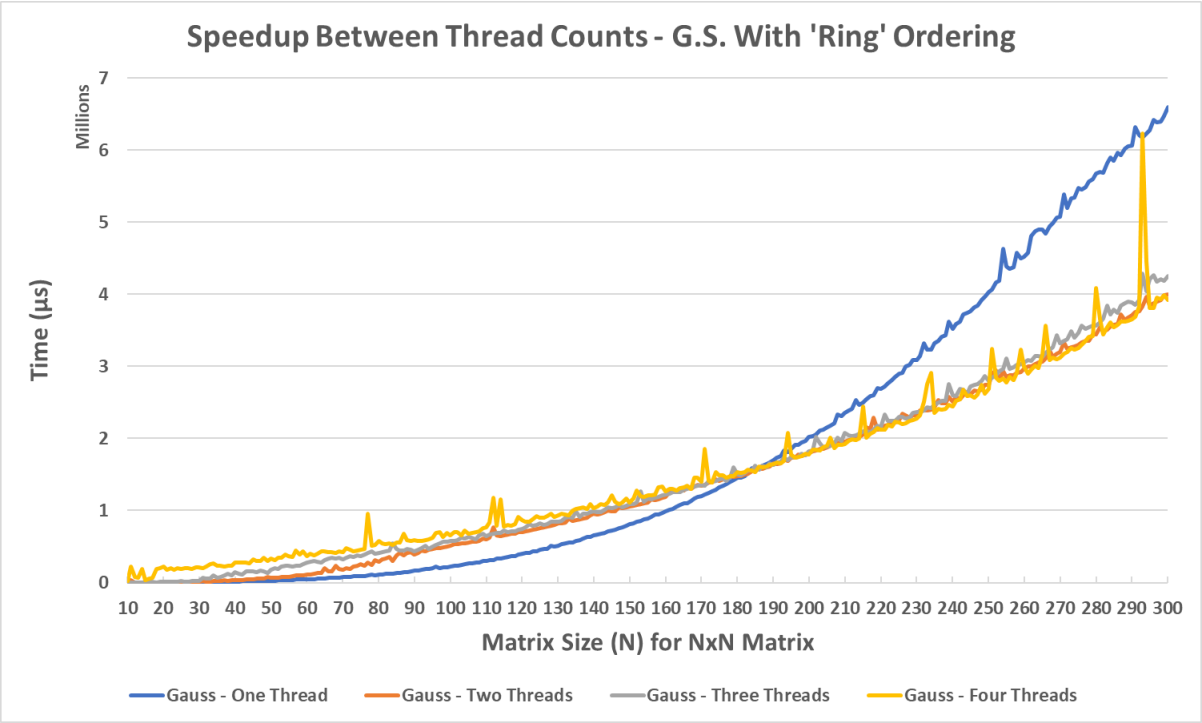

Figure 9: Experiment 2

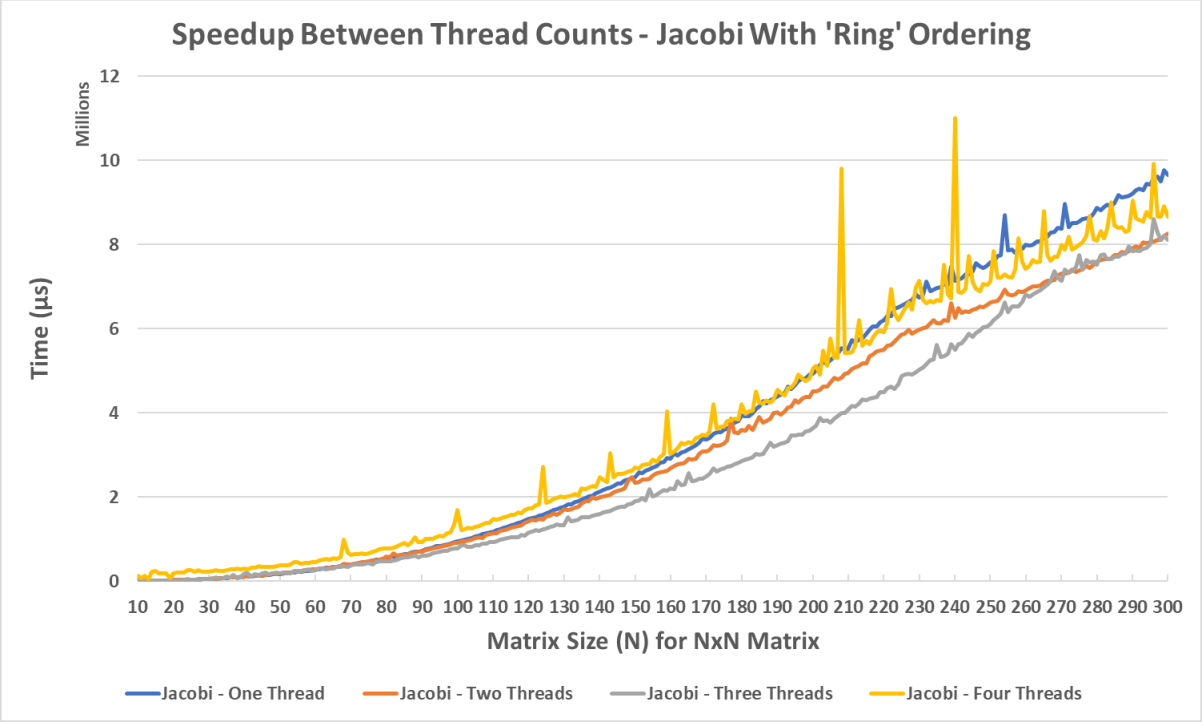

Figure 10: Experiment 2

ANALYSIS

Looking at the matrix outputs for individual runs of the parallel programs, P.R. believes, and M.H. concurs, that the values returned in the matrices our programs calculated are subject to some manner of race condition. P.R. suspected this to be the case as he anticipated the output of the parallel programs to require the same number of iterations to convergence as their serial counterparts. However, this is not the case. For example, consider the output below from the serial Gauss-Seidel 'ring' program, and the parallel Gauss-Seidel 'ring' program:

```
Current settings:
Matrix Dimension=3
TOLERANCE=0.000100
MAX_ITERATION=10000
MAX_TESTING=1
OVERFLOW=100000000.000000

CalcOrder{6,7,8,11,13,16,17,18,12,}

Print matrix x[][]
100.00 100.00 100.00 100.00 100.00
100.00 1.000  1.000  1.000  100.00
100.00 1.000  1.000  1.000  100.00
100.00 1.000  1.000  1.000  100.00
100.00 100.00 100.00 100.00 100.00

Gauss Seidel Solution:
100.00 100.00 100.00 100.00 100.00
100.00 99.995 99.995 99.997 100.00
100.00 99.995 99.995 99.996 100.00
100.00 99.997 99.996 99.998 100.00
100.00 100.00 100.00 100.00 100.00

Results:
GaussSBC: Iterations=15 Time Cost=0
```

```
Current settings:
Matrix Dimension=3
TOLERANCE=0.000100
MAX_ITERATION=10000
MAX_TESTING=1
OVERFLOW=100000000.000000

CalcOrder{6,7,8,11,13,16,17,18,12,}

Print matrix x[][]
100.00 100.00 100.00 100.00 100.00
100.00 1.000  1.000  1.000  100.00
100.00 1.000  1.000  1.000  100.00
100.00 1.000  1.000  1.000  100.00
100.00 100.00 100.00 100.00 100.00

Gauss Seidel Solution:
100.00 100.00 100.00 100.00 100.00
100.00 99.989 99.986 99.987 100.00
100.00 99.983 99.981 99.986 100.00
100.00 99.991 99.985 99.987 100.00
100.00 100.00 100.00 100.00 100.00
Results:
gaussSBC_TestOMP_RING:
    Iterations=124
    Time Cost=4000
    NUM_THREADS=4
    MATRIX_DIM=3
```

The serial version of the Gauss-Seidel 'ring' program returns the same solution values each time it is run. However, the parallel version of this program returns answers that differ from run to run. This is indicative of an unhandled race condition somewhere in the parallel program.

The data shown in **Figures 3 - 8** for the parallel version of the Gauss-Seidel 'iterative' program indicates an issue with this program as well. The regular up-and-down 'sawtooth' pattern to the data points indicates that something is consistently happening to the calculation depending on the size of the matrix. Examining several instances of matrix output of adjacent 'sawtooth' points indicates that the Gauss-Seidel 'iterative' program converges prematurely before ending the program. This means that the values at the boundaries do not fully propagate through the matrix as expected.

Analyzing the data values present in all the runs of the parallelized Gauss-Seidel 'iterative' program (available in the repository linked to in **Appendix A**) using two or more threads indicates that the 'sawtooth' pattern in the graphs is fairly consistent. For example, in the 'Iterations to Convergence' graph for the two threaded trials, **Figure 3**, the parallel Gauss-Seidel 'iterative' program had data values that prematurely converged at odd-numbered matrix sizes. For example, the first seven data points can be seen in **Table I**, and clearly demonstrate the pattern.

Matrix Size (N)	Gauss - Parallel Iterative
10	318
11	60
12	416
13	20
14	540
15	18
16	652

Table I: Gauss-Seidel Parallel Iterative Output

The above pattern also occurs in the trials for four

threads when using odd-numbered matrix size (**Figure 7**).

Interestingly, the pattern also occurs in the trials using three threads (**Figure 5**), first appearing to begin when the matrix size is 12×12. The pattern in this case repeats every *three* data points instead of every *two* as it did with the trials using an even number of threads.

Given that the parallelized Gauss-Seidel 'ring' program did not produce the 'sawtooth' pattern, and the only significant difference between the 'iterative' program and the 'ring' program is the ordering of matrix indices in the '`o[]`' array, then this ordering, and the cyclic way the threads interact with it, must contribute to the reason the 'iterative' program converges prematurely. The primary difference between the 'iterative' ordering and the 'ring' ordering is that the 'iterative' ordering will always have matrix indices updated in an exactly sequential fashion relative to the thread numbering as such:

```
Iterative updating with four threads:

xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      4       1       2       xx
xx      3       4       1       xx
xx      xx      xx      xx      xx
```

The 'ring' ordering, however may have matrix indices that are *not* necessarily updated in a sequential way ({1, 2, 3, 4, 1, 2, 3, 4, 1} vs. {1, 2, 3, 4, **1, 1**, 2, 3, 4}):

```
Iterative updating with four threads:

xx      xx      xx      xx      xx
xx      1       2       3       xx
xx      4       1       1       xx
xx      2       3       4       xx
xx      xx      xx      xx      xx
```

The timing data for the parallelized Gauss-Seidel 'iterative' tests using two or more threads also reflects the 'sawtooth' pattern seen above. However, it is not quite as pronounced in the data.

Regarding the timing of the Jacobi implementations, it is possible that the overhead associated with splitting up the ringed matrix actually takes longer than just iterating over a one dimensional array at relatively small matrix sizes. Each time a new iteration occurs, the parallel threads have to evaluate their new starting and ending locations, which the iterative code does not.

The results of the second experiment show that using more threads in a parallelized implementation can increase speedup at larger matrix sizes. However, these improvements were not drastic. P.R. initially believed that using four threads would cause the matrices to converge the most quickly. Despite the graph of the data, however, he came to the conclusion that this may actually be the case, as he had other applications open and running at the time the tests took place. As such, it is possible that the fourth core was busy handling other processes while it was also running the tests. With the OpenMP barriers in the Jacobi implementation, the other threads would be blocked by the barrier as the fourth core is busy with other work.

CONCLUSIONS

While the boundary condition problem is open to parallelization, our results indicate that doing so may not be a worthwhile endeavor unless the parallel code can be optimized in such a way that it converges more quickly than the serial code. In completing our implementations, we found that parallelizing the Gauss-Seidel and Jacobi methods was quite complicated. While a problem might be parallelizable, it may not be easy to parallelize it in a way that is more efficient that a serialized version. The Gauss-Seidel and Jacobi methods appeared to be easy to parallelize. In terms of accessing matrix values and determining when the algorithm should terminate, parallelizing these two methods was quite difficult.

In the end, however, we believe that parallelizing the solutions to problems is a worthwhile endeavor. Although our results were not what we anticipated, we have both learned more than we knew previously about using OpenMP and how to parallelize a problem.

Appendix A - Github Repository

https://github.com/mholman5721/CS511_FinalProject