[4005-735-01] Parallel Computing I
Programming Project 1

Christopher A. Wood
caw4567@rit.edu
4/1/2013

Overview

This report documents my work on the first Parallel Computing I programming project. It includes answers for the required questions, data to support my answers, and a personal reflection describing what I learned throughout the course of this work.

Questions

Question 1

Describe how your parallel program is designed. As part of your description, address these points:

1. Which parallel design patterns did you use?
2. What data structures did you use?
3. How did you partition the computation among the parallel threads?
4. Did you need to synchronize the threads, and if so, how did you do it?
5. Did you need to do load balancing, and if so, how did you do it?

Answer 1

The Jacobi algorithm for solving a linear system of n equations described by an n-by-n diagonally dominant coefficient matrix **A**, solution vector **x** of n elements, and result vector **b** of n elements is a very common technique used in elementary linear algebra. As presented in [1], the control flow for the algorithm, which corresponds to the number of iterations that are required before termination, is determined by convergence of the relative difference of the solution vector **x** and previously computed solution vector **y**. Faster convergence will lead to less iterations of the main body of the algorithm, which is composed of a modified matrix-vector multiplication step to compute a new temporary solution vector **y**, swapping of the old solution vector **x** and new solution vector **y**, and checking to see if the relative differences between the **x** and **y** vector components converged to a specific value, referred to as epsilon.

There are two main tasks, or sections of code, in the parallel program that implement the Jacobi algorithm. The first task (initialization) is responsible for populating the matrix **A** and vector **b**, and the second task (solution) is responsible for computing the final value of the solution vector **x**. I will use this description of the algorithm and parallel program to answer the following questions.

Answer 1a

I used the result parallelism design pattern in my program for both the initialization and solution tasks. The reason for this decision is as follows. In the initialization task, all threads (or virtual processors) run the same block of code to compute a specific chunk of the **A** matrix and **b** vector. We are interested in the result of every thread's computation because they all contribute to some part of the data used in the subsequent solution task. Therefore, this is clearly result parallelism.

In the solution task, all threads run the same block of code to compute a specific chunk of the temporary **y** vector and determine if the relative difference between every element of this **y** vector and the corresponding elements in the **x** vector is less than the epsilon value. Once all thread computations are complete, a sequential piece of code must swap the **x** and **y** vectors and reset the iteration sentinel flag accordingly. If the algorithm did not converge, the **x** vector is then recalculated using the same computations based on the results of the previous iteration of the algorithm, which means there is a sequential dependency that exists between successive computations. To sum up, since each thread is responsible for only a chunk of the resulting solution vector **x** no matter how many times it is recalculated, and we are interested in all such chunks, this is clearly result parallelism.

It is important to not mistake this task for agenda parallelism. The convergence check may incorrectly be construed as a form of reduction, since each thread checks its own chunk's relative difference and stores the result in a shared variable, but in fact it is not. This is because the output of the task is not a Boolean flag indicating whether or not the algorithm converged. Rather, the output is the entire solution vector **x**. The convergence check is merely part of the sequential dependency that exists between successive computations of the solution vector **x**.

Answer 1b

The Jacobi algorithm takes as input a the n-by-n coefficient matrix **A** and n-dimensional result vector **b,** both of which are shared variables. Given the requirements specification for the project, the elements in these data structures must be double-precision floating point values. Although these variables are shared among the threads, there are no write conflicts that emerge because each thread writes to a separate entry in the initialization task. Therefore, no synchronization is needed, and they can be stored as type `double[][]` and `double[]`, respectively.

Also, based on the algorithm, this means that the shared vectors **x** and **y** must also store double-precision floating point values. Following similar reasoning, **x** and **y** require no synchronization because there are no write conflicts that emerge (parallel threads only read from **x** and write to different chunks of **y**). Thus, they are both stored as type `double[]`

without synchronization. Of course, swapping **x** and **y** modifies both of these data structures, but this is done in a sequential block of code and therefore will not cause any read-write conflicts.

The only other shared variables in my parallel program are the solution dimension n, epsilon value, and the iteration (`converged`) and convergence (`iterConverged`) sentinel flags. The solution dimension is only ever written in a single thread at the beginning of the program, so it requires no synchronization and is stored as a primitive `int`. The epsilon value is set at compile time as a static variable and is only ever read, so no synchronization was required. And finally, the iteration flag is only written in a single thread and read by multiple threads, so it also requires no synchronization and can be stored as a primitive `boolean` type.

Interestingly, the convergence variable is read and written to by multiple threads. However, given the nature of the algorithm and my implementation, all write instructions in the parallel section of code store the same value (false) in the variable. Therefore, it does not matter if multiple threads try to write to this at the same time; the result will always be false. Also, this value is only ever set to true in a single thread. Therefore, no synchronization is required, and the variable is stored as a primitive `boolean` type.

To sum up, no data structures from the Parallel Java library were required for the correct behavior of this program. Only arrays (or primitive types) and individual primitive types were used in my parallel program.

Answer 1c

Prior to running the solution task that takes the coefficient matrix **A** and vector **b** as input, these data structures must be populated with meaningful test data in the initialization task. Since matrix traversal requires $O(n^2)$ steps if done sequentially, and each row in the matrix is initialized independently from the rest, this initialization task was transformed into a parallel loop that evenly distributes the computational task of initializing the rows to separate threads. In other words, the outer loop that traverses the rows of **A** and **b** was made into a parallel for loop, as shown in Figure 1.

```
Parallel for i = 0 to N-1: // row-based splitting
      Initialize the PRNG using the seed and skip ahead to the correct spot
      For j = 0 to N-1:
            A[i,j] <- PRNG.nextDouble() * 9.0 + 1.0
      A[i,i] <- A[i,i] 10.0 * N
```

```
b[i] <- (PRNG.nextDouble() * 9.0) + 1.0
```

Figure 1. Initialization task pseudocode.

This row-based splitting for initializing **A** and **b** is shown graphically in Figure 2. It's important to note that since each thread uses its own thread-local PRNG initialized with the same seed to generate the random values for **A** and **b**, and the random numbers generated for each element in these data structures must match the random numbers used in the sequential version of the program, it was necessary to use a sequence-splitting PRNG. By skipping each thread's PRNG ahead to the correct spot in the sequence of random numbers based on the row chunk they are assigned, we can safely generate random number sequences that, together, match the sequence in the sequential version. Furthermore, since each thread computes n random numbers in the inner for loop and one additional random number for the **b** coordinate, the skip value for each thread is `(n+1)*first`, where `first` is the index of the first row in the thread's chunk.

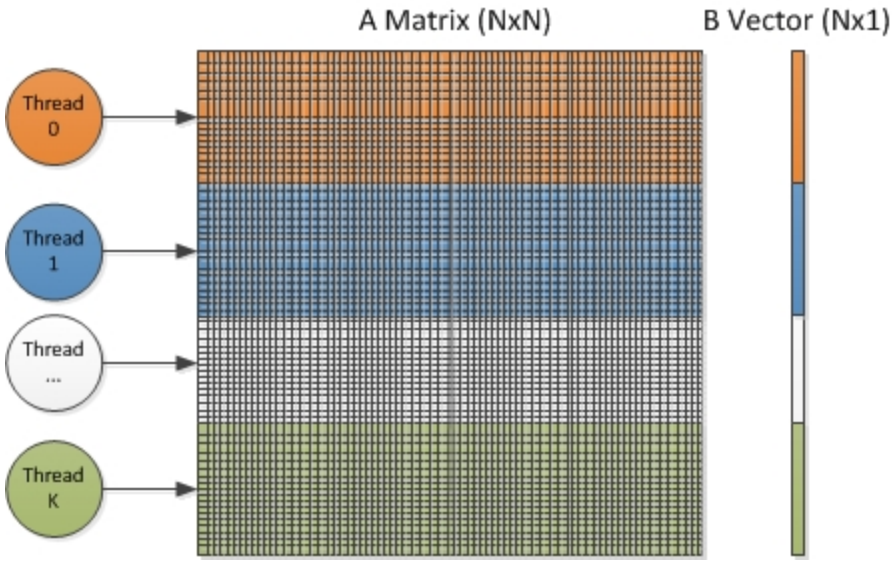

Figure 2. Row-based splitting of the initialization (and solution) task.

After the test data was initialized, the parallel team then executes the solution task until the solution vector converged. Since there is a sequential dependency of swapping the **x** and **y** vectors and checking to see if the relative difference has converged for all threads, the only candidate for parallelism in this task is the computation of the **y** vector in the inner loop and chunk-wise check for relative difference convergence (we cannot parallelize the outer loop because of the sequential dependencies). Therefore, my parallel program partitions the computation of the **y** vector into evenly distributed chunks across all threads using the same row-based splitting strategy as in the initialization task, resulting in the design shown in Figure 3.

```
iterConverged <- true // only written to true by a single thread
converged <- false // always false to start
While not converged:
      Parallel for i = 0 to N-1:
            t_iterConverged <- false // per-thread variable
            For j = 0 to N-1:
                  y[i] <- (b[i] - lowerProduct - upperProduct) / A[i][i]
            if (reldif(x[i], y[i]) < epsilon):
                  t_iterConverged <- false
            if (t_iterConverged == false):
                  iterConverged <- false // avoid freq. shared writes
      swap(x, y)
      converged <- iterConverged // reset iteration sentinel value
      iterConverged <- true // only written to true by a single thread
```

Figure 3. Solution task pseudocode. The variables *lowerProduct* **and** *upperProduct* **correspond to the sums** $\sum_{j=0}^{i-1} A[i][j] * x[j]$ **and** $\sum_{j=i+1}^{n-1} A[i][j] * x[j]$**, respectively.**

Again, the swap and convergence check for all threads must be done sequentially after all threads have finished their computation. Therefore, the outer loop remains a sequential loop. The more computationally expensive part of the Jacobi algorithm, namely, computing the **y** vector, is done in parallel.

Answer 1d

Within the initialization parallel task of the program, there was no synchronization needed among the threads. There are no sequential dependencies that exist between the threads computing values for **A** and **b**, since each thread receives its own row and the values of a row do not depend on any other row. Also, since each thread only writes to the rows to which they are assigned, no two threads would ever attempt to write to the same elements in **A** or **b**. So, even though these are shared variables, they did not need to be synchronized in any way. Therefore, all of the threads in the initialization task were allowed to execute without synchronizing with any other thread until they reached the implicit barrier at the end of the execute() method. The threads are required to synchronize at the end of this task's implicit barrier because it prevents other threads from proceeding on to the solution task until all **A** and **b** have been correctly initialized. Put another way, if a thread was allowed to start computing the solution and **A** and **b** were not fully initialized, then such thread might compute an incorrect result.

Synchronization was much more of an issue in the solution task. Given the nature of the Jacobi algorithm, there exists an implicit sequential dependency between each iteration of the outermost loop [1]. That is, the temporary solution vector **y** depends on the values of **x**, which are determined in the previous iteration of the algorithm. Furthermore, the task of swapping the **x** and **y** vectors inside the outermost loop cannot be done until every element of the **y** vector is calculated. This is because **y** depends on **x**, and if **x** were to be different for threads within the inner loop, then an incorrect result would be computed.

In terms of variable synchronization, both **A** and **b** are never written to within this task; they are only read. Therefore, synchronization is still not required for these shared variables. Also, since the work of the inner loop is partitioned among the threads by splitting up the rows, which means that each thread is responsible for computing a different chunk of the **y** vector, it is not necessary to synchronize access to this variable either. No two threads will ever attempt to write to the same element of **y**. The two Boolean sentinel flags, namely the convergence and iteration flags, are also shared among the threads. No synchronization is needed for the iteration flag since it is read by all threads but only written in a single thread. The convergence flag also does not require synchronization because no two threads will ever write different values to this variable, which means that write-write conflicts will never result in an incorrect value being stored in the variable (see Figure 3). However, to avoid unnecessary writes to shared variables that lead to cache interference, a thread-local copy of the convergence flag is maintained until the end of the thread's row chunk, at which point the thread will then store its local copy (if false) into the shared variable. The other variables used in the algorithm, namely the `lowerProduct` and `upperProduct`, can be kept as thread-local variables, and thus require no synchronization.

Given this design and variable sharing scheme, thread synchronization for the solution task occurs as follows. The inner loop is transformed into a parallel for loop with an explicit `BarrierAction` at the end which is responsible for swapping **x** and **y** and resetting the iteration and convergence flags. Swapping **x** and **y** is done by reference reassignment, which takes constant time (as opposed to walking the vectors and doing a pair-wise swap). This barrier ensures that no thread can proceed to the next iteration of the algorithm without the swap and flag update operations (based on the relative difference convergence) being executed, which ensures that the **x** vector will contain the correct values for every iteration. This design is shown in Figure 4. After the single-threaded barrier is complete, the threads return to the start of the parallel for loop and, if the algorithm hasn't converged, compute new values for **y** in parallel before stopping at the swapping barrier again.

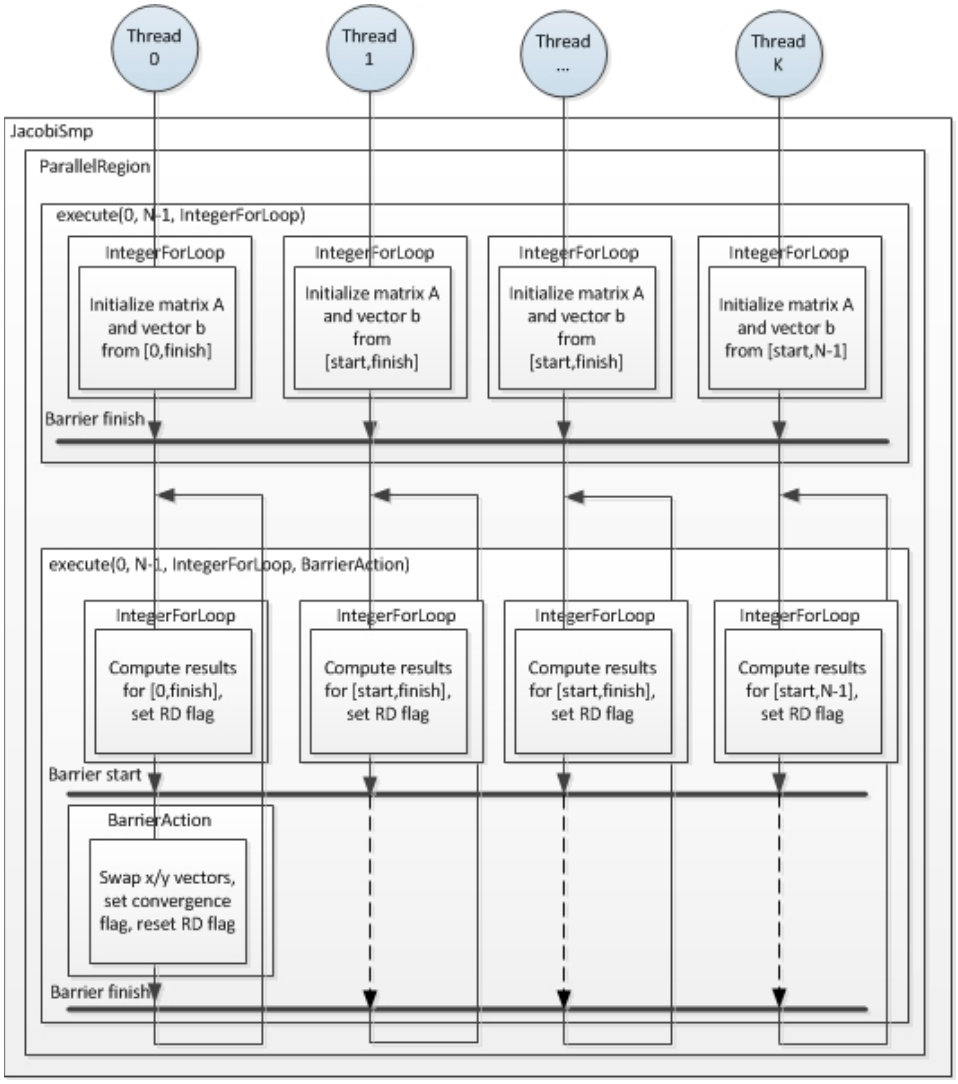

Figure 4. Thread synchronization in the JacobiSmp program. The initialization task uses an implicit barrier and the solution task uses an explicit barrier.

Answer 1e

Since the two main parallel tasks of my program are the initialization and solution tasks, there were at most two opportunities to perform load balancing. In the case of the test data initialization block, every element of matrix **A** and vector **b** must be initialized (none can be skipped), which means that every thread must initialize n columns for each row they are responsible for. Therefore, since each thread computes data for the same number of columns, and the number of rows are evenly distributed among the threads using the `IntegerForLoop` class in the Parallel Java library, no load balancing was needed (at first!) for this block of parallel code.

However, since each calculation is dealing with randomly generated floating point values, the

complexity of the arithmetic may vary. For example, multiplying by 0.0 may be faster than multiplying by 1.54×10^6 depending on the CPU architecture, and for this reason a guided schedule was included in the `IntegerForLoop` to ensure that such arithmetic anomalies do not create an unbalanced load.

Following similar reasoning, with the added fact that some threads may only execute the relative difference code once for their chunk of the vector (after a thread finds that it hasn't converged it does not repeat this calculation for the remainder of its chunk) and others may need to perform this calculation for every value in the chunk, there is also the possibility of an unbalanced load in the solution task. Therefore, a guided schedule was also used in this task's `IntegerForLoop` to ensure that the computations are evenly balanced among all of the threads.

In both cases, including a guided schedule required me to override the schedule() method in the `IntegerForLoop` class, as shown in Figure 5.

```
execute(0, n - 1, new IntegerForLoop()
{
   ...
   public IntegerSchedule schedule() { return IntegerSchedule.guided(); }
   ...
}
```

Figure 5. Specifying a guided schedule for the IntegerForLoop anonymous classes in the initialization and solution tasks.

Question 2

Table 1. Data set 1 results.

NT	T1 (msec)	T2 (msec)	T3 (msec)	T (msec)	Speedup	Efficiency	ESDF
Seq	54746	54368	53660	53660			
1	36345	37265	36857	36345	1.476	1.476	
2	23184	23401	23510	23184	2.315	1.157	0.276
3	16480	16503	16526	16480	3.256	1.085	0.180
4	13271	13138	13230	13138	4.084	1.021	0.149
8	7643	7629	7588	7588	7.072	0.884	0.096

Question 3

Table 2. Data set 2 results.

NT	T1 (msec)	T2 (msec)	T3 (msec)	T (msec)	Speedup	Efficiency	ESDF
Seq	109870	109561	108718	108718			
1	67455	66295	67914	66295	1.640	1.640	
2	36087	36070	36061	36061	3.015	1.507	0.088
3	26672	26512	26742	26512	4.101	1.367	0.100
4	20764	20972	20937	20764	5.236	1.309	0.084
8	12138	12057	12118	12057	9.017	1.127	0.065

Question 4

Table 3. Data set 3 results.

NT	T1 (msec)	T2 (msec)	T3 (msec)	T (msec)	Speedup	Efficiency	ESDF
Seq	166646	164901	166563	164901			
1	109224	109676	109049	109049	1.512	1.512	
2	58491	58877	58640	58491	2.819	1.410	0.073
3	41231	41496	41333	41231	3.999	1.333	0.067
4	32905	32603	32928	32603	5.058	1.264	0.065
8	18377	18862	18724	18377	8.973	1.122	0.050

Question 5

Summarize your measurements from Questions 2-4. As part of your summary, address these points:

a. How close to the ideal speedup and efficiency did your parallel program achieve as the number of parallel threads increased?
b. What is causing the discrepancy if any between the ideal and the measured speedup and efficiency in your parallel program?
c. Is this problem a good candidate for an SMP parallel program? Why or why not?

Answer 5a

Generally speaking, my parallel program surpassed the ideal speedup and efficiency (that is, a speedup of K and efficiency of 1.0). For all cases other than NT = 8 with the first data set, my efficiencies were greater than one, indicating a superlinear speedup compared to the sequential version of the program. However, in all three data sets, as the number of parallel threads increased, the efficiencies began to decrease slightly, approaching the ideal values, as a result of Amdahl's Law. For example, in the second data set, my efficiency started out at 1.640 for NT = 1 and ended up at 1.127 at NT = 8. Even though both of these values still indicate superlinear speedups, it is clear that the speedups and efficiencies were approaching Amdahl's limit and would soon drop below the ideal values as the number of parallel threads increased.

Answer 5b

There are two main causes for the superiority between the ideal and measured speedup and efficiency in my parallel program:

1. Cache memory data storage and access for each processor thread.

   In my parallel program, each thread is able to cache both the program JVM instructions and data for the **A** and **b** data structures after the completion of the initialization task. Then, when the solution task begins executing, the needed data for these computations is already in cache memory, enabling much faster reads than the sequential program was able to achieve (since the single-threaded sequential version probably cannot fit all of the data in the cache). Although each thread keeps the their chunk of the **y** matrix in cache during the solution vector, it is probably the case that the sequential swap inside the `BarrierAction` is causing these cache lines to be invalidated and thus reread at the start of every iteration of the solution task.

2. Early Java JIT optimizations (the JIT compiler effect).

   After running the JVM profiler on my sequential version of the program I was able to determine that the hotspots of my program consist of the inner loop of the initialization task (i.e. the one that computes all **A** entries for a given row index) and the matrix-vector product in the solution task (i.e. the one that computes the new **y** value for a given row index). Therefore, by running these tasks in parallel, the JVM was able to recognize that these tasks are hotspots much sooner than in the sequential version, which enabled the JIT to compile the Java bytecode down to native code to be executed directly on the processor. This led to superlinear speedups that could not be achieved through the sequential version. This fact was verified by disabling the JIT

and running both the sequential and parallel programs to see the resulting times. In this case, I did not see superlinear speedups.

3. Guided schedule to balance the load.

   Since some threads may notice that the relative difference was not within the epsilon bound earlier than other threads, and each thread will avoid doing the expensive relative difference computation when such condition is met, using a guided schedule helped ensure a more balanced load among the set of threads. This improved the running time in the parallel versions of the program.

The decreases in speedups and efficiencies that resulted from increasing the number of parallel threads was largely due to the barrier synchronization at the end of every iteration of the solution task. With more threads there was more fine-grained partitioning of the **y** vector for computation, and since each thread might have a different amount of work to compute, the latency between the first and last threads reaching the barrier likely increased. While the guided schedule may have helped alleviate this problem, it does not remove it entirely.

Furthermore, based on Amdahl's Law, as the number of parallel threads increases, the total running time for the program approaches the asymptotic limit of $1/F$. This is because the sequential portion of the program running time, which includes initializing the Parallel Java threads and objects, reading the command line parameters, running the `BarrierAction` inside the solution task, and printing the solution, does not decrease as the number of parallel threads is increased; only the parallel portion running time decreases. The drops in my speedups and efficiencies are indicative of this property, and thus support Amdahl's Law.

Answer 5c

Yes, this problem is a good candidate for an SMP parallel program for a couple of reasons. First, given the type of problems being solved by the Jacobi algorithm, as well as the size and type of the of the data structures for the Jacobi algorithm (an n-by-n array of doubles and three one-dimensional arrays of doubles), it is unlikely that the memory limitation in an SMP parallel program will be a limiting factor in the usefulness of an SMP implementation. For example, on an SMP parallel computer with 8GB of main memory, it would be possible to hold $(2^3)(2^{30}) = 2^{33}$ bytes of data, we can solve problems up to size $n \approx 92680$, which is a realistic bound for this type of problem.

Second, since all threads in a single process belong to the same address space on a SMP computer, each thread can communicate with each other using shared memory and also quickly access shared data without having to reach out across the network (as is the case in a

cluster). Given that each calculation of **y** must use a significant chunk of **A**, **b**, and all of **x** during each iteration of the solution task, fast access to this data is necessary for efficient parallel implementations of the Jacobi algorithm. SMP machines enable this type of data access by keeping everything in shared global memory, as opposed to distributed memory spread across clusters of machines that require message passing techniques for access. Also, by keeping all of the the program data in main memory on the same machine, each thread can cache the data it frequently accesses, thus enabling even faster memory reads with minimal performance hits caused by invalidating cache lines.

Question 6

Write a paragraph telling me what you learned from this project.

Answer

This project was an excellent educational experience. Although implementing the sequential version of the program was a straightforward process given the algorithm description in [1], there were many subtleties in my parallel program that drastically affected the overall performance. Without considering issues such as cache interference, shared memory access, thread synchronization, and sequential dependencies, it would have been difficult to obtain the performance metrics (i.e. sizeup and efficiency) shown in the previous answers. It also forced me to think critically about the complexity and CPU instructions required for simple mathematical operations. For example, computing the relative difference is an expensive operation since it requires a multiply, divide, and absolute value computation. To avoid these operations I modified my code to minimize the number of times it is executed. As a result, I saw how simple design changes can improve the program's performance. I also gained hands on experience with a parallel computing library, and with this experience came the realization of exactly how costly parallelism can be for certain applications. In the case of the Jacobi algorithm, the sequential dependencies that exist between consecutive iterations of the main loop forces thread synchronization to take place to compute the correct result. This additional overhead, on top of the overhead of starting up and tearing down the parallel threads, must be carefully managed in order to produce ideal speedups and efficiencies. Finally, I learned how beneficial the Java JVM profiler can be when trying to optimize code. Iteratively optimizing a program is a far quicker way to obtain an optimal program than performing many different optimizations at once, and the profiler helps focus the attention on hotspots that come from poor design or implementation strategies.

References

[1] - Alan Kaminsky. Solution of a Linear System Using Jacobi Iteration. Online Lecture Notes. Available at: http://www.cs.rit.edu/~ark/531/p1/jacobi.pdf. Accessed: 3/27/13.