Jacobi Project

CS 322

Kathryn McClintic
Aaron Tuor

10th September, 2016

1 Introduction

The Jacobi iterative algorithm is a simple numerical solution used to solve Laplace's Equation. Jacobi takes in a matrix of values where the peripheral edges are filled in with known values but the inner values are unknown. Through a series of iterative averages, Jacobi eventually converges and fills in the inner matrix values. Jacobi can be applied to problems in many fields including electrical engineering, environmental analysis, physics, and chemistry to determine unknown values in solids or surfaces. Jacobi's method is often chosen because it is easy to implement and relatively fast for small matrices. However, once the size of the input matrix increases, Jacobi can quickly become a large $O(n^2)$ operation which is too slow for feasible use in scientific analysis. Because each cell can be computed independently, Jacobi can be easily parallelized by partitioning the matrix into columns or rows. For large matrices (which are expected in the real world), it is important to be able to speed up performance so that results can be achieved in a realistic time period.

In our project, we first started by creating the most basic sequential version of the Jacobi iteration. We used loop unrolling, function in-lining and multiplication by a fractional value to optimize our sequential code further. After we had an optimized version of the sequential code, we added in threading to see if we could speed it up. The speedups we saw were not linear but there was identifiable speedup between 1 and 2 threads. Our implementation and experiment sections will describe design decisions, problems encountered, and testing strategies which were used while we tried to speed up Jacobi using threads. In our discussion we will discuss possible reasons for why the speedups we saw were not linear and problems we encountered during testing. Finally, our conclusion will cover the disparity between the ideal machine and the sub par conditions that are often encountered in a shared computing environment with a limited number of processors.

2 Implementation

The parallel and sequential algorithms were implemented in Java. The parallel version employs the java Thread class for concurrent execution of the Jacobi Iteration on partitions of the matrix. The matrix is partitioned by rows. For $n =$ the number of threads, $i =$ the number of rows of the matrix, threads $2, \ldots, n-1$ compute the values of the updated matrix for $\lfloor \frac{i}{n} \rfloor$ rows. Thread 1 computes the values of the updated matrix for $\lfloor \frac{i}{n} \rfloor - 1$ rows. Thread n computes the values of the matrix for $(i \bmod n) - 1$ rows.

For the updated matrix M_k, at iteration k, the algorithm stops when $||M_k - M_{k-1}||_1 < \epsilon$ where epsilon is a small number input as a program argument. For efficiency the stopping criterion is only evaluated every P iterations where P is a parameter gotten from the command line.

In order to synchronize the threads so that each thread is always on the same iteration of the algorithm we implemented a reusable barrier with the java.util.concurrent.Semaphore class. Our reusable barrier is implemented using a turnstile paradigm where every thread must arrive at the first lock before the second lock is released to allow the threads to continue.

To test the correctness of our parallel implementation we did some preliminary printing of matrices to files and used vimdiff, comparing the output from execution using 1 and 4 threads. We also used htop to ensure that we were utilizing all cores for an eight thread execution.

3 Experiment

3.1 Machines

We tested timing results running the Jacobi program on two machines, C_1, C_2 whose specifications are displayed in the chart below.

4 Results

Machine	Processor Type	Speed	# Processors	Floating Point Units	L1	L2
C_1	Intel(R) Core(TM) i7-3770	3.40GHz	8	4	32KB	256KB
C_2	Intel(R) Xeon(R) CPU	2.80GHz	8	4	32KB	256KB

4.1 Timing

Program executions were clocked from the beginning of the loop calling start on each Thread to the end of the loop when all threads are joined after completion.

4.2 Methods

We conducted and timed twenty runs on each machine for thread count $1, \ldots, 16$. Each run processed the 2048×2048 matrix read from the file input.mtx. We set $P = 200$, $\epsilon = 0.0001$, for all program executions. We executed these experiments using a simple 8 line python shell script, `jacobi_experiment.py`. On each machine `jacobi_experiment.py` was the only process running during experiment runs.

5 Results

Table A	C1
Threads	AverageTime (s)
1	42.24
2	36.37
3	36.66
4	37.21
5	37.89
6	37.63
7	37.54
8	37.38
9	39.72
10	39.67
11	39.71
12	39.64
13	39.64
14	39.64
15	39.75
16	39.90

Table B	C2
Threads	AverageTime (s)
1	72.8
2	48.8
3	44.35
4	45.15
5	43.7
6	42.6
7	47.25
8	49
9	48.9
10	47.2
11	46.95
12	47.95
13	48.75
14	49
15	48.9
16	49.15

Average execution times over 20 runs using 1 - 16 threads on C_1 and C_2.

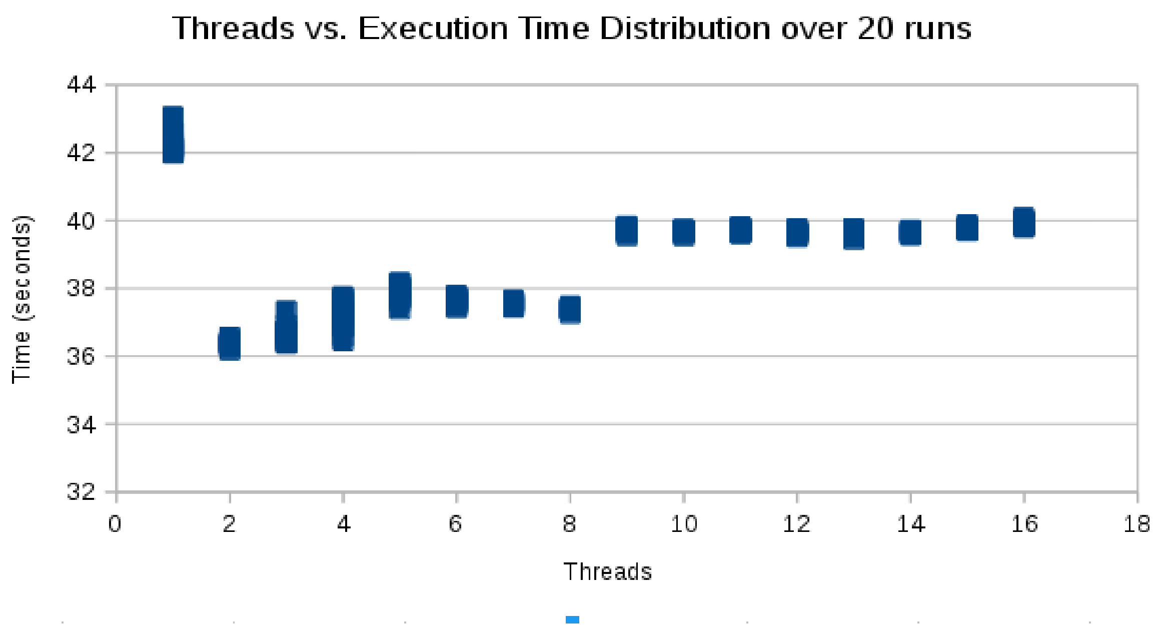

Figure 1: Distribution of execution times over 20 runs for each number of threads. This shows the variance of time within each thread count group.

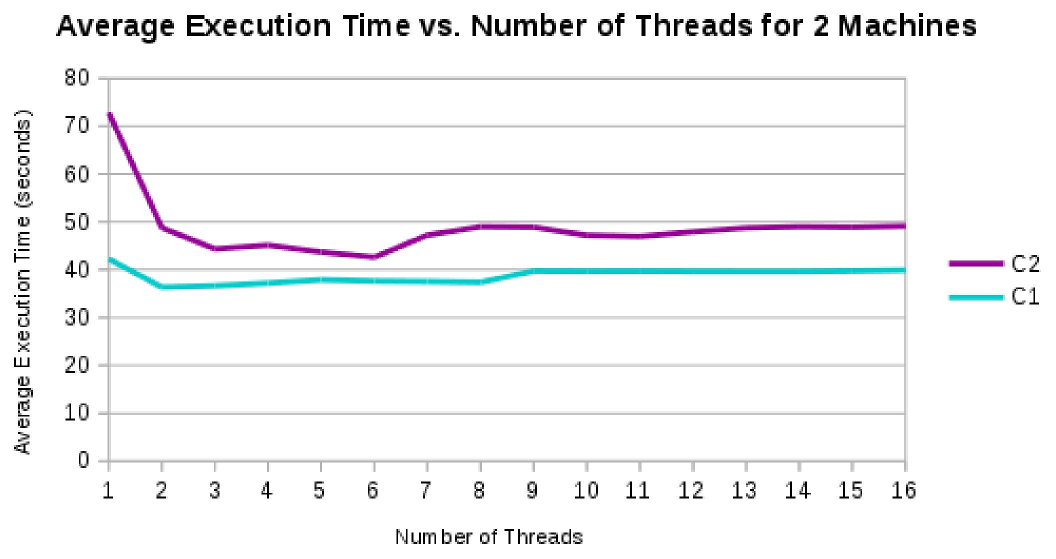

Figure 2: Average execution time vs. number of threads curve for each machine C1 and C2. This shows that C2 started out slower(due to slower clock speed) and thus experienced a more dramatic speed up when using threads as opposed to C1, the lab machine.

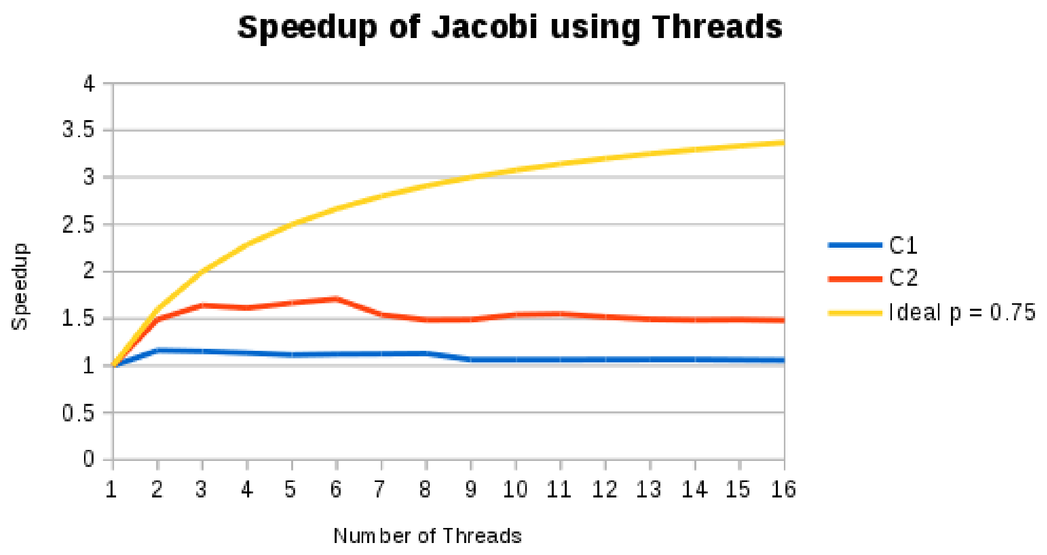

Figure 3: Speedup of Jacobi algorithm using threads on C1 and C2 and ideal speedup. Assume that 1 = regular speed. So a speedup of 1.5 means the program went 1.5 times faster.We computed speedup for each machine by dividing time for one thread by the time for n threads. We computed the ideal speedup assuming that the proportion of work that could be parallelized was 0.75, so p=0.75. We used the equation $\frac{1}{(1-p)+p/n}$ where p=0.75 and n = number of threads to compute the ideal speedup.

6 Discussion and Conclusion

From the experiments conducted we see a speedup, more significant for C_2 than for C_1. In fact the only speedup for C_1 is going from 1 thread to 2 threads, while for C_2 speedup continues to increase up to 6 threads before declining. As C_2 has the same cache sizes, number of processors, and floating point units, this may indicate that the server grade XEON processors which C_2 uses take better advantage of concurrent execution.

One thing we noticed was that due to the machines only having 4 floating point processing units, we didn't see any speedup after around 4 threads on C1 and around 6 threads on C2. It would be interesting trying to accomplish this problem on a machine with more floating point processing units.

It was surprising that the average execution time of 2 threads on C1 (36.37 seconds) was only 13% faster than one thread on C1 (42.24 seconds). After examining the algorithm, it looked like there would have been a more dramatic speedup, however, we hypothesized that our use of barriers may have been a bottleneck on the speedup. Overall, we expected more dramatic speedups when implementing threads.

We will derive an approximation of the amount of serial work $(1-p)$ from our results by solving for p the amount of parallelizable work in the equation: $\frac{1}{(1-p)+p/n} = \texttt{speedup}$. We will use the results for $p = 2$ on machine C_2.

$$\frac{1}{(1-p)+p/2} = 1.49 \Rightarrow p = 0.6577$$

Assuming that about 65 percent of the work is parallelizable, we might expect to see a greater speedup for either machine. We may not be seeing the greatest possible speedups because of the small size of the matrix we are testing on. In a real world scientific computation the matrices are likely to be much larger, meaning that the ratio of time spent waiting for other threads to complete over time spent doing arithmetic will be smaller.

Testing in the real world environment versus the environment of the theoretical textbook is filled with hidden variables which can contribute to varied results. We tested on machines which were not guaranteed to be quiet while we were running our tests. In addition, there is overhead associated with creating threads, starting them, and waiting for them with barriers which is not accounted for in idealized speedup calculations. This project gave us a great understanding of how semaphores and threads can be used to partition the work of a large scale real world calculation. The project also showed us that more threads doesn't necessarily mean faster computation as there are constraints in any real world computational system. For scientific problems which employ the Jacobi Iteration on large matrices constructed from real world data, parallelization via threads is a good choice, even if the speedup does not reach theoretical limits.