ECE 420 Lab 3 Report

Authors: Noah Batiuk, Steven Jiao, Justin Boileau

Description of Implementation

Objective of Lab

In this lab, we implemented a routine to solve a linear system of equations through Gauss-Jordan elimination. Next, we implemented 4 different optimized routines using OpenMP.

Implementations

Base Implementation

Our base implementation accepts a single command-line parameter for the number of threads to be used, however this parameter is only meaningful in the optimized versions. The program begins by loading the input matrix representing a system of linear equations, which is assumed to have been generated already using the provided datagen.c program. Next, it solves the system of linear equations using the Gauss-Jordan method as described by the pseudocode from the lab manual. Finally it stops timing and saves the result.

Best Optimized Implementation (main_opt1.c)

The best implementation of Gauss-Jordan Elimination utilized the closure feature of OpenMP to spawn a single thread pool using **pragma omp parallel** for all calculation sections including the Gaussian Elimination, Jordan Elimination, and the final calculation steps. In Gaussian Elimination, we have a **pragma omp for** to split the team further to calculate the local max for each thread, and then the setting of the max element and index within each thread team is protected with **pragma omp critical**. A **pragma omp barrier** is implemented after the critical section so we ensure that the maximum element row and index is of the max within the entirety of the thread team. **Pragma omp single** is then used for the actual row swap as we only need to swap it once for each team. Finally the elimination portion is split using **pragma omp for** with the **schedule(guided)**. Jordan elimination for the resultant array uses the same thread pool initially spawned and has a **pragma omp for** to split the elimination work along each row also utilizing **schedule(guided)**. Finally, another **pragma omp for schedule(guided)** is used for calculating the values of the unknown variables.

Other Implementations

Additionally, we constructed 3 alternate optimization methods (main_opt2.c, main_opt3.c, and main_opt4.c).

main_opt2.c

The first optimized version declares a pool of threads for each outer loop in the gaussian calculation. Within each loop we parallelize the max element calculation by adding a **pragma omp for** and giving each thread a local max variable, which each thread then compares to the max for the whole row in a critical section. Afterwards, we do the row swap with a single thread only using the **pragma omp single** directive. For the swap, we operate on the indices instead of the actual array rows, which further improves runtime. Lastly, we parallelize the calculation step using another **pragma omp for**.

main_opt3.c

The second alternate (main_opt3) approach targeted only the inner calculating loop of the Gaussian Elimination section. This section calculates the value in each column of each row, so it can be safely parallelized at the row level. We added a **pragma omp for** to achieve this, which yielded a 1 order of magnitude increase over the unoptimized version.

main_opt4.c

The last approach (main_opt4) is similar to our first alternate approach (main_opt2) except this time we wrapped the entirety of the Gaussian Elimination section, Jordan Elimination, and the final calculation in a **pragma omp parallel** to spawn our thread pool. Gaussian Elimination calculations using a local max and local max index up to the row swap were kept the same, and **pragma omp for** is called before the Jordan Elimination to split our threads again to calculate their respective sections, as well as for calculating the final result.

Performance Discussion

Speedup of Best Implementation

The best speedup time we saw from our Gauss-Jordan Elimination programs led to a speedup of almost a factor of 3 as seen in main_opt4.c running 4 threads. 4 threads appeared to be the minimum for this program, with execution time jumping up with fewer threads, and gradually increasing with an increasing thread count. However, main_opt1 and main_opt3 were the outperformers for tests with more than 8 threads. Each saw an approximate 2x speedup across 12 and 16 threads.

This improvement can be attributed to multiple factors, including improved scheduling, reducing the number of implicit joins, parallelizing certain sections of the calculations, and operating on indices of the array matrix instead of the actual matrix in memory. The

number of implicit joins is reduced by creating a “team” of threads at the beginning of the program, and reusing this team for subsequent. This reduces the number of implicit joins and means we don’t need to spawn new threads for every loop. After creating this team of threads, we were able to parallelize the row max calculation and second calculation loop in the Gaussian section, and the main calculation loop in the Jordan elimination section. Another way we were able to improve the runtime is by tracking the indices from the matrix array and swapping just those indices, instead of the actual array rows in memory.

One reason main_opt4.c outperformed main_opt1.c in 4 threads is because main_opt4.c does not contain scheduling in its for-loops, and thus less overhead at lower thread counts and better performance while having all the other speedup benefits we put into the program. We can also confirm this when looking at main_opt4.c vs. main_opt1.c when running 8, 12, and 16 threads in the 1000x1000 matrix where main_opt1.c is gradually able to outperform all other programs with main_opt4.c gradually having worse performance. The next highest performer was main_opt3.c, which only had the Gaussian elimination parallelized. This program outperformed others in increased thread counts as well as matrix sizes. A reason for this is because the most heavy workload for calculations occurs in the elimination portion, and by parallelizing just this portion balances the overhead required for thread spawning for the increased efficiency in calculations. We can also see that main_opt2.c spawning teams of threads for each for loop individually causes slowdowns as compared to main_opt1.c and main_opt4.c which has just 1 thread spawn at the beginning of the entire calculations and all threads join afterwards.

In terms of array sizes, we can clearly see that between the 100x100 matrix and 1000x1000 matrix, thread spawning with its overhead heavily underperforms as compared to serial when thread counts increase, however we do see a slight speedup with just 2 threads. We can also see that because main_opt3.c only has 1 thread parallelization section, its runtime is not as heavily affected by the overhead of more threads.

Parallelism Strategy	2 threads	4 threads	8 threads	12 threads	16 threads
Unoptimized (serial)	2.160683	2.161767	2.162407	2.162811	2.154831
main_opt1.c	1.572963	0.828143	0.850935	0.942297	0.978851
main_opt2.c	1.552429	0.791295	0.967609	1.304495	1.451663
main_opt3.c	1.585950	0.780203	0.854036	0.899369	1.095825
main_opt4.c	1.445031	0.724994	0.841200	1.205063	1.224931

Table 1. Runtime of parallelization optimizations in seconds of main_opt1 through main_opt4 for performing Gauss-Jordan elimination on a 1000x1000 matrix for varying thread numbers.

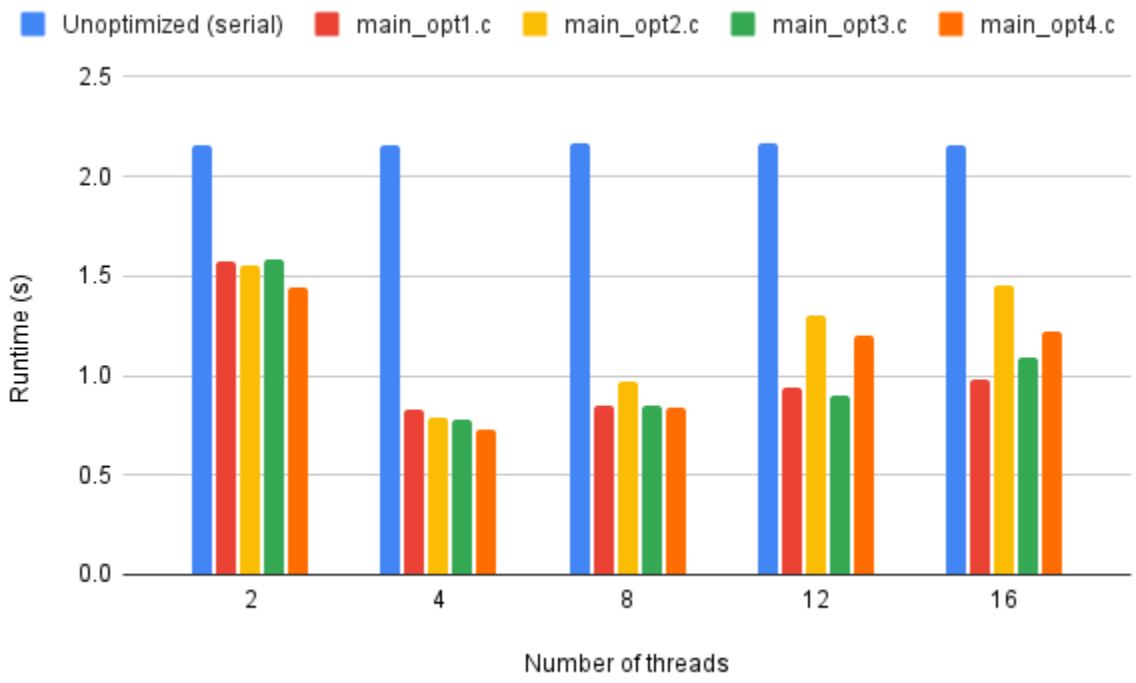

Figure 1. Bar graph of runtimes for programs main_opt1 through 4 for performing Gauss-Jordan elimination on a 1000x1000 matrix for varying thread numbers.

Parallelism Strategy	2 threads	4 threads	8 threads	12 threads	16 threads
Unoptimized (serial)	0.003186	0.003184	0.003168	0.003158	0.003173
main_opt1.c	0.003096	0.002334	0.014207	0.019062	0.027655
main_opt2.c	0.003249	0.002288	0.018020	0.024902	0.030407
main_opt3.c	0.002730	0.001760	0.008464	0.010015	0.013022
main_opt4.c	0.002836	0.001897	0.013868	0.020983	0.027777

Table 2. Runtime of parallelization optimizations in seconds of main_opt1 through main_opt4 for performing Gauss-Jordan elimination on a 100x100 matrix for varying thread numbers.

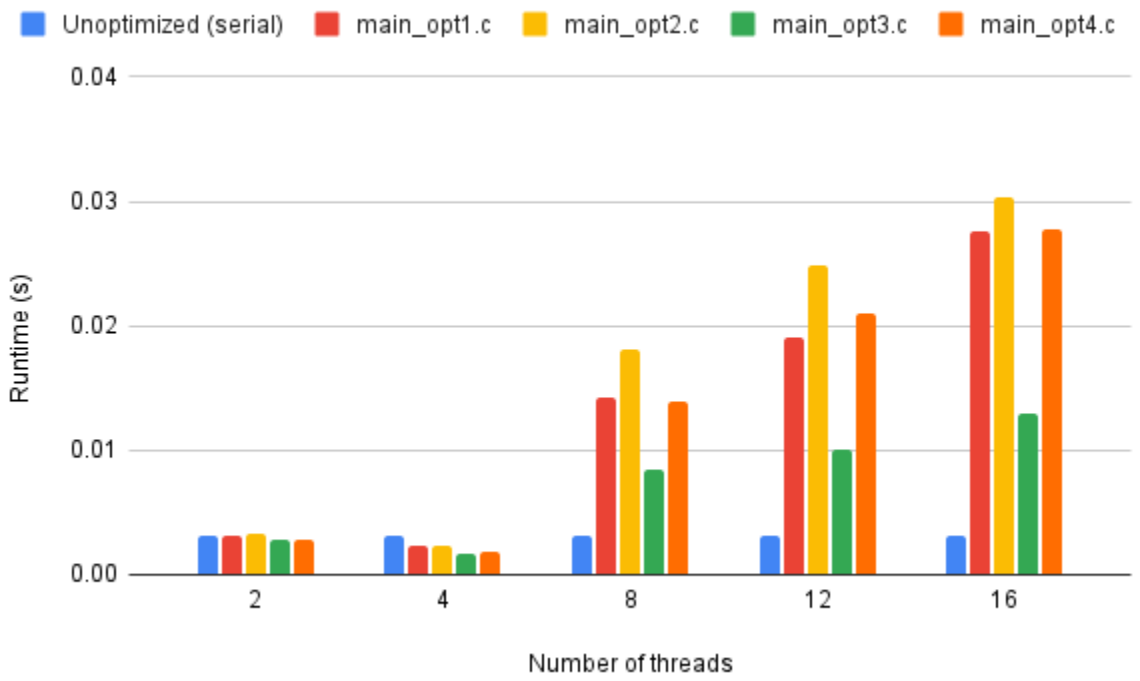

Figure 2. Bar graph of runtimes for programs main_opt1 through 4 for performing Gauss-Jordan elimination on a 100x100 matrix for varying thread numbers.