ECE 420

Lab 3: Solving a Linear System of Equations via Gauss-Jordan Elimination using OpenMP

Lab section H42

Kelly Luc
1498694
Nicolas Serrano
1508361

Implementation methods:

The implementation is based on a serialized version of a linear system of equations using Gauss-Jordan Elimination with partial pivoting. However, this implementation can be designed parallely to increase the efficiency of the system. This can be achieved using Openmp to distribute its threads and divide the workload to provide an increase in speed for performance. The requirement for this lab is to parallelize the Gauss-Jordan Elimination with partial pivoting and optimize it using Openmp.

Solution 1: adding parallelization on the entire algorithm

The for loops in the serialized version of the algorithm has a run time of $O(n^2)$, this makes it the main area that needs to be optimized. A critical section needs to be made when it is finding the row with the maximum absolute value. Afterwards a single thread can swap the new maximum row that can be used for the elimination portion. In the elimination and Jordan elimination portion this can be parallelized using the scheduling type guided to optimize it as much as possible by dynamically allocating the iterations to threads sharing only the matrix and its size being used and the row with the highest value.

Solution 2: parallelizing only in the elimination loops

In the Gauss-Jordan Elimination algorithm another method to parallelize it is by only parallelizing the main portion of the calculation which is in the elimination steps. By doing this we predict that the runtime will not be as good as solution 1 due to less parallelism being implemented. Both of the elimination steps have nested for loops which causes an increase in run time. By initializing Openmp to start running in parallel this can reduce the run time by distributing chunks to each thread. However no scheduling type was specified.

Solution 3: parallelizing with the guided scheduling, and reusing one thread pool

In our previous implementations, we had a method for Gauss Eliminations and a method for Jordan Elimination, where each of them called pragma parallel to create their own thread pool. The issue with these solutions is that there are “implicit barriers” at the end of each directive causing a possible increase in runtime. So this implementation, we decided to try combining Gauss and Jordan into one method, and having each process use the same thread pool. We also decided to try to use the “guided” scheduling in the for loops to compare with other scheduling types. We expect this method to scale well with different problem sizes.

Solution 4: parallelizing with the static scheduling, and reusing one thread pool

Just like solution 3, we will be sharing a threadpool, but instead using the “static” scheduling. Because it's static, we can expect longer run times with increased problem size.

Performance Discussion:

We expect to see that the parallelized versions of the algorithm will run slower than the serial version when the input matrix size is very small. This is because of the extra level of overhead that is required to keep a threadpool. However if the matrix size is large, the parallel version of the algorithm will run faster, as repetitive tasks are run in parallel. So even with the extra level of overhead in openmp, a large table will indeed be faster with parallelism when compared to just the serial version.

From our results, Main2 stayed consistent when it was using 2-5 threads. However, we notice that when the number of threads increases the average time it takes is longer. This is likely due to the task for all the threads being really small, which then causes it to have more overhead time slowing down the run time for the program. For Main3 the program average time for the program seems to decrease when there are more threads available. When two threads are available it shortens the run time by 4 seconds. The average run time decreases even further when we slowly add more threads. However we believe at a certain point, if the program has a lot of excess threads this can potentially affect the average run time.

For main4, we used a shared thread pool and the "guided" scheduling type in the for loops. We noticed from our data that as the number of threads increases, along with bigger problem sizes, we notice a consistent decrease in runtime, resulting in higher speedups. This makes sense because the "chunk sizes" in guided scheduling scale better with more threads and larger problem size, resulting in an increase in speedup. Our charts show main4 initially slower than other solutions, but then significantly faster with more threads. For main5 we followed a similar procedure but instead just using the "static" scheduling type. We noticed that an increase in threads actually reduced the speedup. This is most likely because of the increase in overhead with more threads. Remember that in Static scheduling, all the iterations for each thread are assigned prior to execution, as opposed to guided. So having less threads in Static actually reduces the overhead and overall produced higher speedups. Overall, the static scheduling shows better speedup for smaller sizes with less threads, and the guided scheduling has a high speedup across all problem sizes when there are a greater amount of threads.

Overall, the highest speed up we were able to obtain came from the third solution as it has the faster average run time in both matrix sizes tested. We found that the first solution would be the slowest as we increased the number of threads available for the program.

Appendix:

Table 1:

Average run time for implementations			
Program method	**# of threads**	**Average run time of matrix size**	
		100	**1000**
Serial	**1**	0.002008	1.7633586
Solution 1	**2**	0.00573715	1.60277145
	3	0.01253495	1.583828
	5	0.0160751	1.0950275
	10	0.0364702	1.1721748
Solution 2	**2**	0.002652	1.27844475
	3	0.003794	1.3394413
	5	0.011479	1.30645405
	10	0.02628	1.1456271
Solution 3	**2**	0.00579345	1.95117155
	3	0.00640465	1.2312686
	5	0.0114464	1.0213769
	10	0.0199032	1.08130945
Solution 4	**2**	0.0046225	1.639202
	3	0.00616515	1.64607925
	5	0.0115193	1.84509975
	10	0.02047285	1.14491355

Chart 1*:

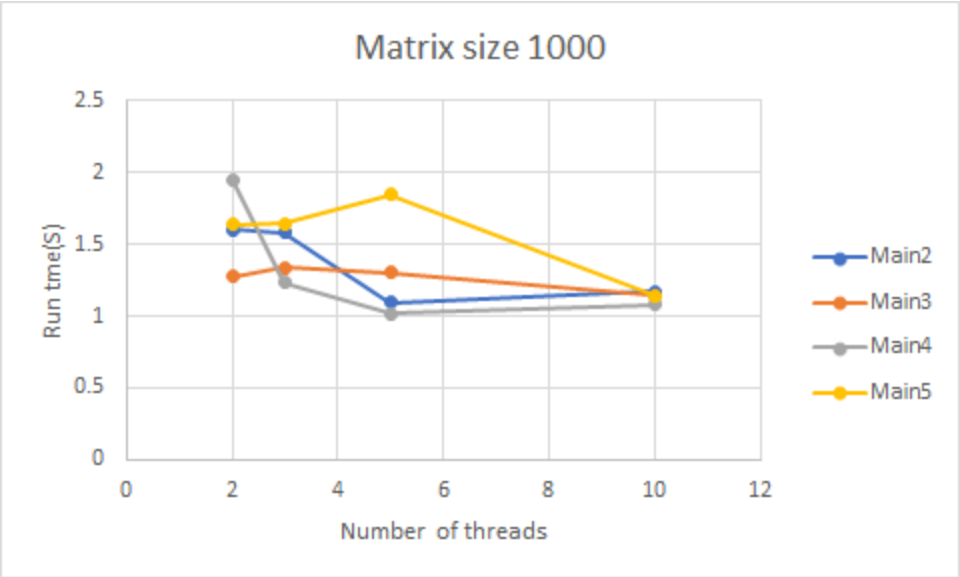
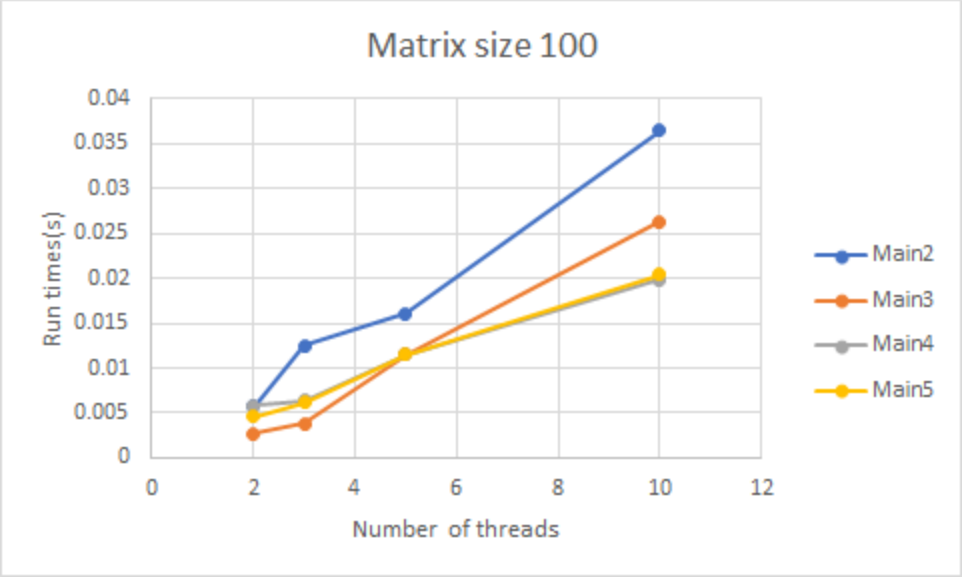


Chart 2*:



*Note solution 1 is Main2 followed by Main3 is solution 2