CS492B Assignment 1

Karl Gylleus
20166041

April 8, 2016

1 Parallelization of Gaussian Elimination

Calculating the solution x of $Ax = b$ through Gaussian Elimination using parallel threads is not completely trivial. Ideally one would like to assign an even amount of rows to each thread and just have them calculate their seperate rows independent of each other. However, every step when creating the upper triangle matrix depends on the previous one, thus it cannot be implemented this way.

For every row that we progress in the calculation we update all values in the same column and right that are under the matrix diagonal. Thus the only way I find to parallelize the calculation is to divide the computations of the current row between the threads.

Since each row beneath the current row and matrix diagonal will need to make updates we can still give each thread a couple of rows to process simultaneously. However, we will encounter work imbalance since different rows will have a different amount of elements to update. One could solve this by balancing the rows to give each thread an equal sum of elements to process. I believe this would ruin the cache locality however, and likely make performance worse. Thus I could not think of any easy way to solve this work imbalance problem in my implementation.

In my implementations I have only parallelized the calculation of the upper triangle matrix, since that is what took the absolute majority of workload.

2 Pthreads implementation

In my pthreads implementation I divide the workload by assigning specific rows to each thread. Since the amount of rows needed to be processed shrinks for

each row iteration I reassign which rows the threads have to make sure that none are idle. If the amount of rows are not divisible by the amount of threads, certain threads (the ones which row number can be rounded up) are assigned an additional row to make sure all gets processed. Before each iteration partial pivoting is performed by the first thread.

The threads are synchronized through the use of pthread_barrier. This is done after each row iteration to make sure that all threads are finished before beginning on the next one. Another barrier is used after the partial pivoting to make the other threads wait for the main one to complete the partial pivoting before moving on to avoid inconsistency. I realize that it is a waste of computational power to have the other threads wait instead of helping with the partial pivoting, but I did not find time to implement support for this.

When the upper triangular matrix is complete the main thread performs the backwards solving and checks the result.

3 Openmp implementation

In the openmp implementation it is the main thread that performs the partial pivoting. For each row iteration during the calculation the main thread creates the specified amount of threads to perform the computation. I believe this creates a lot of overhead to thousands of time create new threads, but I could not find any other way to make it work.

I use the dynamic keyword and let the library perform its magic to distribute the workload evenly. I would not prefer to use static since the workload between chunks is unevenly divided as mentioned earlier. The process is synchronized in that all threads need to finish before the main process will advance to the next row.

4 Results

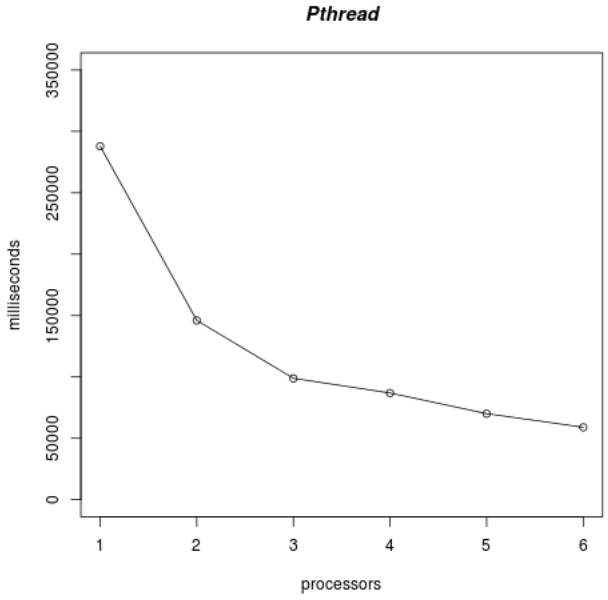
For pthread with 8000x8000 matrix using 6 cores: 237310 ms

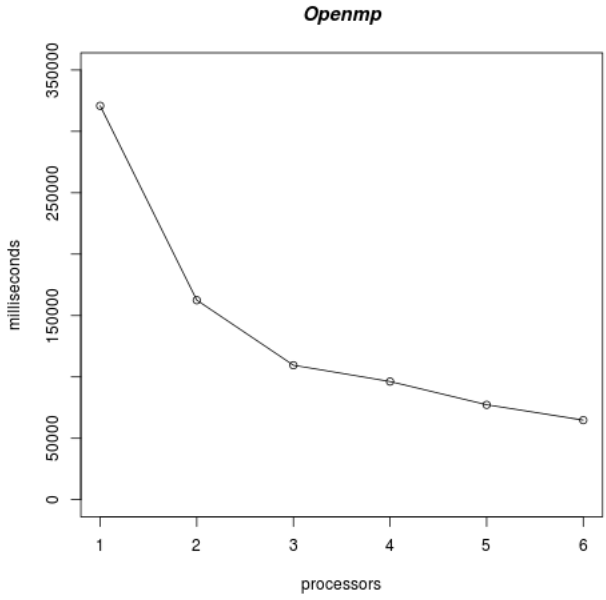
For pthread with 6000x6000 matrix using 6 cores: 101170 ms

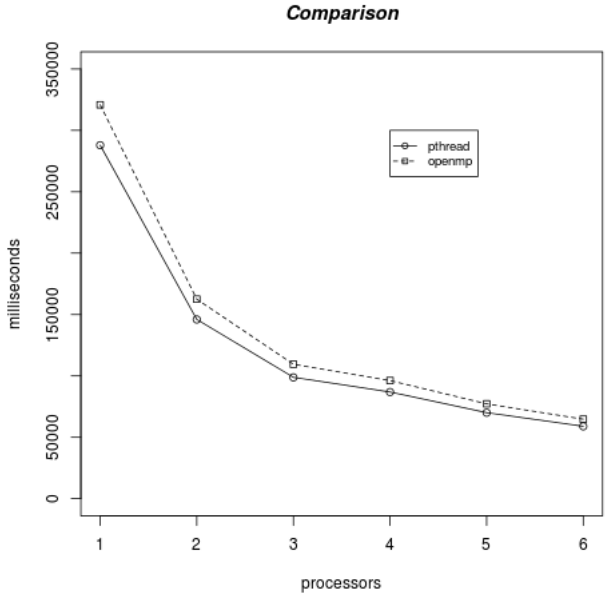
For pthread with 8000x8000 matrix using 6 cores: 58889 ms

The following results are run on matrix size $n = 5000$.

Cores	Pthread (ms)	Openmp (ms)
1	287944	320862
2	146005	162514
3	98852	109540
4	86917	96250
5	70050	77296
6	58970	64812







As the results show the parallelization of the two methods are both successful in that the execution time gets lower with a higher amount of processors. This process follows as it should, as the execution time with 2 cores is around half of that of one and for 4 cores it is half of that of 2.

The comparison graphs illustrates how the pthread implementation performs better for all amount of cores 1-6, but not by much. Towards 6 cores the openmp implementation seems to convergate towards the pthreads one, showing that as more cores are utilized they get closer in performance.