LU Decomposition Group 4

Contact info for questions: [joseph.chamberlain@ucdenver.edu](mailto:joseph.chamberlain@ucdenver.edu), [benjamin.lemarc@ucdenver.edu](mailto:ben.lemarc@ucdenver.edu), [michael.k.chen@ucdenver.edu](mailto:michael.k.chen@ucdenver.edu)

**Overview**: In numerical analysis and linear algebra, lower–upper decomposition or factorization factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. LU decomposition can be viewed as the matrix form of Gaussian elimination. (basically we take the diagonal of the matrix and split it in two, one gets all the values shoved in the upper triangle and one gets the values pushed in the lower triangle using maths.)

**Sequential**: (how it works, speed test/analysis, how to run it)

How it works: Simply put, we take matrix A and find the diagonal of the matrix, then we split matrix into l and u and by pivoting and doing matrix multiplication/division and subtraction we separate l and u to their separate triangles.

Test/Analysis: We ran the code in the small scale and tested for accuracy. To do this if you multiply l and u together you should get the original matrix A back. Which is what we got from our tests. From there we tested efficiency and with 1000x1000 matrix we got an average speed of a little over a second. Uploaded a few test files to see our results. Unfortunately the 1000x1000 tests with printed matrices are over the 25mb limit that github allows a file to be. Posted a 100x100 test to compare with open mp and extra proof of working code.

One change that will probably occur before final hand in, we will probably change the values from floats to doubles. Just so the numbers are a little less messy.

How to run it: Compile and run the code below. Currently it will print the matrix, if you comment out the following lines in the main function (print\_matrix(a, size); , print\_matrix(l, size); , print\_matrix(u, size);) if you want to run something super large. Most of our large tests were 1000x1000 but it can go bigger, above 5000 and the file will take a while to open and were very large, but running around 1000 is easy to open.

Compile: g++ -O lud\_sequential.cpp -o lud\_sequential

Run: sbatch c\_slurm.sh (insert a number here, it will make an nxn matrix of that size.)

**openMP**: (how it works in parallel, speed test/analysis, how to run it)

How it works:

Algorithm is the same as sequential, but it is brought into a parallel region where seperate threads calculate l and u faster than sequential. It’s fairly straight forward, the amount of threads are hard coded, it’s currently set to 40, a simple lock is initialized before initializing the parallel region, then the loop threads are scheduled statically. The initialization scheduling of the matrix is also static. From there it does l and u, just slightly faster.

Test/Analysis:

Code runs and runs slightly faster than sequential. A 1000x1000 matrix at 40 threads runs about .8 seconds. Speed can be altered by adjusting threads. So, the openMP code is working. Submitted an example program. At larger matrices openMP performed better. Sometimes at smaller sequential would win. Like the 100x100 example I’ll post. But this is to be expected, because parallel is supposed to be for larger data.

How to run it: Compile and run the code below. Currently it will print the matrix, if you comment out the following lines in the main function (print\_matrix(a, size); , print\_matrix(l, size); , print\_matrix(u, size);) if you want to run something super large.

Compile: g++ -O -fopenmp lud\_openMPpar.cpp -o lud\_openMPpar

Run: sbatch openmp\_slurm.sh (insert a number here, it will make an nxn matrix of that size.)

**MPI**: (how it works in parallel, speed test/analysis, how to run it)

How it works:

Same algorithm as the sequential one. At the beginning, a bcast sends the A matrix to each thread. In the first stage, each thread computes a certain number of rows of L (equal to matrix size / number of processes). A gather operation then sends then back to process 0, who then bcasts the result back out. In the second chunk, each thread computes a certain number of columns in U. Then another gather call is made to collect up each new column of U, and then the updated matrix is sent out to the other threads.

Speed test/analysis:

This program is much slower than the sequential. This I’m sure is partially because I wasn’t efficient enough in terms of sending the least amount of data out to the worker nodes, but also because distributed memory is just not the right model for performing this type of operation. A shared memory form of parallelism is preferable for this type of thing.

How to run it:

I had trouble getting a slurm job to work with this so please use the following command to run the program:

mpirun -print-rank-map -n [ntasks] -ppn [ntasksPerNode] -f hosts ./[executable name] [matrix size]

Also use this to compile:mpicxx lud\_mpiv4.cpp -o [executable name]

Please only run it when matrix size is divisible by ntasks.

The max matrix size I've been able to run was of size 1,544, but it takes a while (much, much longer than the sequential version (seriously you might think it was diverging but it's just takes that long)).

**Cuda**: (how it works in parallel, speed test/analysis, how to run it)

How it works:

The CUDA program will make a kernel call that takes the matrix and begins finding the pivot row. After finding the pivot row each thread will then use the same pivot row (shared memory) to calculate its matrix index. The kernel calls to find a pivot is done N times and spread across the threads for each iteration.

Speed test/analysis:

CUDA ended up being twice as fast as sequential at 1,000 x 1,000 matrix.

When the matrices are small CUDA is actually slower. This is shown in the sample slurm output.

How to run it

Once lu\_slurm.sh and luCuda.cu are in the same directory

The CUDA program must have at least 1 argument, and the 2nd argument is optional.

If the second argument is 1 then it will print the L and U matrices, otherwise it will only give the run time.

ssh node18 nvcc -arch=sm\_30 /path/to/directory/luCuda.cu -o /path/to/directory/lu

sbatch lu\_slurm.sh [matrix size] [optional]

sbatch lu\_slurm.sh 10 1 #Prints a 10x10 matrix and its L and U matrices