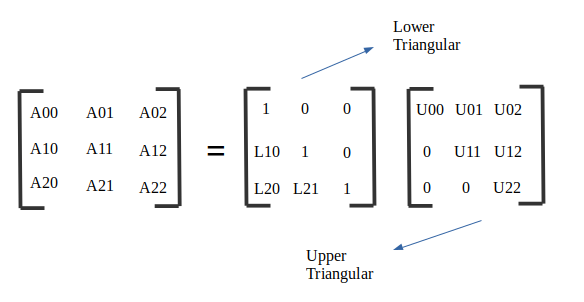
**Abstract:**

For our final project, we were tasks to build solvers for computing the lower and upper matrices of a matrix sequentially and in three implementations for parallel programs: OpenMP, MPI, and Cuda. This is known as LU decomposition, and it is used for solving linear equations, computing the determinant of a matrix, inverting a matrix, and others. What follows is a report that discusses this more in-depth and shares our implementations of this process in sequential and in the three parallel programming paradigms.

**Introduction:**

The problem we’re tackling is Lower-Upper Decomposition (LU decomposition). LU decomposition can be used to break a matrix into two different matrices: (1) the lower matrix which has non-zero values in a lower diagonal region of the matrix and (2) the upper matrix which has non-zero values in the upper diagonal region of the matrix. A picture will likely do better than just words at explaining the forms these matrices take:



The goal, however, is not just to right a regular, sequential algorithm that computes the L and U matrices for any non-invertible matrix A. That is just step 1. The next 3 steps are to implement three parallel programs based on the sequential programs, one for each of the paradigms of parallelism that we’ve learned this fall. Those are Cuda (GPU-based), OpenMP (shared memory), and MPI (distributed memory). This is important because LU decomposition traditionally has a O(n^3), so for large matrix sizes, it can become infeasible to run them in sequential. Any processor time that parallel programming can save us would make larger matrix sizes more practical to solve.

**Background:**

Often in computer science, it is important to solve a linear equation of the form A\*x = B, with A being a matrix, x being a vector, and b being the solution vector. Rather than directly solve this using matrix multiplication, A can be broken down into matrices L and U. Then we can take we can solve for y by using L\*y = b (forward substitution), and use U\*x = y to find x (background substitution). The reason to do this is that L and U can be used to solve for many vectors.

LU decomposition also has other uses, for instance to find the determinant of a matrix. After finding L and U, we can get the product of their diagonal indexes, which can be used to find the determinant of A. It can also be used to invert a matrix in a less expensive.

**Implementation:**

Sequential (program written as a team effort):

We implemented a sequential version to base our parallel version off of. First, we fill the array randomly. Then, we ensure the matrix is diagonally dominant. This ensures the matrix is guaranteed. This was added by Michael near the end to make sure we didn’t run into any issues. This is done when filling the matrix with pseudo-random values rather than during LU decomposition.

Implemented sequentially, LU Decomposition has a O(n^3). It consists of an outer loop that iterates a number of times equal to the size of the matrix. Then there are two for loops at the same scope level inside the outer loop. One of those goes through rows of the matrix and sets the values in the lower matrix. It uses the i from the outer loop to iterate through columns. If the row index lower than the column index, then that cell gets set to 0. Otherwise, the cell in the L matrix is set by first setting it equal to the same cell in the A matrix. It then goes to another inner for loop, one that iterates through a column in L and a column in the U matrix, doing a dot product and taking the product out of the value in the lower matrix.

The second middle loop in the function is similar to the first, but iterates through columns and sets values for the upper matrix, and uses the i from the outer loop to go through rows. It has the same condition of the first which is that if it’s own index is less than the outer loop’s index, the value is set to 0. If they’re equal to each other (the longest diagonal of the matrix) it gets set to 1. Otherwise, a similar calculation is done as the first middle loop - involving its own inner loop - but the product is taken divided by a corresponding cell in the lower matrix. So after an outer loop, two middle loops, and two inner loops, we have L and U, where A = L\*U.

MPI (program written by Ben LeMarc):

To be completely honest, the MPI implementation was not easy. It was a considerable challenge to suss out which threads needed what data. First off, like the MPI lab, I had difficulties getting a 2D array to work with MPI communication functions, so I had to implement the matrix as a 1D array with the matrix stored in row-major order. An equivalent matrix represented as a 2D array would have its A[i][j] index correspond to A[i \* size + j] in this single array implementation.

I decided the best way to get any implementation working was to first just concentrate on getting all of the data to the threads, so there was no issue getting the calculation right. I gathered and broadcast the L and U array once for each iteration of the outer loop, which is very costly. Assembling the columns, which is necessary for calculating U, and then transposing the result of the gather operation from column-major order to row-major order is also costly.

First I implemented an MPI version that works as described where each worker thread ends up sending back a single row of L (calculated in the first middle loop) and a column of U (calculated in the second middle loop). This worked as long as the threads used was equal to the amount of rows used. During this version of the program, the middle loops were removed, replaced by the MPI functions for sending and receiving data.

To handle situations with more rows than threads, I adjusted it to calculate rows\_per\_process, calculated by taking the number of rows divided by the number of processes. This dictates how many rows (or columns) are calculated per thread and sent back to process 0. As long as rows is divisible by the number of threads used, this works. With this new update, the middle loops returned, as each worker thread iterates through it a number of times equal to the number of rows (or columns) it has to calculate.

Due to the issues described in the second paragraph, as well as the fact that distributed memory is easily the worst choice of the three programming paradigms considered for implementing LU decomposition, the MPI program runs much worse than the sequential version. If I had another semester to work on this, I would trim down as much as possible how much data from the matrices are sent to the various threads. This is what I assume is responsible for much of the time lag involved.

Cuda (program written by Michael Chen):

The CUDA implementation was done differently than how the lower and upper triangular matrices were calculated in the other programs. The implementation also uses shared memory method with each thread having a copy of the pivot row. Since the time and overhead to read and write to global memory is higher than shared memory, using shared memory improved the efficiency of the CUDA program and therefore also helped lower the runtime.

There are two kernels that the program uses. The “scale” kernel helps to find the appropriate index values that each thread has to use for its corresponding index values for whichever row that it is operating on. The “elim” kernel is used to do the row reduction in order to calculate the lower and upper triangle. Rather than cause the overhead of also creating separate matrices for the lower and upper triangles, and having each thread write to its appropriate element of the matrix, which would also require every thread to have a copy of the matrix to write its value it, the threads all write to the same original matrix that the GPU receives at the start of the kernel call. This also helps to reduce overhead and improve the runtime.

Each thread then does the appropriate row operation for the rows it calculates, and does this so long as the “my\_row” value is greater than “pivot\_row”, which just means that it calculates each row below the pivot row. The program calls both the kernels N times. Which is just the number of rows in the matrix. It has to do this because for every time the “elim” kernel is called, it is only finding 1 pivot row, which means it has only eliminated 1 column for each of the N-1 remaining rows. Since there needs to be N pivots for a triangular matrix, it must do this N times.

Since the resulting matrix is no longer the original matrix, but a matrix with the lower and triangular matrix values in one, the lower and upper triangles must be extracted from it. This is done starting at line 176 where there is a double nested for-loop (3 for loops in total), where the indexes for the lower and upper triangles are found and placed into their respective matrices.

The program also can take two inputs, the 2nd input is optional as it is used to print the ‘A’ matrix, ‘L’ matrix , and ‘U’ matrix.

OpenMP (program written by JD Chamberlain):

With openMP I had to choose between Dynamic and Static scheduling methods. At first I tried Dynamic to see if I could get the basics of the scheduling down. I would eventually turn everything static as it would end up being slightly faster especially at larger datasets.

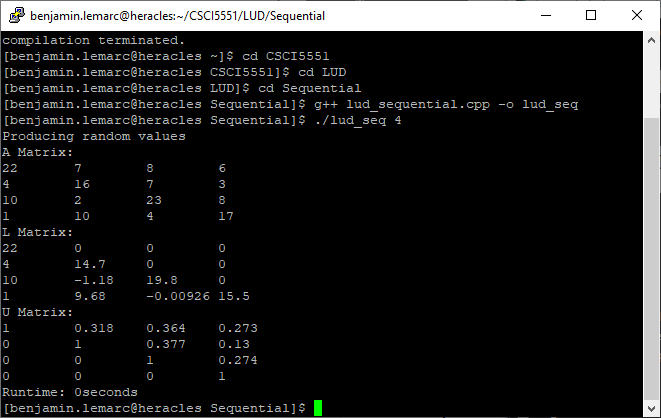
So, I took our sequential code and next was to figure out what sections needed to be parallelized and exactly where. So, because we were taking the matrix processes and and splitting them up I knew there was 2 places we needed to create the parallel sections. The main one was the for loops of our main LU Decomposition function and the other part was the initialization function of the matrix itself. So, at the beginning of the lu function I initialize a lock so when we enter the critical section we guarantee some instructions can only be performed one process at a time. Then I initialized the main function for lu decomposition as the parallel region with the matrix variables a,l,u as shared variables. These with be the processes that will be parallelized. Then on the u for loops set up a pragma schedule, which started off dynamic, but made it static, and did the same for the l for loops. This is how the main parallel region works. One last note here, we set up a pragma schedule in the initialization of the matrix too, cause variables a,l,u were used here as well.

So, the last thing to do was in main, we needed to set the number of threads for omp. I hard coded the amount of threads in, for the datasets I was running, I found 40-42 threads worked really well. But too many threads and you start losing efficiency, and to few you’re not using enough parallel power. But as you can see from the report, openMP worked really well for running the processes of l and u.

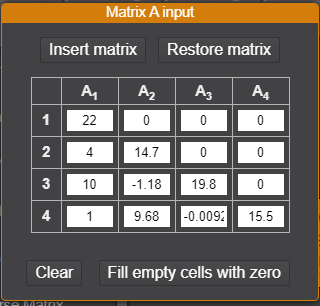
**Results and Analysis:**

Sequential proof of correctness:

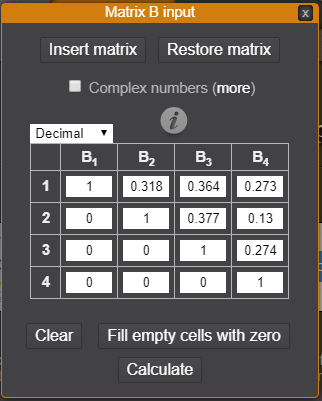
First, we had to build a sequential program for performing LU decomposition. If we can prove the sequential program is correct, we can use it for comparison to make sure the other programs are correct. Here is the sequential program performing LU decomposition on a 4 x 4 matrix:



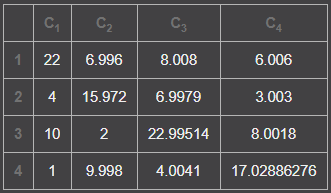
In order to check this, we will utilize a matrix multiplication calculator accessible for free from the internet. A = L\*U, so if we have L and U in their correct forms and they can be multiplied together to get back the original A matrix, we know it is correct. Here is the L matrix:



Here is the U matrix:



And finally, here is the result of multiplying L and U:

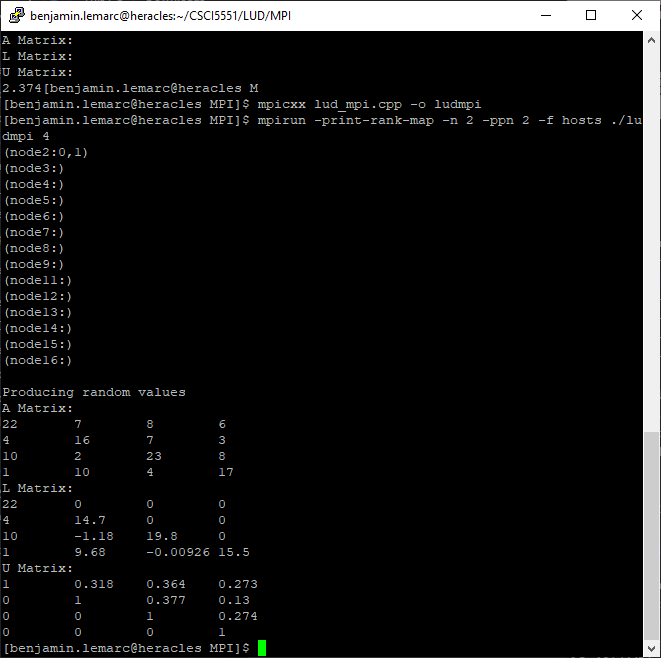


We don’t get all the whole numbers we started at, but the rounded values shown above are extremely close for floating point math.

Now we can use our sequential program going forward to verify correctness of the parallel programs.

MPI proof of correctness:

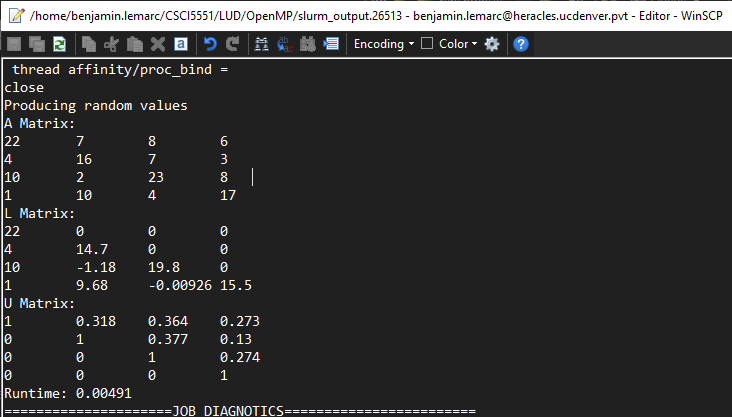
Here is the same 4 x 4 matrix shown above calculated by the MPI program:



As you can tell, we start with the same A matrix as the sequential program, and we end up with the same L matrix and U matrix as well.

OpenMP proof of correctness:

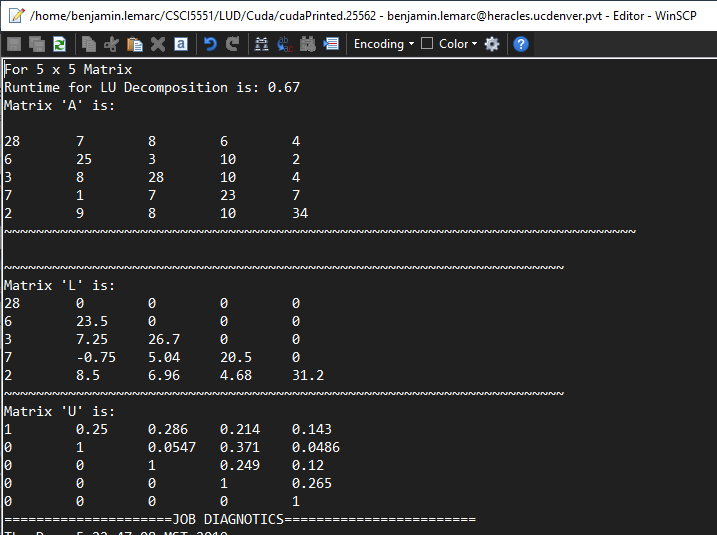
We will again run the same 4 x 4 matrix, this time with OpenMP:



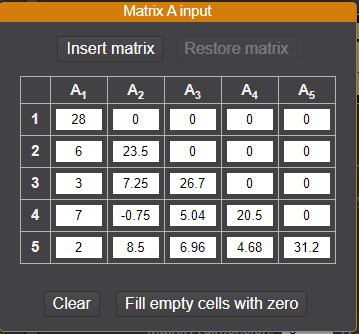
Cuda proof of correctness:

The sequential program’s algorithm was not well suited for Cuda, so a different LU decomposition was used when implementing it. Thus, we will not use the sequential program to verify its correctness, but instead use the matrix multiplication calculator to make sure that it is indeed splitting A up into valid forms of L and U.

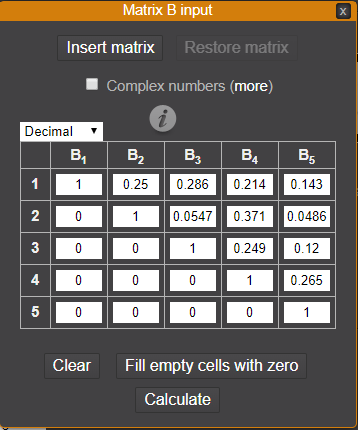
Here is Cuda’s LU decomposition on a 5 x 5 matrix:



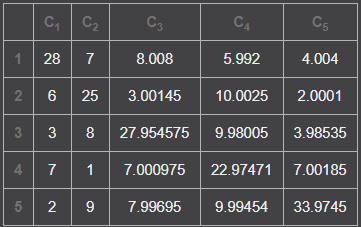
And the same matrix being solved by the online matrix multiplication tool, first L:



And then U:



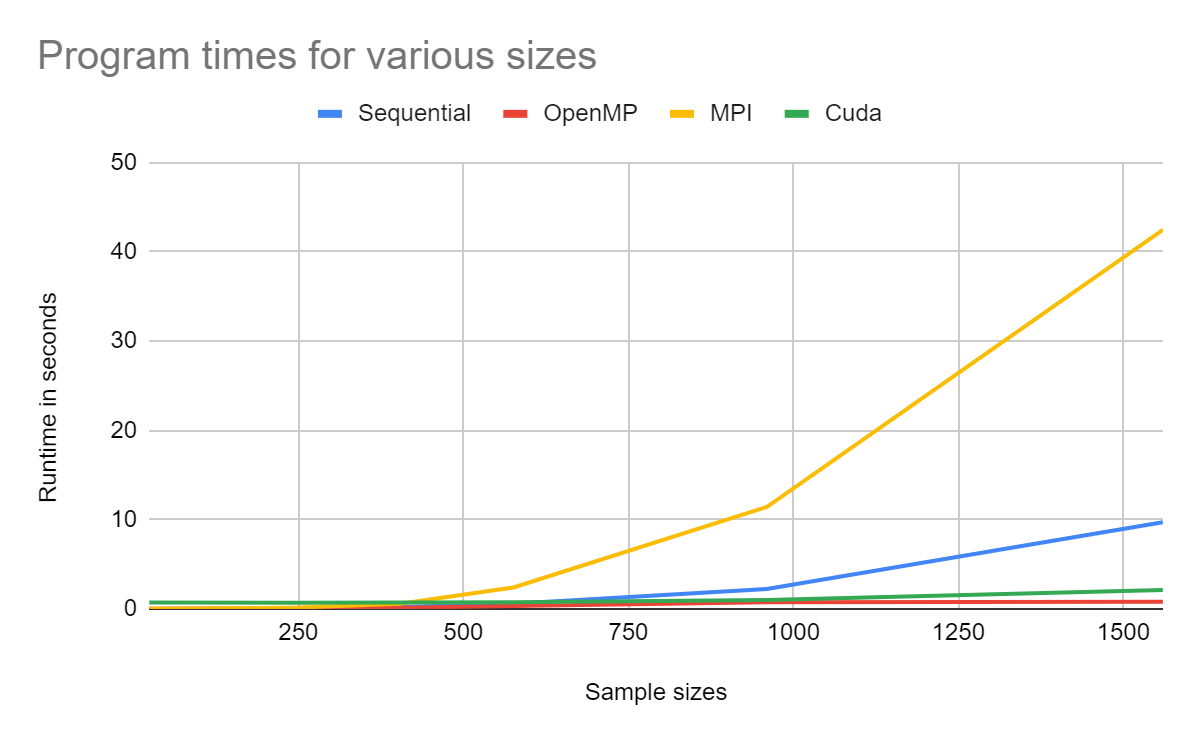
And lastly, reconstructed A matrix:



Again, like with the sequential check, a lot of these numbers aren’t exact, but they’re about the best we can do with floating point math.

Runtime Analysis:

Here are all four programs compared to each other in run time over a different matrix size:



As you can see, the results of the MPI times are not pretty. The Cuda and OpenMP programs however do much better than the sequential program even at sizes as (relatively) small as a 1,500 matrix size.

**Conclusions:**

To review, our mission was to implement LU Decomposition first in sequential, and then use that as the basis for building three parallel programs. We succeeded in fulfilling this goal. However, only two of those actually ended up being quicker than the sequential version. MPI was considerably difficult to implement, and the need to send data around to different nodes often combined with not employing an optimally efficient method for sending and receiving data meant that it ended up much slower than the sequential version. OpenMP may have performed better than CUDA due to the fact that CUDA must write all the data to the GPU before it can begin its calculations. When examining the CUDA slurm error file, the cost to allocate GPU memory, and copy the CPU data to the GPU was about 62%+ of the runtime. Since just those two tasks made up for almost two-thirds the runtime, it is what had the greatest impact on CUDA’s performance when compared to OpenMP.