## Authors:

Andrew Kizzier

Rachel Krohn

James Tillma

Jonathan Tomes

## Included:

Makefile

prog4\_shared.c

prog4\_dist.c

make\_matrix.c

Prog4.pdf

Matrices for testing:

large\_matrix.txt

small\_matrix1.txt

## Compiling:

Simply use the included make file.

Or for the individual programs:

Shared Memory Version:

gcc –g –Wall –fopenmp –o prog4\_shared prog4\_shared.c

Distributed Memory Version:

mpicc –g –Wall –std=c99 –lm –o prog4\_dist prog4\_dist.c

The make\_matrix program:

gcc –g –Wall –o make\_matrix make\_matrix.c -lm

## Usages:

Shared:

prog4\_shared n

Where n is the number of processes.

Distributed:

mpiexec –np n prog4\_dist

Where n is the number of processes.

NOTE: The number of rows in the matrix must be divisible by the number of processes.

MakeMatrix

make\_matrix n

Where n is the size of the desired matrix.

## Problem:

Perform LU decomposition of a matrix of a set number of rows. And we must perform this using both a shared memory version and a distributed memory version. We are requiring that the matrix be a square matrix as many of the applications for the L and U matrices require that the matrix be square.

## Algorithms:

We will use Gaussian row elimination in order to create the upper right matrix that is U and the lower left matrix that is L. This requires that we do row swapping as we discover the need to. Upon eliminating a row we insert the scalars used into L. After we have completed all row eliminations we print each matrix (A, P, L, and U) to output. We then print to output that A = (LU)\* P’ to show that LU decomposition was successful.

## Timings:

### Shared:

This is a chart of the time the LU Decomposition takes versus the size of the matrix. Note that a barely visible trend line based on the power of X (so y=x^2, y=x^3, etc.) has been drawn on this graph. The trend line can be seen to follow a power of roughly 2.5 when based on 2, 4 or 8 threads. Each thread count was run on the same number of items, and same matrix 5 times. If the trend line were placed of the 1 Thread line, it would very closely follow x^3. This gives us an idea that our parallel algorithm is in the category of O(n^3), but it is to the lower end of the category. Whereas a sequential solution would be on the larger side of O(n^3).

This is a graph of speedup versus size of the matrix. For the smallest size, the speedup is very high compared to the others. This is probably because the outermost loop in the entire algorithm has many data dependencies and it is actually the second layer loop that gets the parallel command. That means that the system takes longer to collect all of the threads after a parallel section before hitting another parallel section. Next is that the 8 thread solution is consistently worse than the 4 thread solution. This is probably because the system only has 4 physical cores and can run 8 threads through hyper threading. Lastly, the overall speedup across all solutions is fairly poor. This may be caused anything from algorithm faults (already discussed) to other students being remotely connected to and over-using the same machine that was used for testing.

This is a graph of efficiency versus size of the matrix. The data for this graph is fairly obvious to look at. The smallest test matrix size was the most efficient for every number of threads. As is to be reasonably expected, the least efficient solution was the one being run on 8 threads.

# Foster’s Algorithm

## Partitioning & Communication

There are three main tasks for this problem of LU decomposition, calculating the U matrix, the L matrix, and the P matrix. Calculating the U matrix is divided into each row and determining how to eliminate the rows in the column below the row’s starting value. Calculating the L matrix, is just a matter of finding the scalar that was used in eliminating the rows in calculating the U matrix. The P matrix can be calculated as we swap rows as necessary, looking to put the max absolute value in the diagonal. So the main tasks we can put into parallel is calculating the scalars to calculate the matrices.

We would only need to communicate a single row of each matrix (A, L, U, and P) to each process. However, after experimenting with rows we attempted to apply the algorithm to columns instead so we communicated the columns instead of the rows.

## Agglomeration

Each, row in U needs to eliminate the values in the column below its starting value. Since L is dependent on these scalar values, we need to calculate matrix U first. However, since we have the scalars at the time of calculating U, we can actually combine the two tasks. As we step down through the rows in the U matrix, we can insert the resulting scalars into matrix L. This means we will have to only communicate each row of U at each phase of the program. This also is true of creating the P matrix. So we can create the L and P matrices at the time of computing U.

## Mapping

We can map each row in U to a process. This would result in solving the U matrix, L matrix, and P matrix in Phases, one for each row. We chose this approach as it was the simplest and only approach for the problem.