Program 4 Documentation – LU Decomposition

Authors:

Andrew Kizzier
Rachel Krohn
James Tillma
Jonathan Tomes

Included:

- Makefile
- prog4_shared.c
- prog4_shared_col.c
- prog4_dist.c
- prog4_dist_row.c
- make_matrix.c
- prog4.pdf
- Prog4Dist.xlsx
- Prog4Shared.xlsx
- Matrices for testing:
  - 3
  - 4
  - 4b
  - 4c
  - large_matrix.txt
  - large_matrix2.txt
  - large_matrix3.txt
  - large_matrix4.txt
  - large_matrix5.txt
  - large_matrix6.txt
  - large_matrix7.txt
  - small_matrix1.txt

Two versions of both shared and distributed memory solutions are given. The two versions that specify column and row are slow solutions and were not timed. make_matrix.c was used to randomly generate test matrices. The pdf and spreadsheets give documentation and timing data. A collection of test matrices are also provided. The large_matrixX.txt were used for timing data collection. Matrices small_matrix.txt, 3, 4, 4b, and 4c were small matrices; the solutions of these cases were verified by hand. Some files include the expected results below the input data. These can be used to verify correctness if the programs are compiled with the DDEBUG option.

Compiling:

Simply use the included make file.
Or for the individual programs:

Shared Memory Versions:

```
gcc –g –Wall –O –fopenmp –o prog4_shared prog4_shared.c
gcc –g –Wall –O –fopenmp –o prog4_shared_col prog4_shared_col.c
```

Program 4 Documentation – LU Decomposition

Distributed Memory Version:

```
mpicc –g –Wall –std=c99 –lm –o prog4_dist prog4_dist.c
mpicc –g –Wall –std=c99 –lm –o prog4_dist_row prog4_dist_row.c
```

The make_matrix program:

```
gcc –g –Wall –o make_matrix make_matrix.c -lm
```

Usages:

Shared:

```
./prog4_shared <n> < <matrix>
./prog4_shared_col <n> < <matrix>
```

Where< n> is the number of processes, and <matrix> is the input matrix file.

Distributed:

```
mpiexec –n <n> -hostfile <list of hosts> ./prog4_dist < <matrix>
mpiexec –n <n> -hostfile <list of hosts> ./prog4_dist_row < <matrix>
```

Where <n> is the number of processes, <list of hosts> is the hostfile, and <matrix> is the input matrix file.

NOTE: For the distributed solutions, 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 square matrix.

Problem:

Perform LU decomposition of a matrix of a set size. Shared memory and distributed memory versions are required. 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 with partial pivoting in order to create the upper right matrix U and the lower left matrix L. This requires that we do row swapping to ensure that the maximum values lie on the diagonal. This means that all values in L will be less than 1. After subtracting a row (performed after any necessary swaps), the scalar multiplier is placed in L. After we have completed all row eliminations the elapsed time is displayed. If the program was compiled in DEBUG mode (only recommended for small matrices), we print each matrix (A, P, L, and U) to output and then print the result of P'(LU), which should be equal to A.

For shared memory, the row-wise partitioning scheme was more efficient (prog4_shared.c). This is because when column-wise division was used, deep swaps were required, impacting performance. Swapping rows required a simple pointer swap, so they were very fast. Additionally, the row to subtract does not have to be sent to the different threads. The column-wise version (prog4_shared_col.c) is also included for completeness.

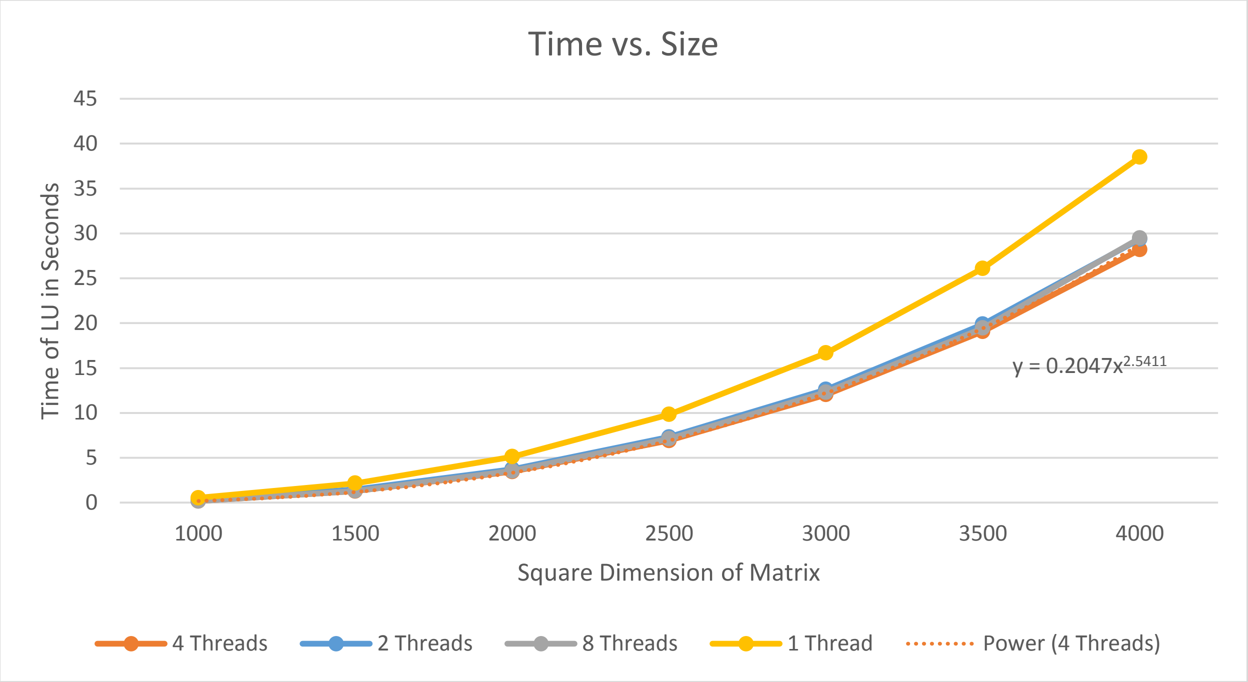
Distributed memory required a different algorithm. In this case, column-wise partitioning (prog4_dist.c) was orders of magnitude faster than row-wise. This is because this scheme limited the communication and increased the parallelism of the deep swaps. The row-wise solution is also included (prog4_dist_row.c).

Program 4 Documentation – LU Decomposition

Timings:

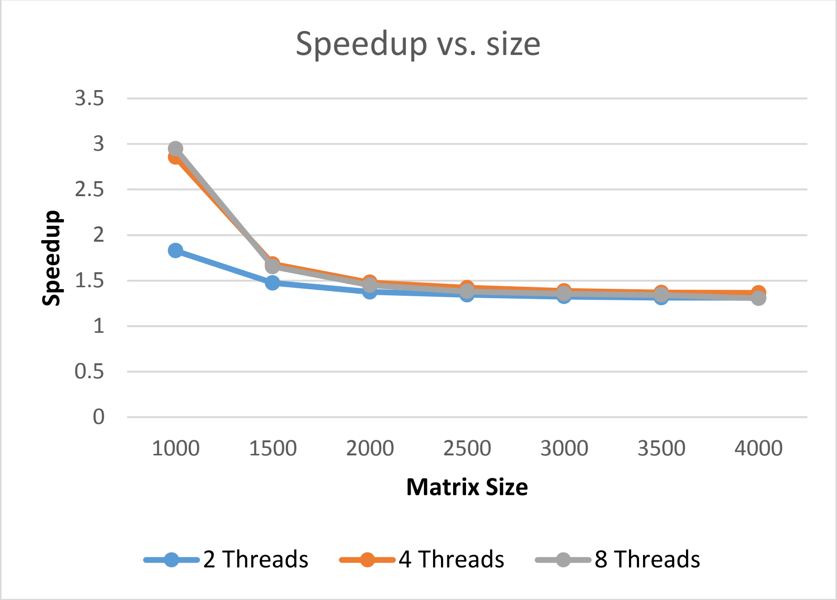
Shared:

Timing data for prog4_shared.c is presented below. prog4_shared_col.c was not timed because of poor performance (roughly 4.4 seconds on a 1000 row matrix on 4 threads).

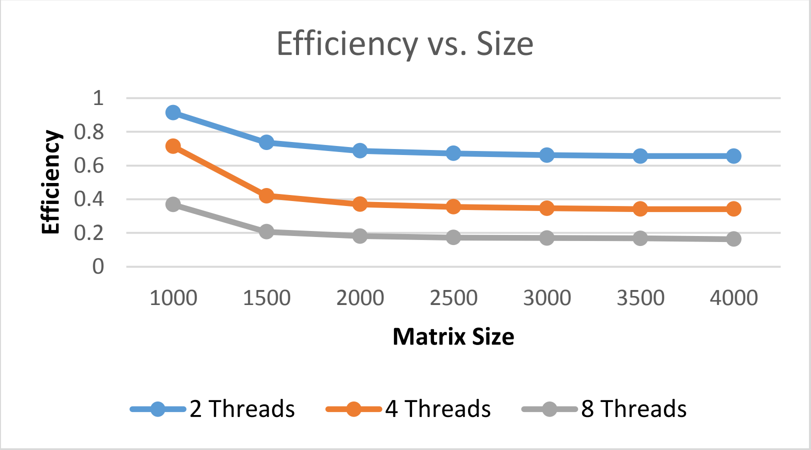


This is a chart of the time the LU Decomposition takes versus the size of the matrix. Note that a barely visible trend line based on the power of X (so y=x^2, y=x^3, etc.) has been drawn on this graph. The trend line can be seen to follow a power of roughly 2.5 when based on 2, 4 or 8 threads. Each thread count was run on the same set of matrices, and each test was repeated 5 times. If the trend line followed the ‘1 Thread’ line, it would very closely follow x^3. Both fit lines give us an idea that our parallel algorithm is roughly O(n^3).

Program 4 Documentation – LU Decomposition



This is a graph of speedup versus size of the matrix. For the smallest size, the speedup is very high for all thread counts. 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. Note that the 8 thread solution is consistently worse than the 4 thread solution for larger matrices. 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 by algorithm faults (already discussed), or other students being remotely connected to the same machine that was used for testing.

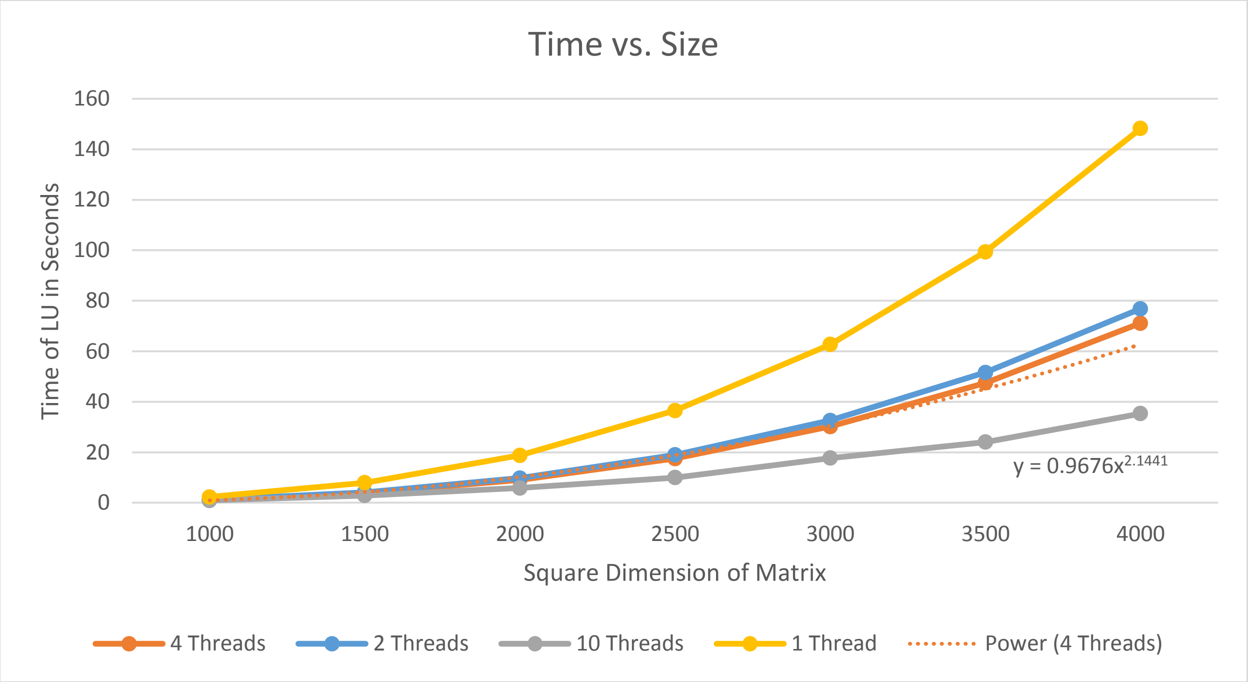


This is a graph of efficiency versus size of the matrix. The smallest test matrix size was the most efficient for every number of threads. As is to be reasonably expected, the least efficient solution was the one being run on 8 threads.

Program 4 Documentation – LU Decomposition

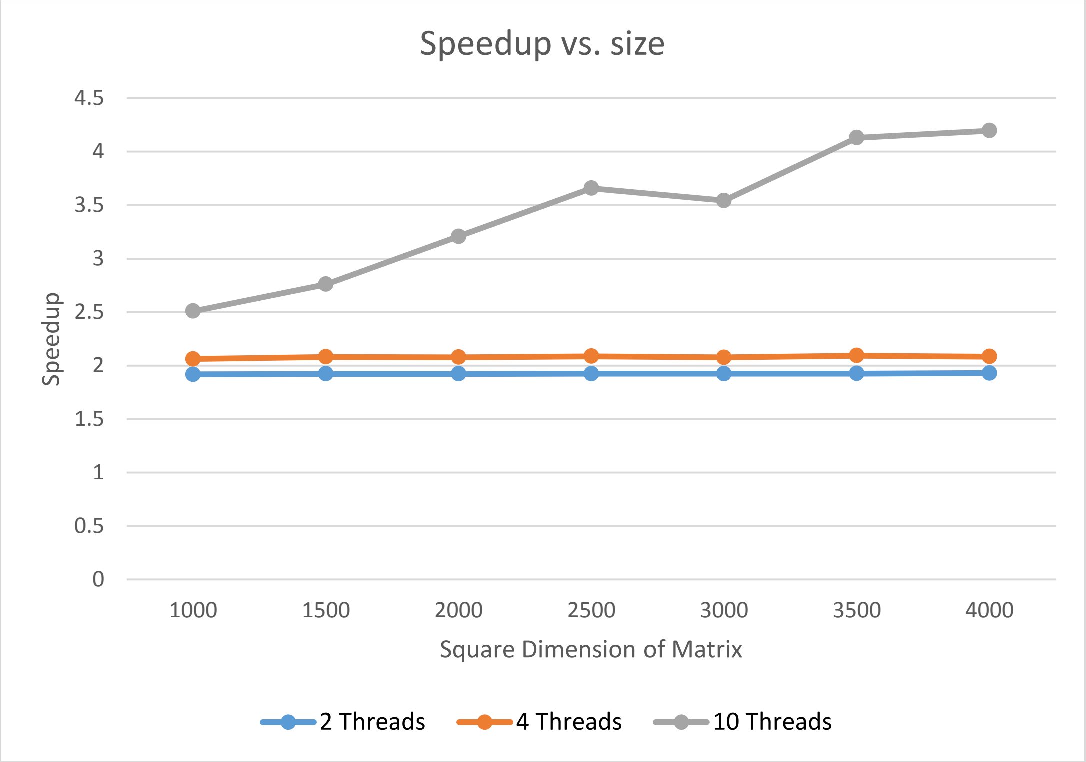
Distributed:

Timing data for prog4_dist.c is presented below. prog4_dist_row.c was not timed because of poor performance (roughly 42 seconds for a 1000 row matrix on 10 processes). For all tests the processes/node ratio was limited to 2.

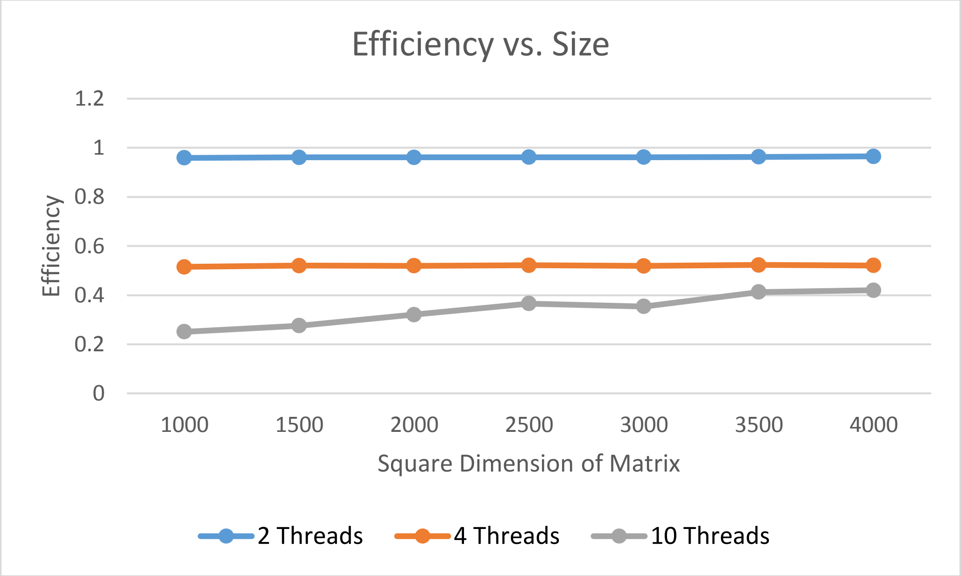


This is a chart of the time the LU Decomposition takes versus the size of the matrix. Again, a trend line has been drawn. The trend line can be seen to follow a power of roughly 2.14 when based on 4 threads. Each thread count was run on the same set of matrices. When runtimes became prohibitive, that matrix-process combination was only run 3 or 1 times; all other tests were repeated 5 times. The included spreadsheet gives exact data. The fit line indicates that our parallel algorithm is roughly O(n^3), but slightly better than the shared solution.

Program 4 Documentation – LU Decomposition



This is a graph of speedup versus size of the matrix. Speedup for 2 and 4 threads remains mostly constant across all matrix sizes. When run on 10 threads, the speedup increases as the matrix size grows. These results suggest that, given enough processes, the algorithm is very efficient.



This is a graph of efficiency versus size of the matrix. As is to be reasonably expected, the least efficient solution was the one being run on 10 threads. However, the efficiency increased as the matrix grew.

Program 4 Documentation – LU Decomposition

Foster's Algorithm

Partitioning

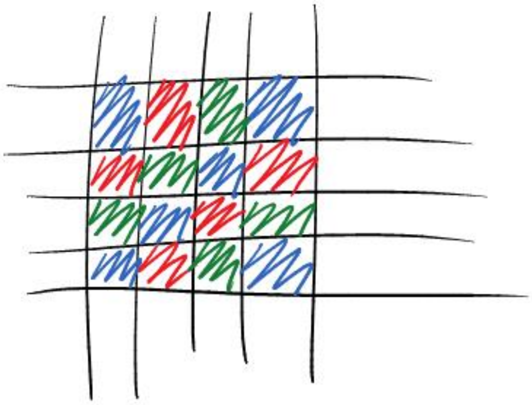
There are three main tasks for this problem of LU decomposition, calculating the U matrix, the L matrix, and the P matrix. The U matrix is constructed with Gaussian row elimination. The multipliers utilized in the elimination are used to construct L. Anytime a row swap is required based on partial pivoting, the rows in P are swapped to eventually produce a permutation matrix.

The maximum potential data partitioning is to give each individual element to a different process. This is totally impractical, and will not be discussed further. There are three potential block-based partitioning schemes: row-wise, column-wise, and cyclic block allocation.

Task parallelism is not really applicable for this problem, because the same set of tasks (row elimination and swapping) is performed many times to produce the final matrices. The data parallelism is much more prevalent here, so task parallelism is not used.

Communication

For the cyclic block allocation scheme, each process receives a collection of square matrix blocks. This could result in an allocation like the one below.



To find the maximum value in a column (to determine if a swap is required), all the blocks below the diagonal in that column would have to communicate their local maximums. To swap a row, blocks owning the first row would have to communicate with blocks below that own the second row. To eliminate a row, the blocks in a single column would have to communicate the scalar multipliers horizontally to all blocks. Additionally, the blocks containing the row to subtract have to communicate the elements of that row to all blocks below it. This is a significant amount of communication, and overhead would greatly impact performance. It would also be very difficult to determine exactly which processes own the data necessary for each communication. Therefore, this method was abandoned.

Now, consider the row-wise partitioning scheme. To determine if a row swap is necessary, each process has to send its local maximum for the column under consideration to a single process, where the global maximum is determined. To perform a row swap, the two processes owning the rows must communicate the data in the rows. In a shared memory solution, this can be accomplished with a simple pointer swap. In a distributed memory solution, however, individual values must be swapped to maintain contiguous memory storage. This means that all elements in the row must be looped and assigned individually, impacting performance. Finally, to eliminate a row, the process owning the row to subtract has to send that row to all other processes.

Finally, consider the column-wise partitioning scheme. To determine if a row swap is necessary, the process owning that column simply loops to find the maximum value. Then, if a row swap is necessary, this process simply sends the indexes of the rows to swap to all the other processes; each process then swaps the individual values in its columns. To perform a row subtraction, the process owning the column with the current diagonal has to compute all the scalar multipliers and then send them to all other processes, where the multipliers are used to subtract the row. In a shared memory solution, this requires deep swaps instead of pointer reassignment, affecting performance. However, in a distributed memory solution, this strategy limits the communication necessary to improve performance.

Agglomeration

It is not practical to assign each row or column to an individual process, so instead a contiguous block of rows or columns is assigned to each process. This helps to further limit communication, since some row swaps will occur within a single process, and the row to eliminate can be sent to fewer processes. To simplify the partitioning in the distributed memory solution, the number of processes must evenly divide the rows in the matrix.

Mapping

For a row-wise scheme, the rows of U, L, and P are mapped to different processes, making sure that each process owns the same section of all three matrices. Similarly, for the column-wise partitioning, the columns are divided across the processes.

The mapping is less obvious in the shared memory solution because OpenMP was used. Parallelism is applied to sections of code rather than the entire program, and OpenMP handles the block assignment itself.

Conclusion

We predict that the row-wise strategy will be more efficient for OpenMP, while the column-wise will be better suited to MPI. However, both versions will be developed to verify this prediction.