Parallel Computing Project

Miles Corn, Shawn George, Michael Lenyszn, and Adelin Owona

April 2023

Introduction and Motivation

An introduction to solving linear systems and an overview of practical applications.

Linear systems of equations arise in many practical applications. Consider the diffusion equation on the spatial domain [0,1]:

$$u_t = \nu u_{xx} \tag{1}$$

where ν is the thermal diffusivity. Consider a time and space discretization $v_j^n = u(j\Delta x, n\Delta t)$, where $j \in \{0, 1, 2, ..., M\}$ and $n \in \mathbb{N}$. One may be tempted to use a simple forward Euler scheme:

$$v_j^{n+1} = v_j^n + \frac{\nu\Delta t\left(v_{j+1}^n - 2v_j^n + v_{j-1}^n\right)}{\Delta x^2} \tag{2}$$

where $\Delta x = \frac{1}{M}$. However, it can be shown that doing so puts a restraint on your time step[3]:

$$\frac{\nu\Delta t}{\Delta x^2} < \frac{1}{2} \tag{3}$$

or

$$\Delta t < \frac{\Delta x^2}{2\nu} \tag{4}$$

Violating the constraint leads to instability. One may note that, in general, implicit discretization schemes are unconditionally stable[3]. Thus, we may instead consider the backwards Euler scheme:

$$v_j^n = v_j^{n+1} + \frac{\nu\Delta t\left(v_{j+1}^{n+1} - 2v_j^{n+1} + v_{j-1}^{n+1}\right)}{\Delta x^2} \tag{5}$$

This scheme is unconditionally stable, but how does one find the next time step? Note that the j subscripts denote the index of the spatial discretization. Thus, we have a system of linear equations:

$$\begin{aligned}
v_1^n &= v_1^{n+1} + \frac{\nu\Delta t\left(v_2^{n+1} - 2v_1^{n+1} + v_0^{n+1}\right)}{\Delta x^2} \\
v_2^n &= v_2^{n+1} + \frac{\nu\Delta t\left(v_3^{n+1} - 2v_j^{2} + v_1^{n+1}\right)}{\Delta x^2} \\
v_3^n &= v_3^{n+1} + \frac{\nu\Delta t\left(v_4^{n+1} - 2v_3^{n+1} + v_2^{n+1}\right)}{\Delta x^2} \\
&. \\
&. \\
&. \\
v_{M-1}j^n &= v_{M-1}^{n+1} + \frac{\nu\Delta t\left(v_M^{n+1} - 2v_{M-1}^{n+1} + v_{M-2}^{n+1}\right)}{\Delta x^2}
\end{aligned} \tag{6}$$

Note that here we have M-1 equations, but M+1 unknowns. We draw two more equations from the boundary conditions. For the purpose of this application, suppose the boundaries are Dirichlet:

$$\begin{aligned} v_0^n &= \alpha(n\Delta t) \\ v_M^n &= \beta(n\Delta t) \end{aligned} \tag{7}$$

Thus, we have M+1 unknowns and M+1 equations. This can be expressed in matrix form, as seen in the Princeton article [2019]. Further applications in one differential equations can be found in diffusion-advection [3] and the Crank-Nicolson scheme [4].

Solving Linear Systems

We will now provide a short mathematical overview of solving linear systems, including stability and the existence of LU factorizations. Numerous methods exist for solving systems of linear equations, such as full and reduced QR, in which a matrix is decomposed into an orthonormal basis Q and an upper triangular R, full and reduced singular value decomposition, in which a matrix is decomposed into matrices of left and right singular vectors and a diagonal matrix of singular values, and Gaussian elimination [1].

Gaussian elimination, or more specifically the LU factorization, is of particular interest in this paper. An LU factorization of a matrix is a decomposition of a matrix into a lower triangular L an upper triangular U such that

$$LU = A \tag{8}$$

Note, however, that Gaussian elimination is not numerically stable [1]. We can solve this issue in practice with partial pivoting, where we swap rows so that our pivot is always the element in a column that has the largest magnitude. Thus, our factorization takes on the form

$$LU = PA \tag{9}$$

where P is a permutation matrix. In theory, this algorithm is also unstable, but is practice it is absolutely stable [1]. In the book by Trefethen and Bau [1997], the following example of a matrix that exhibits worst-case instability is given:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix} \tag{10}$$

After Gaussian elimination, the resultant matrix U is

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & 0 & 16 \end{bmatrix} \tag{11}$$

A doubling occurs in the final column in each step. The growth factor of an mxm matrix of this form is 2^{m-1}. A growth factor of 2^k corresponds to a loss of precision on the order of k bits. However, as noted by Trefethen and Bau [1997], these matrices never arise in practical problems.

Another factor to consider is the uniqueness of an LU factorization. Indeed, if a matrix has an LU factorization, it will have infinitely many such factorizations [1]. That said, if the matrix L is made to have only ones along its main diagonal, then the factorization will be unique[1][8].

The Algorithm in Serial

Before we get into the parallel part of this paper, we mus lay the serial foundations of this algorithm. Consider the mxm matrix A for which we want to find the previously described LU factorization. The following pseudocode from Trefethen and Bau [1997] describes the serial LU factorization with partial pivoting:

```
Step 0. A is an mxm matrix, L, P are the mxm identity matrix, U = A
for k = 1 to m-1 do
  select i≥k to maximize $|u_i k|$
  swap $U_{k,k:m}$ and $U_{i,k:m}$
  swap $L_{k,1:k-1}$ and $L_{i,1:k-1}$
  swap $P_{k,1:m}$ and $P_{i,1:m}$
  for j = k+1 to m do
    $L_{j,k} = U_{j,k}/U_{k,k}$
    $U_{j,k:m} = U_{j,k:m} - L_{j,k}U_{k,k:m}$
  end for
end for
```

Of course, certain optimizations can be made if one knows a bit more about the matrix. If one know the matrix is symmetric, a Cholesky factorization may be used for a speedup of a factor of 2. Additionally, if the matrix is sparse or banded, certain algorithms may be able to get even greater performance[5]. For the purpose of this project, we assume very little about the structure of the matrix. The only thing we guarantee is that the LU factorization exits. This we guarantee by the construction of our test cases: our test cases are strictly diagonally dominant, and thus according to Farid [2011], have an LU factorization.

Operation Count of the LU Factorization

In Trefethen and Bau [1997], the operation count is derived geometrically. During the derivation, it is stated that "the work is dominated by the vector addition in the inner loop" [1]. The number of elements being operated on is l = m-k+1, where m is the dimension of the matrix and k is the current column. Note that during the row operation, we multiply each element by a scalar, then add two vectors together. Thus, we expend 2l flops conducting the row operation: 2 for each entry. The work is repeated for rows k+1 through m. In Trefethen and Bau [1997], this is represented by each layer of the following solid:

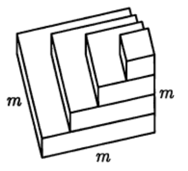

Figure 1: Source: Trefethen and Bau [1997]

As $m \to \infty$, the volume of this solid (the number of elements to operate on) approaches $\frac{m^3}{3}$. Since we have two operations per element, the operation cost of Gaussian elimination is $O\left(\frac{2m^3}{3}\right)$ [1][10].

LU in Parallel

The LU factorization is not, in its entirety, parallelizable. Each column must find its pivot and eliminate in sequence. However, the most expensive part of the factorization, the row addition, is parallelizable. Additionally, the row swaps are parallelizable. Thus, the new operation count can be derived in a similar manner. With the row operations and swaps now theoretically negligible, the serial operation cost is now the area of the right side of Figure 1. This is a right triangle with base and height m, so the area is $\frac{m^2}{2}$, and so the serial cost is $O\left(\frac{m^2}{2}\right)$ (note that we do not pick up the factor of two since that is eliminated by our parallelization).

Additionally, we can shave of another dimension of the pyramid by observing that while we must iterate across the columns in sequence, the row operations on each individual column can be done simultaneously, thus converting the time requirement of this loop to constant time and the time requirement of the algorithm as a whole to O(m). This is more or less the situation described by Santos and Muraleetharan[2000] and by Rana and Lin[2016] .

Implementation

For the purposes of this paper, we started from a serial LU factorization code, and then wrote CUDA kernels for the parallel functions. These functions are tailored to the specific row swaps, since they require different numbers of elements for maximum efficiency, and the row operation.

To ensure our matrices have an LU factorization, the program generates a strictly diagonally dominant matrix to factor. Then, we check if $PA = LU$ to verify that the factorization was successful.

Additionally, many practical applications may need to reuse a factorization more than once. For example, in the aforementioned numerical partial differential equation scheme, the matrix of coefficients need only be created and factored once. Thus, it need not be calculated at every time step if it can be stored. As such, we have also created an MPI I/O program to write the large matrices to storage in parallel so that they be used again.

Parallel I/O using MPI is performed as follows to start, all MPI ranks read in an equal portion of the data except for the last rank which reads in its portion of the data plus whatever portion of the data did not divide evenly among the ranks. MPI_read_at_all is used for reading in the data as it gives the file system a more holistic view of the data being read in allowing for optimizations that could not be made by just using MPI_read_at. Next all the data is given to MPI rank 0 using the MPI_Send and MPI_Recv functions. Once all the data is received by MPI rank 0 it is then reformatted into a 2-dimensional array and the LU factorization kernel is called. Once the LU factorization is completed, rank 0 reformats the data back into a 1-dimensional array so it can be evenly re-disputes amongst the other MPi ranks.Rank 0 re-disputes all the data back to the other MPI ranks using MPI_Send and MPI_Recv. Once all ranks have their portion of the data they perform an MPI_write_at_all call to write the data back out to a file. Once again MPI_write_at_all is used in place of MPI_write_at for the performance optimizations it allows for.

Parallel LU Experiments

For the MPI I/O reading experiments to get the starting times clock_now() was called on rank 0 right before the MPI_read_at_all call and to get the ending times clock_now() was called on rank 0 again right after the data was received by rank 0 and formatted into a 2-dimensional array. For the MPI I/O writing experiments to get the starting times clock_now() was called on rank 0 right before the calls to MPI_send and to get the ending times clock_now() was called on rank 0 again right after the call to MPI_write_at_all. To get the times for each of the experiments the starting time was subtracted from the ending time. The number of clock cycles resulting from that difference was then converted to milliseconds.

We ran a number of factorizations for matrix sizes of m = 50, 100, 200, 400, and 800 for both serial and parallel. Each factorization was timed using the supplied clock_now() function. We calculated the mean and standard deviation, then plotted the results. We anticipated that the parallel algorithm will perform better by almost a factor of m^2.

Additionally, we timed the MPI I/O operations.

Results

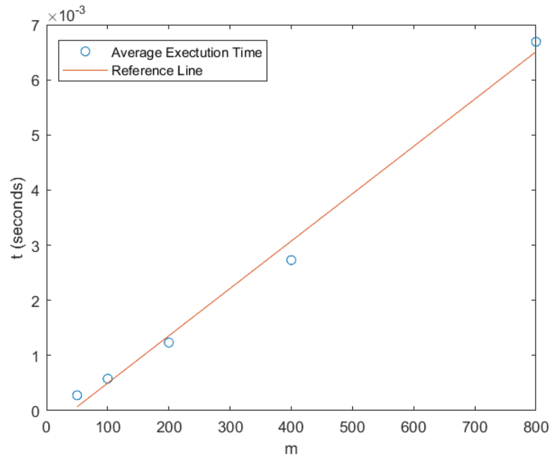

Figure 2: A plot showing the time requirements for the parallel LU factorization of matrices of dimension m. Observe the linear growth.

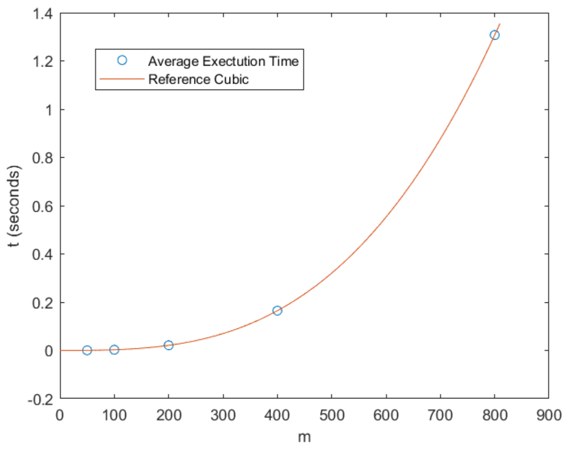

Figure 3: A plot showing the time requirements for the serial LU factorization of matrices of dimension

m. Observe the cubic growth.

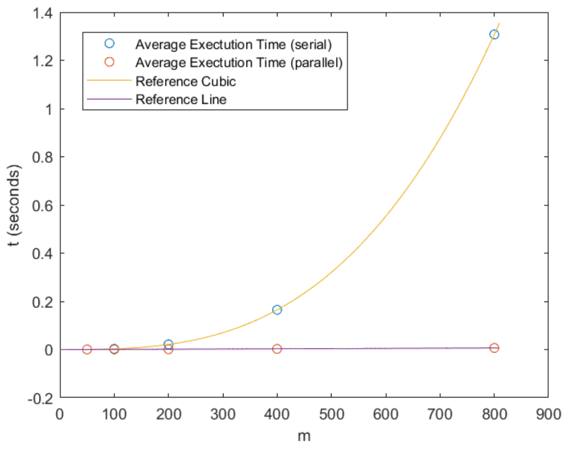

Figure 4: A plot showing the time requirements for the serial and parallel LU factorization of matrices of dimension m plotted together.

Dimension	Parallel (s)	Serial (s)
50	0.0003	0.0004
100	0.0006	0.0028
200	0.0012	0.0212
400	0.0027	0.1647
800	0.0067	1.3073

Figure 5: Computation times for serial and parallel algorithms to 4 decimal places.

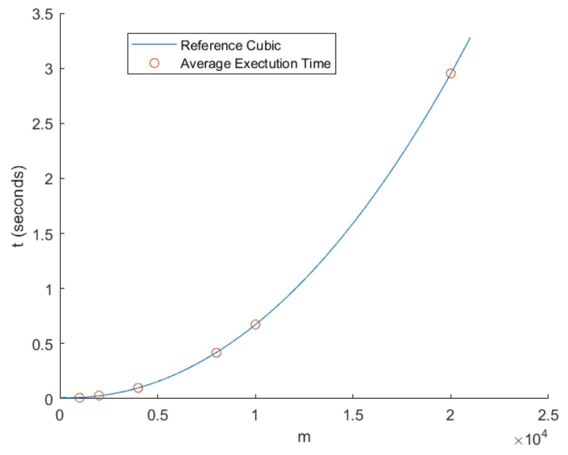

Figure 6: A plot showing the parallel algorithm after the GPU resources have been used up (i.e. the maximum threads and blocks are fully utilized). Note the return to cubic growth, as expected.

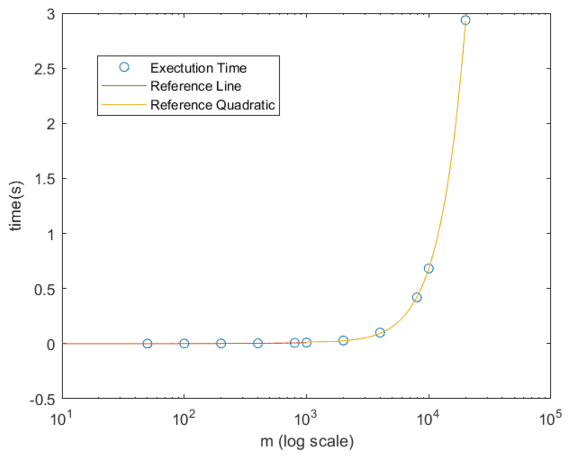

Figure 7: A plot showing the parallel algorithm for some very large matrix sizes running on 8 nodes with 4 GPUs each. Note it takes longer for the faster growth to resume, and the growth is still not as fast.

Parallel IO Experiments

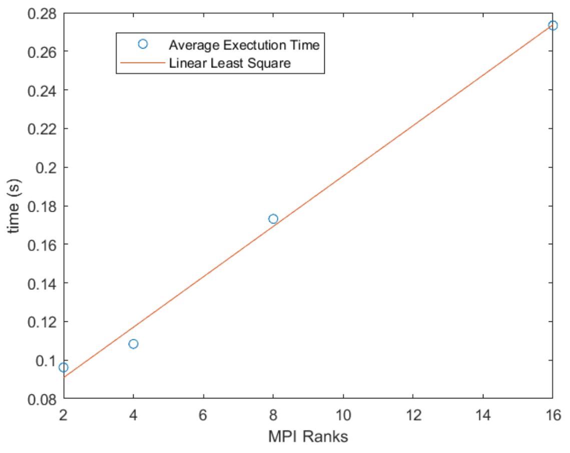

Figure 8: MPI write times with m = 50.

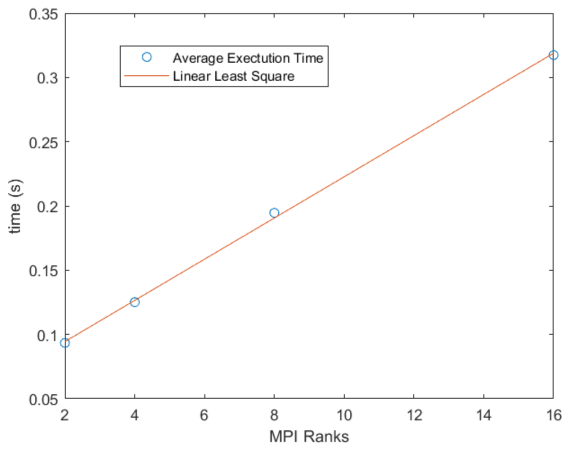

Figure 9: MPI write times with m = 100.

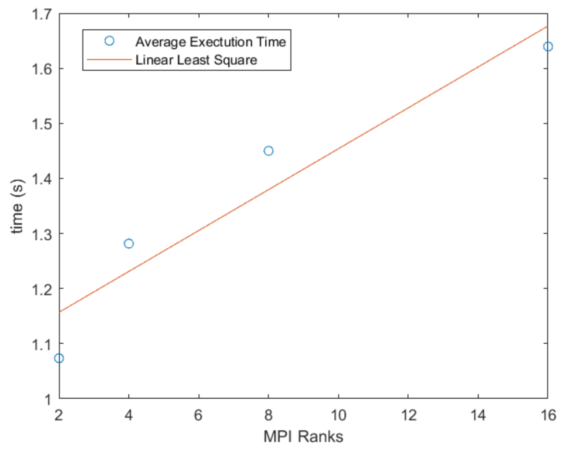

Figure 12: MPI write times with m = 800.

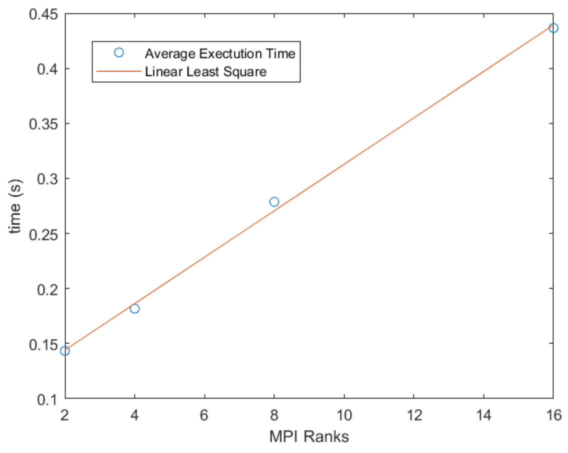

Figure 10: MPI write times with m = 200.

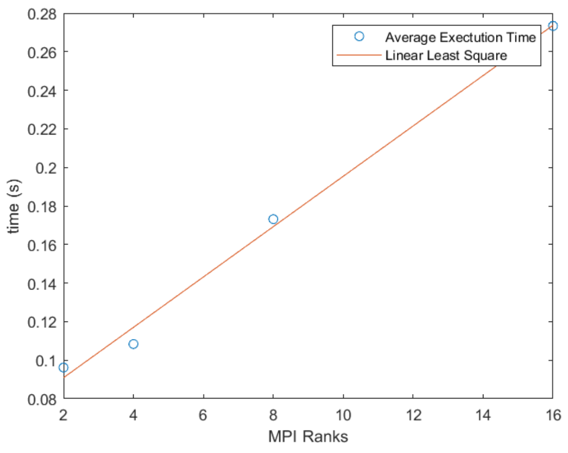

Figure 13: MPI read times with m = 50.

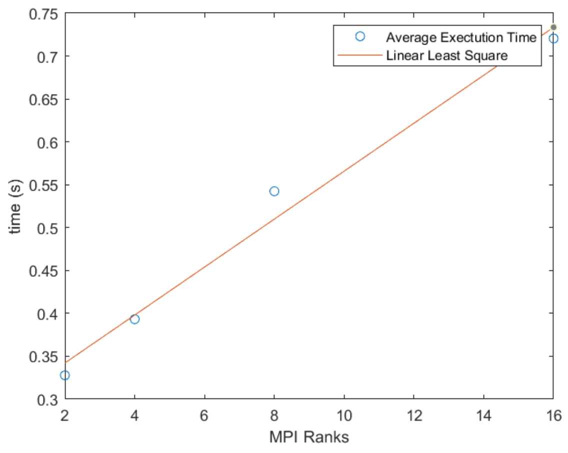

Figure 11: MPI write times with m = 400.

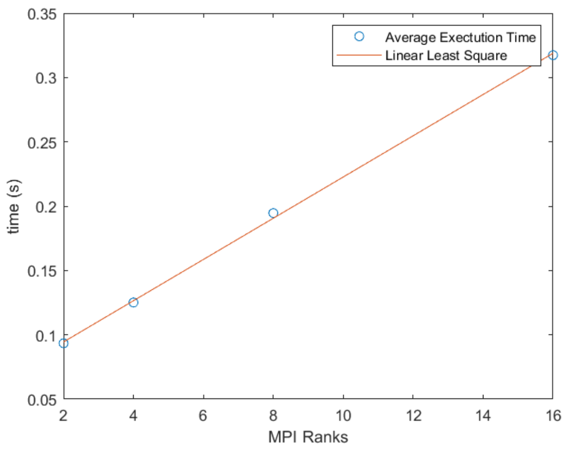

Figure 14: MPI read times with m = 100.

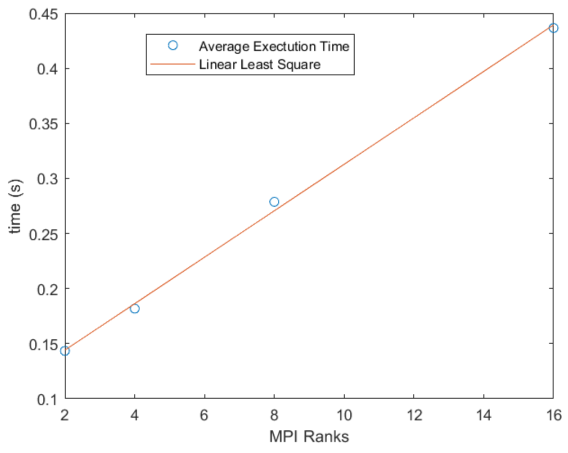

Figure 15: MPI read times with m = 200.

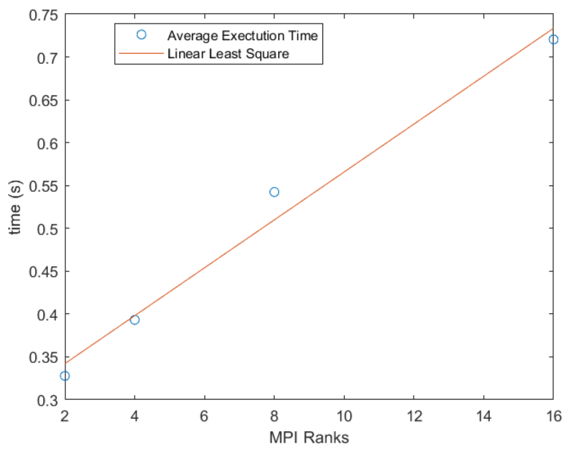

Figure 16: MPI read times with m = 400.

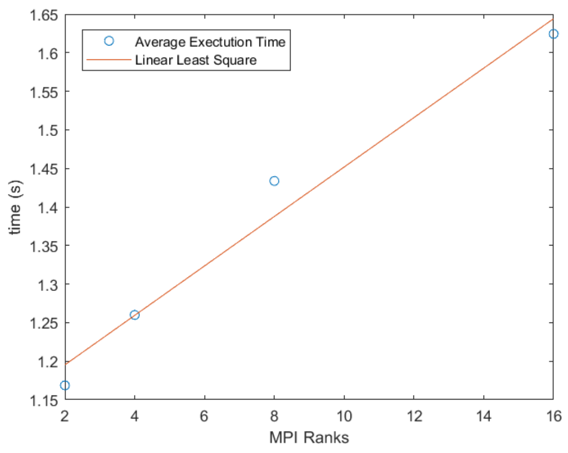

Figure 17: MPI read times with m = 800.

Analysis of Parallel LU

As we predicted analytically in the Implementation section, we have reduced the LU factorization to linear time. Indeed, for matrices of size 1000x1000, the serial algorithm completes in over 1.3 seconds, while our algorithm completes in less than one hundredth of a second. Consider the following table:

Dimension	Percent Speedup	Speedup Factor
50	25%	$\sim \frac{4}{3}$
100	78.57%	$\sim \frac{14}{3}$
200	94.34%	$\sim \frac{53}{3}$
400	98.36%	~ 61
800	99.49%	~ 195

It can be seen that as the matrix increases in size, we asymptotically approach 100% speedup. Note that this is relative to the serial program; the parallel program will of course grow in execution time as the matrices increase in size as well.

We can calculate the expected speedup at a given time. Consider Amdahl's law:

$$S(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{12}$$

where p is the proportion of execution time that the part of the program that benefits from improved resources originally occupied. Earlier, it was shown that the task requires O($\frac{2m^3}{3}$) operations, of which O(m) must be done in serial. Thus, the parallel portion is

$$p \sim \frac{\frac{2m^3}{3} - m}{\frac{2m^3}{3}} = 1 - \frac{3}{2m^2} \tag{13}$$

and the serial portion is

$$1 - p \sim \frac{3}{2m^2} \tag{14}$$

Note that p is always less than 1 and greater than 0 since m is a natural number greater than or equal to 2. Thus, we can write Amdahl's law

$$S(s) = \frac{1}{1-p+\frac{p}{s}} \sim \frac{1}{\frac{3}{2m^2} + \frac{1-\frac{3}{2m^2}}{s}} = \frac{2sm^2}{3s+2m^2-3} \tag{15}$$

Here, the $\sim$ symbol is doing some heavy lifting; all we really mean to say is that the theoretical speedup is quadratic in nature, which a plot of the data shows is true:

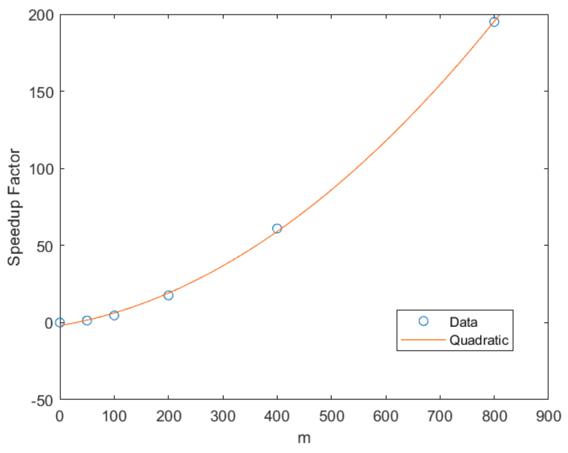

Figure 18: A figure showing quadratic growth in the speedup factor.

This is consistent with the theoretical results found by Yang et. al [2010].

Comparison to Other Papers

In Galoppo et. al [2005], another experiment in parallelizing LU factorizations was conducted. Their method included reduces the factorization to a series of rasterization problems on the GPU. Their method includes "new techniques for streaming index pairs, swapping rows and columns and parallelizing the computation to utilize multiple vertex and fragment processors."[7] While their approach does indeed achieve some speedup, the growth in time for each matrix size is greater than the linear growth in our algorithm, and their algorithm is always slower.

Analysis of Parallel IO

In general we see that as we increase the number of MPI ranks the slower our I/O times become. However, this is to be expected given the nature of the problem and our implementation. The main reason we see this decrease in performance is the additional overhead incurred by adding more MPI ranks. There are sections of the LU factorization algorithm that require serial operation to be performed across the entire matrix. This necessitates that one MPI rank obtains the entire matrix at some point.

In order for one MPI rank to obtain the entire matrix we must perform message passing operations to collect all the data at one rank. As shown in our performance results the overhead generated by these message passing operations outweighs the performance benefits of performing the reading in parallel. We see the same performance degradation when writing to files as well. This is because once all information is congregated in one rank the data then needs to be once again split up and sent to the other ranks to write out again.

Future Work

In the future, we think we might have been able to achieve a slightly faster performance by parallelizing the max function used in pivoting. Our implementation used a loop to find the element of maximum magnitude in each column. Loops execute in O(m) time, but through a pairwise reduction algorithm, we might have gotten this down to O($\log_2(m)$).

In the future, we would have liked to fully implement the backwards Euler scheme described in the motivation. That particular scheme can be shown to be second order accurate in space[3]. That is, the error of the scheme due to spatial discretization is $O(\Delta x^2)$. Thus, increasing the resolution of your grid by a factor of 2 decreases the size of your error due to spatial discretization by a factor of 4. Normally, the computational cost of doubling the resolution, and thus doubling the number of x_j, goes up by a factor of approximately 8 since the LU factorization scales cubicly. Our algorithm reduces this to linear. Contrast with the algorithm proposed by Liu and Cheung[1996], in which a block LU factorization is imposed on a hypercube system, and each block is assigned to a processor. This algorithm operates on $O((m/n)^2)$ elements, where m is the matrix dimension and n is the dimension of the blocks.

Summary

Linear systems of equations arise in many practical applications. Consider the diffusion equation on the spatial domain [0,1]. A forward Euler scheme imposes a time step restriction in order to maintain stability, so one may use an unconditionally stable backwards Euler scheme. This scheme produces a system of linear equations to solve, and the size of this system scales quadratically with the resolution of the spatial discretization.

Numerous methods exist for solving systems of linear equations, such as full and reduced QR, in which a matrix is decomposed into an orthonormal basis Q and an upper triangular R, full and reduced singular value decomposition, in which a matrix is decomposed into matrices of left and right singular vectors and a diagonal matrix of singular values, and Gaussian elimination. Gaussian elimination, or more specifically the LU factorization, is of particular interest in this paper. An LU factorization of a matrix is a decomposition of a matrix into a lower triangular L an upper triangular U such that LU = A. Since a naive factorization algorithm is unstable, we use partial pivoting (LU = PA). This is theoretically unstable, but absolutely stable in practice.

LU factorizations of an mxm matrix complete in $O(2m^3/3)$ time. The most expensive part of this algorithm is the row additions. These are just vector additions, and can be parallelized. Additionally, each operation on a column may be parallelized, though the columns must be traversed in serial. Additionally, finding the element of maximum magnitude restricts how much we can speed up the program. Our implementation removes the vector addition from the time calculation, as well as the iteration down the rows, so we predict $O(m)$ time for completion.

We ran a number of factorizations for matrix sizes of $m = 50$, 100, 200, 400, and 800 for both serial and parallel. Each factorization was timed using the supplied clock now() function. We calculated the mean and then plotted the results. Additionally, we wrote an MPI I/O program to write the factorizations to storage so that they may be used later. In the future, we would have liked to fully implement the backwards Euler scheme described in the motivation. That particular scheme can be shown to be second order accurate in space.

Overall we were able to observe a speedup from cubic to linear timing when moving from serial to parallel implementations. In addition, although MPI I/O is useful, it may not have been too useful when considering the structure of our project but did give us ways to take in matrices of varying sizes. In conclusion, the results show that when using it correctly, parallel computing can very much speed normally exponential and tedious processes.

Contributions

Miles Corn - Author of serial LU factorization, some kernels for the parallel function, and all plots. Also wrote Introduction and Motivation, Solving Linear Systems, LU in Parallel, Analysis of Parallel LU, Future Work, and Summary.

Shawn George - Author of LU factorization verification function, CUDA kernels for LU factorization, and ran timing tests for the CUDA LU factorizations.

Michael Lenyszn - Author of MPI I/O program, Parallel IO experiments, Analysis of Parallel IO, conducted IO tests, and combined the CUDA and MPI portions of this project.

Adelin Owona - Author of LU factorization verification function, CUDA kernels for LU factorization, and ran timing tests for the CUDA LU factorizations.

Bibliography

[1] Lloyd Trefethen, David Bau, III. 1997. Numerical Linear Algebra. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[2] Princeton. 2019. Basic Numerical Solution Methods for Differential Equations. Retrieved from https://scholar.princeton.edu/.

[3] D. Levy. Numerical Methods for Linear PDEs. Retrieved from http://www.math.umd.edu.

[4] J.W. Thomas. 1995. Numerical Partial Differential Equations: Finite Difference Methods. Springer-Verlag New York, Inc, New York.

[5] Florida State University. LU Factorization of Banded Matrices. Retrieved from https://www.math.fsu.edu/.

[6] F.O. Farid. 2011. Notes on matrices with diagonally dominant properties. Retrieved April 1, 2023 from https://www.sciencedirect.com/science/article/pii/S002437951100365X

[7] Nico Galoppo, Naga K. Govindaraju, Michael Henson, Dinesh Manocha. 2005. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. Retrieved April 2, 2023 from http://gamma.cs.unc.edu/LU-GPU/lugpu05.pdf

[8] Jincai Chang, Xiaoqiang Guo, Chunfeng Liu, Aimin Yang. 2010. Research on Parallel LU Decomposition Method and It's Application in Circle Transportation. Retrieved April 16, 2023 from https://www.researchgate.net/publication/220346657_Research_on_Parallel_LU_Decomposition_Method_and_It%27s_Application_in_Circle_Transportation

[9] E. E. Santos, M. Muraleetharan. 2000. Analysis and Implementation of Parallel LU-Decomposition with Different Data Layouts. Retrieved April 16, 2023 from https://math.ucr.edu/∼muralee/LU-Decomposition.pdf

[10] V.S. Rana, M.Lin. 2016. A Scalable Task Parallelism Approach for LU Decomposition With Multicore CPUs. Retrieved April 16, 2023 from https://bpb-us-e1.wpmucdn.com/you.stonybrook.edu/dist/6/1671/files/2017/11/rana2016scalable-v3cvc2.pdf

[11] Zhiyong Liu, D.W. Cheung. 1996. Efficient Parallel Algorithm for Dense Matrix LU Decomposition with Pivoting on Hypercubes. Retrieved April 16, 2023 from https://www.sciencedirect.com/science/article/pii/S0898122197000527