CS 420 Project Report: LU Decomposition

Alina Kononov, Zicheng Li, and Nora Quillman

1 Introduction and Background

The LU decomposition is a way of factoring a square matrix A into a lower triangular matrix L and an upper triangular matrix U. This factorization is typically the first step in a numerical method for solving a system of linear equations, a ubiquitous procedure in scientific computing. However, sequential algorithms for computing the LU decomposition of a dense $N \times N$ matrix generally take time $\mathcal{O}(N^3)$ [1]. For large matrices, sequential algorithms are prohibitively slow and a parallel implementation is essential.

1.1 Gaussian Elimination

A typical sequential algorithm for computing the LU decomposition of a matrix A resembles Gaussian elimination. First, U is initialized as a copy of A. Then we iterate over columns of U, eliminating elements in column k below the diagonal by subtracting a multiple of row k from each row $i > k$. Specifically, we calculate a multiplier $l_{ik} = u_{ik}/u_{kk}$ for each row $i > k$ and update the elements in row i using $u_{ij} = u_{ij} - l_{ik}u_{kj}$. The l_{ik} form the lower triangular matrix L, and the u_{ij} will form the upper triangular matrix U at the end of the elimination process [2].

Although this algorithm can be parallelized as-is, the elimination steps primarily consist of matrix-vector multiplications when matrix-matrix multiplications are better-suited for high performance computing [3]. Therefore, we instead use a block algorithm for finding the LU decomposition as described in Section 1.2.

1.2 Block LU Decomposition

To parallelize efficiently, we partition A, L, and U into blocks:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2n} \\ A_{31} & A_{32} & A_{33} & \cdots & A_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \cdots & A_{nn} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 & \cdots & 0 \\ L_{21} & L_{22} & 0 & \cdots & 0 \\ L_{31} & L_{32} & L_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & L_{n3} & \cdots & L_{nn} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} & \cdots & U_{1n} \\ 0 & U_{22} & U_{23} & \cdots & U_{2n} \\ 0 & 0 & U_{33} & \cdots & U_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U_{nn} \end{bmatrix}. \quad (1)$$

By inspection, we have $A_{i1} = L_{i1}U_{11}$ and $A_{1j} = L_{11}U_{1j}$. To find the blocks of L and U, we first find L_{11} and U_{11} by computing the LU decomposition of A_{11} using the procedure described in Section 1.1. Then, since $L_{i1} = A_{i1}U_{11}^{-1}$ and $U_{1j} = L_{11}^{-1}A_{1j}$ we can invert L_{11} and U_{11} and use them to solve for the remaining L_{i1} and U_{1j} blocks in parallel. From there, we can subtract the

contributions of L_{i1} and U_{1j} from the lower-right $(n-1)^2$ blocks of A to get

$$\begin{bmatrix} \bar{A}_{22} & \bar{A}_{23} & \cdots & \bar{A}_{2n} \\ \bar{A}_{32} & \bar{A}_{33} & \cdots & \bar{A}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{n2} & \bar{A}_{n3} & \cdots & \bar{A}_{nn} \end{bmatrix} = \begin{bmatrix} L_{22} & 0 & \cdots & 0 \\ L_{32} & L_{33} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n2} & L_{n3} & \cdots & L_{nn} \end{bmatrix} \begin{bmatrix} U_{22} & U_{23} & \cdots & U_{2n} \\ 0 & U_{33} & \cdots & U_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{nn} \end{bmatrix}, \tag{2}$$

where $\bar{A}_{ij} = A_{ij} - L_{i1}U_{1j}$. Now the same process can be repeated to factor $\bar{A}_{22}$ into L_{22} and U_{22}, solve for L_{i2} and U_{2j}, update the lower-right $(n-2)^2$ blocks with $\bar{\bar{A}}_{ij} = \bar{A}_{ij} - L_{i2}U_{2j}$, and so on until all blocks of L and U have been obtained [3].

2 Implementation Details

Our implementation of parallel LU decomposition using a block algorithm contains subroutines for each of the following basic single processor, single matrix block tasks:

- `LU.c`: LU decomposition using Gaussian elimination as described in Section 1.1. This routine is used on diagonal blocks of A to find diagonal blocks of L and U.
- `invert.c`: Inversion of lower triangular and upper triangular matrices using forward substitution and back substitution, respectively. These routines are used to invert diagonal blocks of L and U.
- `matrix_product.c`: Matrix multiplication optimized for a lower triangular matrix times a dense matrix, a dense matrix times an upper triangular matrix, and a dense matrix times another dense matrix. The first two cases are used to solve for off-diagonal blocks of L and U, while the third case is used to update the remaining blocks of A (the subtraction involved is also integrated into the routine).

Each of these tasks is parallelized using OpenMP. The tasks are combined into an implementation of the parallel block LU decomposition in `main.c`, which follows the procedure described in Section 1.2 to compute blocks of L and U in stages. Stage n of the algorithm consists of the following main steps:

1. Compute L_{nn} and U_{nn} using the LU decomposition routine on A_{nn}.
2. Invert L_{nn} and U_{nn} using the inversion routines.
3. Broadcast L_{nn}^{-1} and U_{nn}^{-1} to all processes.
4. Compute $L_{in} = A_{in}U_{nn}^{-1}$ and $U_{nj} = L_{nn}^{-1}A_{nj}$ for all $i, j > n$ in parallel using the first two matrix multiplication routines.
5. Communicate L_{in} and U_{nj} between processes.
6. Update A_{ij} for all $i, j > n$ in parallel by subtracting $L_{in}U_{nj}$ using the third matrix multiplication routine.

We use MPI for communications between processess and we use a master-slave scheme for load balancing within each stage. The master process performs Steps 1, 2, and 3, and then dynamically distributes Step 4 across the other processes. In Step 5, all processes communicate over a 2D torus network topology with L_{in} gathered across rows and U_{nj} gathered across columns. This communication pattern ensures that each pair of L_{in} and U_{nj} meets at a unique processor, which updates the corresponding A_{ij} in Step 6.

3 Code Optimizations

3.1 OpenMP Parallel Loop Schedule

The basic routines listed above frequently use OpenMP to compute the iterations of various loops in parallel. Preliminary tests determined that the static and guided schedules result in comparable performance, but the guided schedule produces slightly faster execution times for most routines. Thus, we use the guided schedule throughout the experiments discussed in the following sections.

3.2 Matrix-Matrix Multiplications

The bulk of the computation in a block algorithm for LU decomposition consists of matrix-matrix multiplications. When decomposing an $N \times N$ matrix using $B \times B$ blocks, stage n of the algorithm involves $N/B-n$ multiplications of the form $L_{nn}^{-1}A_{nj}$, $N/B-n$ multiplications of the form $A_{in}U_{nn}^{-1}$, and $(N/B-n)^2$ multiplications of the form $L_{in}U_{nj}$. Each of these multiplications takes time $\mathcal{O}(B^3)$.

In contrast, each stage of the algorithm only involves a single LU decomposition and two inversions of triangular matrices, which each take time $\mathcal{O}(B^3)$. Thus, when the original matrices are partitioned into many blocks (or when N/B is large), optimizing the matrix multiplication routines is imperative for good performance.

Algorithm 1 Basic algorithm for matrix-matrix multiplication: compute $C = AB$ where C is $M \times N$, A is $M \times K$, and B is $K \times N$.

```
for (i=0; i<M; i++) do
    for (j=0; j<N; j++) do
        C[i*N+j]=0;
        for (k=0; k<K; k++) do
            C[i*N+j]+=A[i*K+k]*B[k*N+j];
        end for
    end for
end for
```

A basic matrix multiplication algorithm involves three loops, as shown in Algorithm 1. Two optimizations are obvious: first, all arrays should be stored contiguously in order to enable prefetching. Secondly, when one of the matrices is triangular, we can avoid looping over the entries above or below the diagonal by adjusting the limits of the `k` loop. For an $L_{nn}^{-1}A_{nj}$ multiplication, only `k` iterates from 0 to `i` need to be included, while for an $A_{in}U_{nn}^{-1}$ multiplication, only `k` iterates from 0 to `j` need to be included[1].

[1]Note that L_{nn}^{-1} is also lower triangular and U_{nn}^{-1} is also upper triangular.

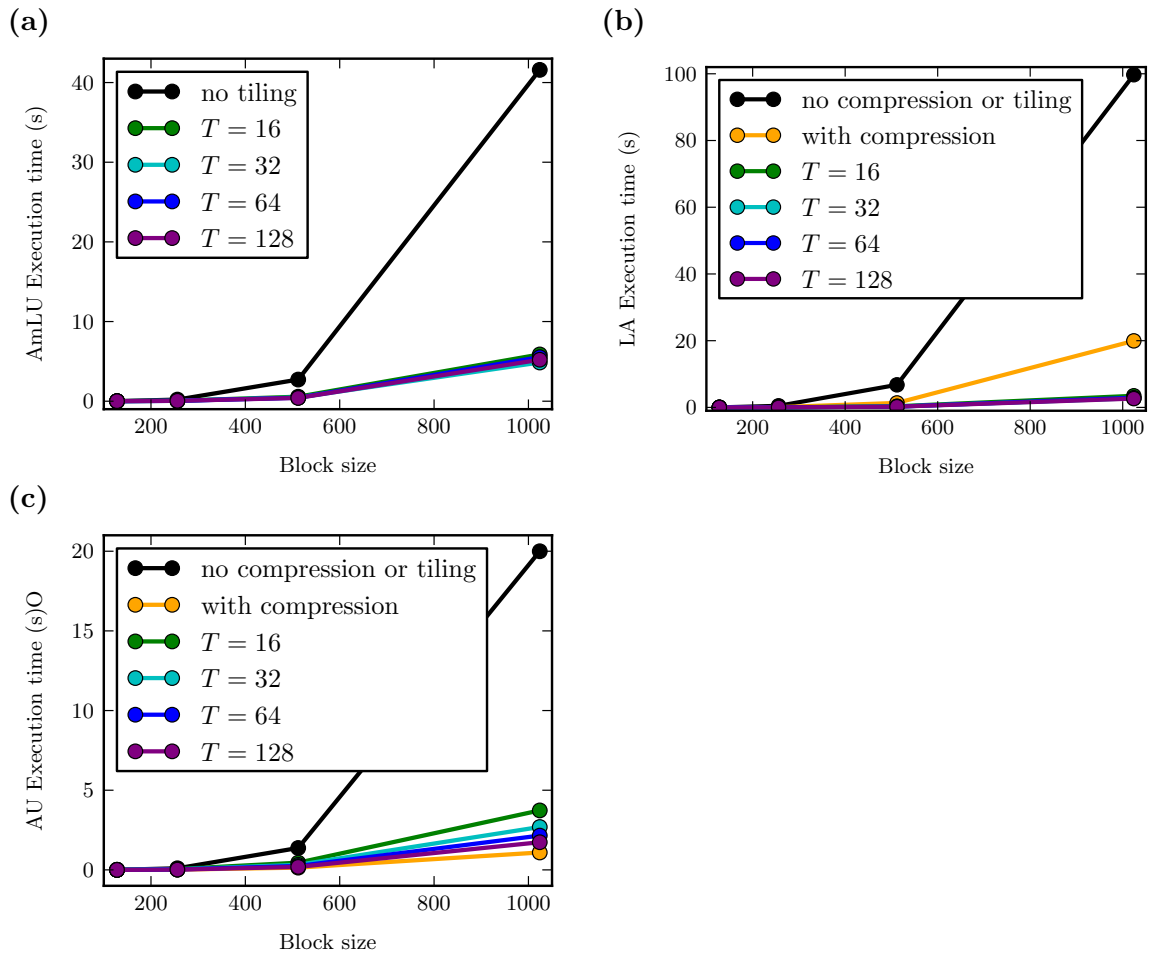

Figure 1: Performance of different implementations of matrix-matrix multiplication where **(a)** both matrices are dense **(b)** the first matrix is lower triangular and **(c)** the second matrix is upper triangular. All experiments shown were conducted on Blue Waters using 1 node and 32 OpenMP threads. Each data point was obtained by averaging the results of five tests, clearing the cache between tests. Error bars corresponding to the standard deviation are too small to see.

We tested several more subtle strategies for optimizing the matrix multiplications. The most impactful of these was tiling over all three loops, which sped up the execution of the more numerous $L_{in}U_{nj}$ multiplications by a factor of 4.4, 6.6, and 8.6 for block sizes of 256, 512, and 1024, respectively. When the blocks are larger than the cache, tiling improves temporal locality of memory accesses and therefore increases the cache hit ratio. We tested tile sizes of 16, 32, 64, and 128 and found that all tile sizes achieve a similar level of performance, with the optimal tile size depending on the block size. The execution time for a single $L_{in}U_{nj}$ multiplication for different block sizes and tile sizes is plotted in Figure 1 (a).

In the $L_{nn}^{-1}A_{nj}$ and $A_{in}U_{nn}^{-1}$ multiplications, L_{nn}^{-1} and U_{nn}^{-1} are potentially reused many times, so optimizing the storage format of these inverted triangular matrix blocks is worthwhile. Since the zeros above or below the diagonal do not contribute to the matrix product and only waste memory, we do not store them. In L_{nn}^{-1}, we also do not store the diagonal elements, which are always 1. Furthermore, we store U_{nn}^{-1} column-wise in order to improve spatial locality during the $A_{in}U_{nn}^{-1}$ multiplications. Thus, when using $B \times B$ blocks, the compressed format of L_{nn}^{-1} contains

$B(B-1)/2$ numbers with the ith row (for $i > 0$) starting at index $i(i-1)/2$, while the compressed format of U_{nn}^{-1} contains $B(B+1)/2$ numbers with the jth column (for $j \geq 0$) starting at index $j(j+1)/2$. The routines used to compress L_{nn}^{-1} and U_{nn}^{-1} are included in `compress.c`.

As shown in Figure 1 (b) and (c), we found that the compressed format improved performance for the $L_{nn}^{-1}A_{nj}$ multiplications by an approximately constant factor of about 5 and for the $A_{in}U_{nn}^{-1}$ multiplications by a factor ranging from only 1.1 for a block size of 128 to 18.3 for a block size of 1024. Furthermore, the execution times with and without compression are both about one order of magnitude larger for the $L_{nn}^{-1}A_{nj}$ multiplication than for the $A_{in}U_{nn}^{-1}$ multiplication. Without compression, the execution time for the $L_{nn}^{-1}A_{nj}$ multiplication even exceeds the execution time of the regular $L_{in}U_{nj}$ multiplication (without tiling)! This result shows that without compression, reducing the number of iterations in the `k` loop generates more overhead from branching than it saves on useless computations.

Meanwhile, the difference in behavior between the $L_{nn}^{-1}A_{nj}$ and $A_{in}U_{nn}^{-1}$ multiplications is explained by the different structures inherent to the respective computations. In the $A_{in}U_{nn}^{-1}$ multiplication with compression, both A_{in} and U_{nn}^{-1} are traversed the same way they are stored: A_{in} row-wise and U_{nn}^{-1} column-wise. This feature of our compressed storage format for U_{nn}^{-1} enables efficient memory access patterns, producing dramatic speed-ups that increase with block size. However, in the $L_{nn}^{-1}A_{nj}$ multiplication, A_{nj} is stored row-wise but traversed column-wise, so compressing L_{nn}^{-1} only saves on inefficiencies associated with cluttering the array with the useless 1s and 0s on and above the diagonal.

Fortunately, tiling the $L_{nn}^{-1}A_{nj}$ multiplication brings its performance to the same level as the $A_{in}U_{nn}^{-1}$ multiplication. Again, we tiled all three loops and tested tile sizes of 16, 32, 64, and 128. For the $L_{nn}^{-1}A_{nj}$ multiplication, we found that using both compression and tiling speeds up computation by a factor ranging from 4.0 to 7.6 (depending on block size) compared to using compression only, and tile sizes of 32, 64, and 128 all achieve comparable levels of performance.

However, no tile size significantly decreases the execution time of the $A_{in}U_{nn}^{-1}$ multiplication below the execution time when using compression only. We postulate that tiling this type of multiplication only disrupts the already optimal memory access patterns.

In light of the results presented in this section, the tests reported in the following sections use the compressed storage format for L_{nn}^{-1} and U_{nn}^{-1}. $A_{in}U_{nn}^{-1}$ multiplications are performed without tiling, while $L_{in}U_{nj}$ and $L_{nn}^{-1}A_{nj}$ multiplications use a tile size of 128.

3.3 In-Place LU Factorization

Although the matrix-matrix multiplications in our algorithm vastly outnumber the LU factorizations, each stage of the algorithm starts with an LU factorization on the master MPI process that must complete before any of the matrix multiplications can begin. Thus, the LU factorization poses a potential bottleneck.

To improve the LU factorization routine, we tested an in-place algorithm where L_{nn} and U_{nn} overwrite A_{nn} rather than being stored in a separate array. This optimization halves the number of cache misses and thus approximately halves the execution time of the routine for block sizes of 2048 or larger. Even larger speed-ups are obtained for smaller block sizes.

3.4 Overlapping Communication and Computation

We take advantage of several opportunities to overlap communication with computation. During the dynamic load balancing in step 4 of the algorithm, each slave process uses a non-blocking send and receive to initiate a new request for work as soon as it receives a task from the master process. Thus, the slave processes simultaneously compute tasks and wait for their next tasks, effectively eliminating scheduling overhead.

We also partially overlapped steps 5 and 6 of the algorithm, where each process gathers L_{in} and U_{nj} blocks from its row and column in the 2D torus and uses each pair to update A_{ij}. Each process first initiates the gathers using `MPI_Iallgatherv` (after having gathered the appropriate block counts), and then performs the A_{ij} updates using its own L_{in} and U_{nj} blocks while the gathers are occurring. Once the gathers have completed, each process can proceed with the A_{ij} updates that involve the blocks it received. Overlapping in this way minimizes any idle time before the communications complete.

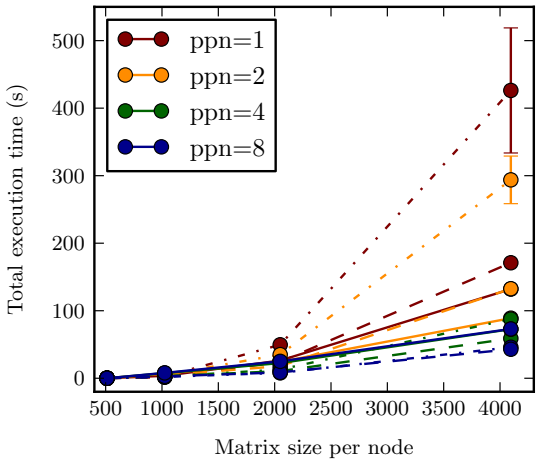

Figure 2: Performance of the LU decomposition for different numbers of ppn, block sizes, and problem sizes. Solid lines, dashed lines, and alternately dashed and dotted lines represent block sizes of 1024, 512, and 256, respectively. All experiments shown were conducted on Blue Waters using 2 nodes and 32 threads per node. Each data point and error bar correspond to the average and standard deviation of the results of five tests, clearing the cache between tests.

4 Run-time Parameters

Once the actual code has been finalized, several parameters remain to be set at run-time. These include the number of MPI processes per node, the number of OpenMP threads per process, the number of nodes, and the block size used to partition the matrix[2].

4.1 MPI Processes per Node and OpenMP Threads per MPI Process

Each Blue Waters XE node contains 32 cores [4], so in order to take advantage of all available parallelism, we always use 32 threads per node. We assume that for large problems, our code will already make maximal use of a single core's cache and computation capacity, especially since each pair of cores shares a single FPU [4]. With this assumption, using multiple threads per core would not improve performance.

Thus, we used p processes per node (ppn) and $32/p$ threads per process and investigated how performance depends on p. Figure 2 plots the results for different block sizes and problem sizes. We found that 4-8 ppn produces the shortest execution times across different block sizes and problem sizes. Although we did not test an even larger number of ppn, the difference in performance between 4 and 8 ppn is not that large, especially when optimizing the block size. We expect that if moving from 8 to 16 ppn improves performance, the impact would be even smaller.

[2]The user also sets the tile size for parallelization within an MPI process, but we discuss tiling in Section 3.2.

With a small number of nodes, having multiple processes per node is especially important for adequate loading of the node containing the master process. For example, with only 2 nodes and 1 process per node, the master process has 32 cores at its disposal, most of which remain idle while the master allocates work to the other process. With more nodes, increasing the amount of computation done on the master node would become less effective, especially with the increased communication cost of a higher number of ppn.

4.2 Nodes and Block Size

For good load balancing, the number of blocks handled in each stage of the algorithm should be much greater than the number of processes. Thus, for a given problem size N and block size B, the number of processes p should be smaller than N/B, and with 8 processes per node, the number of nodes should be smaller than $N/(8B)$.

Meanwhile, B should be an integer multiple of the vector hardware capacity; each 256-bit SSE vector FPU on the Blue Waters XE nodes [4] can compute on 8 single precision floats at a time. B should also be an integer multiple of the cache line size, which is 16 singles. While larger blocks allow better memory access patterns, $2B^2$ (the combined size of corresponding A and L/U blocks) should not exceed the total cache size per process. Given that each of the 32 cores on an XE node has a 1MB L2 Cache [5], an upper bound on the optimal block size is given by

$$2B^2 < \frac{32 \cdot 1024^2\text{B}}{4\text{B/single} \cdot \text{ppn}} = \frac{2^{23}}{\text{ppn}} \text{ singles.}$$

For 8 ppn, this gives $B < 724$, so the optimal block sizes should be 512.

To verify our hypothesis for the optimal block size, we timed our implementation using block sizes of 256, 512, and 1024 for different problem sizes. We ran these tests using 2 nodes (16 processes) and 4 nodes (32 processes) using $N/p = 128$, 256, 512, and 1024. Figure 3 plots the results of these tests.

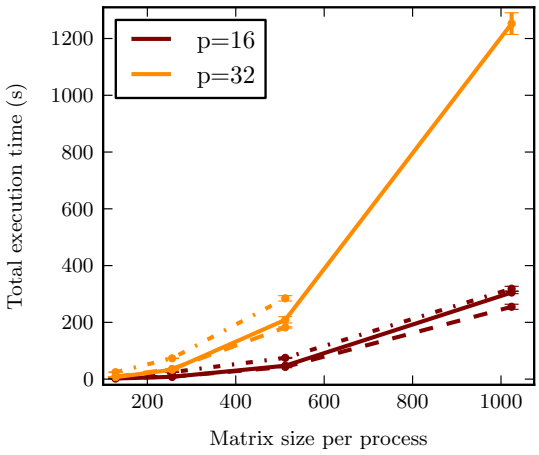

Figure 3: Performance of the LU decomposition for different block sizes, numbers of nodes, and problem sizes per processes. Solid lines, dashed lines, and alternately dashed and dotted lines represent block sizes of 256, 512, and 1024, respectively. All experiments were conducted on Blue Waters using 8 processes per node and 4 threads per process. Each data point and error bar correspond to the average and standard deviation of the results of four tests, clearing the cache between tests.

We found that when $N/p < 512$, $B = 256$ gives best performance because a larger block size would prevent proper load balancing. However, for $N/p \geq 512$, $B = 512$ indeed yields best performance because this is the largest block size for which the data associated with one block still fits in the $L2$ caches within a process.

5 Weak Scaling

To evaluate the efficiency of our parallel implementation of LU decomposition, we measured how the execution time varies with the number of nodes for $N/p = 128$, $B = 256$, ppn $= 8$, and $T = 128$. The results of these tests are plotted in Figure 4.

Although the upward trend in Figure 4 looks alarming, it should be noted that the LU decomposition actually has complexity $\mathcal{O}(N^3)$, so keeping N^3/p constant instead of N/p would provide a more accurate evaluation. Unfortunately, we exhausted the number of node hours allocated per group after trying to measure the execution time with 256 nodes (2048 processes) and thus could not perform this second type of experiment.

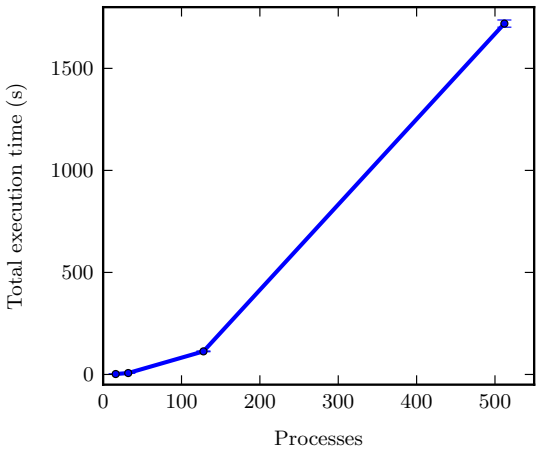

Figure 4: Performance of the LU decomposition for different numbers of processes, keeping $N/p = 128$ constant. All experiments were conducted on Blue Waters using 8 processes per node and 4 threads per process. Each data point and error bar correspond to the average and standard deviation of the results of four tests, clearing the cache between tests.

6 Further Possible Improvements

6.1 I/O

Our code completely bypasses the extensive I/O that would be required by a real application in order to read in A and write back L and U. Instead, we generate a random block of A at a given MPI process whenever necessary. While this scheme is suitable for testing some implementation options that affect computation and communication cost, I/O should also be included before making a meaningful comparison to a production-scale code like ScaLAPACK. A good option to manage the I/O would be HDF5, which is particularly efficient at accessing subarrays such as the matrix blocks involved in our algorithm for LU decomposition.

6.2 Dynamic Allocation

Our scheme for dynamically allocating $L_{nn}^{-1}A_{nj}$ and $A_{in}U_{nn}^{-1}$ multiplications to MPI processes could be improved in a couple of ways. First, our code allocates the multiplications one at a time, but it may be preferable to allocate the multiplications in contiguous chunks. This would reduce the scheduling overhead, and more importantly, when an I/O mechanism is included, we expect that it would improve memory performance. The optimal chunk size and its dependence on the number of processes would need to be determined through testing.

Secondly, our code continues allocating these tasks until the very last stage of the algorithm, when only one block of A remains. The master process then factors this last block. Stopping the dynamic allocation earlier, at the point when the communication cost exceeds the cost of completing the remaining computation on a single process, would improve performance. Again, this threshold size for the remaining matrix would need to be determined through testing.

6.3 Eliminating Bottlenecks

As described in Section 3.3, the LU factorization at the beginning of each stage as well as the subsequent inversion of L_{nn} and U_{nn} could pose bottlenecks because these steps are performed on the master process while the slaves wait to receive the results. This could be mitigated by treating the LU factorization as a task for the slaves and computing it within the previous stage of the algorithm. The code would need to explicitly enforce the dependency between the stages: L_{nn} and U_{nn} cannot be computed until until $L_{n,n-1}$ and $U_{n-1,n}$ have been calculated and used to update A_{nn}.

6.4 Pivoting

When a pivot (an element on the diagonal of U) is zero or very small compared to the elements it eliminates, a regular LU decomposition would incur a division by zero or loss of precision [1]. This problem is solved using pivoting, which permutes rows and/or columns so that the pivots are always nonzero and as large as possible. In order to make our implementation more numerically stable, pivoting should be included at the start of each stage.

7 References

[1] M. T. Heath. *Scientific Computing: An Introductory Survey*. McGraw-Hill, 2002.

[2] D. C. Lay. *Linear Algebra and its Applications*. Pearson/Addison-Wesley, 2006.

[3] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[4] NCSA. *Node and Core Comparison*. https://bluewaters.ncsa.illinois.edu/node_core_comparison.

[5] AMD. *AMD Opteron 6200 Series Processor Quick Reference Guide*. https://www.amd.com/Documents/Opteron_6000_QRG.pdf.