Communication Avoiding LU Factorisation

[redacted]

github.com/[redacted]

July 2023

Abstract

LU factorisation is an algorithm of fundamental importance in the field of numerical analysis, with applications including solving systems of linear equations and computing matrix inverses. For large problems, a stable, parallel algorithm that minimises inter-process communication is required. The cost of inter-process communication typically forms a bottleneck that can be reduced by incurring extra computational work. This is achieved by a Communication Avoiding LU algorithm, which implements a tournament pivoting strategy. The aim of this is to maintain stability while greatly reducing the communication required by other pivoting strategies such as GEPP (Gaussian Elimination with Partial Pivoting). This report strives to implement a 1D version of the algorithm outlined in "CALU: A Communication Optimal LU Factorisation Algorithm" from L.Grigori et. al, factorising an arbitrary $m \times n$ matrix A distributed among p processes using MPI.

Contents

Chapter 1

Introduction

This report outlines the implementation of the Communication Avoiding LU factorisation (CALU) algorithm in C, with MPI used for inter-process communication. Although the algebra of the algorithm is outlined in Grigori and Demmel's paper[1], no implementation is provided. The implementation described in this report factorises an arbitrary $m \times n$ matrix using a 1D grid of p processes.

The primary focus of this report is to present the details of the implementation's parallelism. The scaling of the algorithm, as well as stability and error will be investigated. Due to time constraints this algorithm works on a 1 dimensional grid of processes, while the original paper discusses a 2D cyclic grid of processes. How this implementation would be adapted for a 2D grid of processes is discussed in chapter 6.

As with many other block LU factorisation algorithms, the core idea of CALU is to compute the LU factorisation of the first panel of b columns, use this to update the first b rows, then update the Schur's compliment. The algorithm then proceeds on the trailing matrix. This algorithm works recursively until the entire matrix has been processed. The difference between CALU and other algorithms lies in the computation of the LU factorisation of the panel.

Using GEPP (Gaussian Elimination with Partial Pivoting), while slow, ensures a level of stability not offered by algorithms such as block pairwise pivoting [2]. TSLU (and CALU) implements a tournament pivoting scheme which aims to reduce communication while preserving the stability of GEPP. Each process performs LU factorisation on its local rows of the panel $PA_{loc} = LU$ with the sole intention of obtaining the pivot matrix of this factorisation P.

In a binomial tree type reduction, two processes' best b rows ($2 * b$ rows in total) are then similarly compared to find the b best rows for stability among them. This method continues until the binomial tree gives the b best overall rows that must be

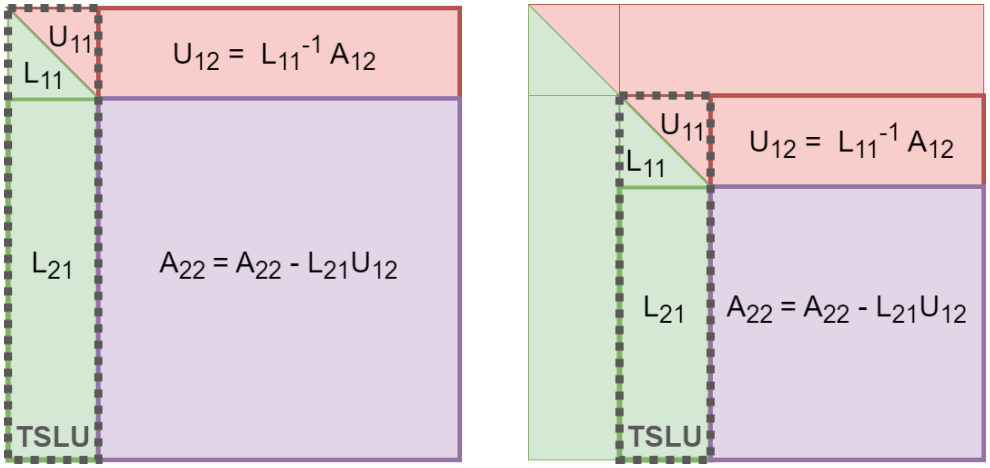


Figure 1.1: Overview of how CALU factorises a matrix

pivoted to the top of the original A matrix. These rows of A are pivoted to the top, allowing for a stable unpivoted LU factorisation to be performed on the panel.

This method only requires b full row pivots per panel. Row pivoting requires communicating entire rows of width m between processes, while tournament pivoting only requires communicating panel rows of width b. Thus, the use of this tournament style pivoting instead of GEPP results in a reduction of the algorithm's overall inter-process communication.

Tournament pivoting combined with the parallel unpivoted LU solver composes the Tall Skinny LU solver (TSLU). TSLU is used to compute the LU factorisation of each panel in the matrix. A detailed outline of how this was implemented, as well as a breakdown of challenges posed by memory allocation, library functions (e.g. `dgetrf`) and the binomial tree reduction of processes is included in chapter 3.

Updates must then be made to U_{12} and A_{22} as shown in Figure 1.1. Recursion of the algorithm must be implemented, both when the next block of rows lies on the same process as well as when it lies on another. The implementation of this recursion was the most technically challenging facet of the project and is outlined in detail in chapter 4.

Additional aspects of the project, such as MPI IO, matrix generation and growth factor/error calculation are discussed throughout the implementation section.

In chapter 5, it is shown that this project's implementation of CALU results in significant $\approx 3\times$ speedup for randomly generated matrices when tested on chuck. On larger matrices tested on Lonsdale, significant speedup is also observed. It is also shown that the algorithm is stable, with low growth factor and low error ($\mathcal{O}(10^{-10})$).

Chapter 2

Background

2.1 Block LU Factorisation

Block LU factorisation forms the backbone of most efficient LU solvers, including LAPACK's `dgetrf` which uses block Gaussian elimination with partial pivoting (GEPP)[3]. Through operating iteratively on submatrices (blocks) of various size, block algorithms can take advantage of cache structure in the architecture and choose their block size to provide optimal performance on that architecture. In the case of `dgetrf`, this is done by the function `ilaenv`. No method is employed for determining the block size for CALU in this implementation and none was mentioned in "CALU: A communication optimal parallel LU factorisation". Implementing this in the future is discussed in chapter 6.

This approach can also be used for parallel algorithms, where the blocks can be distributed among processes and their size determined by the local architecture of the core. Since block size must remain the same throughout the algorithm, it is easier to find the correct block size on homogeneous architectures. Performance for various block sizes are discussed in chapter 5.

As shown in Figure 1.1, block LU factorisation works by first factorising the first b columns (b representing the block size) and updating the first b rows. After performing an update on the final $(n-b) \times (m-b)$ block, the algorithm proceeds recursively on this Schur's complement (trailing matrix).

2.2 CALU Matrix Algebra

Note: The following section is based on the matrix algebra discussed in "CALU: A Communication Optimal LU Factorisation Algorithm" by L. Grigori et.al.[1]

Once a block size b has been chosen, the $n \times m$ matrix A is partitioned as follows.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

where A_{11} is a square submatrix of size $b \times b$, A_{12} is of size $b \times (m-b)$, A_{21} is of size $(n-b) \times b$ and A_{22} of size $(n-b) \times (m-b)$. The general outline of the algorithm is to perform TSLU on

$$A(:, 1:b) = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

and perform the necessary updates on A_{12} and A_{22}. The algorithm then proceeds recursively on A_{22}.

2.2.1 Tournament Pivoting

Tournament pivoting is the critical step that differentiates CALU from other block LU factorisation algorithms. The primary benefit of tournament pivoting is to make communication for the panel factorisation independent of the number of columns in the overall matrix. The parallel case of tournament pivoting will be outlined below, where each of the processes performs a reduction of its local block of $A(:, 1:b)$ as the first step of the algorithm. The aim is to obtain obtain the b best pivot rows that can be pivoted to the top of the panel to ensure stability while performing unpivoted LU factorisation. In the case of 4 processes, a local LU factorisation is performed on each of the blocks of $A(:, 1:b)$. Note that for the sake of the clarity, the array indices will be the indices in the global matrix as opposed to local indices.

$$\begin{aligned} A(:, 1:b) &= \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} \Pi_{00} L_{00} U_{00} \\ \Pi_{10} L_{10} U_{10} \\ \Pi_{20} L_{20} U_{20} \\ \Pi_{30} L_{30} U_{30} \end{bmatrix} \\ &= \begin{bmatrix} \Pi_{00} & & & \\ & \Pi_{10} & & \\ & & \Pi_{20} & \\ & & & \Pi_{30} \end{bmatrix} \cdot \begin{bmatrix} L_{00} & & & \\ & L_{10} & & \\ & & L_{20} & \\ & & & L_{30} \end{bmatrix} \begin{bmatrix} U_{00} \\ U_{10} \\ U_{20} \\ U_{30} \end{bmatrix} \\ &= \Pi_0 L_0 U_0 \end{aligned}$$

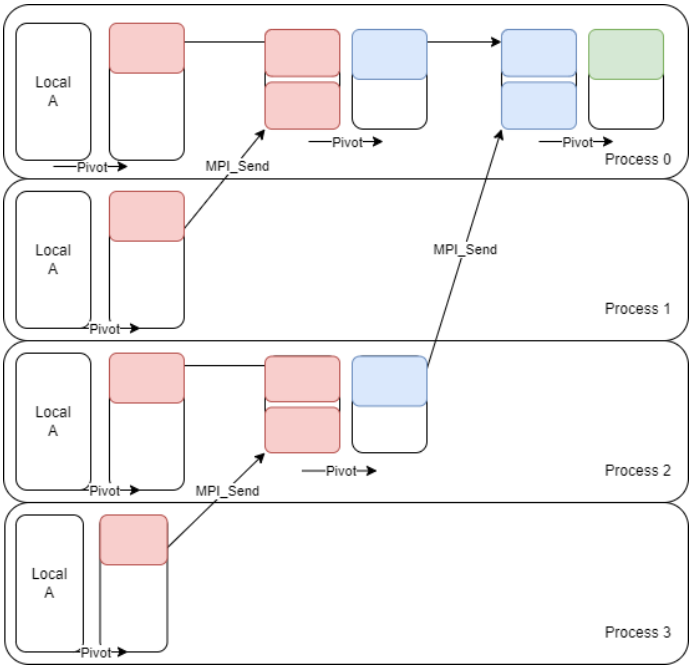


Figure 2.1: Tournament pivoting: Each process performs LU decomposition on the local A matrix to find the pivot matrix P. PA is then calculated and the top b rows of A progress to the next round

Here, Π_{0i} are pivot matrices. The pivot matrix Π_0, composed of these Π_{0i} along the diagonal, is a permutation matrix and thus has the property $\Pi_0^T = \Pi_0^{-1}$.

Next, this is applied to A to obtain $\Pi_0^T A$. The b top rows of $\Pi_0^T W$ on each of the 4 processes enter into a binomial tree reduction, whereby the top b rows of two processes are compared until only one remains. So, in the example above the second step would be

$$\begin{bmatrix} (\Pi_0^T A)(1:b, 1:b) \\ (\Pi_0^T A)(n/p+1:n/p+b, 1:b) \\ \hline (\Pi_0^T A)(2n/p+1:2n/p+b, 1:b) \\ (\Pi_0^T A)(3n/p+1:3n/p+b, 1:b) \end{bmatrix} = \begin{bmatrix} \Pi_{01} L_{01} U_{01} \\ \Pi_{11} L_{11} U_{11} \end{bmatrix}$$

$$= \begin{bmatrix} \Pi_{01} & \\ & \Pi_{11} \end{bmatrix} \begin{bmatrix} L_{01} & \\ & L_{11} \end{bmatrix} \begin{bmatrix} U_{01} \\ U_{11} \end{bmatrix} = \Pi_1 L_1 U_1$$

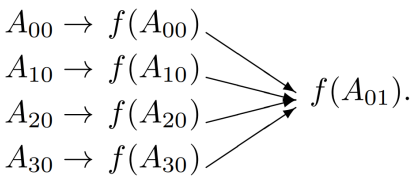
Proceeding to the final step of the binary tree, the top b rows of the two remaining

processes are compared and the best b rows of them is chosen.

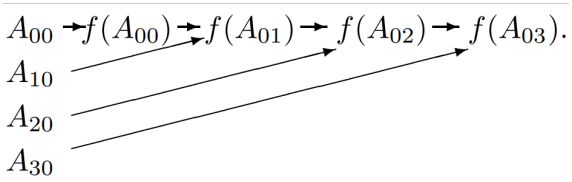
$$\begin{bmatrix} (\Pi_1^T \Pi_0^T A)(1:b, 1:b) \\ (\Pi_1^T \Pi_0^T A)(2m/p+1:2m/P+b, 1:b) \end{bmatrix} - \Pi_{02} L_{02} U_{02}$$

Applying this final obtained pivot matrix to A, the pivoted matrix for proceeding with TSLU is obtained $\Pi_0^T \Pi_1^T \Pi_2^T A$. This is an example of using a binomial reduction tree for calculating the pivot rows. A more visual representation is shown in **??**

Other forms of reduction tree are also possible such as a reduction tree of height one:



Or a flat reduction tree.



For this implementation, a modified binomial tree is used. The modifications are included to take advantage of this versatility in tournament pivoting, allowing the reduction tree to work for cases where $p \neq 2^q$ for any $q \geq 0$.

This reduction method is ideal for parallelism and results in much less communication than the flat reduction tree and less communication and memory footprint than the height one reduction tree.

Note that this method is equivalent to GEPP for $b = 1$. The U_{12} matrix is then updated as shown in Figure 1.1 with

$$U_{12} = L_{11}^{-1} A_{12}$$

and the trailing matrix updated with

$$A_{22} = A_{22} - L_{21} \cdot U_{12}$$

An important factor for stability is the rank of this trailing matrix update[4]. The elimination of each column of A results in a rank one update (for example, for $b = 1$, $A_{22} = A_{22} - l \cdot u^T$ where l, u are vectors) which is shown experimentally to be more stable than the high rank update present in block parallel pivoting.

2.3 Stability of CALU

2.3.1 Stability and Growth Factor

The stability of an algorithm measures its sensitivity to perturbations during computation. Computational methods and calculations have an inherent level of error not present in analytic solutions due to the finite level of precision achievable on machines (namely machine epsilon ϵ_m). Two major factors that make a pivoting strategy necessary in Gaussian elimination are:

- Catastrophic cancellation; when two nonequal numbers subtract to give 0 due to round off error.
- Overflow and loss of precision: Due to the way doubles are represented (IEEE754 format), there is a loss of precision when operating with larger numbers, as well as a limit on how large a number can be represented on a computer. For this reason, division by very small numbers can lead to catastrophic failure.

Error propagates, so stability is a critical concern for any linear algebra algorithm. The growth factor of an algorithm is an important measure of numerical stability. In the case of LU factorisation, the growth factor is defined as follows:

$$g_W = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{i,j}|}$$

where $|a_{ij}^{(k)}|$ is the infinity norm (L_∞) of the matrix obtained at the k^{th} step of the algorithm, while $|a_{i,j}|$ is the infinity norm of the matrix to be factorised. Note here that the infinity norm is defined as the max row sum of the matrix:

$$|a|_\infty = \max_i |a_i|$$

where $|a_i|$ is the sum of the i^{th} row. It plays an important role in backward error analysis (if a slightly perturbed solution is the exact solution to a slightly perturbed problem). Limiting the upper bound of the growth factor, as well as ensuring that

the growth factor stays relatively low for the majority of cases, is an important factor in designing an algorithm. Too large of a growth factor means that at certain iterations of the algorithm, matrix values are large enough to result in a significant loss of precision.

2.3.2 Growth factor of CALU

Theoretically, GEPP has an upper bound on the growth factor of 2^{m-1} while for CALU it is 2^{mH}, where H is the depth of the reduction tree. For GEPP, the upper bound is achieved for special matrices known as Wilkinson matrices. CALU performed similarly to GEPP for these cases, as well as all others tested by the authors. Due to the lack of experimental evidence of this higher upper bound, it is conjectured that CALU also has an upper bound of 2^{n-1} [1].

Although CALU implemented in L.Grigori's paper is performed on a 2d grid of processes, it has the same growth factor constraints as the 1D implementation discussed in this report. This is due to the tournament pivoting step being performed on a 1D grid in both implementations, and the remaining algebraic operations being equivalent from a stability standpoint, regardless of how many cores they are distributed among.

The growth factor of this implementation was calculated for a sample of random matrices of varying sizes among varying numbers of processes in chapter 5. An overview of how the growth factor was computed is outlined in chapter 4.

As expected, none of the results obtained come anywhere close to the upper bound, as this upper bound is primarily achieved by Wilkinson matrices. Should this bound be achieved on ordinary matrices, overflows would occur for small matrices rendering the algorithm useless. The growth factor staying relatively small for randomly generated matrices is a good outline of the algorithm's stability.

Chapter 3

TSLU Implementation

3.1 Matrix IO and Load Balancing

3.1.1 Random Matrix Generation

Initial test case matrices A were generated randomly in C and written to a `.txt` file. The C `rand()` function was used to generate uniformly distributed random numbers between 0 and 1. Along with the $n \times m$ A matrix, an x vector of length m was also generated, with constant values, for example $x[i] = 1.0$ for $0 \leq i \leq n$.

A b vector was then generated by using the `dgemv` BLAS function to calculate $b = Ax$. Both A and b were written to text files, to be read into the CALU solver.

3.1.2 Load balancing matrix distribution among processes

To ensure an even load of work between processes, and thus minimal overall runtime along the critical path, a general $n \times m$ matrix must be divided as evenly among the processes as possible. CALU is a block matrix factorisation algorithm that requires first, the factorisation of panels of width b, and second the update of the corresponding b rows. To avoid incurring the huge communication overhead of having a block of b rows distributed across two processes, it is of vital importance to load balance the processes in units of b rows.

It is also vital for the sake of minimising communication to be able to compute the start and end row indices not only locally, but for any processor in the communicator. There will be multiple instances in the program where it will be necessary, for the sake of communication, to compute the number of rows held by other processes. Communicating the start and end point of a process' rows incurs pointless latency

penalties and should be avoided, even at extra computational cost. Instead, the sending/receiving process calculates the start and end indices of its destination/source process locally. This is done using the load balancing function outlined below.

Algorithm 1 Determining the start and end index of a process' rows of the matrix

$n_{original} = n$ ▷ Store n value
if $n \mod b \neq 0$ **then** ▷ If $n \nmid b$, must adjust n then offset back
  $n = n + b - (n \mod b)$ ▷ Add to n so that $n|b$
end if
$n_b = n/b$ ▷ Number of chunks of b in matrix
$c = \lfloor bChunks/p \rfloor$ ▷ Minimum no. of chunks per processor
$r = n_b \mod p$ ▷ Remainder
if $rank < r$ **then** ▷ Load balance extra chunks between first processes
  $s = rank + c * rank$ ▷ Start index
  $c \mathrel{+}= 1$ ▷ Update no. chunks to account for extra one
else
  $s = r + c * rank$ ▷ Now r processes already have extra chunk
end if
$e \mathrel{+}= s + c$ ▷ End index
$start \mathrel{*}= b$ ▷ Convert chunk to row index
$end \mathrel{*}= b$
if $rank = p - 1$ **and** $n_{original} \mod b \neq 0$ **then**
  $e = e - (b - (n_{original} \mod b))$ ▷ Offset back what was added
end if

3.1.3 MPI IO for file writing to/reading from file

Initially, matrices A and b were read in from `.txt` files and stored in arrays on the root process. The root process then looped through all processes, calculated the number of rows they should receive and sent them using `MPI_Send`. To avoid unnecessary communication, `memcpy` was used to copy the local rows from the global to local array. Each receiving process would also call `calcBlock` to determine how many rows to receive.

As the test matrices became bigger, it became clear that this solution was no longer optimal. For one, `.txt` files require more memory, placing a lower limit on the size of matrices that can be tested. Secondly (and much more importantly) binary files can be read using MPI IO, which has 2 significant benefits.

1. Provides a significant reduction in the amount of data any one process must read, thus significantly decreasing runtime along the critical path. Binary files also incur less overhead while reading in due to lack of formatting. Unlike `.txt` files, they do not need to ensure architecture independence and human readability.

2. Completely eliminates the need for the communication necessary to distribute the matrix from root process. This communication previously dominated the runtime of the entire program. The cost of reading in the matrices becomes dramatically reduced when using MPI IO instead.

While the speedup from more efficient IO operations does not actually improve runtime of the CALU solver, it allows for much faster testing on large matrices. This is essential for the investigations into the relationship between n, b and the number of processes in the results section.

To implement this, each process opens the binary file. The number of elements in that processes chunk, as well as a displacement from the start of the file is calculated. The file view is then set using `MPI_File_set_view` and all processes collectively read from the relevant sections of the file using `MPI_File_read_all` (which is significantly more efficient in this case than `MPI_File_read`). Note that to store these generated matrices in the binary files to begin with, a similar function was constructed using `MPI_File_write_all` and is included in the github repository.

3.2 Tall Skinny LU General Strategy

The implementation of tall skinny LU has three primary components.

1. A binomial tree reduction to find the best b pivot rows of the panel - the "tournament pivoting" introduced in Grigori and Demmel [1].

2. A full row pivoting scheme which pivots the rows given by the tournament pivoting to the top.

3. An unpivoted parallel LU solver which can solve perform the necessary communication and compute the unpivoted LU of the prepivoted panel.

All of these steps are done with aim to both minimise communication and minimise memory footprint. To avoid the massive inefficiencies incurred by allocating separate L, U and P matrices, the LU factorisation is performed in place (i.e. L and U are

stored in A). This avoids unnecessarily allocating $2 * n * m$ doubles, reduces the amount of copies and increases cache efficiency.

Similarly, the P matrix is not formed explicitly. For a tall skinny matrix of size $n \times m$ with $n >> m$, it would be disastrous to performance to explicitly form a $n \times n$ matrix and perform matrix multiplication to obtain PA, all with the sole aim of pivoting b rows to the top. Instead, P is represented as an array of b integers. This array contains, in order, the row indices of the b rows which must be pivoted to the top i.e. if $p = [5, 3, 8]$, rows 5,3,8 must be pivoted with rows 0,1,2.

For the reasons outlined above, this method of in place LU factorisation with a pivot index array is also used in LAPACK's LU factorisation `dgetrf`, although there is slight differences in how the pivots are stored.

3.3 Tournament Pivoting

Tournament pivoting is the most technically intricate and challenging of the three sections of TSLU. The simpler case of setting up a binomial tree assuming $p = 2^q$ (where p is the number of processes and q is some integer) will be solved first. This will subsequently be generalised to work on any number of processes, a necessary step for implementing CALU.

3.3.1 Inputs and Outputs

The `tournPivot` function has the following declaration

```
tournPivot(locA, n, b, locBlockNum, indices, comm);
```

where:

- `locA` is the array containing the process' local chunk of A.
- `n` is the total number of rows of A.
- `b` is the block size of the TSLU/CALU algorithm
- `locBlockNum` is the number of panels that have already been completed on the root processor[1].
- `indices` is an array containing the indices of the local rows of A owned by the processor.

[1]This variable is necessary for adapting TSLU to be a step in the CALU recursion. It will be further discussed in chapter 4.

- `comm` is the MPI communicator through which communication should be performed. This is relevant in the recursion section, where the communicator gets redefined to exclude finished processes.

. The only relevant output of the function is the updated indices array on the root process, which dictates which rows should be pivoted into the top b rows of this panel. Note that these pivots should be applied to the **entire matrix**, not solely the current panel (again, relevant in CALU).

This indices array is passed into the `pivotFullRow` function which will be discussed later.

3.3.2 Controlling which processes send, receive and remain dormant

As seen in subsection 2.2.1, a recursive algorithm must be implemented where each process calculates its b best pivot rows. It then either receives b rows from another process and continues or sends its best b rows to another process and terminates. The method through which this is implemented is through the introduction of two variables: `isActive`$= 2^i$, `wasActive`$= 2^{i-1}$, where i is the current iteration starting at 1. These are used to control whether a process sends or receives through the `if` statement in the pseudocode Algorithm 2.

In the case of subsection 2.2.1 for example, process 2 receives on iteration 1 since $2 \pmod{2^1} = 0$, and sends on iteration 2 since $2 \pmod{2^2} \neq 0$.

The source rank for a receiving process is calculated as `sourceRank = rank +` 2^{i-1}. The destination rank for a sending process is calculated as `destRank = rank -` 2^{i-1}. This is due to the binomial tree eventually reducing onto rank 0, processes must always send upward to the nearest available active process.

3.3.3 Accounting for cases where $p \neq 2^q$

In the vastly more common case where $p \neq 2^q$ for $q \in \mathbb{N}_0$, the binomial tree will eventually reach one or multiple stages where the number of active processes is not even. In these cases, the bottom process must be paused until an odd number of processes is once again encountered. It then must rejoin the tree, as shown in figure 3.1. This is possible since the ordering of the comparisons does not matter. As long as all rows are involved in the comparison, the most stable should rise to the top regardless.

However, one faces the issue of how to know what the source and destination ranks are when this actually rejoins the tree. Attempting to precompute which rank

Algorithm 2 Controlling which processes remain active

$i = 1$
while $p_{active} > 1$ **do**
    $isActive = 2^i$
    $wasActive = 2^{i-1}$

    Perform local LU and pivot b rows to top

    if $rank \mod isActive = 0$ **then**
        Receive from neighbouring active process and continue
    else if $rank \mod wasActive = 0$ **then**
        Send to neighbouring active process and terminate
    end if
    $i \mathrel{+}= 1$
    $p_{active} \mathrel{/}= 2$
end while

will rejoin and when is difficult. Similarly, it is difficult for a process to know that it is the odd process without communication. Many previous odd processes may have paused and rejoined so it would be difficult to run any sort of numerical check to test if a process was odd based on its rank.

The solution takes advantage of the fact that if a process is an odd process, it will attempt to receive from an out of bounds process. It can then be paused until there is an even number of processes (including the paused process), at which point the original logic of send/receive ranks holds true. This can be seen clearly in Figure 3.1. Had there been 8 processes, process 4 would still have sent to process 0 on iteration 4.

3.3.4 Local LU with `dgetrf`

To perform local LU factorisation of the panel's chunk in the tournament, LAPACK's `dgetrf` routine was employed. It operates by calculating an efficient block size and performing a serial block LU factorisation algorithm, not dissimilar in a matrix algebra sense to what was outlined in section 2, albeit without tournament pivoting. By design, tournament pivoting incurs more computational cost to decrease inter-process communication, so it stands to reason that other methods such as Gaussian Elimination are more efficient for serial algorithms. The high level of optimisation

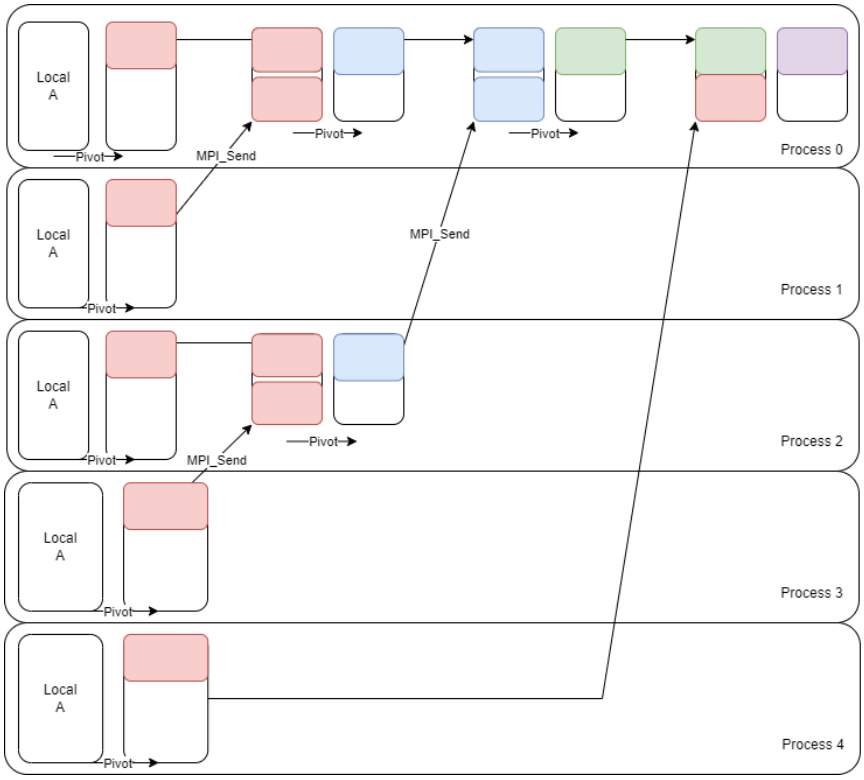


Figure 3.1: An example of a case where $p \neq 2^q$. The 5th process waits until an odd number of processes remains (in this case until 1 process remains), then rejoins the tree

present in LAPACK BLAS routines such as `dgetrf` make them an ideal tool for matrix-matrix and matrix-vector operations and thus they will be employed where possible to perform local operations.

However, some slight overhead comes from the lack of available customisation present in these routines. The LAPACK `dgetrf` routine performs LU in place, resulting in the original matrix A being overwritten. For this reason the extra cost of copying the A matrix must be incurred before passing it into the function. However, the more significant problem present concerns the format of the returned pivot array. LAPACK returns a pivot array `ipvt` representing the pivot matrix P such that row `i` and row `ipvt[i]` must be swapped. The pivot matrix P represented by this array obeys the equation $A = PLU$, not $PA = LU$.

Additionally, although not specified in the documentation, it seems that the pivot array returned does not represent the indices that must be pivoted to the top to

ensure that unpivoted LU factorisation is stable. Rather, it gives the list of indices representing the pivots that must be applied iteratively to the returned L and U to recover A. Applying the pivots iteratively rather than merely giving the list of indices that must be pivoted to the top clearly yields different results that must be accounted for. For example, consider the following pivot array P being applied to the matrix A, with $b = 3$ (i.e. 3 rows should be pivoted to the top). [2]

$$P = \begin{bmatrix} 2 \\ 0 \\ 4 \end{bmatrix} \qquad A = \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 2 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix}$$

Iterative pivoting results in the following

$$A = \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 2 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P[0]}} \begin{pmatrix} 2 & \dots \\ 1 & \dots \\ 0 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P[1]}} \begin{pmatrix} 1 & \dots \\ 2 & \dots \\ 0 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P[2]}} \begin{pmatrix} 1 & \dots \\ 2 & \dots \\ 4 & \dots \\ 3 & \dots \\ 0 & \dots \\ 5 & \dots \end{pmatrix}$$

While ensuring that the indices of A given in the `ipvt` array are pivoted to the top gives

$$A = \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 2 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P}} \begin{pmatrix} 2 & \dots \\ 0 & \dots \\ 4 & \dots \\ 3 & \dots \\ 1 & \dots \\ 5 & \dots \end{pmatrix}$$

The absence of any specification of which strategy is employed in the LAPACK documentation is presumably due to the returned `ipvt` array being primarily used as an input to LAPACK's triangular system solver `dgetrs` or inverse calculator `dgetri`.

[2] Note that due to this being a LAPACK library function initially designed for Fortran, array indexing begins at 1. However, this implementation is written in C. In the interest of avoiding unnecessarily complicating manners, the C convention of array indexing will be followed. Before attempting to use the pivot array, it must be ensured that the array is looped through and 1 subtracted from all entries to convert it to C style indexing.

Regardless of cause, this led to the initial implementation of the CALU solver assuming the format of the latter strategy, which we will henceforth refer to as static pivots. This could clearly lead to instabilities which should otherwise be avoided through correct tournament pivoting.

To rectify this issue, the pivots had to be transformed from the iterative form with $A = PLU$ to the form with $PA = LU$, be it static or iterative. This must be done at low computational cost and avoid any extra communication cost, so the communication can be kept at a minimum. Should too much computational cost be introduced, it may negate some of the value of using the LAPACK library function to begin with.

Initially, since the pivoting functions for serial and parallel pivoting had been written for static pivoting arrays, the pivots were transformed into a static array. However, this led to needing to allocate n integers. The disadvantage of this was two fold:

1. It led to an increase in computational work, both in terms of cache efficiency and larger loops of computation.

2. It led to the algorithms only n dependence for memory allocation.

The second of these is by far the most concerning and motivated rewriting the algorithm to work with iterative pivoting. This allows the issue of converting from a returned `ipvt` array obeying $A = PLU$ to one obeying $PA = LU$ to be resolved easily. This is achieved simply by applying the pivots in reverse order, as shown in the example below.

Let

$$A = \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 2 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix}$$

and let `dgetrf(A)` return

$$P = \begin{bmatrix} 2 \\ 0 \\ 4 \end{bmatrix} \qquad L \cdot U = \begin{pmatrix} 4 & \dots \\ 0 & \dots \\ 1 & \dots \\ 3 & \dots \\ 2 & \dots \\ 5 & \dots \end{pmatrix} \implies P \cdot L \cdot U = \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 2 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix} = A$$

Applying P in reverse order to A yields

$$A = \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 2 & \dots \\ 3 & \dots \\ 4 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P[2]}} \begin{pmatrix} 0 & \dots \\ 1 & \dots \\ 4 & \dots \\ 3 & \dots \\ 2 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P[1]}} \begin{pmatrix} 1 & \dots \\ 0 & \dots \\ 4 & \dots \\ 3 & \dots \\ 2 & \dots \\ 5 & \dots \end{pmatrix} \xrightarrow{\text{Apply P[0]}} \begin{pmatrix} 4 & \dots \\ 0 & \dots \\ 1 & \dots \\ 3 & \dots \\ 2 & \dots \\ 5 & \dots \end{pmatrix} = L \cdot U$$

so clearly applying P iteratively in reverse to A recovers $L \cdot U$ (i.e. $P \cdot A = L \cdot U$). Note that throughout, the `ipvt` array and the permutation matrix P are used interchangeably. While perhaps not mathematically rigorous, in practice they are equivalent and are treated as such in the interest of clarity.

3.3.5 Applying local pivots and keeping track of global indices

Once the correct pivots have been obtained from the local LU solver, they must be applied to A. This is done by iterating backwards through the `ipvt` array and using `memcpy` to perform the row swap (using a `temp` array).

Previous sections outline how a processor calculates if it will send or receive, and to which processor it will do so. If the process is sending its top b rows to its destination rank, it sends then terminates. If receiving, the process:

- Reallocates the `locA` array to hold $2b$ doubles using `realloc` to keep the contents of the first b rows intact (only necessary on the first iteration).
- Receives from its calculated source rank into the b rows underneath its own top b rows.

Once the recursion ends, the root process performs its final local LU to determine the best b overall rows of the matrix (with regards to keeping the LU stable).

Similarly, the `indices` array containing the rows owned by this processor gets updated at every step that the A matrix does above. This is a vital step, if not *the* vital step of the TSLU algorithm. As above, the `indices` array gets pivoted with the same pivots that are applied to A, and the top b entries progress in the tournament.

After the recursion and final update, the b entries at the top of the `indices` array contain the global indices of the most stable rows in the panel. These entire rows will be pivoted to the top of the overall matrix A (for TSLU at least, for CALU this changes slightly) using the `pivotFullRow` function.

3.4 Full row pivoting scheme

3.4.1 Overview of function

The `fullRowPivot` function aims to pivot entire rows of the matrix A using the overall best pivot indices returned by the `tournPivot` function. To do this it must

- Iterate backward through the `indices` array.
- Determine on root which rank owns that row, then send to and receive from that process.
- Determine on other processes if that row is in their local chunk. If so, receive from and send to that process
- If root owns `indices[i]`$^{\text{th}}$ row, use `memcpy` to swap rows.

3.4.2 Determining which rank owns `indices[i]`$^{\text{th}}$ row

On the root process, it must be determined in advance which process owns the row `indices[i]` that row `i` is to be pivoted with. While an MPI Remote Memory Access (RMA) may have been utilised to allow passive communication whereby the root process would not *need* to know which process was accessing its memory, this has an unacceptable level of overhead and synchronisation.

Instead, the `calcBlock` function described previously is used in a loop from `0:nprocs`. The starting index of all process' local chunks are stored in a `starts` array. This array is then looped through until a process `j` is encountered such that `starts[j]`$\leq$`indices[i]`$<$`starts[j+1]`. This process becomes the rank with which the swap will occur.

3.4.3 Performing the swap

Once the rank owning the row has been determined, provided it isn't 0, the root process opens an `MPI_Send` to send the `i`$^{\text{th}}$ row to the swapping process. Once this send is complete, it opens a `MPI_Recv` to receive the `indices[i]`$^{\text{th}}$ row from that process.

The other processes run a check to see if they own the `indices[i]`$^{\text{th}}$ row by checking if `start`$\leq$`indices[i]`$<$`end`. If this is true, the local index of the row is obtained as `indices[i] - start`, where here `start` denotes the global index of the first row on this process. The swap is then initiated. The row is copied into a `temp`

array. A `MPI_Recv` call is made to match the `MPI_Send` on the root processor. A `MPI_Send` call is then made to send the `temp` array to root, as shown below.

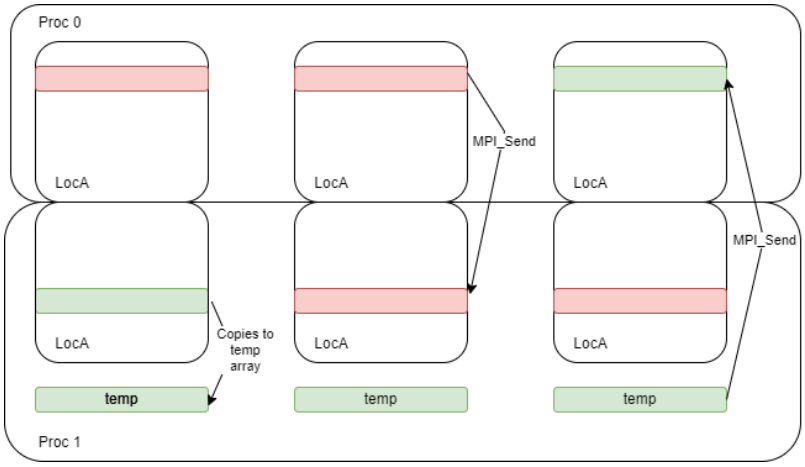


Figure 3.2: How the `fullRowPivot` function swaps rows with other processes

Should the swapping rank be 0, the root process uses `memcpy` to perform the swap. This is achieved as above through

1. Copying the `i`$^{\text{th}}$ row to a `temp` placeholder array.
2. Copying the `indices[i]`$^{\text{th}}$ row to the `i`$^{\text{th}}$ row.
3. Copying the `temp` array to the `indices[i]`$^{\text{th}}$ row, thus concluding the swap.

3.5 Parallel unpivoted LU solver

3.5.1 Overview of function

To obtain the LU factorisation of the panel, the local unpivoted LU factorisation must first be calculated on the root process. The obtained U matrix is then extracted and broadcast to all other processes. This is the only necessary communication for this part of the algorithm, leading to only $(nprocs-1)*b^2$ doubles needing communicated (can be reduced further by not broadcasting 0s, but benefit is small when $b << n$). The remaining processes then calculate their chunk of the L matrix, concluding the LU factorisation.

3.5.2 Unpivoted LU on root process

To perform LU on the root process, the following algorithm is performed on a $(n \times b)$ matrix with *block* rows locally on process:

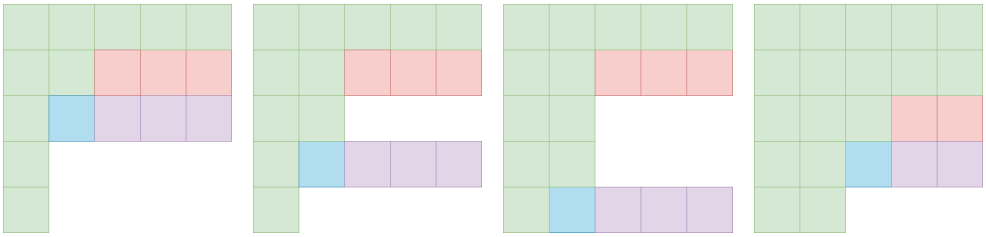


Figure 3.3: Visualisation of how algorithm performs in place unpivoted LU factorisation on root process. Performs updates then proceeds to the next diagonal element. Continues recursively until full matrix is factorised.

1. The diagonal elements are iterated across $0 \leq k < b$.
2. All elements on the column of the diagonal (column k) are iterated over $0 \leq j < block$ and divided by the diagonal element $A_{j,k} = A_{j,k}/A_{k,k}$ (shown in blue in Figure 3.3).
3. For each of these elements $A_{j,k}$, the rest of the row iterated over $k < i < b$.
4. An update is then performed $A_{j,i} = A_{j,i} - A_{j,k} * A_{k,i}$ ($A_{k,j}$ shown in red in Figure 3.3, $A_{i,j}$ is shown in purple).

This is outlined in Figure 3.3. The upper triangular matrix is then extracted and broadcast to the other processes by collectively calling `MPI_Bcast`.

3.5.3 Updates on other processes

Other processes perform a similar algorithm to the one on root process, with some changes. On root the algorithm iterates along the diagonal and updates elements underneath and to the right of the current diagonal element, Off process, the algorithm uses the U communicated by the root process to update entire columns to the right of the current entry (more clearly illustrated in the diagram below). As seen in the figures above, the entire column is updates, not just the subdiagonal entries. The algorithm is outlined below

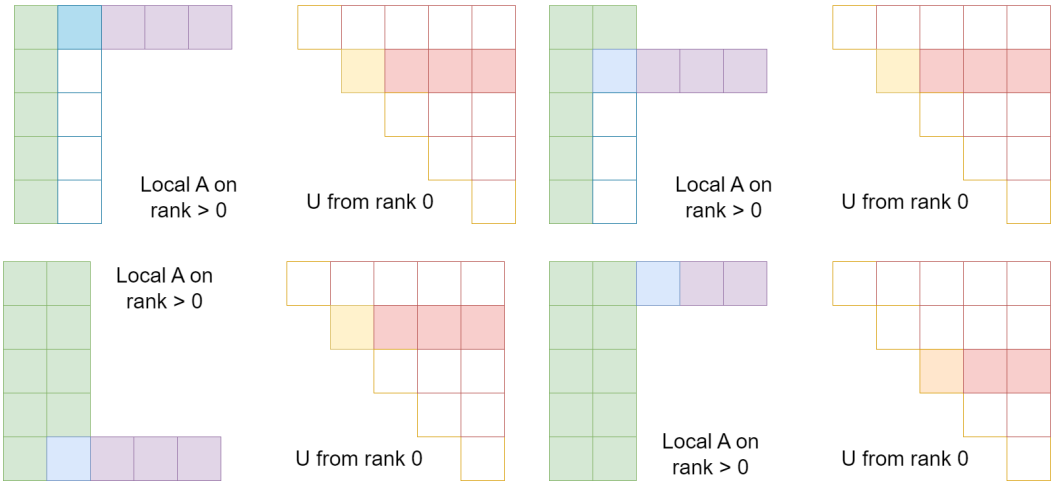


Figure 3.4: Visualisation of how algorithm performs in place unpivoted LU factorisation off root. Receives U from root process and uses it to update its chunk of $A \longrightarrow L$

1. Iterate over $0 \leq k < b$.
2. For each k, iterate over all rows $0 \leq j < b$ and perform the update $A_{j,k} /= U_{k,k}$.
3. For each row j, iterate over all columns $k < i < b$ and perform the update $A_{j,i} -= A_{j,k} * U_{k,i}$

Once these updates have been performed on all processes, the TSLU algorithm is complete. It runs for a matrix of arbitrary dimensions (provided the matrix is tall and skinny) on any number of processes. Both of these factors are important, as TSLU will become one of the critical steps of CALU's recursive block factorisation.

Chapter 4

CALU Implementation

4.1 CALU algorithm overview

The core idea of the CALU algorithm is the following

1. Perform TSLU with tournament pivoting on the first panel of b columns.
2. Update the first b rows.
3. Update the trailing matrix.
4. Proceed recursively on this trailing matrix (Schur's complement).

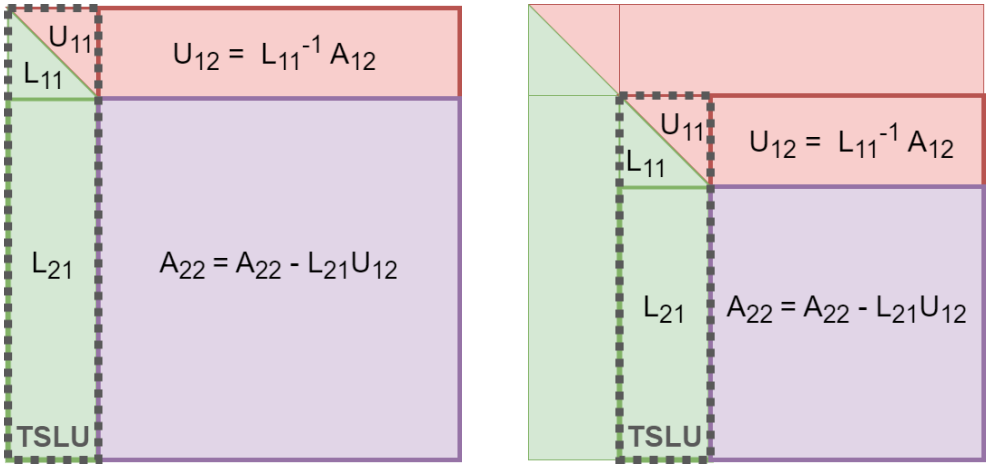


Figure 4.1: CALU algorithm's recursion

This leads to a number of implementation challenges for practically every step of TSLU already discussed, which will be explored in the following section. Other challenges involve performing the updates to U_{12} and A_{22} as well as handling the two kinds of recursion at play here; on process recursion (where the next block in the recursion is on process), and inter-process recursion (where the next block resides on the next process).

Three critical variables which will be seen throughout this chapter are:

- `blockNumber` (n_b): Records the total number of blocks of b rows and columns already factorised.
- `locBlockNum` (n_{lb}): Records the number of blocks factorised on the currently active process.
- `block`: Records the number of rows still requiring factorisation on the local process.

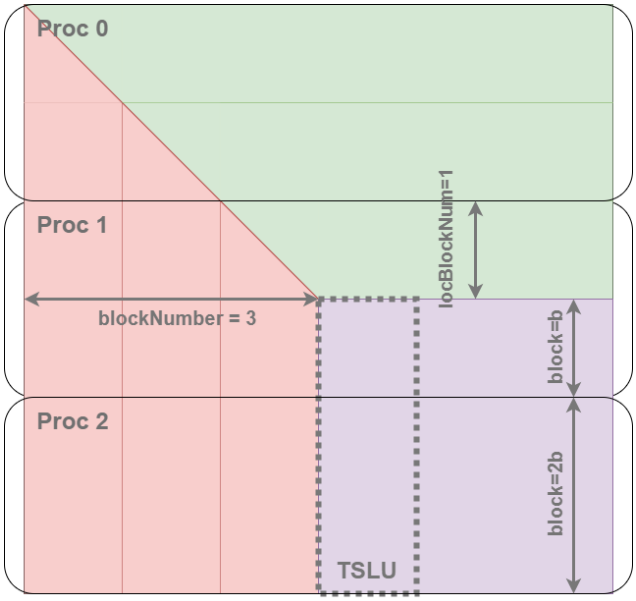


Figure 4.2: Visualisation of the variables `blockNum`, `locBlockNum` and `block`

The purpose of these variables is best understood visually, illustrated in Figure 4.2. `locBlockNum` and `blockNum` give the block row and block column displacements of the current panel in the local chunk of the A matrix. `block` gives the height of the panel on each process.

4.2 Updating TSLU for recursion

4.2.1 Updating tournament pivoting for recursion

To begin, the panel to be factorised with TSLU is copied from the global matrix to a new array. It is not possible to perform this in place, as the tournament constantly overwrites and pivots rows in the panel. The following pseudocode outlines how the correct elements of A are accessed to extract the panel.

Algorithm 3 Extracting panel W from A

if $rank = 0$ **then** ▷ On root, account for column *and* row offset
  $s = n_{lb} * b * m + n_b * b$ ▷ Calculate start index
else ▷ Off root account for column offset
  $s = n_b * b$
end if

for $i = 0 : (\text{block}-1)$ **do**
  for $j = 0 : b - 1$ **do**
    $W[i * b + j] = A[s + i * m + j]$ ▷ Note stride difference
  end for
end for

For the `tournPivot` function, caution must be taken when using the `calcBlock` function. This function determines the start and end indices of each process' chunk. It is an essential step in filling the local `indices` array with the correct global indices[1].

The panel passed in may not be evenly distributed among processes, as the root process has already processed $n_{lb} * b$ rows. Should `calcBlock` be used naively in this context, it would incorrectly assume that the panel is evenly distributed among processes. This is shown in Figure 4.3.

To remedy this, the length passed into `calcBlock` is the original length of the process' block; $block + n_{lb} * b$. The calculated start and end indices are then offset back to account for the added inclusion of the $n_{lb} * b$ extra rows.

This is the only necessary alteration to the `tournPivot` function. Once complete, the allocated memory for the panel can be freed. The pivot indices are the only output of the function required for the algorithm.

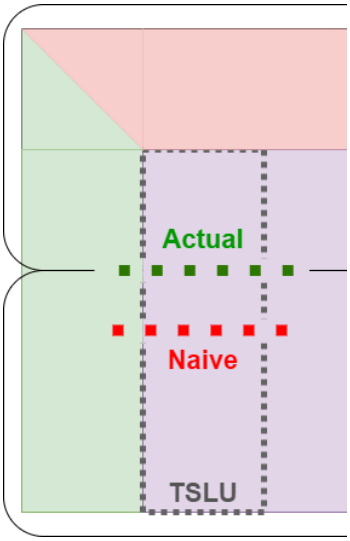


Figure 4.3: Naive use of `calcBlock` would assume that the panel is distributed evenly between processes

[1] Recall each process passes an array with the global row indices it contains into the tournament

4.2.2 Updating full row pivoting and unpivoted parallel LU for recursion

For updating the full row pivot function, recall that to determine which process owns the pivoting rank `calcBlock` was used in a loop. As above, the change in local block number n_{lb} must be accounted for. Aside from that, this function remains the same. Similarly for the parallel unpivoted LU factorisation, all that is updated is the `startIndex` (s) in the same fashion as shown in Algorithm 3.

Note that for the above three functions, only adaptations to account for *on-process* recursion have been discussed. This is due to both of these functions accepting a MPI communicator as an argument. As will be discussed later, the redefinition of this communicator once a process is finished (as well as some variable updates) are all that is necessary for recursion between processes.

4.3 Performing matrix updates on U_{12} and trailing matrix A_{22}

4.3.1 Updating U_{12}

After TSLU has been performed on the panel, the remainder of the first b rows must be operated on to convert A_{12} to U_{12}. The equation representing the update (as seen in Figure 1.1) is $U_{12} = L_{11}^{-1}A_{12}$. The calculation of the inverse of the lower triangular matrix L_{11} incurs unnecessary cost. Instead, the LAPACK function `dtrsm` is employed to efficiently solve the triangular system.

Initially, both L_{11} and A_{12} were extracted and stored in separate arrays using user defined functions. The extraction of L_{11} is trivial, For A_{12}, this is achieved by using n_{lb} and n_b to calculate the starting index as follows

$$\texttt{startIndexA12} = n_{lb} * b * m + (n_b + 1) * b$$

then performing a simple copy with row length $m - (n_b + 1)$ and leading dimension m.

The arrays were subsequently passed into `dtrsm`, which overwrites the result onto A_{12} (i.e. $A_{12} \longleftarrow L_{11}^{-1}A_{12}$). The A_{12} matrix, now updated to U_{12}, is broadcast to other processes to update the trailing matrix. This is the only communication necessary for the trailing matrix update, as illustrated in the diagram below. Finally, U_{12} was copied back into the A matrix on root which concludes the U_{12} update.

However, this results in the unnecessary computational work of copying the U matrix into a separate array and copying it back into A once the update has been

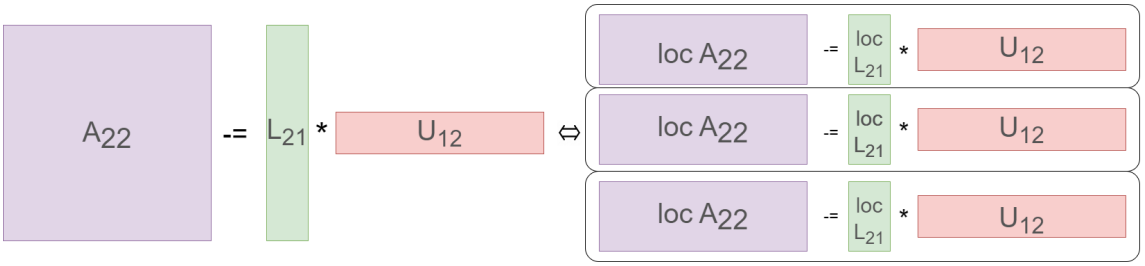


Figure 4.4: Parallelism of A_{22} trailing matrix update

performed. This is remedied by updating A_{12} in place with `dtrsm` and using the noncontiguous `MPI_vector_type` to broadcast U_{12} to non-root processes. For the former, the `dtrsm` function allows row length and leading dimension as separate arguments. Thus, using the `startIndexA12` variable defined earlier in combination with leading dimension m and row length

$$\texttt{rowLength} = m - (n_b + 1) * b$$

this allows the function to update A_{12} to U_{12} in place.

The next issue is how to broadcast this noncontiguous chunk of the array to other processes. The solution lies in `MPI_Datatype`, which allows the user to define data types which can be sent/received. In this case, `MPI_Type_vector` is used. This is the ideal option when one is dealing with a regular data pattern, where there is a block of a basic `MPI_Datatype` (in this case `MPI_DOUBLE`) of constant length with the start of each block separated by a constant amount. This type is committed and used in the `MPI_Bcast` on the root process.

The processes which will receive from the `MPI_Bcast` call however, will be storing U_{12} separately rather than in the A array. Thus, other processes allocate for U_{12} and set it as the receive buffer for `MPI_Bcast`. Since U_{12} is stored separately on these processes, it is more efficient to have it stored contiguously (for both memory footprint and cache efficiency). To achieve this, the non-root processes call `MPI_Bcast` with type `MPI_DOUBLE`. The `count` argument reflects the number of doubles stored in the vector type sent by the root process. This allows U_{12} to be received contiguously on the other processes, without ever having to explicitly extract it on the root process.

4.3.2 Performing A_{22} trailing matrix update

There are two forms of matrix update that must be considered:

1. The update of the trailing matrix A_{22} on the root process.

2. The update of the sections of A_{22} owned by remaining nonroot processes.

Once again, a CBLAS function will be used to perform the necessary update

$$A_{22} \longleftarrow A_{22} - L_{21} * U_{12}$$

For the local matrix multiplication and subtraction, the `dgemm` function is used. Note that the parallelism of this operation can be seen in Figure 4.4. Once U_{12} is broadcast, each process can perform an update of it's own section of the trailing matrix.

The `dgemm` function performs the following operation

$$C \longleftarrow \alpha * A * B + \beta C$$

so clearly by setting $\alpha = -1$ and $\beta = 1$, the desired equation is recovered with $A = L_{21}$, $B = U_{12}$ and $C = A_{22}$.

In addition to the `startIndexU12` (on root) and `rowLength` variables defined earlier, the following variables will be used to facilitate performing this matrix multiplication in place (i.e. with no copying necessary outside of the temporary workspace inherent to the `dgemm` function):

- `startIndexL21`, denoting the local start index of the L_{21} block of the matrix. On root, accounting for row and column offsets gives

  $$\texttt{startIndexL21} = (n_{lb} + 1) * b * m + n_b * b$$

  Off root, where there are no row offsets this reduces to

  $$\texttt{startIndexL21} = n_b * b$$

- `startIndexA22`, denoting the local start index of the A_{22} block of the matrix. Similarly, accounting for row and column offsets on root gives

  $$\texttt{startIndexA22} = (n_{lb} + 1) * b * m + (n_b + 1) * b$$

  while off root gives

  $$\texttt{startIndexA22} = (n_b + 1) * b$$

These variables give the starting address (eg. `locA + startIndexA22`) of the matrix blocks. Clearly there is b rows in U_{12} and columns in L_{21}. The number of columns in U_{12} and A_{22} is given by the `rowLength` variable defined earlier. The number of rows in L_{21} and A_{22} is given by the `block` variable defined previously (subtracting b on root to account for the block just processed). The final value needed to perform the multiplication in place is the leading dimension of these blocks. This is m in all cases, as A has m columns.

Thus, everything necessary to perform in place matrix multiplication for the trailing matrix update can be calculated at every step of the recursion. On root, U_{12} is in the local chunk of the matrix and is specified similarly to the L_{21} and A_{22} blocks. Off root, it is held in a contiguous array and is thus trivial to pass as an argument.

This concludes the final step of an iteration of the recursion.

4.4 Recursion updates

4.4.1 On process recursion

With all matrix updates finished, some basic variable updates must be performed to prepare for the next step of the recursion (assuming the next block resides on the same process). The `locBlockNum` (n_{lb}) and `blockNum` (n_b) variables are increased by one to account for another block having been completed. This is essential for accessing the correct memory in the next iteration. On root, b is subtracted from the `block` variable to account for there being b fewer rows left to factorise.

This concludes the on process recursion (with the exception of the final block, which will be considered in a later section). The issue that remains is, once one process has completed the factorisation of its chunk of the matrix, how does the algorithm continue?

4.4.2 Inter-process recursion

To handle inter-process recursion, MPI communicators are used. Through redefining the communicator to exclude the finished processes, the process that is currently factorising blocks of rows can always be the root process. This greatly simplifies communication and makes it highly suitable for recursion. The alternative would be rife with issues attempting to calculate rank offsets for communication, as well as for functions such as `calcBlock` which has been utilised regularly throughout the implementation.

With all functions that involve communication accepting this communicator as an argument, the communication space is always limited to processes that are active (i.e. communicating and performing updates on their rows in the matrix).

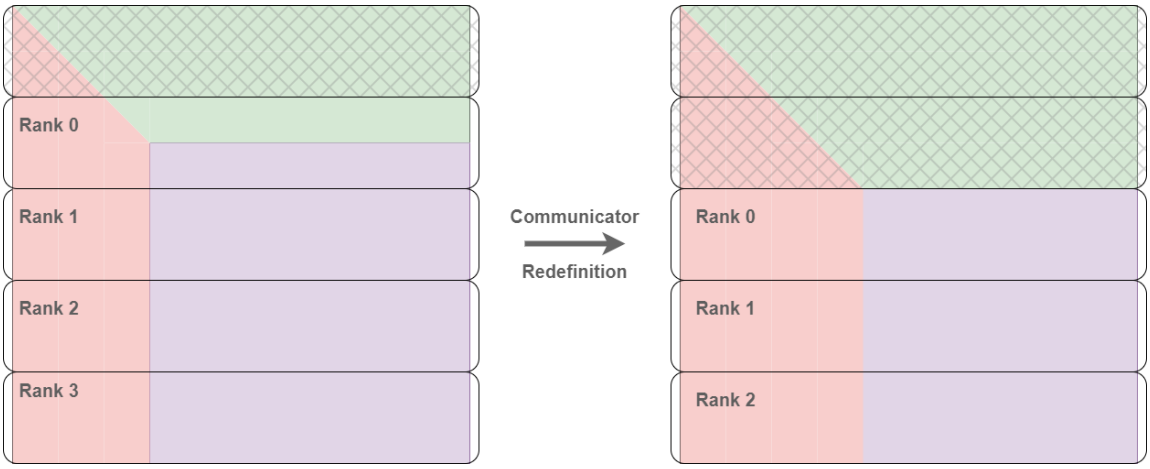


Figure 4.5: How the communicator gets redefined once the root process finishes factorising its chunk of rows

To implement this communicator redefinition, a communicator must be declared and initialised to `MPI_COMM_WORLD`. Each process then needs to know when the first process' block is finished. Ideally this is done without communication, as to not unnecessarily increase latency (number of messages sent). With this aim, each process calls `calcBlock` passing 0 in as the rank, and storing the end index in a variable `rootEnd`.

An if statement is then employed at the end of each step in the recursion, testing if

$$\texttt{rootEnd} = n_{lb} * b$$

When this condition is satisfied, the root process is finished factorising and the communicator must be redefined. The new communicator must exclude the root process and make the previous rank 1 processor the new root, as seen in Figure 4.5.

MPI Groups are used to achieve this as follows:

1. An MPI Group is formed from the processes in the communicator using `MPI_Comm_group`.
2. The root process is excluded from this new group using `MPI_Group_exclude`.
3. A new communicator is formed from this group using `MPI_Comm_create`.

4. The old root process exits the recursion.

5. The `rank` variable is updated to be the process' rank in the new communicator. The `locBlockNum` counter (n_{lb}) is reset to 0, as no blocks have been processed on the new process. The block variable is reset with `block = e-s`.

6. Finally, the end index of the new root process is calculated on all processes as above using `calcBlock`. This time however, care must be taken to pass in the number of processes in the new communicator, and the number of rows present on those processes ($n - n_b * b$).

The algorithm then proceeds to the next iteration. This recursion continues while both of the following hold:

$$(n_b + 1) * b < n \tag{4.1}$$

$$(n_b + 1) * b < m \tag{4.2}$$

This is to prevent going out of bounds when accessing L_{21}, U_{12}, A_{22}.

4.5 Handling final block of matrix

4.5.1 Considerations for factorising final block

For the final block, two things must be considered:

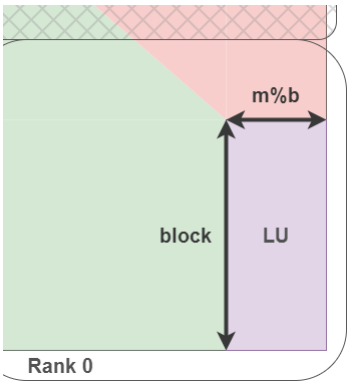


Figure 4.6: Case of tall matrix with $n \nmid b$

- The shape of the overall $n \times m$ matrix. If the matrix has $n \geq m$ the last block must be handled differently than if $m > n$. One requires performing LU on a tall matrix, the other requires performing LU on a square matrix, then updating the U_{12} block.

- The number of rows/columns left. If, for example, there is 11 rows and 12 columns with block size 3, the rows will be the terminating factor for the recursion. The recursion will end with 2 rows unfactorised. This is not enough for a block of $b = 3$ rows. However, b cannot be naively replaced with $m \mod b = 2$ in the functions, as b is used for calculating offsets for memory access. Thus any functions used will need to be rewritten to accept b and size of block (denoted `rem` in code) as separate arguments.

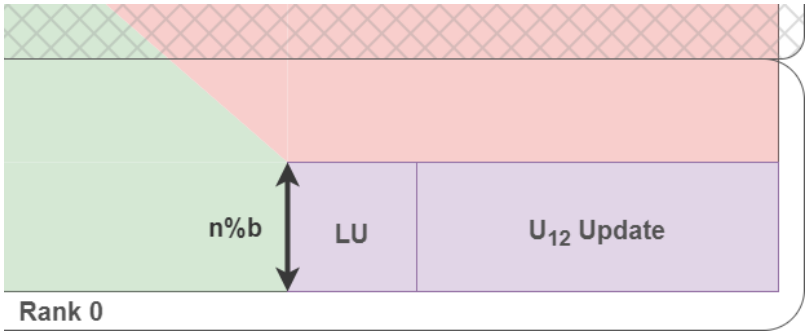


Figure 4.7: Case of a wide matrix with $n \nmid b$

Additionally to these concerns, it is not possible to simply run a local LU factorisation with CBLAS on the remaining panel. Any pivots applied must be applied to the entire matrix, not solely to the panel. Only *then* can unpivoted LU be performed.

4.5.2 Performing LU on final block

To begin, the number of columns in the LU factorisation must be ascertained, denoted `rem` (note n denotes number of rows and m number of columns).

- If $n < m$ and $n|b$, clearly `rem`$= b$. Otherwise, `rem`$= n \mod b$.
- Similarly, if $m \leq n$ and $m|b$, `rem`$= b$. Otherwise `rem`$= m \mod b$.

With the number of rows known, CBLAS' `dgetrf` is called to calculate the `rem` best pivots. These pivots are then applied to the full matrix using the `pivotMatrix` function (the local pivot function utilised in TSLU. Can use local pivot function since all pivot rows are guaranteed to be on process). A local unpivoted LU function `finalLU` is then used, which can accept b and `rem` as separate arguments to correctly calculate offsets[2]. This function completes the cases where $n \geq m$.

For $m > n$, the remaining $U_{12} = L_{11}^{-1} A_{12}$ (seen in Figure 4.7) must be calculated. With a slight change of offsets to account for a possibly different block size;

$$\texttt{startIndexA12} = n_{lb} * b * m + n_b * b + rem$$

$$\texttt{rowLength} = m - (n_b * b + rem)$$

[2]Since implementation of LU have been discussed at length, a description of this functions implementation is omitted. It can be found in the file `func/luFunc.c` in the github repository

L_{11} is extracted and, through using the CBLAS `dtrsm` function, U_{12} is updated in place.

This concludes the factorisation of the final block of A and thus the CALU algorithm.

4.6 Input and Output Format

4.6.1 CALU function Usage and Additional Functions

The CALU function has the following declaration:

```
calu(locA, locb, n, m, b)
```

where;

- `locA` is the local chunk of the A matrix held on process, as calculated with `calcBlock`. On output, this contains that process' section of the in place LU factorisation.
- `locb` is the local chunk of the $n \times 1$ b vector (again as calculated by `calcBlock`). This has the primary function of keeping track of all row pivots that occur throughout the factorisation. It has two formats, depending on desired use.

  Should CALU factorisation be used to solve a system of linear equations $Ax = b$ with b being passed as the argument, on output b will be correctly pivoted to match the pivots applied to A during LU factorisation. This allows the system to be solved immediately with back substitution.

  Alternatively, should CALU simply be needed to factorise A, b should be the list of indices from $0 : n - 1$. On output, it will hold the new order of the indices. For example, should

$$A = \begin{pmatrix} 2 & \dots \\ 4 & \dots \\ 6 & \dots \\ 8 & \dots \\ 10 & \dots \end{pmatrix} \xrightarrow{CALU} L \cdot U = \begin{pmatrix} 8 & \dots \\ 2 & \dots \\ 10 & \dots \\ 6 & \dots \\ 4 & \dots \end{pmatrix}$$

  Then

$$b = \begin{bmatrix} 3 \\ 0 \\ 4 \\ 2 \\ 1 \end{bmatrix}$$

- `n` is the number of rows in the global matrix A and vector b.
- `m` is the number of columns in A.
- `b` is the block size for the algorithm.

4.6.2 Utility Functions: Triangular system solver

Several functions are included in the repository which process and test the returned matrix. Since they are not directly a part of the CALU algorithm, they will only be mentioned briefly.

As an initial test of the accuracy of the LU factorisation, the matrix A and vector b were generated such that $Ax = b$ for some constant vector x. Upon output from CALU solver, the matrices A and b were gathered onto root. A simple, serial triangular system solver was written to calculate x and write it to file, also displaying the average x value for convenience.

4.6.3 Utility Functions: Growth Factor Calculation

Recall that the growth factor of the algorithm g_W, an important measure of algorithmic stability, is measured as follows:

$$g_W = \frac{max_{i,j,k}|a_{ij}^{(k)}|}{max_{i,j}|a_{i,j}|}$$

where $|a_{ij}^{(k)}|$ is the L_∞ norm of the matrix a on iteration k of the algorithm and $|a_{i,j}|$ is the L_∞ norm of the matrix A. A serial function was written, calculating the max row sum of the matrix (implementation is trivial).

Gathering the matrix to root process every iteration would result in a catastrophic level of slowdown for calculating $|a_{ij}^{(k)}|$ for any reasonably large matrix. With this in mind, the function is implemented in parallel.

Clearly, the overall max row sum will be the max row sum on some process' local block. Therefore the global max row sum can be found as follows:

- The local infinity norm (max row sum) is calculated on each process.
- The maximum of these local infinity norms is calculated using `MPI_Allreduce` with the `MPI_MAX` operation.
- The global infinity norm is now held on each process.

This global infinity norm is calculated on each iteration, overwriting a `maxInfNorm` variable if it is the largest encountered so far. The result is returned from the CALU function and divided by the infinity norm of the A matrix to calculate the growth factor.

4.6.4 Utility Functions: Pivoting $L \cdot U$ back to A

As discussed in subsection 4.6.1, for general applications of CALU the b input vector can be a list of row indices from $0 : n-1$. Upon return, it contains the new order of the matrix $L \cdot U$. To be able to compare this $L \cdot U$ matrix with A, the matrix must be pivoted so that its row order is reverted to $0 : n-1$. Going forward the b array will also be referred to as the pivot array.

To do this, the rows of $L \cdot U$ are iterated over. For each row i, `pivot[j]=i` is found. This tells us that row i in A is located at row j of $L \cdot U$.

Row `j` of $L \cdot U$ is then pivoted with row i, placing it in the correct place in the matrix. However, this means that row i of $L \cdot U$ has moved. This must be reflected in the pivot array, to keep its map of row indices correct. The rest of the pivot/b array is iterated through in this fashion.

This may be easier to follow through an example. Below, one iteration of this strategy is shown.

$$A = \begin{pmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \qquad L \cdot U = \begin{pmatrix} 2 & 4 & 6 \\ 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix} \qquad b = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

$$L \cdot U = \begin{pmatrix} 2 & 4 & 6 \\ 0 & 2 & 4 \\ 1 & 3 & 5 \end{pmatrix} \xrightarrow[\text{pivot rows 1 and 0}]{b[1]=0} \begin{pmatrix} 0 & 2 & 4 \\ 2 & 4 & 6 \\ 1 & 3 & 5 \end{pmatrix}$$

Now updating b

$$b = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix} \xrightarrow[Update]{b[1]=0} \begin{bmatrix} 0 \\ 2 \\ 1 \end{bmatrix}$$

Clearly iterating through $L \cdot U$ in this fashion will recover A, allowing for error analysis using $||A - PLU||_\infty$.

Chapter 5

Results

5.1 Strong and Weak Scaling of CALU

5.1.1 Strong scaling

In strong scaling, the runtime of the algorithm is measured for an increasing number of processes while problem size remains constant. Ideal strong scaling is achieved if the following holds

$$\tau(p) = \frac{\tau(1)}{p}$$

Only embarrassingly parallel problems achieve close to this. Problems which are interesting to parallelise typically require communication and synchronisation between processes. Additionally, since inter-process communication is a huge bottleneck for algorithms, they often incur additional computation to avoid this (as is the case in CALU).

The speedup achieved by an algorithm is a measure of how much faster the algorithm runs with p processes instead of 1.

$$s = \frac{\tau(1)}{\tau(p)}$$

Strong scaling tests run on chuck's 12 processes are shown below, running on a variety of random matrices of different size. The algorithm is primarily suited to larger problems for which parallelism is necessary, so it is expected that the strong scaling improves with problem size. Note that for each matrix size, the same randomly generated matrix is used to avoid noise in the timings.

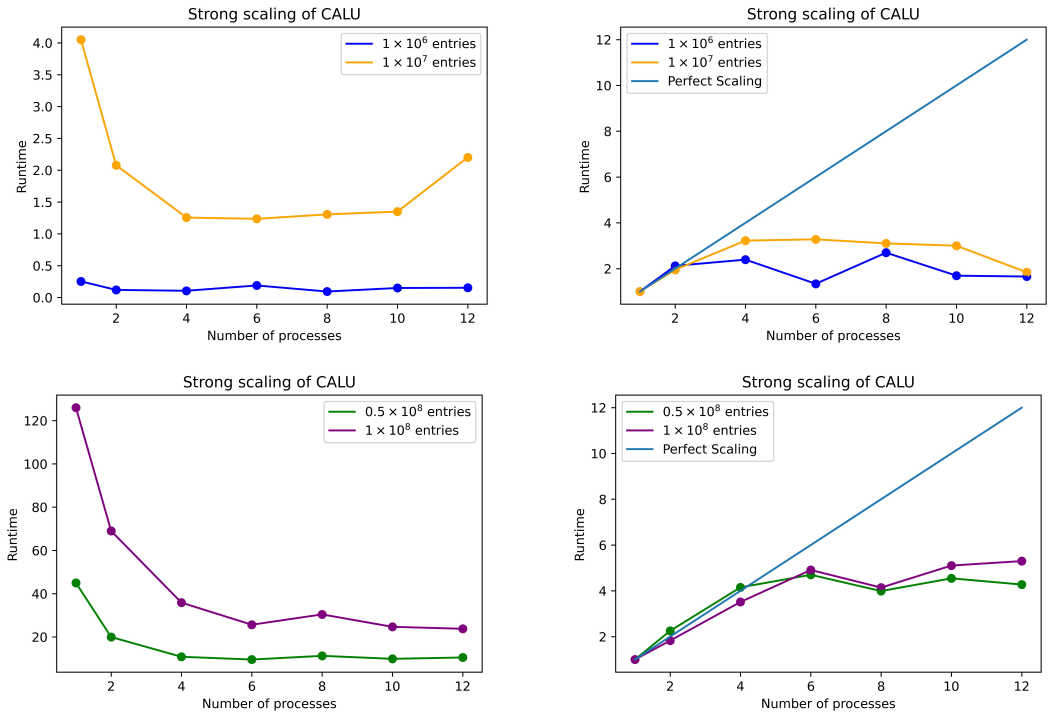


Figure 5.1: Strong scaling runtime (left) and speedup (right) for small (top) and large (bottom) matrices. Note that as the matrix size increases, the strong scaling gets better. This is due to the overhead of setting up the communication no longer outweighing the computational benefit of the extra processes

Clearly, the runtime increases unrealistically at the beginning. This is caused by the fact that the algorithm is designed to work on multiple processes, and incurs a lot of pointless overhead setting up communication without any of the upside that this brings in parallel.

This misleading initial speedup aside, the results are as expected. For smaller problem sizes (top), the overhead of setting up and executing the inter-process communication negates much of the benefit parallelism offers. The speedup decreases as more processes are used, as the overhead of using more processes outweighs the computational benefit of having them.

However, when there is more computational work (i.e. when the matrix is larger, as is the case in the bottom plots), the additional processes are worth the overhead. Large reductions in runtime and significant speedup is observed when working with

these larger matrices. Even when the $p = 1$ case is omitted, a

$$s = \frac{\tau(2)}{\tau(12)} \approx 2.9$$

speedup is achieved when going from 2 to 12 processes for a $10^4 \times 10^4$ randomly generated matrix (10^8 entries). This clearly shows that the algorithm has merit as a parallel solver.

For larger problem sizes tested on a larger number of processes, significant speedup is again observed. Seen below is the runtime and speedup for between 16 and 80 processes for a variety of block sizes. Note that the speedup is relative to the runtime on 16 processes.

$$s_{max} = \frac{\tau(16)}{\tau(45)} \approx 1.8$$

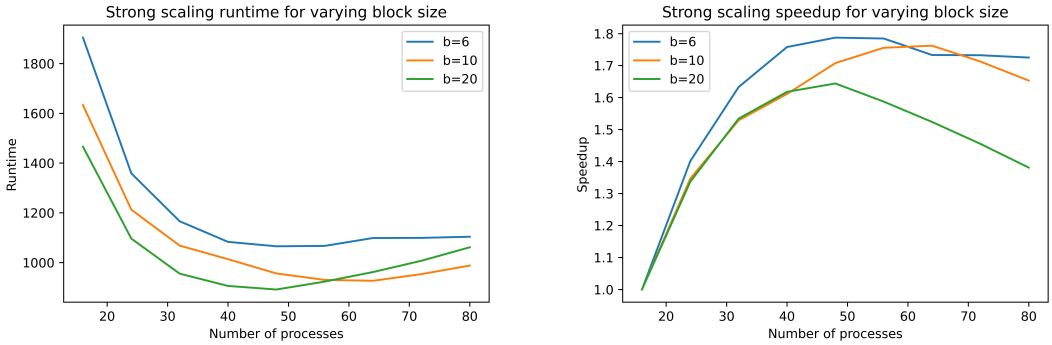


Figure 5.2: Strong scaling tests run on Lonsdale. Note here that b is the number of blocks per process, *not* the number of rows in the block.

It is clear from the plots that there is an optimal number of processes to use for a particular problem size. The benefit of the distribution of computational work must outweigh the overhead of performing the communication/synchronisation between processes. Above, for $b = n/20p$ (the best choice of block size tested above, with 20 blocks per process), the optimal number of processes is ≈ 45. After this, there is not enough work per process to justify the extra communication.

Also note the effect of block size on runtime and speedup. Choice of block size has a significant effect on the runtime of the algorithm. This will be investigated further in subsection 5.1.3. Optimal block size depends on both the problem size and the number of processes in use.

In the current implementation, the optimal block size and number of processes to use for a particular problem size must be approximated empirically. It is left as future work to design methods which allow the optimal values for these parameters to be calculated in advance.

5.1.2 Weak Scaling

The weak scaling of an algorithm denotes how the algorithm scales as the number of processes increases when the problem size per process is fixed. For an algorithm with perfect weak scaling, there is no increase in runtime as the number of processes (and thus global problem size) is increased. Again, this is usually not realistic for all but embarrassingly parallel problems. It requires there to be no extra communication, synchronisation or computation per process when increasing the overall problem size.

This is certainly not true for CALU, where extra processes

1. Increase the height of the binomial tree used in TSLU.

2. Increase the number of steps of the overall CALU recursion.

The runtime along the critical path of CALU is directly controlled by the problem size, since the number of active processes decreases linearly with time.

This is evident in the plot below, where the weak scaling is not anywhere close to perfect. A potentially more useful method of visualising weak scaling is through

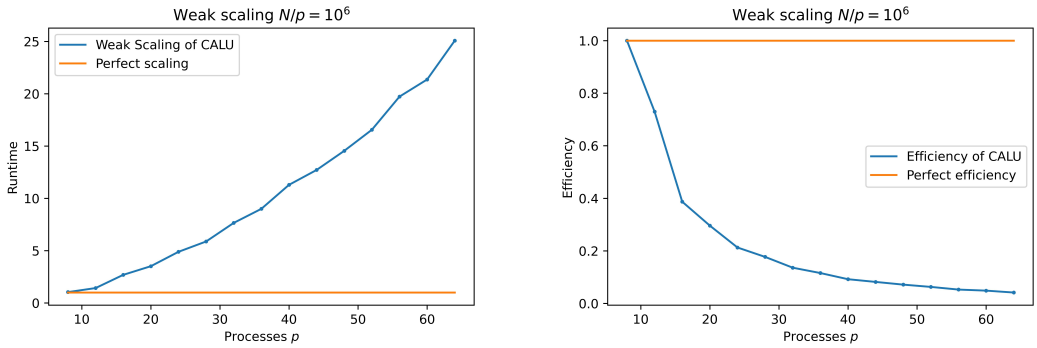


Figure 5.3: Weak Scaling runtime (left) and efficiency (right) of CALU with 1 million matrix entries per process. Tested on Lonsdale

measuring the efficiency

$$Efficiency = \tau(1)/\tau(p)$$

where p is the number of processes and $\tau(p)$ is the runtime for p processes (where again, problem size per process is constant). In this case, to avoid the unreliable timings for running the algorithm on one process, the weak scaling testing begins at 8 processes[1].

The efficiency drops rapidly with the number of processes. This is not surprising considering the mitigating factors mentioned above. Few algorithms dedicated to real life problems exhibit anywhere near 1.0 efficiency.

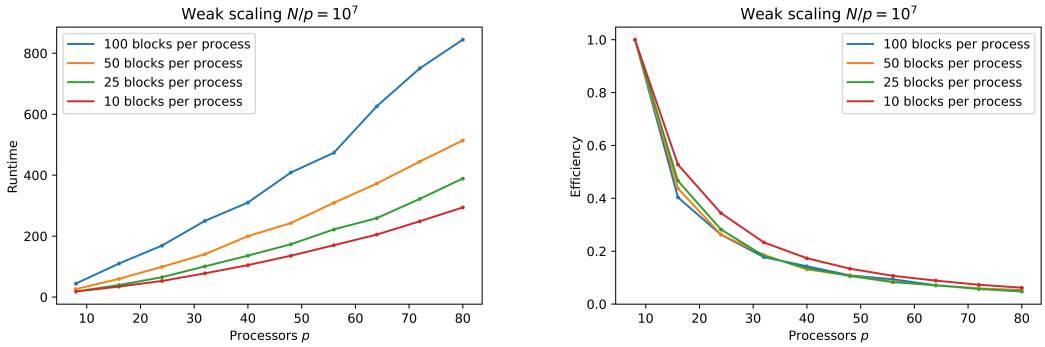


Figure 5.4: Effect of block size on weak scaling runtime (left) and efficiency (right) with 10 million matrix entries per process.

Again, it is observed that the choice of block size has a marked effect on runtime. However, it does not appear that the runtime has much of an effect on weak scaling, as shown by the efficiency chart.

5.1.3 Effect of Choice of Block Size on Runtime

As mentioned previously, choice of block size has a significant effect on the runtime of the CALU algorithm. However, the optimal choice is difficult to determine in advance. For this implementation, it is left up to the user to choose an optimal block size empirically. An analysis like that shown in Figure 5.5 can be used to determine the optimal block size for a particular problem size on a set number of processes (in this case 20 blocks per process being optimal).

From testing, it appears that this result is not matrix specific, with similar results for different randomly generated matrices (with $n = 50,000$ and $p = 32$ as in Figure 5.5). This empirical method will prove sufficient when the user wants to run

[1]Since efficiency is a relative measure, this still provides reliable efficiency results

CALU on many matrices of equal size on the same architecture. Determining an optimal block size in advance is left as further work and is discussed in chapter 6.

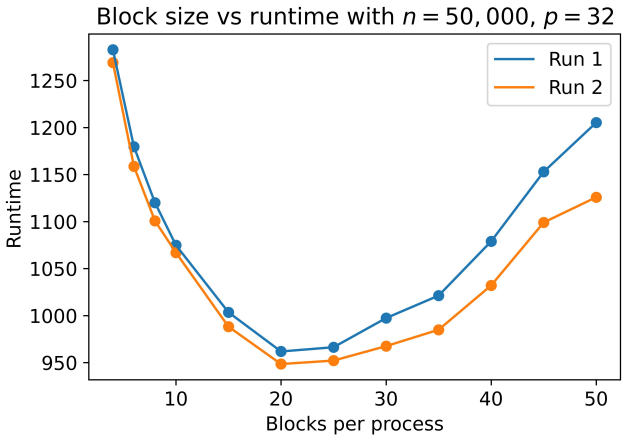


Figure 5.5: Effect of block size on runtime for fixed n and p. Different randomly generated matrices are used for each data point.

5.2 Analysis of Growth Factor in CALU

As previously discussed in section 2.3, the growth factor

$$g_W = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{i,j}|}$$

is an important measure for the stability of a linear algebra algorithm [1]. The growth factor is calculated first for a variety of problem sizes on chuck's 12 processors.

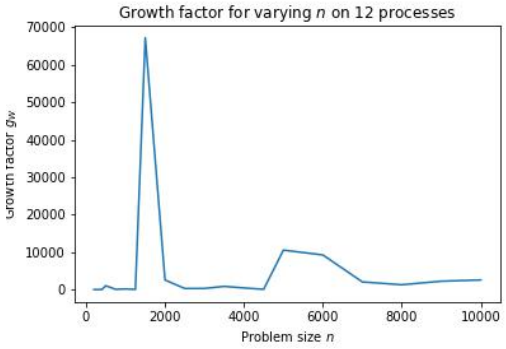


Figure 5.6: Growth factor for varying problem size on 12 processes

The results clearly do not come anywhere close to the theoretical upper bound of GEPP (2^{n-1}) or CALU (2^{nH}) and overall the growth factor is reasonably small. A growth factor of $\mathcal{O}(10^4)$ is unlikely to cause overflow or catastrophic loss of precision.

A more conclusive analysis is shown in Figure 5.7. It takes the average of ten runs on different randomly generated matrices for a varying numbers of processes and problem sizes. Note a problem size n implies it is being run on a $n \times n$ matrix.

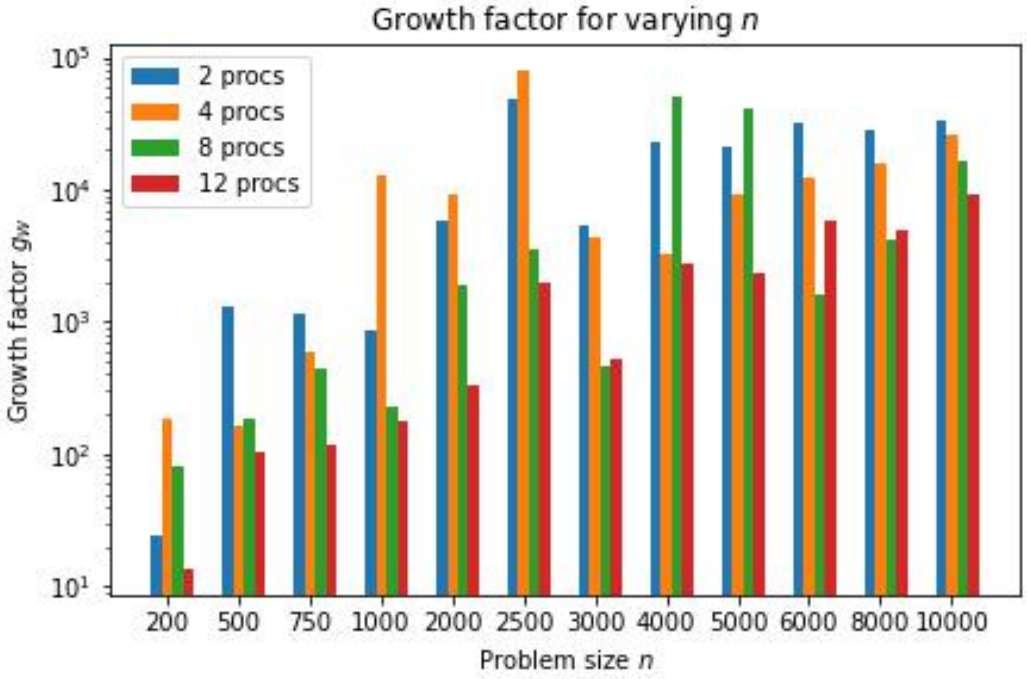


Figure 5.7: Average growth factor for varying problem size and number of processes. Average shown is taken from 10 runs on different random matrices.

The growth factor seems well behaved on all of these samples. Interestingly, the growth factor appears to be consistently lowest on 12 processes, despite the fact that the upper bound is greater for 12 processes ($H = 5 \implies g_W \leq 2^{5n}$). The reason for this is not quite clear, perhaps being due to there being less rows per process.

Regardless of the different patterns for different core counts, it is clear that the growth factor of CALU when run on random matrices is not concerning. Critically, growth factor does not seem to continually, dramatically increase with problem size. For $n = 10,000$ (100 million entries), the average growth factor is roughly the same order of magnitude, $\mathcal{O}(10^4)$, as for $n = 2500$ (6.25 million entries).

5.3 Error Analysis of CALU

The same measure of backward error will be used as in the paper on which this report is based[2], namely

$$e_{backward} = \frac{||A - PLU||_\infty}{||A||_\infty}$$

This relative error is tested across a number of different matrix sizes. Ideally, the size of the error will be near $\mathcal{O}(\epsilon_m)$, where ϵ_m is machine epsilon. However, in practice roundoff and propagated errors throughout the algorithm typically result in slightly higher errors.

The growth factor analysis in section 5.2 shows that as the size of the matrix A increases, so does the maximum size of the matrix elements observed throughout the algorithm. Since larger numbers are used in the intermediary steps of the algorithm, a loss of precision is expected. As seen in Figure 5.8, this holds true.

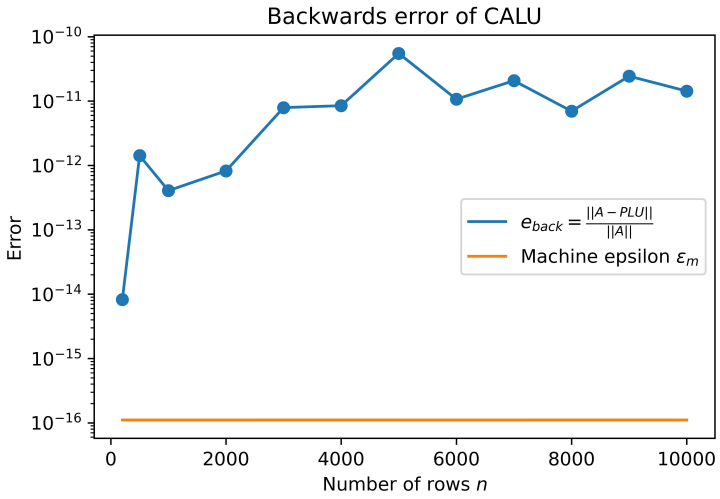


Figure 5.8: Backwards error analysis on 12 cores with 5 blocks per process

However, while the error is several orders of magnitude away from ϵ_m, it is still $\mathcal{O}(10^{-10})$ for a matrix with 100 million entries. Furthermore, the rate of increase does not indicate that this error will rise significantly for larger matrices.

[2]Note that in the original paper $||PA - LU||/||A||$ is used. However, this is entirely equivalent as the pivoting step involves no computation on the elements of A and incurs no error. Whether the pivots are applied to LU or to A is irrelevant, as long as the two matrices have the same row order.

Additional error analysis was run for a fixed matrix of size (5000×5000) on a varying number of processes. From the growth factor analysis, it appeared that the growth factor decreased with the number of processes. This is clearly not the case for the error, as shown by Figure 5.9.

So, in summary, the size of the error increases with problem size and minimally increases with number of processes. For all cases, the error stays relatively low ($> \mathcal{O}(10^{-9})$), and appears to scale reasonably well across both of these metrics.

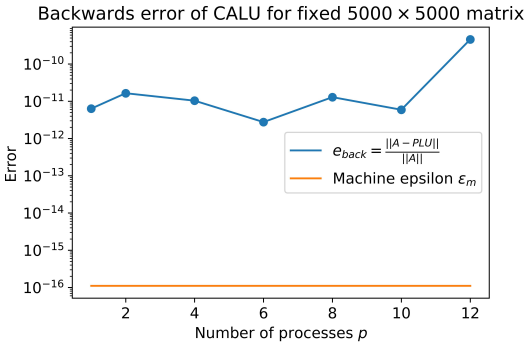


Figure 5.9: Error analysis of varying numbers of processes on a fixed 5000×5000 matrix

Chapter 6

Discussion

6.1 Further Work

Throughout the algorithm, a variety of MPI concepts (MPI Datatypes, MPI IO, collective calls, MPI communicators) have been used along with CBLAS functions to design an efficient and fully functioning implementation. However, although the algorithm provides an accurate, efficient parallel LU solver, there are some potential improvements that can be implemented. Time constraints prevented the implementation of the following changes, so they are listed here as future work.

6.1.1 2D and Beyond

Although the TSLU and tournament pivoting outlined in this implementation is fully inkeeping with that in "CALU: A Communication Optimal LU Factorisation Algorithm"[1], the implementation of CALU is not. This is due to the dimension of the process grid employed. In this report's implementation, a 1D process grid is used. While all of the matrix algebra remains the same, the necessary communication changes as shown in Figure 6.1. Time constraints prevented the adaptation of the 1D version to 2D. The following steps would have to be taken to adapt the algorithm:

1. Update the communicator to be a cartesian grid with `MPI_Cart_create` and adapt communicator redefinition to omit a row and column of processes each time a row of processes finish factorising.

2. Update communication patterns as shown in Figure 6.1.

3. Extend the start and end indices to the second dimension, giving each process its start and end indices for the x and y dimensions of the global array.

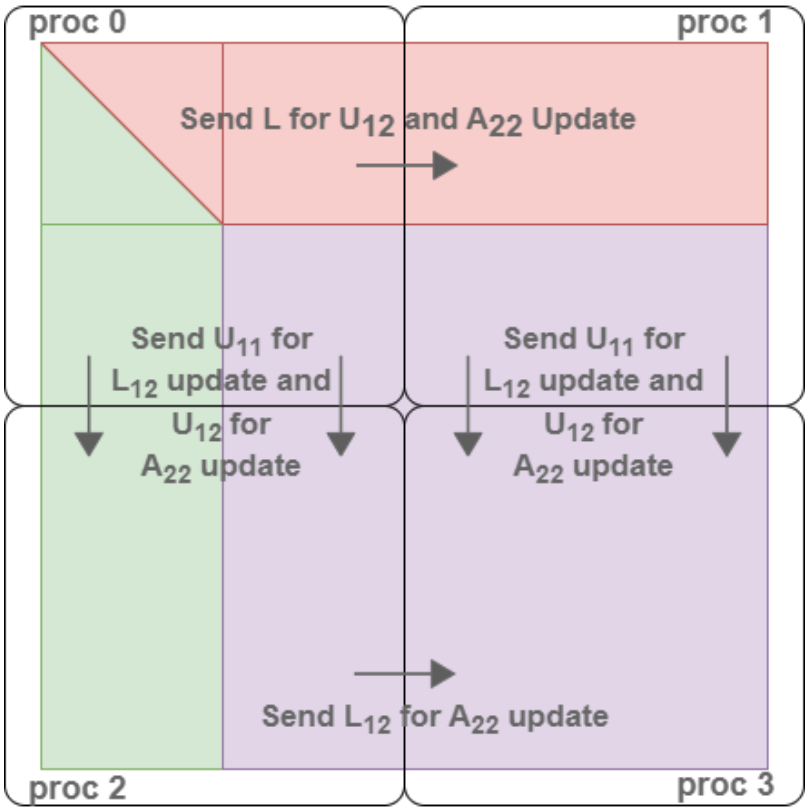


Figure 6.1: Example of communication pattern in 2D CALU on 4 processes. Note that this does not include tournament pivoting communication, as tournament pivoting is identical to the 1D case.

4. Update all recursion offsets to account for the above changes.

This 2D implementation has the benefit that the process grid more accurately reflects the shape of the recursion. Once a row of processes finish factorising, a row and column of processes can be terminated, allowing for only the unfactorised sections of the matrix to remain on active processes. 1D CALU has a the first `blockNumber*b` columns of its chunk already factorised.

6.1.2 Determining block size using `ILAENV`

Determining an optimal block size for CALU is an important factor in algorithmic performance, as seen in Figure 5.4. A further optimisation would be to test the use of the `ILAENV` LAPACK function to determine the optimal local block size for

the algorithm. This is a member function of the `dgetrf` function which tests local architecture to determine an optimal block size for serial LU.

Although it could be the case that an ideal serial block size is not ideal in parallel, it would provide an architecture specific method of determining block size which could increase performance.

6.2 Testing

Perhaps the most critical work required to expand on this project is the testing and profiling of this algorithm. Due to time constraints, as well as the lack of a profiling software compatible with Intel compilers (necessary for MKL) on chuck, the profiling of this algorithm was not investigated.

One area that could benefit from profiling is the communicator redefinition present inside the CALU recursion. It is clear that this function comes with some synchronisation overhead, but the level of overhead present is not clear. Only one redefinition is necessary per process. Therefore, it is assumed that the large amount of work per process on problems that require parallelism render this communicator redefinition irrelevant when n/p is somewhat large. However, this may not be the case. If so, a method of keeping track of which rank was root is required. Additionally, the relative and global ranks of each process must be recorded, and all collective communication calls avoided.

In addition to the profiling, for completeness the algorithm should be tested on special classes of matrices. Methods of calculating the growth factor and error have already been implemented, as well as parallel IO for reading in binary files. Thus, all that is required is a script to generate binary files of special matrix classes. These should be tested and their error/growth factor analysed.

Chapter 7

Conclusions

Throughout the course of this project, a 1d implementation of first Tall Skinny LU factorisation (TSLU), and subsequently Communication Avoiding LU factorisation (CALU), were successfully implemented. The final TSLU and CALU implementations were fully general, working for any $m \times n$ matrix distributed among any number of processes. By performing all local factorisations and block updates in place, as well as taking advantage of MPI IO to read in the matrices, the algorithms memory footprint was close to optimal. Not only independent of the global number of rows, it was limited to $\approx \frac{nm}{p} + mb$.

The algorithm takes advantage of the tournament pivoting scheme introduced by L.Grigori et. al.[1]. It proves stable and highly accurate for all randomly generated test cases. Weak and strong scaling tests show that the implementation provides significant speedup when run on an increasing number of processes.

Versatility was a critical concern in the design of this algorithm. A large proportion of the overall project time was spent ensuring maximum generality. Beginning with TSLU, the reduction tree was adapted to work for any number of processes. Load balancing of the matrix was made account for cases where $n \nmid b$.

Similarly, CALU was designed to work on rectangular $n \times m$ matrices, accounting for cases where $n, m \nmid b$. A major lesson taken from undertaking this project was the importance of edge cases, and the effort required to remove limitations on a function's usage.

The extensive detail of both the implementation section and documentation of the code (both as Doxygen and in file commenting) mean that this code base can be easily extended to a 2D algorithm.

Overall, this project provides a fast, stable and general parallel LU factoriser for general $n \times m$ matrices. Significant speedup is achieved with small growth factor and

error, making this implementation a highly viable algorithm for parallel architectures.

Bibliography

[1] Grigori, L., Demmel, J.W. and Xiang, H. (2011). CALU: A Communication Optimal LU Factorization Algorithm. SIAM Journal on Matrix Analysis and Applications, 32(4), pp.1317–1350.

[2] Chu, Eleanor, and Alan George. "Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor." Parallel Computing, vol. 5, no. 1-2, July 1987, pp. 65–74,

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, 1999.

[4] L. N. Trefethen and R. S. Schreiber, Average-case stability of Gaussian elimination, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 335–360