Trinity College Dublin

High Performance Computing

Master of Science

Communication-Avoiding Tall-Skinny LU Factorisation

William O'Sullivan

15. September, 2021

Acknowledgements

Special thanks to my supervisor, Kirk Soodhalter, for patience and guidance, and for many good conversations along the way. Your sunny perspective genuinely helped me to keep my spirits up throughout the whole project.

I'd like to thank Matt Boyer for his intellectual and emotional support. Whenever I struggled to manage professional and academic work, you'd always manage to help me resolve whatever problem was at hand.

I'd like to thank my parents, John and Yvonne, for their steady support. From good advice to much needed hugs, knowing that I could always rely on you pushed me to work all the harder to make you both proud.

And finally, I'd like to thank my close friends, who reminded me of the fact that there was more to life than work, and who kept me in good company throughout what was a sincerely difficult year. Here's to many more late nights, long conversations, and good laughs.

Abstract

Communication-Avoiding (CA) LU is a method that aims to optimise the communication of parallelised LU factorisation. The literature describes a successful implementation whereby numerical accuracy and stability are maintained when compared with existing non-communication-avoiding methods. This project would aim to develop a Tall-Skinny LU factorisation utilising the pivoting technique known as tournament pivoting. Having successfully developed a new variant of the algorithm dubbed TSLU 2021, it was shown that the TSLU implementation successfully computed the LU factorisation of both sparse and dense matrices with a growth factor of no greater than 1.0 for panels of size 4096×256 and 16384×2048. Whilst the tournament pivoting operation demonstrated scaling consistent with expectations for a binary reduction tree, the implementation was hindered by a runtime bottleneck due to a Gaussian elimination step at the end of the process. As a result, whilst the algorithm is shown to be well behaved and stable in the multi-core regime, running on multiple cores does not lead to a desirable decrease in runtime.

A link to the public repository containing the production build of TSLU 2021 can be found here:

TSCALU GitHub Repository

Contents

List of Algorithms

Listings

List of Tables

1 INTRODUCTION

1.1 Communication Minimisation

Algorithms can be described in terms of two costs - an arithmetic cost and a communication cost. Arithmetic cost refers to the required number of floating point operations required to achieve a desired solution, whilst communication cost can have one of two meanings. In a serial case this communication refers to the movement of data between different levels of the memory hierarchy, whilst in the parallel case this communication refers to the movement of data between different processors on a network.

The communication cost can be further split into two components - bandwidth and latency. Bandwidth describes the total number of words of data moved, and latency is defined by the number of messages in which the words are organised and sent. It is then possible to model the cost of sending w words as $\alpha \cdot \frac{1}{\beta} w$, where α represents the latency measured in seconds and β represents the bandwidth in words per second [1].

The arithmetic cost of an algorithm is frequently overshadowed by its communication cost, and so there is an impetus to either reduce both the latency and bandwidth or to shift a greater degree of the computation onto the arithmetic elements of the algorithm. Here we see the utility of communication-avoiding algorithms - in exchange for a greater volume of arithmetic, we perform fewer time-consuming communications which can result in an overall speed-up in the algorithm.

1.2 Approaches to LU Factorisation

It is important to understand that different strategies for LU factorisation result in the formation of different L and U matrices. This means that different strategies cannot be compared on the basis of their unique L and U matrices, but on whether they are able to accurately regenerate the matrix from which they are factored. Several matrices, known as pathological matrices, cannot be factorised in a stable manner, such as the Wilkinson matrix [2], the Foster [3] matrix and the Wright matrix [4].

1.2.1 Gaussian Elimination with Partial Pivoting

Gaussian elimination with partial pivoting (GEPP) is a very stable algorithm for performing LU factorisation. As GEPP passes through the columns of the matrix being factorised, it permutes the row containing the largest element in that column to the diagonal of the matrix. These permutations are applied to L and U, and depending on the implementation also a permutation matrix P or Π. After this partial pivoting is complete, the Gaussian elimination takes place, updating the entries of L and U as required. This algorithm has efficient implementations in both serial and parallel with the DGETRF from LAPACK [5] which employs a block GEPP factorisation and recursive GEPP [6] for the serial case, and ScaLAPACK's PDGETRF routine, which avails of a block cyclic layout [7].

1.2.2 CALU

Implementations of communication-avoiding LU factorisation aim to reduce the runtime of LU factorisation on large matrices by shifting the balance of the algorithmic cost from the communication contribution to the arithmetic contribution. In particular, block pivoting methods have attained communication optimality in both serial and parallel.

The key idea behind CALU as it was first presented in the 2011 paper by Grigori et al [8] was to find pivots in a way that minimised communication. The result of this investigation was the development of "tournament pivoting" (which had previously gone by the name "ca-pivoting"). Tournament pivoting was a method by which each set of block rows in a panel could put forth their most suitable pivots, which would then engage in a competition to evaluate which pivots were best for a given panel. With this information, it then became possible to permute the entire panel without having to look down the entire column for each pivot row, therefore saving on communication. After these pivots had been applied, the panel would then be suitable for the use of Gaussian elimination without the need for partial pivoting, as the tournament pivoting had already satisfied that pre-conditioning requirement. In this two-step manner, the LU factorisation could be computed.

1.2.3 LU_PRRP

LU_PRRP and CALU_PRRP were initially presented in the 2012 paper by Khabou et al [9]. These two methods are more stable than both GEPP and CALU respectively, with a greater resistance to so-called pathological cases whereby GEPP and CALU fail to successfully perform the LU factorisation.

LU_PRRP uses a block algorithm, whereby the LU factorisation of a panel is performed, and then used to update the trailing matrix after which the next LU factorisation can take place. Each panel is initially factored by transposing the panel, and then calculating its rank revealing QR (RRQR) factorisation [10]. The permutations recovered were then applied to the entries of the input matrix, whilst the L matrix is derived from the R factor of the RRQR factorisation, after which the trailing matrix was updated.

CALU_PRRP is the communication avoiding variant of LU_PRRP based on the tournament pivoting technique pioneered in the development of CALU. At each stage of the tournament, each processor utilises RRQR to determine the best pivot rows, before passing this information to the processors in the next round of of the tournament until a final set of pivots are generated. Once these pivots have been obtained, the panel is permuted accordingly, and the QR decomposition of the transpose of the now permuted panel takes place without pivoting.

2 THEORY

2.1 GEPP

Outside of a few specific cases, GEPP is by and large a simple and effective method for performing LU factorisation. The GEPP algorithm described in Algorithm 1 [11] details the LU factorisation of square matrices.

Algorithm 1: Gaussian Elimination with Partial Pivoting

Require: Dimension m;

$U = A, \quad L = I, \quad P = I$;

for $k = 1 \rightarrow m - 1$ **do**

  Select $i \geq k$ to maximize $|u_{ik}|$;

  $u_{k,k:m} \longleftrightarrow u_{i,k:m}$ (interchange two rows);

  $l_{k,1:k-1} \longleftrightarrow l_{i,1:k-1}$;

  $p_{k,:} \longleftrightarrow p_{i,:}$;

  for $j = k + 1 \rightarrow m$ **do**

    $l_{jk} = u_{jk}/u_{kk}$;

    $u_{j,k:m} = u_{j,k:m} - l_{jk}u_{k,k:m}$;

  end

end

This algorithm would later be adapted into a rectangular version of GEPP for the purposes of this project.

2.2 CALU

CALU is a communication avoiding LU factorisation algorithm based on the tournament pivoting strategy (previously referred to as ca-pivoting) [8]. This algorithm performs an optimal amount of communication with the added benefit (compared to previous techniques) of being stable in practice [12].

In this section we will examine the operation of this algorithm.

2.2.1 Brief Overview

At the first iteration, the $m \times n$ matrix A is partitioned in the following way:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

Where the submatrices possess the following dimensions:

- A_{11} is of size $b \times b$
- A_{12} is of size $(m - b) \times b$
- A_{21} is of size $b \times (n - b)$
- A_{22} is of size $(m - b) \times (n - b)$

Once the partition of A has been completed, the LU factorisation of the first panel (the block-column formed by A_{11} and A_{21}) is computed. This allows us to form the block-upper submatrix U_{12}, before the trailing matrix A_{22} is updated to reflect the permutations that occurred as a result of the panel factorisation.

2.2.2 Tournament Pivoting

The panel undergoing LU factorisation itself can be deemed a tall and skinny matrix, whereby the dimensions of the panel $m \times b$ are such that $m >> b$. Accordingly, the factorisation can be referred to as TSLU factorisation.

TSLU uses tournament pivoting in order to compute the LU factorisation of the panel. Tournament pivoting is the key ingredient for distinguishing CALU from other kinds of block algorithms such as block parallel pivoting.

TSLU is performed in two steps. In the first step, good pivots are determined by tournament pivoting at a low communication cost and used to permute the rows in the panel into a desirable configuration. In the second step, Gaussian elimination is then employed on the "unpivoted" panel in order to produce the LU factorisation.

Taking a panel W, we can explore the operation of tournament pivoting in terms of explicit matrix formation - of course, we never perform this explicit matrix formation in practice,

instead opting for implicit methods in order to avoid unnecessary additional computation and to conserve memory.

First, the panel W is decomposed into a number of block-rows, N; in this instance we are splitting the panel into $N = 4$ block-rows - for convenience sake we can specify that m is divisible by N.

$$W = \begin{bmatrix} W_{00} \\ \hline W_{10} \\ \hline W_{20} \\ \hline W_{30} \end{bmatrix}$$

For each block-row we perform GEPP:

$$W = \begin{bmatrix} W_{00} \\ \hline W_{10} \\ \hline W_{20} \\ \hline W_{30} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{00}\bar{L}_{00}\bar{U}_{00} \\ \hline \bar{\Pi}_{10}\bar{L}_{10}\bar{U}_{10} \\ \hline \bar{\Pi}_{20}\bar{L}_{20}\bar{U}_{20} \\ \hline \bar{\Pi}_{30}\bar{L}_{30}\bar{U}_{30} \end{bmatrix}$$

We can expand this factorisation to collect the terms of $\bar{\Pi}_{i0}, \bar{L}_{i0}$ and $\bar{U}_{i0}$ in terms of $\bar{\Pi}_0\bar{L}_0$ and $\bar{U}_0$:

$$\begin{bmatrix} \bar{\Pi}_{00}\bar{L}_{00}\bar{U}_{00} \\ \hline \bar{\Pi}_{10}\bar{L}_{10}\bar{U}_{10} \\ \hline \bar{\Pi}_{20}\bar{L}_{20}\bar{U}_{20} \\ \hline \bar{\Pi}_{30}\bar{L}_{30}\bar{U}_{30} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{00} & & & \\ & \bar{\Pi}_{10} & & \\ & & \bar{\Pi}_{20} & \\ & & & \bar{\Pi}_{30} \end{bmatrix} \cdot \begin{bmatrix} \bar{L}_{00} & & & \\ & \bar{L}_{10} & & \\ & & \bar{L}_{20} & \\ & & & \bar{L}_{30} \end{bmatrix} \cdot \begin{bmatrix} \bar{U}_{00} \\ \bar{U}_{10} \\ \bar{U}_{20} \\ \bar{U}_{30} \end{bmatrix} = \bar{\Pi}_0\bar{L}_0\bar{U}_0$$

As all $\bar{\Pi}_{i0}$ are of dimension $m/N \times m/N$, $\bar{\Pi}_0$ itself is of dimension $m \times m$. The elements $\bar{L}_{i0}$ are of dimension $m/N \times b$ and therefore $\bar{L}_0$ is of dimension $m \times b$ whilst all $\bar{U}_{i0}$ are of dimension $b \times b$, meaning $\bar{U}_0$ takes on a dimension of $N \cdot b \times b$.

At the same time, it is then possible to take the top b rows of the product of $\bar{\Pi}_{i0}\bar{L}_{i0}\bar{U}_{i0}$ to form W_{i1}:

$$\begin{bmatrix} \bar{\Pi}_{00}\bar{L}_{00}\bar{U}_{00} \\ \hline \bar{\Pi}_{10}\bar{L}_{10}\bar{U}_{10} \\ \hline \bar{\Pi}_{20}\bar{L}_{20}\bar{U}_{20} \\ \hline \bar{\Pi}_{30}\bar{L}_{30}\bar{U}_{30} \end{bmatrix} \rightarrow \begin{bmatrix} W_{01} \\ \hline W_{11} \\ \hline W_{21} \\ \hline W_{31} \end{bmatrix}$$

Whereby the dimensions of all W_{i1} are then $b \times b$. The creation of these new W_{i1} is the result of the most suitable pivots "winning" their round of the tournament, and therefore passing on to another round of tournament pivoting. From this point, we employ a binary tree to perform the subsequent rounds of the tournament. The block-rows are stacked on top of one another, and the rows present are then pivoted and the winners collected as before:

$$\begin{bmatrix} W_{01} \\ W_{11} \\ \hline W_{21} \\ W_{31} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{01}\bar{L}_{01}\bar{U}_{01} \\ \hline \bar{\Pi}_{11}\bar{L}_{11}\bar{U}_{11} \end{bmatrix} \rightarrow \begin{bmatrix} W_{02} \\ \hline W_{12} \end{bmatrix}$$

Once again, the matrices can be expanded and multiplied together to produce the next level of $\bar{\Pi}_i, \bar{L}_i$ and $\bar{U}_i$:

$$\begin{bmatrix} \bar{\Pi}_{01}\bar{L}_{01}\bar{U}_{01} \\ \hline \bar{\Pi}_{11}\bar{L}_{11}\bar{U}_{11} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{01} & \\ & \bar{\Pi}_{11} \end{bmatrix} \cdot \begin{bmatrix} \bar{L}_{01} & \\ & \bar{L}_{11} \end{bmatrix} \cdot \begin{bmatrix} \bar{U}_{01} \\ \bar{U}_{11} \end{bmatrix} = \bar{\Pi}_1\bar{L}_1\bar{U}_1$$

The final round of the tournament (in this scenario) is then performed on the stacked winners W_{02} and W_{12}, immediately giving the final level of $\bar{\Pi}_i, \bar{L}_i$ and $\bar{U}_i$:

$$\begin{bmatrix} W_{02} \\ W_{12} \end{bmatrix} = \begin{bmatrix} \bar{\Pi}_{02}\bar{L}_{02}\bar{U}_{02} \end{bmatrix} \rightarrow \bar{\Pi}_2\bar{L}_2\bar{U}_2$$

At this point the tournament has been completed, with the various $\bar{\Pi}_i$ tracking the permutation of rows throughout the matrix. These $\bar{\Pi}_i$ can then be used to permute the original panel W using the following multiplication:

$$\bar{\Pi}_2^T\bar{\Pi}_1^T\bar{\Pi}_0^T W$$

It is important to understand that whilst the dimensions of the block matrices within $\bar{\Pi}_0$ are still of dimension $m/N \times m/N$, the block matrices within $\bar{\Pi}_1$ and $\bar{\Pi}_2$ have dimensions of

$2 \cdot b \times 2 \cdot b$. As such, $\bar{\Pi}_1$ and $\bar{\Pi}_2$ must be extended by identity matrices of the appropriate dimension:

$$\begin{bmatrix} \bar{\Pi}_{01} & \\ & \bar{\Pi}_{11} \end{bmatrix} \rightarrow \begin{bmatrix} \bar{\Pi}_{01} & & & \\ & I & & \\ & & \bar{\Pi}_{11} & \\ & & & I \end{bmatrix} \quad \& \quad \begin{bmatrix} \bar{\Pi}_{02} \end{bmatrix} \rightarrow \begin{bmatrix} \bar{\Pi}_{02} & & \\ & & \\ & I & \end{bmatrix}$$

Once W has been pivoted, it can then undergo step two: the Gaussian elimination to form the desired LU factorisation. Upon the completion of this elimination, TSLU has successfully factorised the panel W.

2.2.3 TSLU Structure

The implementation developed as the main focus of this dissertation differed from the implementation described by "Communication Avoiding Gaussian Elimination" [8], and instead more closely resembled the later implementation in "CALU: A communication optimal LU factorization algorithm" [12].

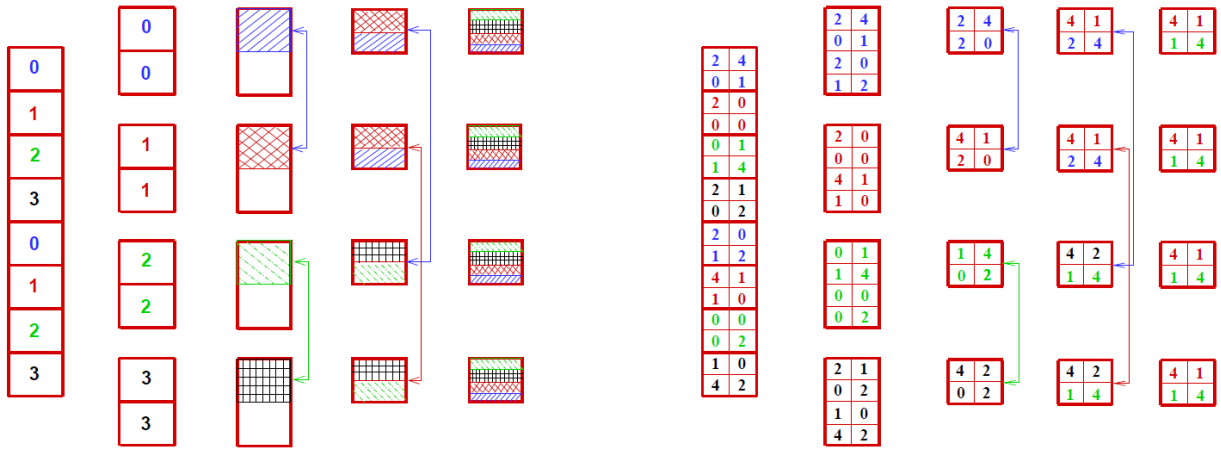

Figure 1: Decomposition of panel W according to the initial implementation of TSLU and the "butterfly" communication pattern. Source:[8]

The first implementation differed in both panel decomposition as well as communication framework. Figure 1 shows a panel W being decomposed into four block rows W_1 through W_4. In this decomposition, the block rows are composed of two sets of further smaller block

rows of size $n \times n$ where one of each is sourced from either the top or bottom half of the panel.

After the decomposition was performed the tournament proceeded. The best b rows were communicated to one-another in such a way that both processors are able to perform the pivoting to determine the winners of the next round. In this butterfly pattern each processor then communicates its winning pivots to the other processor. After this final round of pivoting for the N scenario, each processor then contains the ideal pivots as determined by tournament pivoting.

The revised parallel implementation instead opts for a parallel binary tree for its communication instead of using the butterfly implementation. Additionally, it does not explicitly describe the construction of the block rows W_1 through W_4. Figure 2 demonstrates how the winners of a tournament are communicated to a neighbour whereby another round of the tournament takes place, until the winners are located on the root or process for the final round of the tournament. This is quite a significant difference from the butterfly communication pattern, as now only a single processor will hold the pivots, but the communication of tournament winners has been minimised even further.

As previously mentioned, the implementation created for this dissertation utilised the binary tree method as shown in Figure 2 with one important distinction. The original binary tree only communicated the top $b = n$ rows, as would be necessary for ensuring that the matrices of winners would always be square before they were combined for the next round of the tournament. The new binary tree was more flexible in that it could communicate the any number of top rows. However, this increased flexibility did create the opportunity to make additional decisions about how many "winners" could be in a given round.

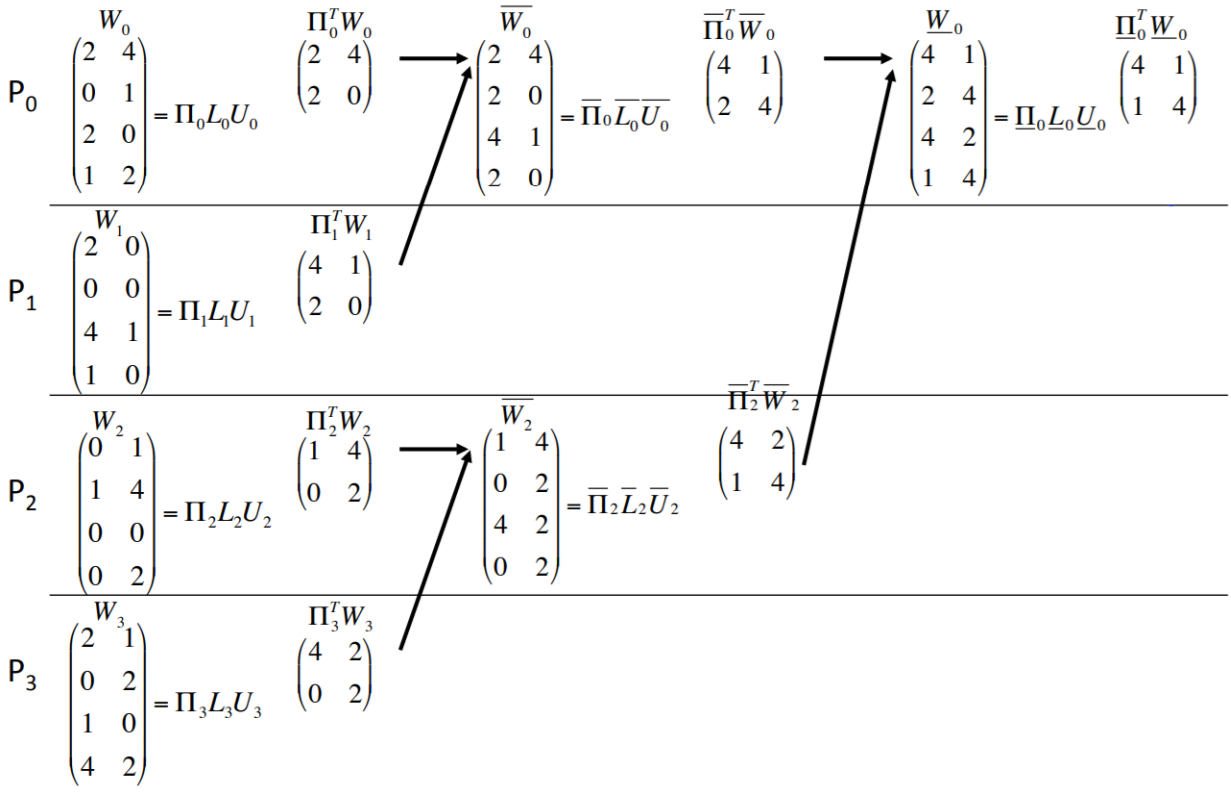

Figure 2: Tournament pivoting as described using a binary tree. In contrast to the butterfly communication pattern, the binary reduction tree only stores the pivots on the root processor rather than having all processors evaluate the pivots. Source: [13]

2.2.4 TSLU Algorithm

Algorithm 2 is the formalisation of the TSLU implementation as presented in Grigori et. al [12]. It specified that the input panel matrix A was distributed using a 1-D block row layout, with each processor containing its own subset of block rows. The algorithm then described the general layout of tournament pivoting across an all-reduction tree, and the required extension of the permutation matrices.

Algorithm 2: TSLU 2011

Require: S is the set of P processors, $i \in S$ is my processor's index;

Require: All-reduction tree with a height H;

Require: The $m \times b$ input matrix $A(:, 1:b)$ is distributed using a 1-D block row layout; $A_{i,0}$ is the block of rows belonging to my processor i;

Compute $\Pi_{i,0}A_{i,0} = L_{i,0}U_{i,0}$ using GEPP;

for k *from* 1 *to* H **do**

  if *I have any neighbors in the all-reduction tree at this level* **then**

    Let q be the number of neighbors;

    Send $\Pi_{i,k-1}A_{i,k-1}(1:b, 1:b)$ to each neighbour j;

    Receive $\Pi_{j,k-1}A_{j,k-1}(1:b, 1:b)$ from each neighbour j;

    Form the matrix $A_{i,k}$ of size $qb \times b$ by stacking the matrices $\Pi_{j,k-1}A_{j,k-1}(1:b, 1:b)$ from all neighbours.;

    Compute $\Pi_{i,k}A_{i,k} = L_{i,k}U_{i,k}$ using GEPP;

  else

    $A_{i,k} := \Pi_{i,k-1}A_{i,k-1}$;

    $\Pi_{i,k} := I_{b \times b}$;

  end

end

Compute the final permutation $\bar{\Pi} = \bar{\Pi_H}; ...; \bar{\Pi_1}\bar{\Pi_0}$ where $\bar{\Pi_i}$ represents the permutation matrix corresponding to each level in the reduction tree, formed by the permutation matrices of the nodes at this level extended by appropriate identity matrices to the dimension $m \times m$;

Compute the Gaussian elimination with no pivoting of $(\bar{\Pi}A)(:, 1:b) = LU$;

Ensure: $U_{i,H}$ is the U factor obtained at the final step for all processors $i \in S$;

2.3 Conditioning, Stability and Growth Factor

In terms of examining the quality of a solution, floating point arithmetic differs significantly from analytical arithmetic. In analytical arithmetic precision can genuinely be lost - not just by the truncation of irrational numbers, but also by the fact that the concept of an infinite number of values residing between any two numbers simply does not translate.

These are not the only concepts that differ between the two kinds of arithmetic, but they serve as an important starting point for describing the difficulties faced when trying to replicate analytical solutions using floating point arithmetic. It is a very real example of the distinction between how we translate somewhat abstract mathematical concepts into the foundations of solving real-world problems.

2.3.1 Conditioning

The conditioning of a problem describes its sensitivity to errors - how either perturbations in the input, or the way in which inputs are combined result in a solution which itself contains errors. More specifically, problems can be defined as well-conditioned or ill-conditioned depending on how perturbations in the input affect the output.

A well-conditioned problem is one where a small perturbation in the input x results in a small perturbation in the evaluation of $f(x)$, and an ill-conditioned problem is one where a small perturbation in the input x results in a large perturbation in the evaluation of $f(x)$. This concept can be extended to multiple variables, whereby a well-conditioned problem can accept multiple perturbed inputs x, y, z and return a small perturbation in the evaluation of $f(x, y, z)$. However, if one the perturbation of one of those variables results in the a large perturbation in the evaluation of $f(x, y, z)$ then despite the fact that it is well-conditioned for two of the three variables, the problem is then described as ill-conditioned. This is a part of the reason why ill-conditioned problems are more common in real-world applications; the real world more often than not requires complex representations in order to describe the system in its entirety. In doing so, the representation then opens itself to more opportunities for ill-conditioning to arise. However, the issue of a single variable resulting in poor conditioning does not grasp the scope of the problem either. Returning to the problem $f(x, y, z)$, it is also possible for all of the variables to individually pose no issue for conditioning; rather it can

become a problem when these perturbed inputs interact with one another to create other kinds of errors that have the knock-on effect of resulting in a highly perturbed evaluation of $f(x, y, z)$.

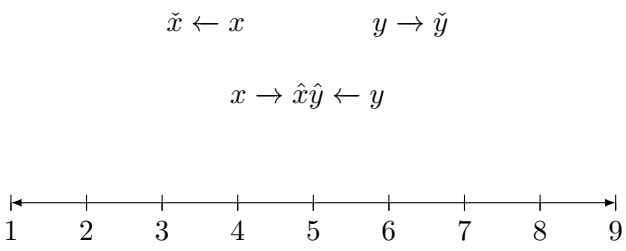

Figure 3: The function $f(x, y, z)$ can be more reliably evaluated for one set of permuted inputs $f(\check{x}, \check{y}, \check{z})$, than for another per set of permuted inputs $f(\hat{x}, \hat{y}, \hat{z})$, demonstrating how problems which are ill-conditioned in certain circumstances can them impart circumstantial ill-conditioning in problems they are built upon.

For example, the evaluation of $f(x, y, z)$ may require the invocation of a subtraction operation between x and y. Subtraction of the form $(x - y)$ is inherently ill-conditioned for $x \rightarrow y$ due to cancellation error. For the set of perturbations $f(\check{x}, \check{y}, \check{z})$ it could be possible for the difference between $\check{x}$ and $\check{y}$ to not bring about catastrophic cancellation. However, another set of perturbations $f(\hat{x}, \hat{y}, \hat{z})$ may bring about catastrophic cancellation when performing the subtraction of $\hat{x}$ from $\hat{y}$. Whilst this is only a toy example, the important takeaway is that ill-conditioning in one part of a problem can affect the conditioning of the problem overall.

It is this notion of inheritable ill-conditioning that brings us to the importance of pivoting in LU factorisation. Matrices can be described by their own condition number. The condition number k, of a matrix A is given by the upper bound of $k \leq ||A||\ ||A^{-1}||$. This quantity in particular describes the max condition number of the problem where A is multiplied by a vector x, but this problem is so common in linear algebra that its use as the basis of conditioning for all matrices is more than suitable.

It is then possible to resolve the ill-conditioning of a matrix by means of a pre-conditioning step. This is an operation whereby some modifications are made to the matrix to impart better conditioning. Of particular relevance in this work is the idea of multiplying one matrix by another, such that the conditioning of the product is improved when compared to the conditioning of the original matrix by itself. In LU factorisation, this is characterised by the formation of pivots - in forming the permutation matrix P and multiplying it by the original matrix A, the product PA will have better conditioning than the original matrix A. Though

the improvement in conditioning itself might not appear to have function immediately, it is when we factorise the matrix that the use of a matrix with better conditioning results in increased stability in the method.

2.3.2 Stability

The stability of an algorithm is a measure of its sensitivity to perturbations. This means that as different algorithms go about solving the same problem in different ways, some algorithms may be more or less stable than their counterparts - a factor that must be accounted for when deciding on the algorithms one wishes to employ. This is then further compounded by the real-world challenges innate to numerical methods, namely a loss in precision. "Machine epsilon" often written ϵ_m is the smallest number which you can add to unity and still create a new number. Whilst this value can vary across different architectures, ϵ_m would commonly be of the order $2 \cdot 10^{-16}$.

ϵ_m then provides us with a lower limit for determining the error of our solutions, by way of quantifying the smallest possible relative error $(r.e)$. If the relative error of a solution is approximately ϵ_m then this is as close to a completely correct solution as one can get. Hence, we deem any method whereby $r.e \approx \epsilon_m$ to be an "accurate method". In practice, this ideally accurate method is infrequently achieved. As most real-world problems are ill-conditioned, algorithms that can act on ill-conditioned problems are of vital importance. This means that propagation error is likely, driving the relative error away from ϵ_m. This issue arises before even touching on the notion of the algorithm taking perturbed inputs. These obstacles will prevent the method from achieving the gold-standard of $r.e \approx \epsilon_m$.

There are different kinds of stability which satisfy different criteria. Forward and backward stability are two ways to describe the behaviour of an algorithm.

Forward stability is achieved if when the perturbation in the inputs is of the order ϵ_m and relative error of the output is also of the order ϵ_m. Whilst this may seem ideal at first glance it fails to capture the specific behaviour of the problem itself. It is giving a perturbed answer to a perturbed problem. Whilst in isolation this may be a perfectly valid requirement to meet when generating a set of independent answers, a more rigorous kind of stability is required to capture the behaviour of the problem itself.

This brings us to backwards stability. Backwards stability is achieved when application of the algorithm to a non-perturbed input gives an accurate output as if the problem itself (not the algorithm, but the problem!) had actually taken a perturbed input, whereby that relative error of that perturbed input was of the order ϵ_m. It is important to note two things about backwards stability. Firstly, algorithms possessing this stability capture the behaviour of the problem itself, not just an approximation of the behaviour of the problem. This allows for the construction of systems of answers that accurately represent the behaviour of the problem as if the inputs to that problem were the ones being perturbed rather than the inputs to the algorithm. In doing so, a backwards stable algorithm generates an exact solution to a perturbed problem. Secondly, the presence of backwards stability implies forward stability, such that if the inputs were actually perturbed themselves, the outputs would still also only have a relative error of the order ϵ_m.

Stability can be described using different metrics in different circumstances, but in LU factorisation measuring the growth factor is a simple way to quantify the propagation of error in a system. The growth factor g is given by:

$$g = \frac{||LU||_\infty}{||PA||_\infty}$$

The lowest bound on this growth factor is unity, indicating that the products of LU factorisation of the permuted matrix can readily be used to reconstruct that permuted matrix without error greater than ϵ_m arising.

In performing the LU factorisation, the conditioning of a matrix directly affects the stability of the solution - if the LU factorisation were to be performed on the matrix A, which has not undergone any preconditioning, then depending on the nature of the matrix A, the stability may not be affected at all or the growth factor could explode out to infinity, indicating a complete loss of stability. In LU factorisation, this loss in stability could most frequently be attributed to the subtraction or division by numbers of greatly differing orders of magnitude.

3 COMPUTATIONAL METHODS

3.1 Special Matrix Generation

Not all matrices are equally well suited to LU factorisation due to the issue of conditioning, and so special matrices with varying condition number k were generated. The following matrices were created using `Matlab`'s test matrix gallery:

Circulant Matrix A 4096×4096 circulant matrix was generated. It is a special case of the Toeplitz matrix - the first row of a circulant matrix is a vector of length n with each entry populated according to that entry's index. All rows afterwards had their entries shifted to the right to create a repeating pattern as shown in Figure 4.

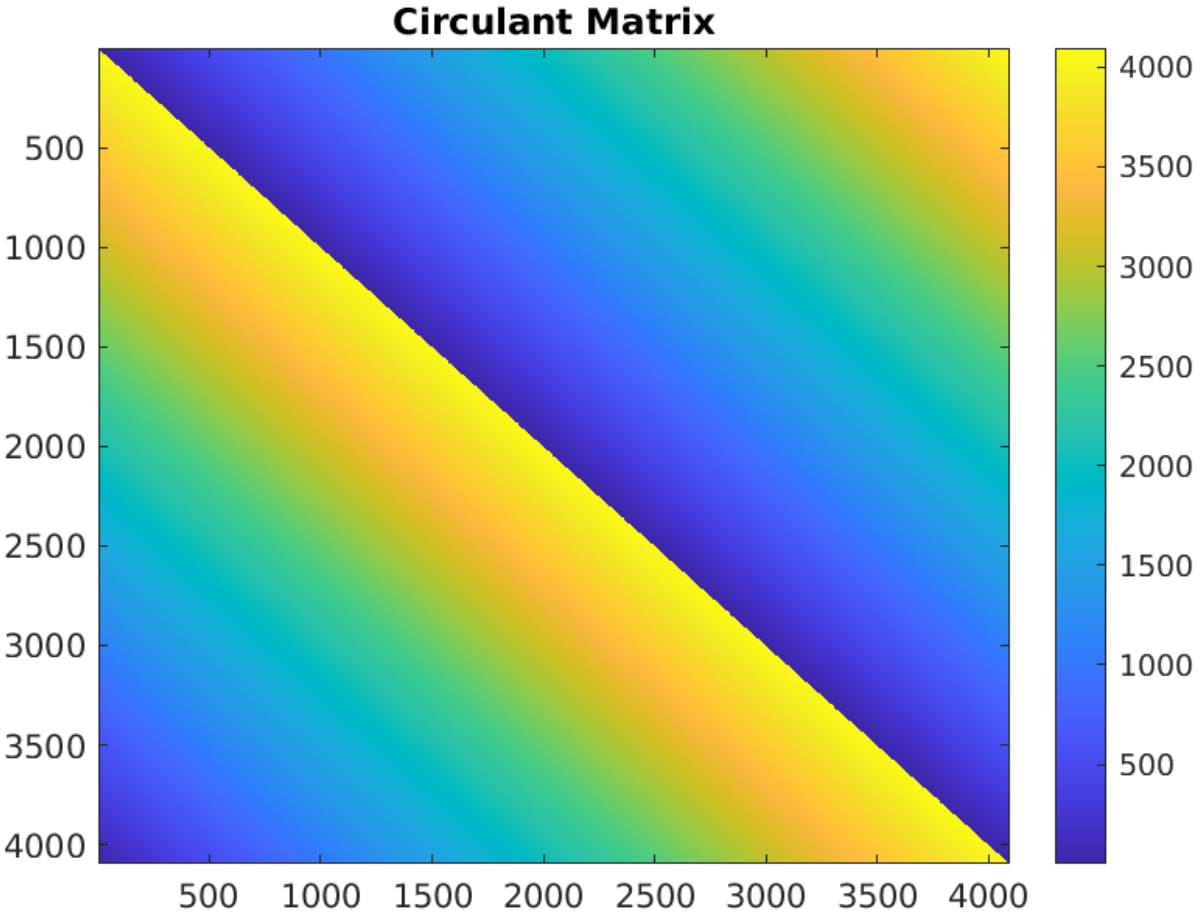

Figure 4: A circulant matrix possesses rows which are a right shift of the row above it, giving it this distinct form. With a value of $k = 4.097010^{03}$, the conditioning is away from perfect - recall that an ideal condition number is unity.

Diagonally Dominant Matrix A 4096×4096 diagonally dominant matrix was generated. This is an ill-conditioned, sparse matrix. Figure 5 shows the tri-diagonal matrix, with great disparity between entries along the diagonal.

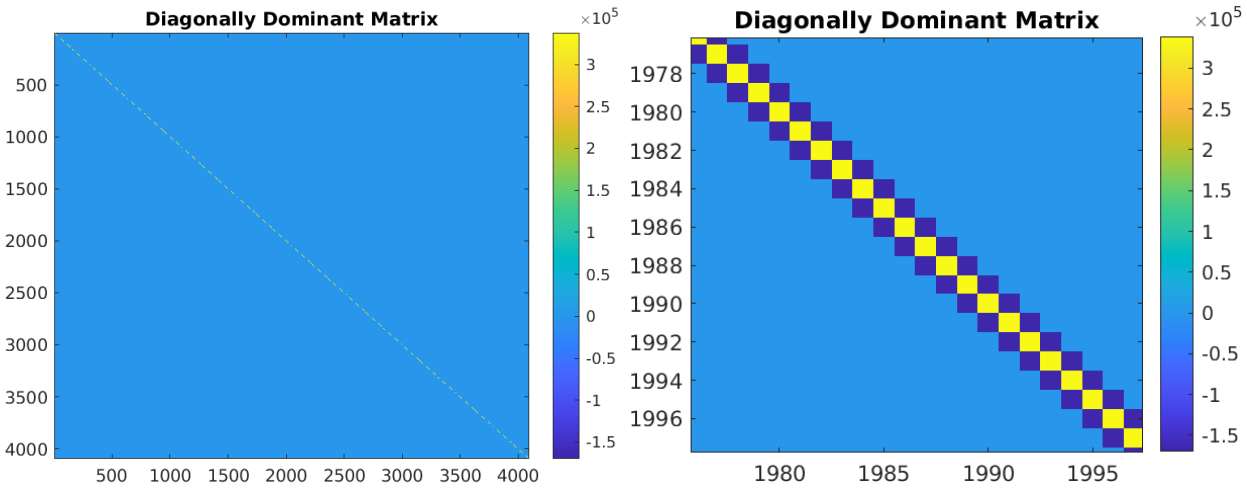

Figure 5: A diagonally dominant matrix is a sparse matrix with a very faint tri-diagonal set of elements. It has a very high $k = 1.733310^{11}$, which is in part due to the large difference in order of magnitude between matrix elements as can be seen in the colour bar.

Jordan Block Matrix A 4096×4096 Jordan block matrix was generated. Though the matrix is sparse like the diagonally dominant matrix, no value goes below zero as can be seen in the colour bar of Figure 6.

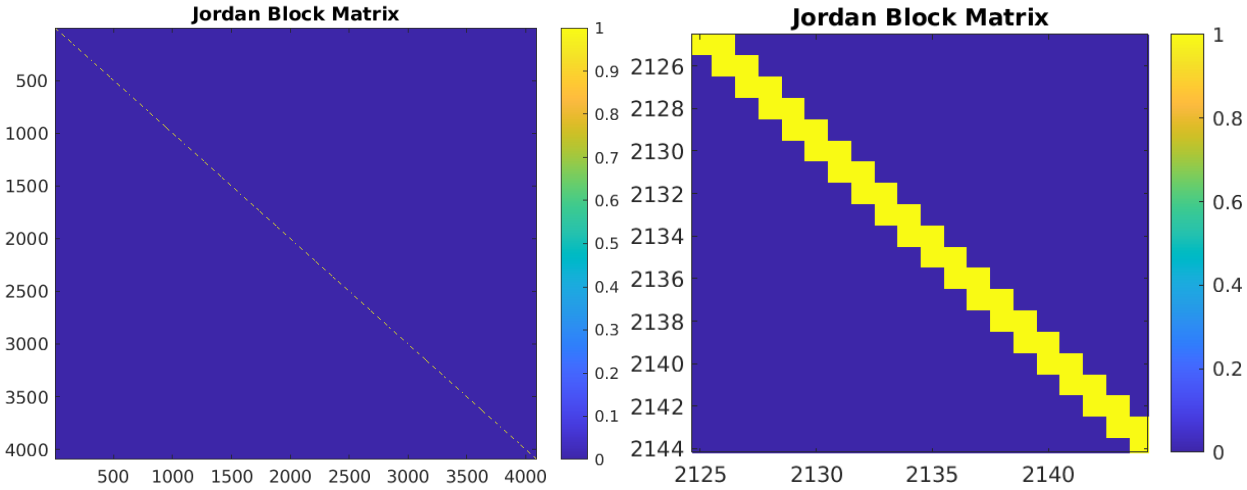

Figure 6: The Jordan block matrix is a non-negative tri-diagonal matrix, with a significantly lower $k = 5.215810^3$ when compared to its diagonally dominant counterpart.

Neumann Matrix A 4096×4096 Neumann matrix was generated. The Neumann Matrix is a singular matrix. This means that the matrix is non-invertible and that its determinant is zero. Showcased in Figure 13, three distinct sets of diagonal elements can be observed.

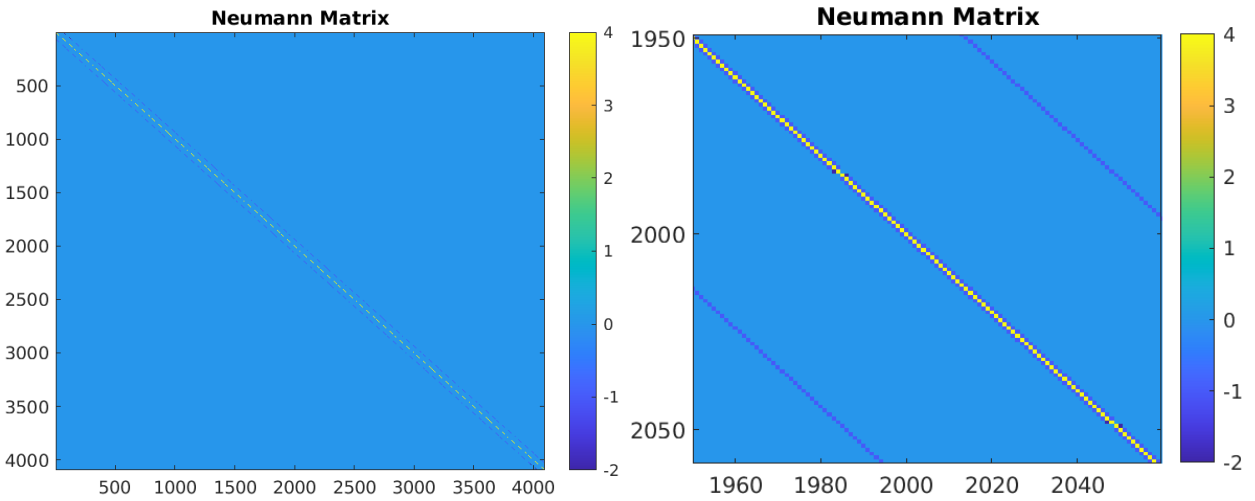

Figure 7: The Neumann matrix is a singular matrix, with an incredibly high $k = 2.957510^{17}$. This matrix would be expected to perform very poorly when subjected to operations like LU factorisation accordingly.

Kac-Murdock-Szegö Matrix A 4096×4096 Kac-Murdock-Szegö matrix was generated. Like the circulant matrix it is another variant of the Toeplitz matrix. Figure 8 shows a decay in values away from the diagonal.

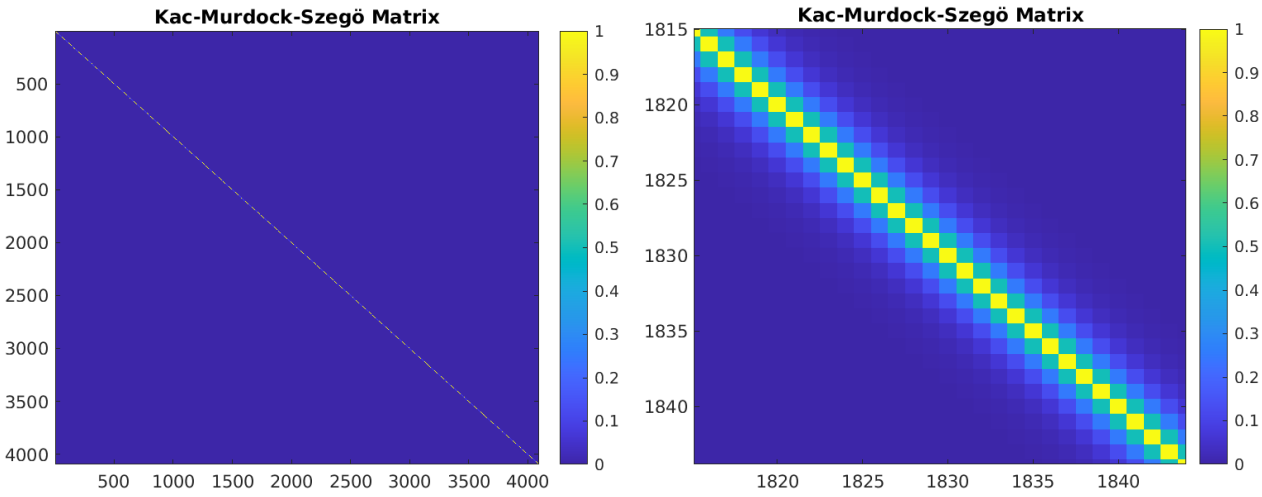

Figure 8: Whilst the matrix achieves a respectable $k = 9.0$ as all entries of the matrix lie between 0 and 1.

For testing the behaviour of the TSLU routine outlined below, n of the left-most columns were selected for factorisation, to give a panel W of shape $m \times n$.

The process of transferring the matrices from `Matlab` to `C` was not entirely trivial, as a `Python` script had to be written in order to convert the `Matlab` output into a format that was suitable for reading into a `C` array. Additionally, whilst other matrices were present in the `Matlab` gallery, only the ones listed above could be loaded onto Chuck and Kelvin via `git`, as only after they had been compressed were they able to be stored on and accessed from GitHub.

3.2 Random Gaussian Matrix Generation

Whilst Section 3.1 Special Matrix Generation described a number of specific matrices with particular properties, these matrices were limited in size, and could not be used to explore the effects of TSLU on larger systems.

As such, matrix generation at runtime was required. For this purpose, `GSL` was utilised. Availing of a L'Ecuyer engine for the random number generation itself, elements of a Gaussian distribution with a mean $\mu = 0$ and a standard deviation $\sigma = 1.0$ were generated and used to populate a panel W. Figure 9 shows a `Matlab` version of this Random Gaussian Matrix (RGM) for a full matrix of dimension 16384×16384, where one can readily note the subtle difference in entry values, with the vast majority lying between ± 2.

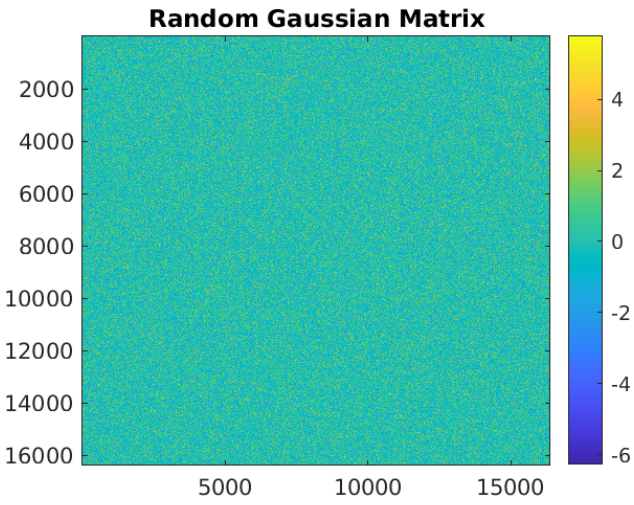

Figure 9: A Random Gaussian Matrix can have a value for k that scales with the size of the matrix i.e. k for a 4096×4096 matrix typically is of the order 10^3, and the k for a 16384×16384 like the one above is typically of the order 10^4.

3.3 LU Algorithms

3.3.1 LUP with GEPP

As a precursor to an implementation of the panel TSLU, an implementation of LUP factorisation availing of the GEPP technique was built, as disclosed in Listing 1. This function was initially based on the `LUP()` function present in `Matlab`, with some adjustments made for the conversion into C - this included the explicit formation of the permutation matrix P.

```
1  void LUP(double * const A, double * const L, double * const U, double *
       const P, const int m_dim, const int n_dim)
2  {
3          /* INITIALISE MATRICES */
4          for(int i = 0; i < m_dim; i++){
5          for(int j = 0; j < n_dim; j++){
6          U[i*n_dim+j] = A[i*n_dim+j];
7          if(i==j) {L[i*n_dim+j] = 1.0;}}}
8          for(int i = 0; i < m_dim; i++){
9          for(int j = 0; j < m_dim; j++){
10         if(i==j) {P[i*m_dim+j] = 1.0;}}}
11
12         /* MAIN LOOP */
13         for(int k = 0; k < n_dim; k++)
14         {
15                 /* Select i >= k to maximise U[i*n_dim+k] */
16                 double tmp = 0; int index = 0;
17                 for(int i = k; i < m_dim; i++){
18                 if(fabs(U[i*n_dim+k])>tmp){
19                 tmp = fabs(U[i*n_dim+k]); index = i;}}
20
21                 /* SWAP U ROWS */
22                 for(int kay = k; kay < n_dim; kay++){
23                 double z_U = U[k*n_dim+kay];
24                 U[k*n_dim+kay] = U[index*n_dim+kay];
25                 U[index*n_dim+kay] = z_U;}
26
27                 /* SWAP L ROWS */
```

```
                for(int kay = 0; kay < k; kay++){
                double z_L = L[k*n_dim+kay];
                L[k*n_dim+kay] = L[index*n_dim+kay];
                L[index*n_dim+kay] = z_L;}

                /* SWAP P ROWS */
                for(int kay = 0; kay < m_dim; kay++){
                double z_P = P[k*m_dim+kay];
                P[k*m_dim+kay] = P[index*m_dim+kay];
                P[index*m_dim+kay] = z_P;}

                /* PERFORM GAUSSIAN ELIMINATION */
                for(int j = k+1; j < m_dim; j++){
                L[j*n_dim+k] = U[j*n_dim+k]/U[k*n_dim+k];
                for(int kay = k; kay < n_dim; kay++){
                U[j*n_dim+kay] -= L[j*n_dim+k] * U[k*n_dim+kay];}}
        }
}
```

Listing 1: LUP Factorisation

In TSLU and indeed many other implementations of LU factorisation by GEPP, P is never explicitly formed, instead either being stored as a string of permutations or by permuting the rows of A during the process of factorisation.

Whilst this LUP implementation may seem mundane, it demonstrates utility that other implementations may fail to capture altogether. As this LUP factorisation was developed with the understanding that the project would eventually expand into panel factorisation, this function is capable of factorising rectangular $m \times n$ matrices as presented in Algorithm 3.

Algorithm 3: Rectangular GEPP

Data: $U = A, \quad L = I, \quad P = I, \quad m, \quad n$

for $k = 1 \rightarrow n - 1$ **do**

  Select $i \geq k$ to maximize $|u_{ik}|$;

  $u_{k,k:n} \longleftrightarrow u_{i,k:n}$ (interchange two rows);

  $l_{k,1:k-1} \longleftrightarrow l_{i,1:k-1}$;

  $p_{k,1:m} \longleftrightarrow p_{i,1:m}$;

  for $j = k + 1 \rightarrow m$ **do**

    $l_{jk} = u_{jk}/u_{kk}$;

    for $q = k \rightarrow n$ **do**

      $u_{j,k:n} = u_{j,k:q} - l_{jk}u_{k,k:q}$;

    end

  end

end

3.3.2 Parallel TSLU

The implementation of parallel TSLU represents the main body of work behind the findings of this dissertation. The main loop performs several functions key to an operational TSLU:

- Tracks the number of rounds in a tournament.
- Informs processors of their neighbours in a given round.
- Identifies best pivots for a given block row.
- Communicates those pivots, and tracks pivot swapping.

Listing 2 shows the start of the TSLU main loop, responsible for setting up the tournament structure, and initialising the entries of local index tracking arrays to -1 to allow for easier parsing of a global index later on.

```
/*
 *       BEGIN TOURNAMENT PIVOTING LOOP
 */

/* Loop passing through layers of the BRT. */
for (int d = 0; d < log2(lastpower); d++){

    /* Setting the values of the local k-i pairs to -1. */
    for(int i = 0; i < n_dim; i++){local_k[i] = -1; local_i[i] = -1;}

    /* Performing communication between senders and
     * receivers in a specified layer. */
    for (int k = 0; k < lastpower; k += 1 << (d + 1)){
        const int receiver = k;
        const int sender = k + (1 << d);
```

Listing 2: TSLU Part 1

The layer loop on `line 6` passes through the layers of the binary reduction tree tracking which round of the tournament is taking place, which in turn is utilised in setting up the communication loop on `line 12`. The value k is used as a rank specifier, and informs processors if they are classified as either a "sender" or "receiver" and then of their relevant neighbour. The `<<` operator is known as a left bit shift; the effect of the operation $1 << n$ is equal to $1 \times (2^n)$. Using bit shifting the binary reduction tree can track which processors should be utilised for the tournament.

Listing 3 then expands on what is taking place inside of the inner loop, in terms of both the communication structure and how the pivoting operations are organised.

```
    /* Performing communication between senders and
     * receivers in a specified layer. */
    for (int k = 0; k < lastpower; k += 1 << (d + 1))
    {
        const int receiver = k;
        const int sender = k + (1 << d);
        if (rank == receiver)
        {
```

```
9           /* Receive "winning" rows from neighbours tournament. */
10          MPI_Recv(W_buffer, b*n_dim, MPI_DOUBLE, sender, tag1,
    MPI_COMM_WORLD, &status);
11          /* Perform pivoting between eligible entries of sub-matrix
    currently located on processor. */
12          pivot_contestants(A, block_height,n_dim, local_index, local_k,
     local_i);
13          /* Collect winning rows from this processors tournament. */
14          double *W = collect_winners(A,b,n_dim);
15          /* Receive indices of the winning rows from neighbours
    tournament. */
16          MPI_Recv(&local_index[b], b, MPI_INT, sender, tag2,
    MPI_COMM_WORLD, &status);
17          /* Form the new sub-matrix from the winners of the round. */
18          combine_winners(A, W, W_buffer, block_height, n_dim);
19          mkl_free(W);
20      }
21
22      else if (rank == sender)
23      {
24          /* Perform pivoting between eligible entries of sub-matrix
    currently located on processor. */
25          /* Send winning rows to neighbour. */
26          pivot_contestants(A, block_height,n_dim, local_index, local_k,
     local_i);
27          MPI_Send(A, b*n_dim, MPI_DOUBLE, receiver, tag1,
    MPI_COMM_WORLD);
28          /* Send indices of the winning rows to neighbour. */
29          MPI_Send(local_index, b, MPI_INT, receiver, tag2,
    MPI_COMM_WORLD);
30      }
31  }
32  /* Barrier to ensure that all ranks have completed their round of
    tournament pivoting. */
33  MPI_Barrier(MPI_COMM_WORLD);
```

Listing 3: TSLU Part 2

The loop and receiver/sender specification was discussed under Listing 2, but are also included in Listing 3 for completeness. The bulk of the work in this method is seated on the shoulders of the receiver ranks. Starting on `line 6` the receiver block is responsible for collecting the winning entries from the sender ranks (`line 9`), performing the block row pivot operation (`line 11`), collecting the global indices of winning rows from the sender ranks (`line 15`) and finally stacking the winning block rows of both the receiver and sender (`line 17`).

By comparison, the sender blocks starting on `line 21` have much less to do. The sender ranks only need to perform their pivoting operations, and are then free to send on the relevant block rows and the global indices of the winning pivots (`line 25-28`).

Listing 4 explores the `pivot_contestants` function invoked on every active rank in every round of tournament pivoting, including the final round that takes place on the root processor.

```
1  void pivot_contestants(double * const A, const int m_dim, const int n_dim,
       int * const local_index, int * const local_k, int * const local_i)
2  {
3          for(int k = 0; k < n_dim; k++)
4          {
5                  /* Select i >= k to maximise A[i*n_dim+k] */
6                  double tmp = 0; int index = 0;
7                  for(int i = k; i < m_dim; i++){
8                      if(fabs(A[i*n_dim+k])>tmp){
9                          tmp = fabs(A[i*n_dim+k]); index = i;}}
10                 /* k-i pairs act as the unit of pivoting,
11                  * denoting the two rows being interchanged. */
12
13                 /* Assign the values of k. */
14                 local_k[k] = local_index[k];
15                 /* Assign the values of i. */
16                 local_i[k] = local_index[index];
17                 /* Swap the rows of the matrix to represent the
       permutation. */
18                 for(int kay = 0; kay < n_dim; kay++){
19                 double z_A = A[k*n_dim+kay]; A[k*n_dim+kay] = A[index*
       n_dim+kay];
```

```
20                  A[index*n_dim+kay] = z_A;}
21                  /* Swap the index rows to represent and track the
    permutation. */
22                  int ind_tmp = local_index[k]; local_index[k] = local_index
    [index]; local_index[index] = ind_tmp;
23          }
24  }
```

Listing 4: Pivot Contestants

At the start of the tournament the `local_index` is an array containing the global indices of the block rows being treated by each individual processor. Using the same partial pivoting scheme as the rectangular GEPP, a given column is examined for suitable pivots, with the current row giving an index k. Rows which are suitable for pivoting give their index i, and then these elements, the $k - i$ pairs, have their values stored in the arrays `local_k` and `local_i`.

After the inner loop has finished, the record of $k - i$ pairs must be transcribed to a global array which is capable of tracking all of the permutations. Listing 5 shows how, with a root at rank 0, the root is able to store the required pivoting information in two arrays, `global_k` and `global_i`, from both all other ranks and itself (`line 6-14`). After all of the pivot information had been collected, one final round of tournament pivoting would have to occur on root before the final GEPP operation could be applied.

```
1     /* Rank 0 collects all pivoting information from a given round into
    the global k-i array */
2     if(rank == 0)
3     {
4           for(int s = 1; s < size; s++)
5           {
6                 MPI_Recv(&global_k[(s*n_dim)+(size*d*n_dim)], n_dim, MPI_INT,
    s, tag3, MPI_COMM_WORLD, &status);
7                 MPI_Recv(&global_i[(s*n_dim)+(size*d*n_dim)], n_dim, MPI_INT,
    s, tag4, MPI_COMM_WORLD, &status);
8           }
9
10          /* Rank 0 must also record its own pivots for a given round. */
```

```
11          for(int i = 0; i < n_dim; i++)
12          {
13              global_k[(d*size*n_dim)+i] = local_k[i];
14              global_i[(d*size*n_dim)+i] = local_i[i];
15          }
16      }
17
18      /* All non-root ranks send their pivoting information to rank 0 to
        track pivots being used. */
19      else if(rank != 0)
20      {
21          MPI_Send(local_k, n_dim, MPI_INT, 0, tag3, MPI_COMM_WORLD);
22          MPI_Send(local_i, n_dim, MPI_INT, 0, tag4, MPI_COMM_WORLD);
23      }
24
25  }
26  /* Final barrier to ensure all communications are complete at this point.
        */
27  MPI_Barrier(MPI_COMM_WORLD);
28
29  /*
30   *      END TOURNAMENT PIVOTING LOOP
31   */
```

Listing 5: TSLU Part 3

After the tournament pivoting loop has concluded and the final `pivot_contestants` has taken place on the root, the information stored in the `global_k` and `global_i` and the `local_k` and `local_i` from the last round of pivoting must be decoded and used to create a set of usable pivots. Listing 6 shows the method at the heart of this process, where `global_index_tmp` is an array initialised with the values of the original unpermuted `global_index`, which had previously been broken down to create the `local_index` arrays owned by each rank in the tournament. This allows for the $k-i$ pairs to reference a consistent array. After the $k-i$ pairs have been exchanged, the `global_index_tmp` array is updated to record how the panel rows themselves are swapped (`line 13-14`).

```
void swap_indices(int * const global_index, int * const global_index_tmp,
    int k, int i)
{
    /* Take the indices of permutation from global_index_tmp*/
    int k_tmp = global_index_tmp[k];
    int i_tmp = global_index_tmp[i];

    /* Swap the indices stored in the global index according to the values
    pulled from global_index_tmp. */
    int ind_tmp = global_index[k_tmp];
    global_index[k_tmp] = global_index[i_tmp];
    global_index[i_tmp] = ind_tmp;

    /* Swap the indices in global_index_tmp to reflect the pivoting of k
    and i. */
    global_index_tmp[k] = i_tmp;
    global_index_tmp[i] = k_tmp;
}
```

Listing 6: Swap Indices

At this point, not only is the tournament pivoting complete, but the entries of the panel W will have been permuted according to the pivots specified. From this point Gaussian elimination was performed to produce the LU factorisation of the panel. In this particular implementation a serial Gaussian elimination algorithm was used, but this acted as a bottleneck to runtime. Future implementations should avail of a parallelised Gaussian elimination to maximise performance.

Due to the differences in implementation between the original TSLU algorithm presented by Grigori et al, the creation of an updated TSLU algorithm was merited, and is described in Algorithm 4: TSLU 2021.

Algorithm 4: TSLU 2021

Require: S, the number of processors $P_{i=1:S}$ assigned to the binary reduction tree;

Require: S be a power of 2;

Require: The input panel $W(1:m, 1:n)$ distributed using a 1-D block row layout;

Ensure: W_{P_i} is the block of rows belonging to P_i;

for d *from* 1 *to* $log2(S)$ **do**

  for k *from* 1 *to* S, *strided by* $1 << (d+1)$ **do**

    Receiver $= k$; Sender $= k << (d+1)$;

    if P_i = *Receiver* **then**

      Pivot rows of W_{P_i} to produce winning rows and row IDs;

      Receive winning b rows from Sender;

      Receive winning row IDs from Sender;

      Combine b rows from W_{P_i} and Sender to form new W_{P_i};

      Combine row IDs from W_{P_i} and Sender to form new set of row IDs;

    else if P_i = *Sender* **then**

      Pivot rows of W_{P_i} to produce winning rows and row IDs;

      Send winning b rows to Receiver;

      Send winning row IDs to Receiver;

  end

  if $P_i =$ *root* **then**

    Receive winning row IDs from all other P_i;

  else

    Sending winning row IDs to P_i = root;

  end

end

Compute ΠW according to pivoting information stored in row IDs;

if $P_i =$ *root* **then**

  Compute $\Pi W = LU$ using GE;

end

4 RESULTS & DISCUSSION

The implementation of a communication-avoiding TSLU factorisation technique was demonstrated under two sets of testing conditions:

1. A variety of panels of size 512×512 to 4096×512, with varying properties for the purposes of examining the strong and weak scaling and growth factor of specific types of matrices.

2. Panels size 2048×2048 to 16384×2048, randomly populated with numbers from a Gaussian distribution for purposes of examining the strong and weak scaling and growth factor on larger matrices.

4.1 Scaling

Scaling is a property of parallel algorithms that serves to measure the effect of the addition of an increasing number of compute cores. Strong and weak scaling each imply different behaviour of the algorithm.

4.1.1 Strong Scaling

Strong scaling examines the behaviour of an algorithm for the use of an increasing core count to treat a problem of the same size. It seeks to measure how the algorithm benefits from increasing computing power. An ideal strong scaling sees perfectly proportional increases in speedup with increasing core count, such that computation runs twice as quickly as the serial computation if using two cores, and four times as quickly for four cores etc.

Figure 10 shows the strong scaling derived from running TLSU on the various kinds of matrices described earlier on, compared with the ideal strong scaling.

As becomes readily evident, the strong scaling for the overall runtime fails to follow the ideal speedup. The consistent, unchanging speedup regardless of core count is an indicator of a bottleneck. This bottleneck in the implementation is discussed below in Section 4.4.1 "Design Error: Serial Gaussian Elimination".

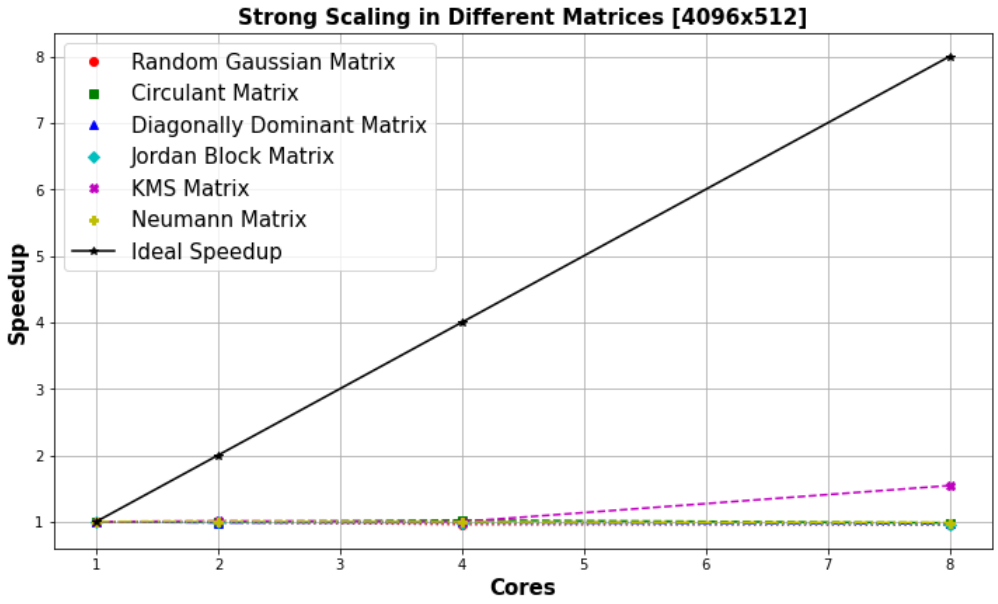

Figure 10: Due to a bottleneck inherent in the GE step of current TSLU 2021 implementation, the strong scaling for the overall runtime was unable to follow the predicted performance curve.

It is possible to see that as the core count increases, the general trend sees the speedup decrease - a behaviour that would be observed if the implementation were not bottlenecked. This decrease in speedup would be attributed to the increasing time spent on communication, and whilst this algorithm is very unfortunately caught in a bottleneck, the comparatively slow degradation in speedup indicates that the communication avoiding technique has played a part in ensuring generally consistent behaviour over an increasing core count.

Of particular note is the behaviour of the KMS matrix. Whilst all of other matrices were processed in 4.5 ± 0.1 seconds for all core counts, the KMS matrix took an extended 7.15 ± 0.06 seconds for one, two and four cores, but then behaved similarly to the rest of the matrices at eight cores. This change could not be reasonably attributed to any one factor - its slow but consistent performance on fewer cores remains a point of further investigation.

Figure 11 then examines the strong scaling for overall runtime on the RGM of size 16384×2048. As before, the strong scaling fails to manifest but the decrease in performance for doubling the number of cores is less than the predicted decrease in performance for a non-communication avoiding algorithm with the same bottleneck.

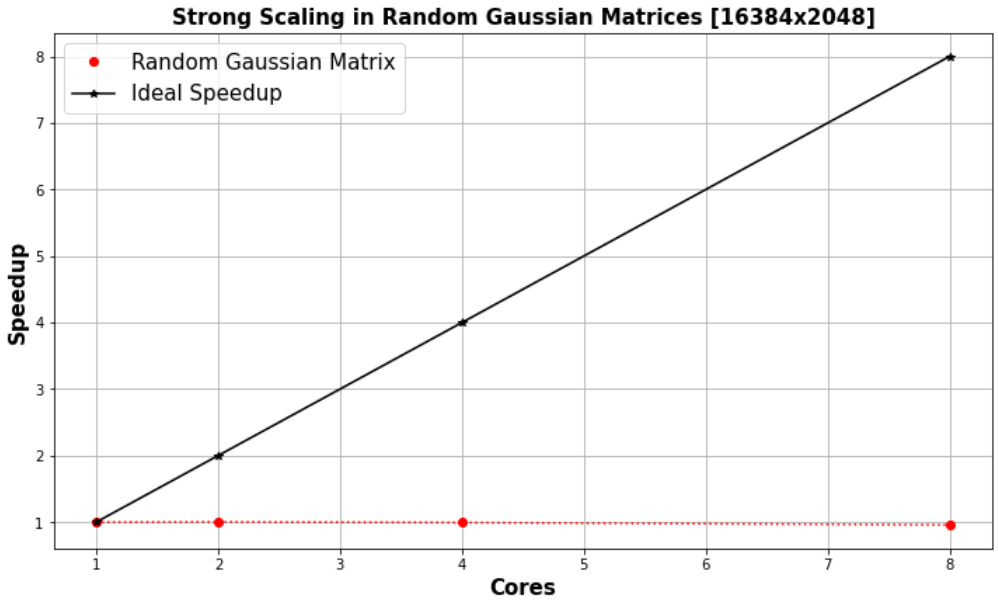

Figure 11: The strong scaling in overall runtime for the large RGM also fails to manifest desirable behaviour, again attributed to the GE bottleneck.

Whilst the performance of strong scaling for overall time is limited by the Gaussian elimination bottleneck it is possible, for analytical purposes, to instead evaluate the strong scaling of the pivoting operations prior to Gaussian elimination.

Figure 12 demonstrates that the portion of the code dedicated to the tournament pivoting begins exhibiting desirable strong scaling behaviour. The most curious element of the scaling curve is in how little the performance seems to differ between the two and four core assessments.

The two core assessments must split the panel into two sub-matrices each of dimension $m/2 \times n$. The best b pivots are selected, and then passed onto the root whereby one more round of pivoting on a $2b \times n$ sub-matrix must be performed before the pivots can be applied to the panel.

On the other hand, the four core assessments split up the panel into four sub-matrices of dimension $m/4 \times n$, at which point the best b pivots are selected and passed into the next round into two separate $2b \times n$ sub-matrices, which in turn go on to have their best b pivots passed onto the root to enable the final pivot calculation after which the pivots can be applied to the panel.

The explanation for this change in behaviour could be that the time spent on communication could have actually balanced out the performance improvement due to increasing the core count. This could then also explain how the larger panels on the 16384×256 scale more consistently, as the performance improvement with a single round of proper tournament pivoting is balanced out by the greater amount of data to parse.

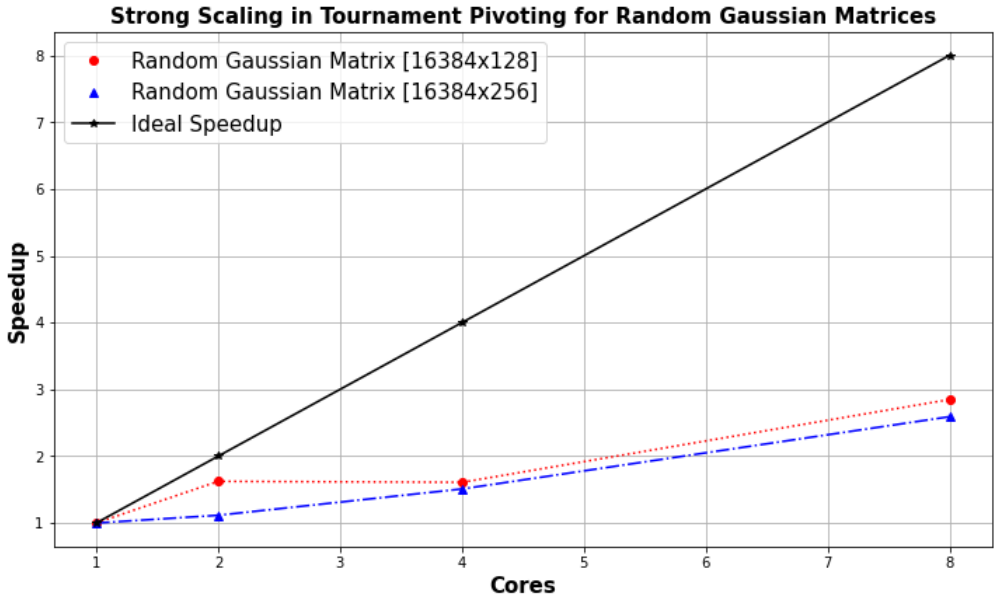

Figure 12: Examining the strong scaling of the tournament pivoting in particular demonstrates that the algorithm's scaling continues to improve for a higher core count, indicating this behaviour may continue for larger systems.

4.1.2 Weak Scaling

Weak scaling examines the behaviour of an algorithm for the use of an increasing core count to treat problems which scale in size with core count. It seeks to measure how the algorithm scales not just in runtime but in efficiency with size. An ideal weak scaling sees runtime remain the same with increasing core count, such that doubling the problem size and doubling the core count at the same time should result in the same runtime for the original problem running in serial.

Figure 13 shows the weak scaling derived from running TLSU on the different types of matrices. As with the strong scaling, the weak scaling also fails to follow the ideal trend.

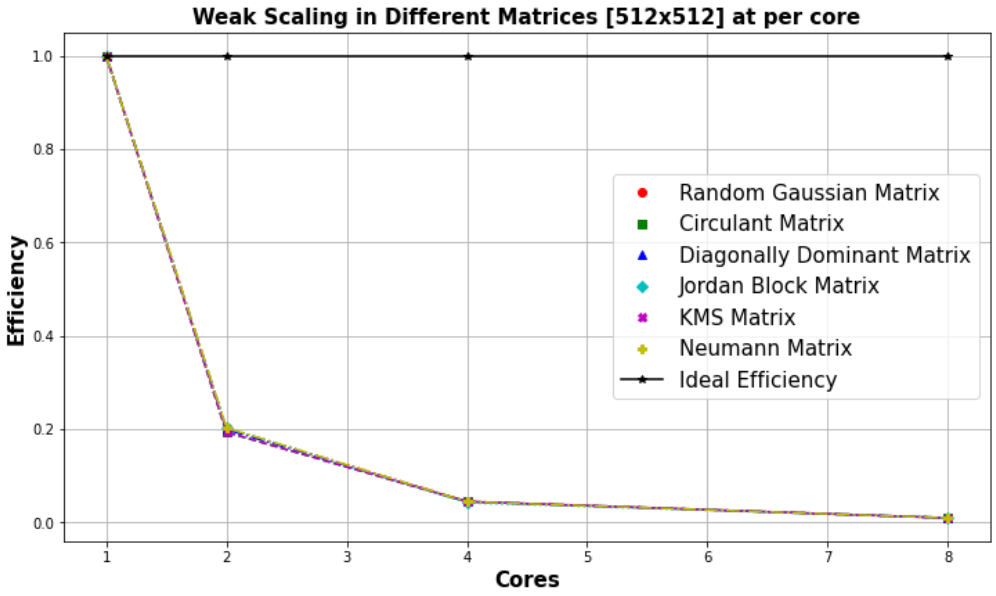

Figure 13: Due to a GE bottleneck in the current TSLU 2021 implementation, the weak scaling in overall runtime was unable to maintain efficiency at scale.

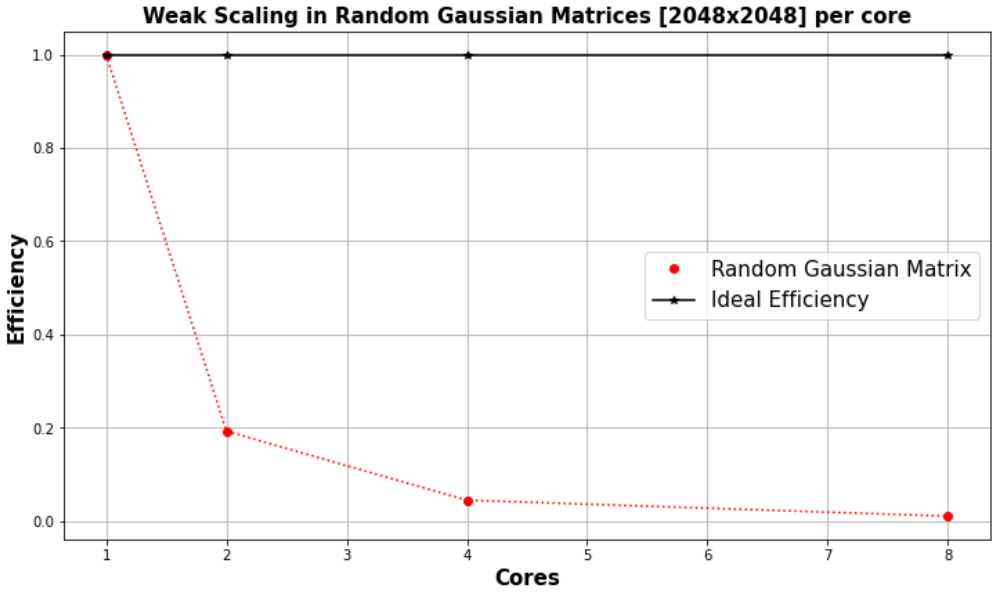

Figure 14: The bottleneck in overall runtime for the TSLU 2021 implementation presented the same weak scaling for the 16384×2048 RGM as well.

The bottleneck described below in Section 4.4.1 "Design Error: Serial Gaussian Elimination" gives further insight into the nature of this problem. The decay in speedup is identical to that seen in examining the weak scaling of a serialised problem, whereby the size of the problem

is increased but the core count remains at one.

Figure 14 examines the weak scaling present on the large RGM. Unsurprisingly, this behaviour is also affected by the bottleneck, and in much the same fashion as the weak scaling present on the varied matrix tests. However, if attention is then turned back to the fundamental pivoting aspect of the TSLU 2021 implementation, it is possible to observe better weak scaling. Figure 15 shows the notable improvement in weak scaling exhibited by the TSLU 2021 algorithm's pivoting operations. Whilst the implementation did not avail of a shared memory structure (hence the sudden drop off upon using two cores), the consistency of the weak scaling for an increasing core count is actually quite well-behaved.

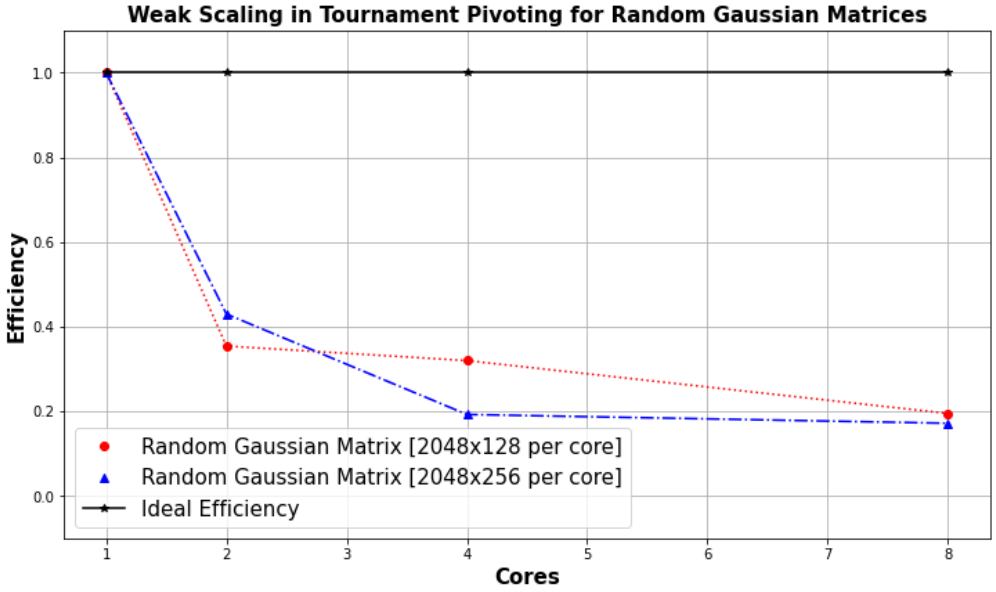

Figure 15: The weak scaling of the pivoting operations in particular proved to be much better than that of the overall runtime. The larger panels performed better at a lower core count, but then performance at the four core tests diverged significantly before steadying out for eight cores.

4.2 Stability Analysis

For each test, the growth factor g was computed as a measure of the stability of the algorithm. The small RGM and the large RGM were shown to be consistently stable with $g = 1$ under all conditions and whilst the special matrices proved well behaved in general, some results were suboptimal.

Table 1 examines the change in g across the varied matrices for the strong scaling tests. The dense circulant and random Gaussian matrices returned $g = 1$ indicating stability in these circumstances. The sparse matrices also behaved well up until the panel was partitioned across eight cores. It is important to recall that g has a lower bound at unity, and so a value of $g = 0$ is inherently erroneous. It would only be able to come about in the circumstance where $||LU||_\infty = 0$. As the infinity norm is the largest sum of elements on any given row, this cancellation could occur when reconstructing a diagonally symmetric LU, whose elements sum to zero.

Matrix Type	4096×512 (1 core)	4096×512 (2 cores)	4096×512 (4 cores)	4096×512 (8 cores)
Circulant	1	1	1	1
Diagonally Dominant	1	1	1	0
Jordan Block	1	1	1	0
KMS	1	1	1	0
Neumann	1	1	1	0
Gaussian	1	1	1	1

Table 1: Growth factors derived during strong scaling tests on varied matrices.

Table 2 shows the growth factor generated when examining the weak scaling across varied matrices, with results matching the strong scaling tests for the 4096×512 on eight cores results.

Again, the dense matrices prove stable with a consistent $g = 1$ for all scenarios except for the circulant matrix of size 2048×512 on four cores. Whilst this behaviour was frankly unexpected, its slight deviation from $g = 1$ means there is little cause for alarm, especially as g returns to unity for 4096×512 on eight cores. If this behaviour were to be attributed to anything, it is believed that it would be the first and only case of numerical instability arising in this algorithm. Tables 3 and 4 demonstrate that the method also remained stable for matrices of higher dimension in both the strong and weak instances.

Matrix Type	512×512 (1 core)	1024×512 (2 cores)	2048×512 (4 cores)	4096×512 (8 cores)
Circulant	1	1	1.792086	1
Diagonally Dominant	1	0	0	0
Jordan Block	1	0	0	0
KMS	1	0	0	0
Neumann	1	0	0	0
Gaussian	1	1	1	1

Table 2: Growth factors derived during weak scaling tests on varied matrices.

Matrix Type	16384×2048 (1 core)	16384×2048 (2 cores)	16384×2048 (4 cores)	16384×2048 (8 cores)
Gaussian	1	1	1	1

Table 3: Growth factors derived during strong scaling tests on the large RGM.

Matrix Type	2048×2048 (1 core)	4096×2048 (2 cores)	8192×2048 (4 cores)	16384×2048 (8 cores)
Gaussian	1	1	1	1

Table 4: Growth factors derived during weak scaling tests on the large RGM.

Whilst the sparse matrices behave well in the serial implementation, all parallel implementations return the problematic $g = 0$ discussed above. Going from these results, it seemed that the issue arose exclusively in the case of tournament pivoting, and for systems where the sub-matrices were square, prompting an investigation into the behaviour of the algorithm.

After several rounds of testing, it was found that for $n \leq \frac{m/2}{\text{nprocs}}$ all matrices demonstrated a value of $g = 1$, but above that the value of g would be driven towards 0, indicating that

the TSLU 2021 algorithm has a strict requirement for the "skinniness" of the panel.

As such, it was possible to generate the data shown in Table 5, showing that provided the panel was of the correct dimension, the algorithm was in fact stable in all instances.

Matrix Type	4096×256 (1 core)	4096×256 (2 cores)	4096×256 (4 cores)	4096×256 (8 cores)
Circulant	1	1	1	1
Diagonally Dominant	1	1	1	1
Jordan Block	1	1	1	1
KMS	1	1	1	1
Neumann	1	1	1	1
Gaussian	1	1	1	1
Matrix Type	512×256 (1 core)	1024×2562 (2 cores)	2048×256 (4 cores)	4096×256 (8 cores)
Circulant	1	1	1	1
Diagonally Dominant	1	1	1	1
Jordan Block	1	1	1	1
KMS	1	1	1	1
Neumann	1	1	1	1
Gaussian	1	1	1	1

Table 5: Growth factors upon appropriately adjusting the width of the input panels W

Interestingly, the adjustment of n did improve g for the circulant matrix, but the random Gaussian matrix, both small and large did not appear to be bound by this same condition in the way that sparse matrices were.

Ultimately, these results show that the TSLU 2021 algorithm is stable in practice for the appropriate panel dimensions as well as for large matrices distributed across a number of processors.

4.3 Limitations and Challenges

There were a number of limitations affecting either the development of the TSLU 2021 implementation or the investigation of the scope of the algorithm.

4.3.1 Developmental Limitations

- There were two main issues with investigating the behaviour of the special matrices. The first was that `Matlab` itself struggled to produce large matrices, with the 4096×4096 matrices representing the upper bound of what could be generated. The second issue was then moving these issues into environments for testing. As it so happens, `git` had a transfer limit of 100MB per file, and as `git` was already a central element of the project's version control this restriction had to be obeyed. Accordingly, if a special matrix was not under 100MB after it had been compressed, it would be discarded. As a result, matrices like the Cauchy and Chebyshev Vandermonde matrices could not be investigated.

- Ultimately, reading in these matrices in larger volumes would never have been feasible. A 16384×16384 RGM had a size of 3.41GB, and trying to read panels from matrices of this size would have readily presented memory problems, especially on larger core counts.

4.3.2 Experimental Limitations

- As the tournament pivoting took place on a binary reduction tree, tests could only take place with a core count that was a power of two. This limited the ability to measure performance on varying numbers of cores, as well as prevented the investigation of the performance of tournaments with a non-power-of-two number of entrants.

- Due to library linking issues, the TSLU implementation could not be run on more than eight cores. The TCDHPC machine "Chuck" was used to generate all of the results discussed above, which only investigated the properties of the algorithm between one to eight cores. Ideally, the tests would have been able to run on the TCDHPC machine "Kelvin" which would have enabled analysis on up to 64 cores.

4.4 Sources of Uncertainty and Error

4.4.1 Design Error: Serial Gaussian Elimination

During the course of this project, the primary focus had been to develop a fully functional implementation of TSLU factorisation, in order to accurately perform a communication efficient LU factorisation.

Accordingly, the overwhelming bulk of development time was spent on getting the tournament pivoting component and then the pivot communication and utilisation operational. Due to this constraint, a simple serial Gaussian elimination was used to compute the LU factorisation of the permuted panel. This enabled the computation of the L and U matrices to a high degree of accuracy, but unknowingly came at the cost of creating the largest contribution to runtime.

This contribution dominated the runtime of the entire program. For example, on the 16384×256 RGM panel the Gaussian elimination took $\approx 4.9 \pm 0.1$ seconds, compared to the time taken for both all of the tournament pivoting, pivot communication and panel permutation at $\approx 0.03 \pm 0.01$ seconds on an eight core assessment. As such, when examining the impact of the running TSLU on an increasing number of cores the actual behaviour of the tournament pivoting had been masked.

So as the focus had been on developing the pivoting operations, by the time the impact of the Gaussian elimination had been discovered, it was deemed too late in the project's lifetime to include the development of a parallelised Gaussian elimination routine in the scope of the TSLU 2021 implementation.

To rectify the analytical issues presented by this oversight, Figures 12 and 15 were created using runtime data that only timed the pivot operations in a bid to show that there were in fact useful parallelised operations taking place.

4.4.2 Other Errors

The inexplicable runtime duration of the KMS matrix on one, two and four cores remained unresolved; the only factor that truly differentiates it from its other special counterparts would be that it actually possesses a low condition number k, but this does little to explain

why it would perform so poorly by comparison.

Additionally the existence of the $g = 0$ events does pose some interesting questions regarding how these errors came about, and why they only existed for sub-matrices possessing more than $\frac{m/2}{\text{nprocs}}$ rows.

Beyond these issues, there were no other readily present errors that merited discussion.

4.5 Future Work

Whilst a great deal of effort has gone into creating a comprehensive body of work, there is more to be done in this field of communication-avoiding LU factorisation. Hopefully through both the GitHub link present in the abstract and the extensive background research prepared for this dissertation, the groundwork has been lain for others to build upon.

Such a future project would need to satisfy two main requirements:

1. Implement a parallel Gaussian elimination algorithm that can avail of pivots created by TSLU 2021, eliminating the current bottleneck.

2. Expand the use of the TSLU panel factorisation to factor an entire matrix, thus achieving a true CALU implementation.

Such an effort would provide ample opportunity to improve and build upon TSLU 2021, and create a whole new parallelised scheme worthy of another dissertation. Additionally, the generation of a greater number of special matrices, especially including pathological matrices, could form a strong basis for a greater degree of analytical work.

5 CONCLUSION

The progress made throughout the duration of this project has manifested itself in the form of clean and commented functional code as available in the GitHub link in the abstract, and as the in-depth background into current methods for pursuing communication-avoiding LU factorisation techniques.

Beyond that, the TSLU 2021 algorithm itself demonstrated both accuracy and scalability. Having determined the upper limit for the width of a TS panel, the growth factor in all instances was equal to unity, indicating that the computation is numerically stable for the all special matrices investigated. Additionally, after the serial Gaussian elimination bottleneck had been identified, it was possible to assess the scalability of the TSLU 2021 algorithm. Whilst binary reduction trees reduce the amount of computing power available to solve problems by a factor of two every round, a reliable pattern of increasing strong scaling and consistently performing weak scaling could be observed.

Satisfaction can be taken in knowing the TSLU method was implemented in a suitably rigorous way to satisfy the requirement that algorithms should not just return fast results, but highly accurate ones as well.

References

[1] Grey Ballard et al. "Minimizing Communication in Linear Algebra". In: *CoRR* abs/0905.2485 (2009). arXiv: 0905.2485. URL: http://arxiv.org/abs/0905.2485.

[2] J. H. Wilkinson. "Error Analysis of Direct Methods of Matrix Inversion". In: *J. ACM* 8.3 (July 1961), pp. 281–330. ISSN: 0004-5411. DOI: 10.1145/321075.321076. URL: https://doi.org/10.1145/321075.321076.

[3] Leslie V. Foster. "Gaussian Elimination with Partial Pivoting Can Fail in Practice". In: *SIAM J. Matrix Anal. Appl.* 15.4 (Oct. 1994), pp. 1354–1362. ISSN: 0895-4798. DOI: 10.1137/S0895479892239755. URL: https://doi.org/10.1137/S0895479892239755.

[4] Stephen J. Wright. "A Collection of Problems for Which Gaussian Elimination with Partial Pivoting is Unstable". In: *SIAM J. Sci. Comput.* 14 (1993), pp. 231–238.

[5] E. Anderson et al. *LAPACK Users' Guide (Third Ed.)* USA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0898714478.

[6] F. G. Gustavson. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms". In: *IBM Journal of Research and Development* 41.6 (1997), pp. 737–755. DOI: 10.1147/rd.416.0737.

[7] The Lu et al. "The Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines". In: (Feb. 1995).

[8] Laura Grigori, James Demmel, and Hua Xiang. "Communication Avoiding Gaussian Elimination". In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008* (Nov. 2008). DOI: 10.1145/1413370.1413400.

[9] Amal Khabou et al. "LU factorization with panel rank revealing pivoting and its communication avoiding version". In: (2012). arXiv: 1208.2451 [cs.NA].

[10] Ming Gu and Stanley C. Eisenstat. "Efficient Algorithms for Computing a Strong Rank-Revealing QR Factorization". In: *SIAM Journal on Scientific Computing* 17.4 (1996), pp. 848–869. DOI: 10.1137/0917055. eprint: https://doi.org/10.1137/0917055. URL: https://doi.org/10.1137/0917055.

[11] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713617.

[12] Laura Grigori, James Demmel, and Hua Xiang. "CALU: A communication optimal LU factorization algorithm". In: *SIAM J. Matrix Analysis Applications* 32 (Oct. 2011), pp. 1317–1350. DOI: 10.1137/100788926.

[13] Laura Grigori. "Dense CALUQR". In: *Who.rocq.inria.fr* (2016). URL: https://who.rocq.inria.fr/Laura.Grigori/TeachingDocs/CS-294_Spr2016/Slides_CS-294_Spr2016/CS294_Spr16_CALUQR.pdf.