Trinity College Dublin

Communication-Avoiding Tall-Skinny LU Factorisation

William O'Sullivan

24. September 2021

Table of Contents

Table of Contents

LU Factorisation

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \\ A_{30} & A_{31} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \\ L_{20} & L_{21} \\ L_{30} & L_{31} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

- LU factorisation is the decomposition of a matrix A into the upper and lower triangular matrices L and U respectively.

LU Factorisation

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \\ A_{30} & A_{31} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \\ L_{20} & L_{21} \\ L_{30} & L_{31} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

- LU factorisation is the decomposition of a matrix A into the upper and lower triangular matrices L and U respectively.
- LU factorisation is used in the computation of determinants, and in inverting matrices. These two operations are cornerstones of linear algebra, and so improvements in LU factorisation are sought out for this purpose.

LU Factorisation

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \\ A_{20} & A_{21} \\ A_{30} & A_{31} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \\ L_{20} & L_{21} \\ L_{30} & L_{31} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

- LU factorisation is the decomposition of a matrix A into the upper and lower triangular matrices L and U respectively.
- LU factorisation is used in the computation of determinants, and in inverting matrices. These two operations are cornerstones of linear algebra, and so improvements in LU factorisation are sought out for this purpose.
- Matrices are typically ill-conditioned. Accordingly, the matrix A needs to be preconditioned using a permutation matrix Π, such that the actual LU factorisation is of the form $\Pi A = LU$

Communication-Avoiding LU Factorisation

- Communication-Avoiding LU (CALU) factorisation is a different approach to performing LU factorisation at scale.

Communication-Avoiding LU Factorisation

- Communication-Avoiding LU (CALU) factorisation is a different approach to performing LU factorisation at scale.
- As the cost of communication typically exceeds the cost of arithmetic, a new computation paradigm has emerged - performing fewer communications at the cost of an increased volume of arithmetic.

Communication-Avoiding LU Factorisation

- Communication-Avoiding LU (CALU) factorisation is a different approach to performing LU factorisation at scale.
- As the cost of communication typically exceeds the cost of arithmetic, a new computation paradigm has emerged - performing fewer communications at the cost of an increased volume of arithmetic.
- CALU as first presented in "Communication Avoiding Gaussian Elimination" by Grigori et al demonstrated the viability of this approach in practice, with performance improvements over existing techniques ranging from a factor of 1.81 to 2.29.

Block Factorisation

- CALU employs a block factorisation technique, whereby the $m \times n$ matrix A is broken into blocks of the specified dimension:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \text{ of dimension } \begin{bmatrix} [b \times b] & [b \times (n-b)] \\ [(m-b) \times b] & [(m-b) \times (n-b)] \end{bmatrix}$$

Block Factorisation

- CALU employs a block factorisation technique, whereby the $m \times n$ matrix A is broken into blocks of the specified dimension:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \text{ of dimension } \begin{bmatrix} [b \times b] & [b \times (n-b)] \\ [(m-b) \times b] & [(m-b) \times (n-b)] \end{bmatrix}$$

- A_{00} and A_{10} are then combined to form a panel W of dimension $m \times b$.

Block Factorisation

- CALU employs a block factorisation technique, whereby the $m \times n$ matrix A is broken into blocks of the specified dimension:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \text{ of dimension } \begin{bmatrix} [b \times b] & [b \times (n-b)] \\ [(m-b) \times b] & [(m-b) \times (n-b)] \end{bmatrix}$$

- A_{00} and A_{10} are then combined to form a panel W of dimension $m \times b$.
- The LU factorisation of W is then computed, and used to update the A_{01} and the trailing matrix A_{11}

TSLU Factorisation

- The panel W has dimensions $m \times b$, where $m >> b$.

TSLU Factorisation

- The panel W has dimensions $m \times b$, where $m >> b$.
- With this constraint in place, W can be described as a Tall-Skinny matrix.

TSLU Factorisation

- The panel W has dimensions $m \times b$, where $m >> b$.
- With this constraint in place, W can be described as a Tall-Skinny matrix.
- Previously performing the LU factorisation of such a matrix required examination of the full column to determine the best pivots, which was a computationally expensive operation.

TSLU Factorisation

- The panel W has dimensions $m \times b$, where $m >> b$.
- With this constraint in place, W can be described as a Tall-Skinny matrix.
- Previously performing the LU factorisation of such a matrix required examination of the full column to determine the best pivots, which was a computationally expensive operation.
- Fortunately, a communication-avoiding technique called "tournament pivoting" has been developed to resolve this issue.

Tournament Pivoting

- Tournament pivoting is a method for finding suitable pivots in a TS matrix.

Tournament Pivoting

- Tournament pivoting is a method for finding suitable pivots in a TS matrix.
- The panel W is partitioned into sub-panels across N processors.

Tournament Pivoting

- Tournament pivoting is a method for finding suitable pivots in a TS matrix.
- The panel W is partitioned into sub-panels across N processors.
- Each processor then determines its best pivots, and communicates with neighbouring processors to share the "winning pivots".

Tournament Pivoting

- Tournament pivoting is a method for finding suitable pivots in a TS matrix.
- The panel W is partitioned into sub-panels across N processors.
- Each processor then determines its best pivots, and communicates with neighbouring processors to share the "winning pivots".
- Combining the newly received pivots with the best pivots on processor, the processor then proceeds to determine which of the present pivots are the best - it is by this method of continually selecting the best pivots that gives tournament pivoting its name.

Table of Contents

Special Matrix Generation

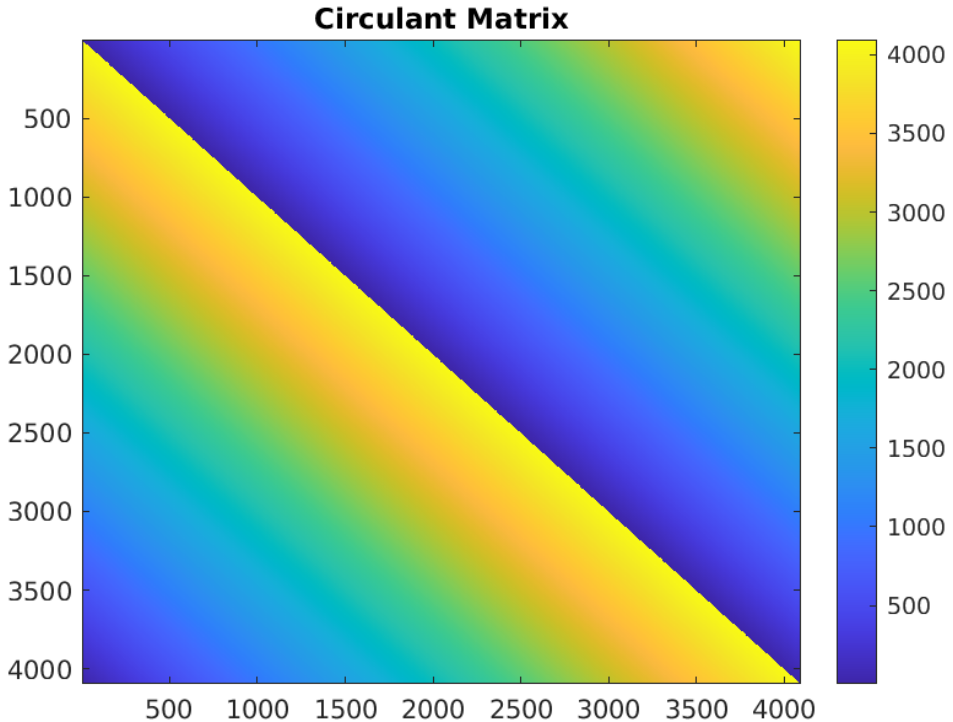
- The investigation called for the generation of a number of "special matrices", matrices with unique properties.

Special Matrix Generation

- The investigation called for the generation of a number of "special matrices", matrices with unique properties.
- To this end, functions available in the MATLAB test matrix gallery were employed to generate special matrices such as the circulant matrix:

Special Matrix Generation

- The investigation called for the generation of a number of "special matrices", matrices with unique properties.
- To this end, functions available in the MATLAB test matrix gallery were employed to generate special matrices such as the circulant matrix:



Random Gaussian Matrix Generation

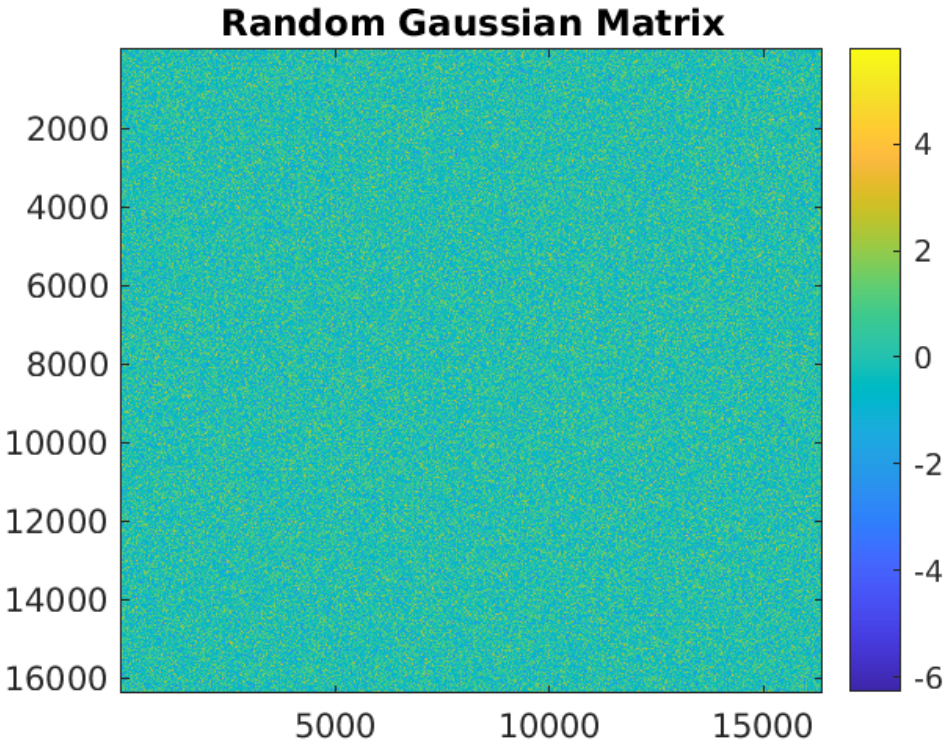
- Due to size limitations with the special matrices, another kind of matrix had to be generated on processor for scaling tests.

Random Gaussian Matrix Generation

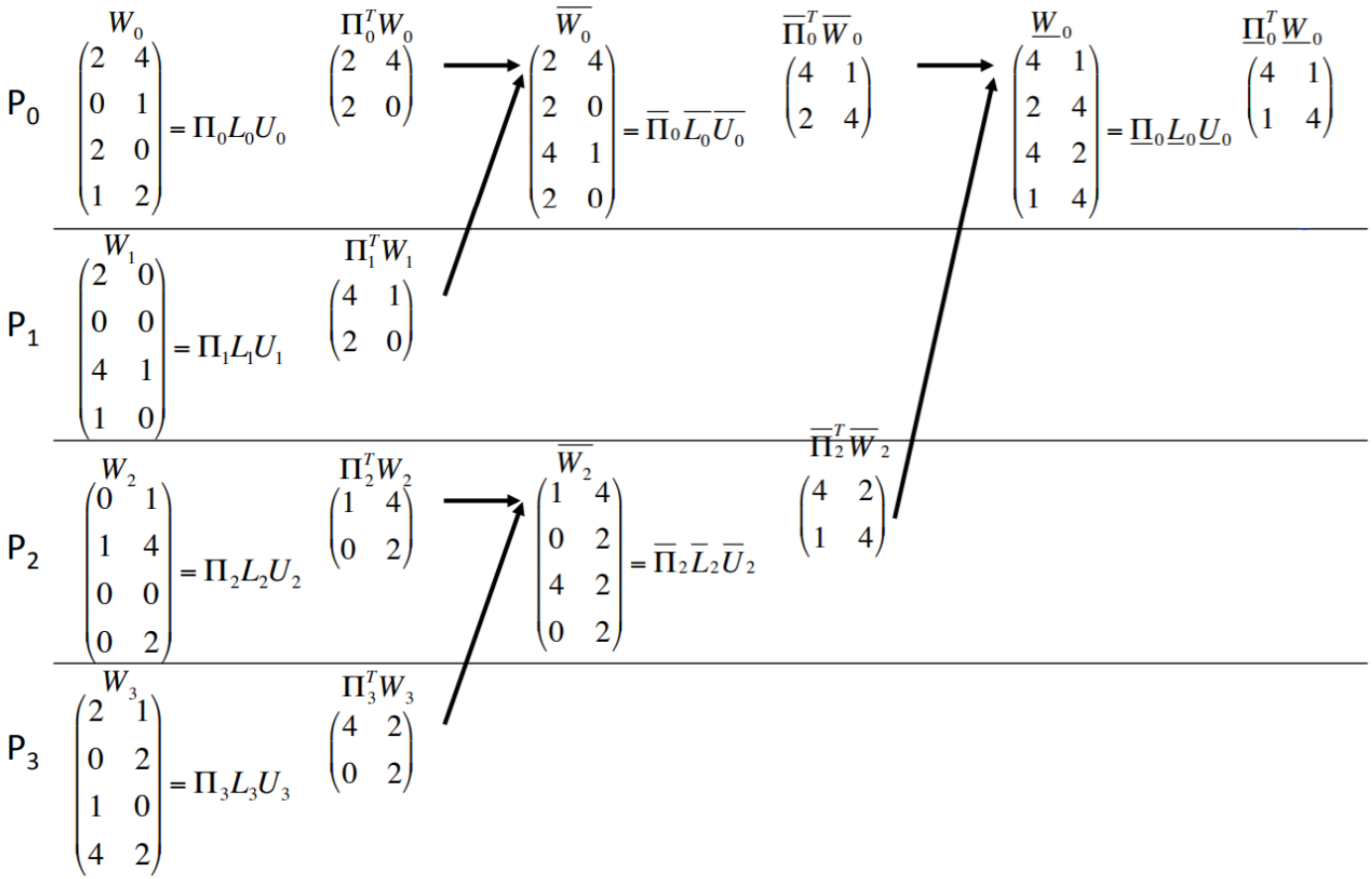
- Due to size limitations with the special matrices, another kind of matrix had to be generated on processor for scaling tests.
- This matrix was a random Gaussian matrix (RGM), whose entries were selected at random from a Gaussian distribution with a $\mu = 0$ and $\sigma = 1$:

Random Gaussian Matrix Generation

- Due to size limitations with the special matrices, another kind of matrix had to be generated on processor for scaling tests.
- This matrix was a random Gaussian matrix (RGM), whose entries were selected at random from a Gaussian distribution with a $\mu = 0$ and $\sigma = 1$:



Tournament Pivoting



- The binary reduction tree is instrumental in tournament pivoting.

Storing Pivots

- In order to permute the panel, the pivots determined by tournament pivoting were tracked by storing them in a set of local arrays.

Storing Pivots

- In order to permute the panel, the pivots determined by tournament pivoting were tracked by storing them in a set of local arrays.
- After each round of the tournament each active processor would store its best b pivots on a global array owned by the root.

Storing Pivots

- In order to permute the panel, the pivots determined by tournament pivoting were tracked by storing them in a set of local arrays.
- After each round of the tournament each active processor would store its best b pivots on a global array owned by the root.
- After the tournament had concluded, these pivots would reference an index of all the rows in the panel in order to create a full array of permutations, which were then in turn used to permute the entries of the panel.

Gaussian Elimination

- After the permuted matrix had been computed, Gaussian elimination was used to factor ΠA into the L and U matrices.

Gaussian Elimination

- After the permuted matrix had been computed, Gaussian elimination was used to factor ΠA into the L and U matrices.
- In this implementation, the Gaussian elimination used was a simple serial elimination on the root process.

Gaussian Elimination

- After the permuted matrix had been computed, Gaussian elimination was used to factor ΠA into the L and U matrices.
- In this implementation, the Gaussian elimination used was a simple serial elimination on the root process.
- Whilst the tournament pivoting and the application of the pivots had been parallelised, this serial Gaussian elimination step resulted in a bottleneck.

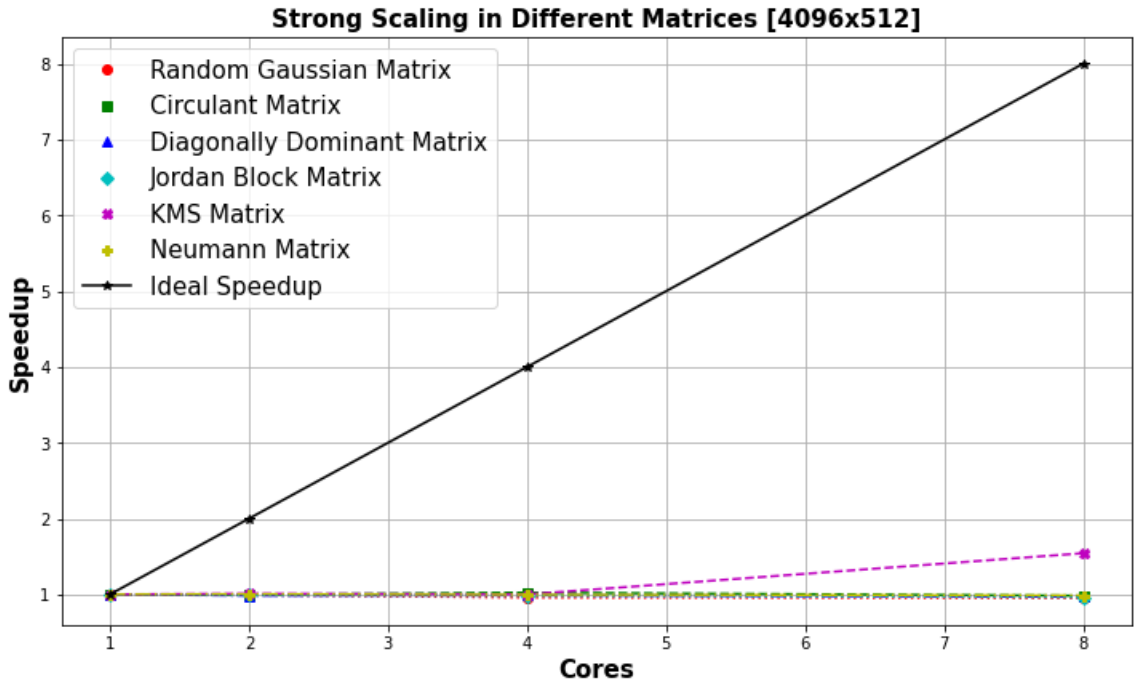
Table of Contents

Strong Scaling - Overall

- The strong scaling overall was shown to be quite poor due to the Gaussian elimination bottleneck.

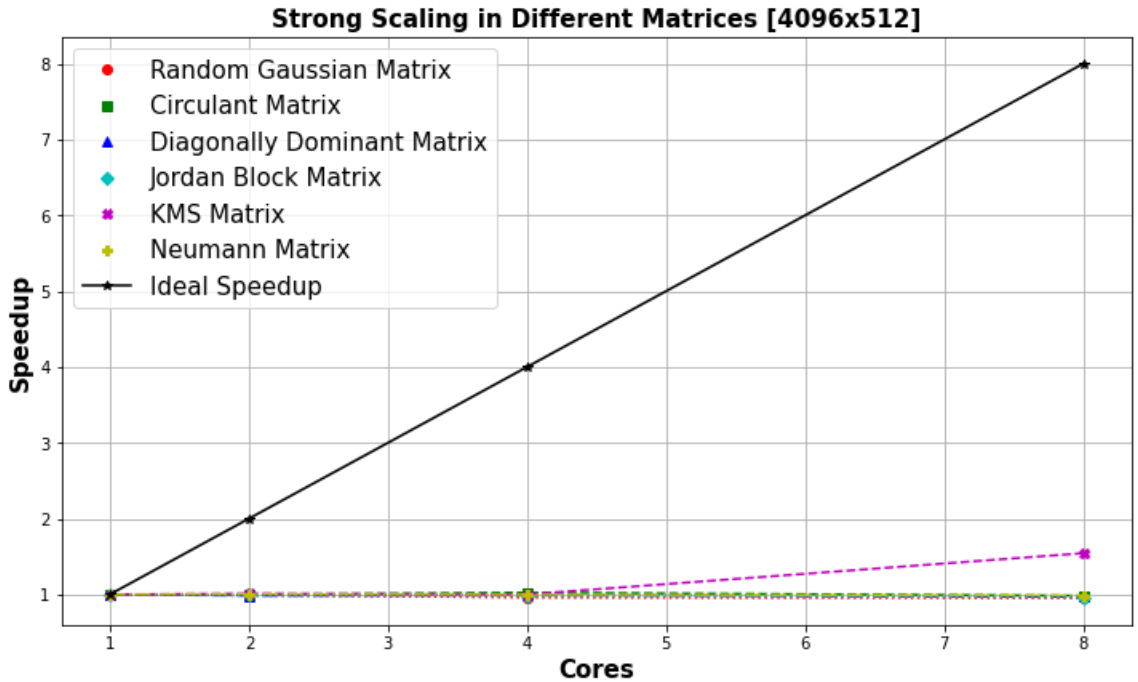
Strong Scaling - Overall

- The strong scaling overall was shown to be quite poor due to the Gaussian elimination bottleneck.
- The below plot of strong scaling demonstrates the problem arising from the bottleneck.



Strong Scaling - Overall

- The strong scaling overall was shown to be quite poor due to the Gaussian elimination bottleneck.
- The below plot of strong scaling demonstrates the problem arising from the bottleneck.



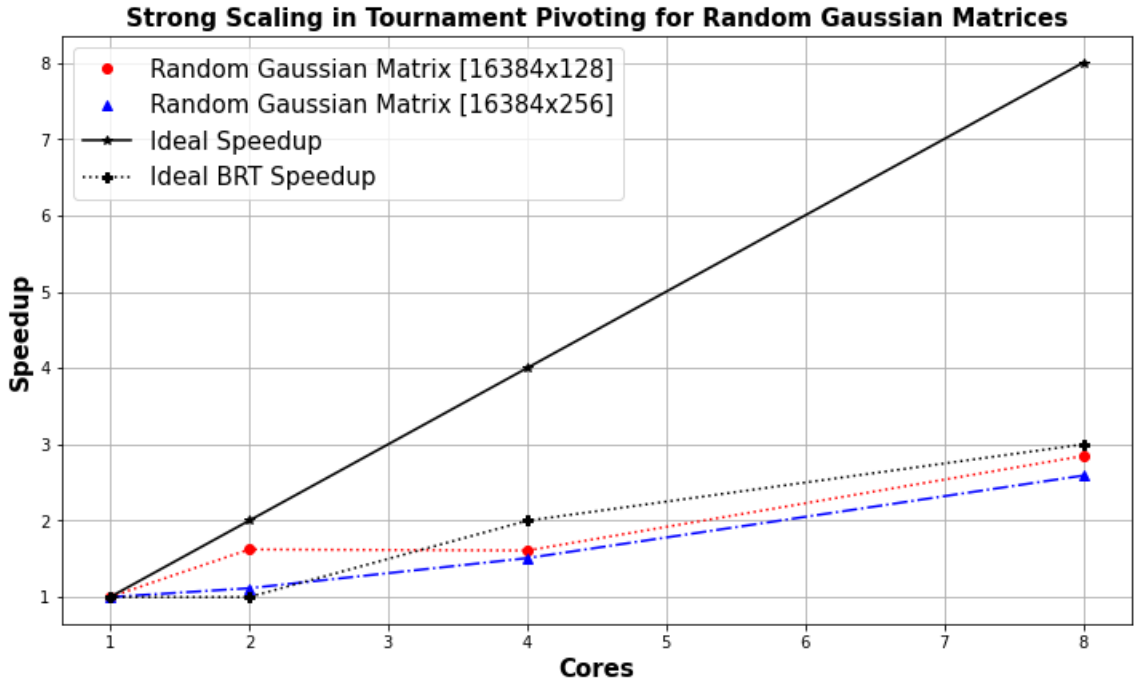
- Note that all speedup values are relative to the uni-core regime; the KMS matrix actually runs slower than other matrices on 1, 2 and 4 cores than on 8, which is a unique behaviour.

Strong Scaling - Pivoting

- The strong scaling for the portion of the code that had been fully parallelised demonstrated much better behaviour.

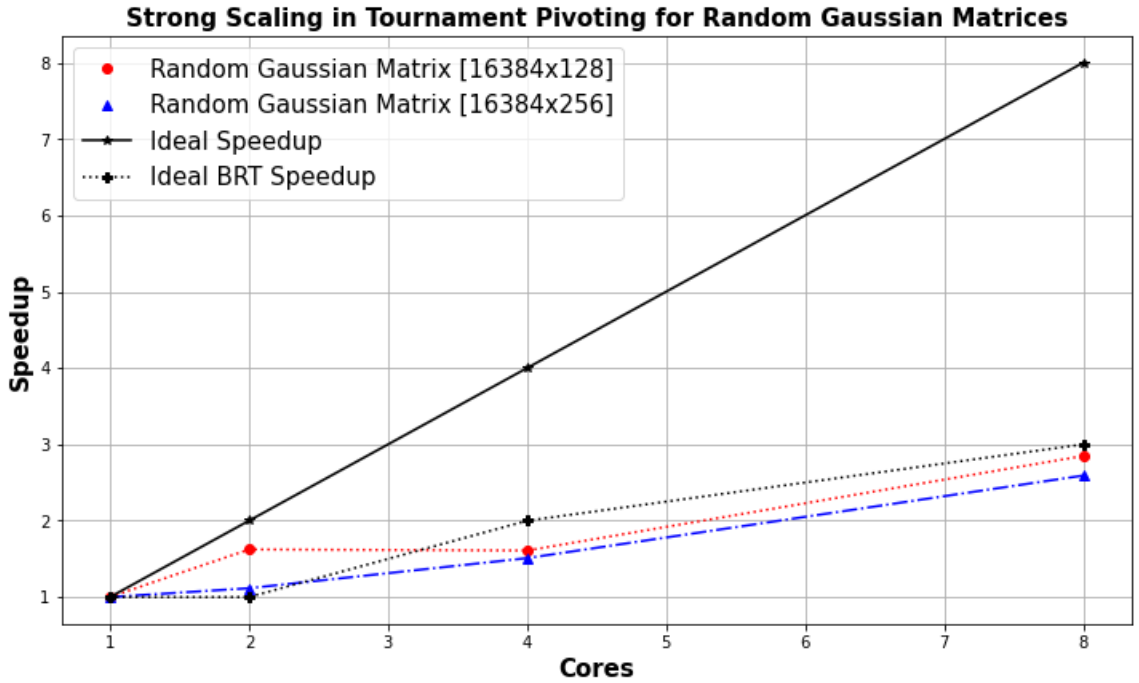
Strong Scaling - Pivoting

- The strong scaling for the portion of the code that had been fully parallelised demonstrated much better behaviour.
- As BRTs halve the number of active cores available to them every round the ideal scaling goes with the $\log_2(N)$.



Strong Scaling - Pivoting

- The strong scaling for the portion of the code that had been fully parallelised demonstrated much better behaviour.
- As BRTs halve the number of active cores available to them every round the ideal scaling goes with the $\log_2(N)$.



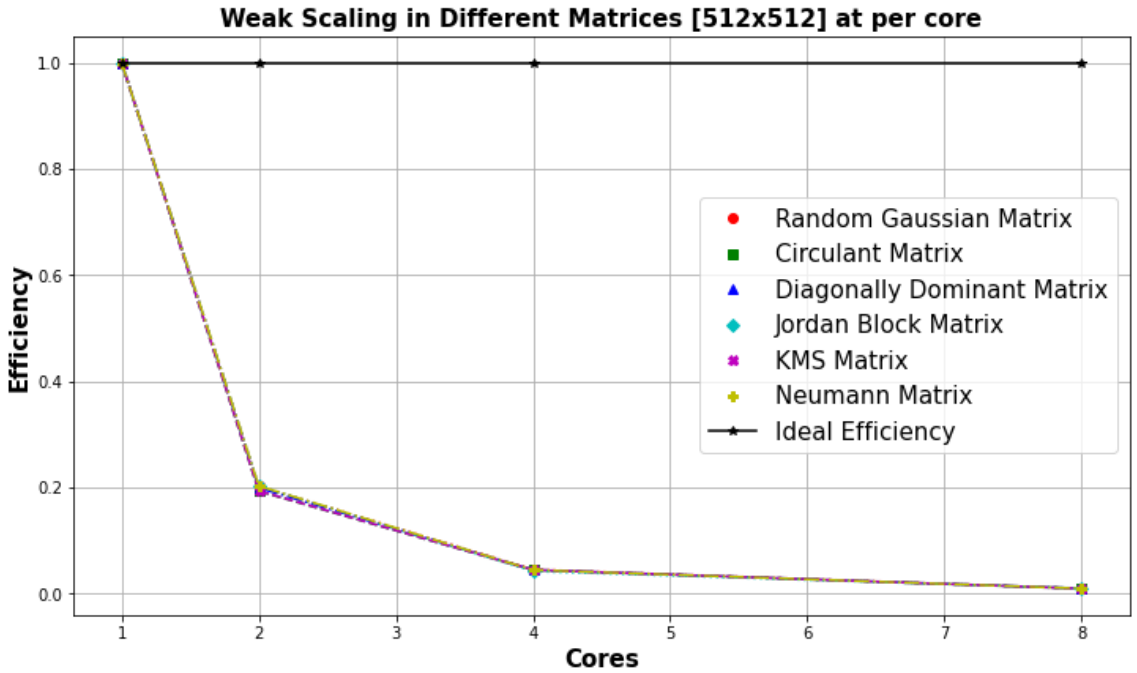
- These times also include the communication performed in permuting the matrix after the best pivots are found, resulting in the discrepancy on the 2-core tests.

Weak Scaling - Overall

- The overall weak scaling also suffered from the Gaussian elimination bottleneck.

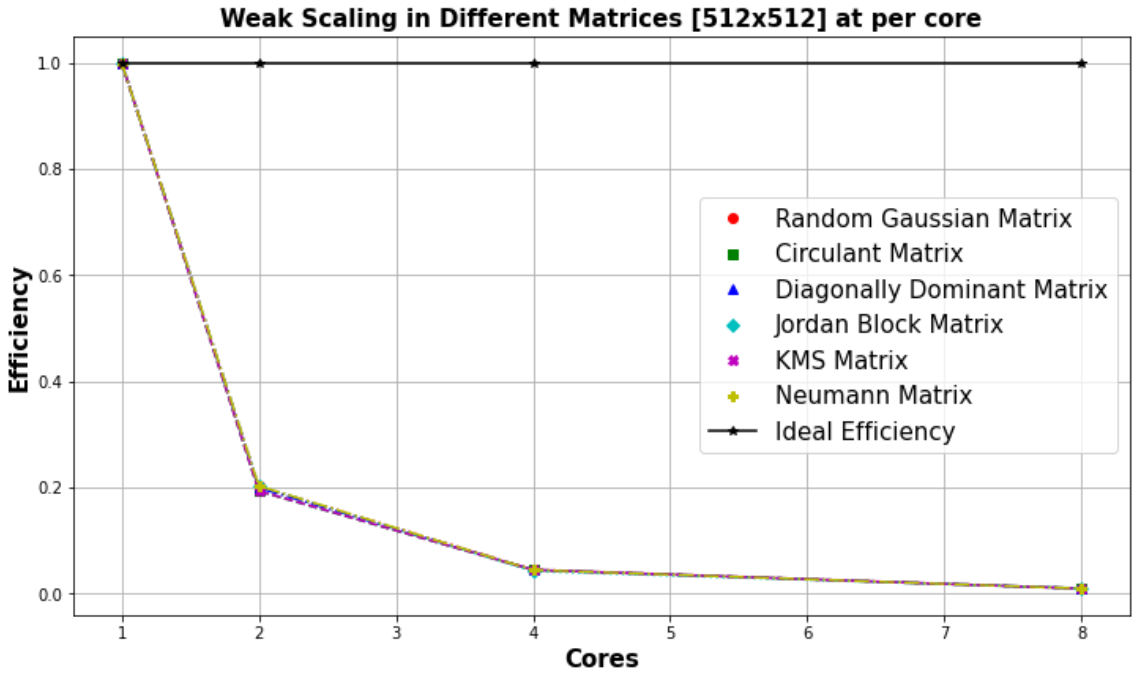
Weak Scaling - Overall

- The overall weak scaling also suffered from the Gaussian elimination bottleneck.
- The below plot of weak scaling demonstrates an exponential decay in efficiency:



Weak Scaling - Overall

- The overall weak scaling also suffered from the Gaussian elimination bottleneck.
- The below plot of weak scaling demonstrates an exponential decay in efficiency:



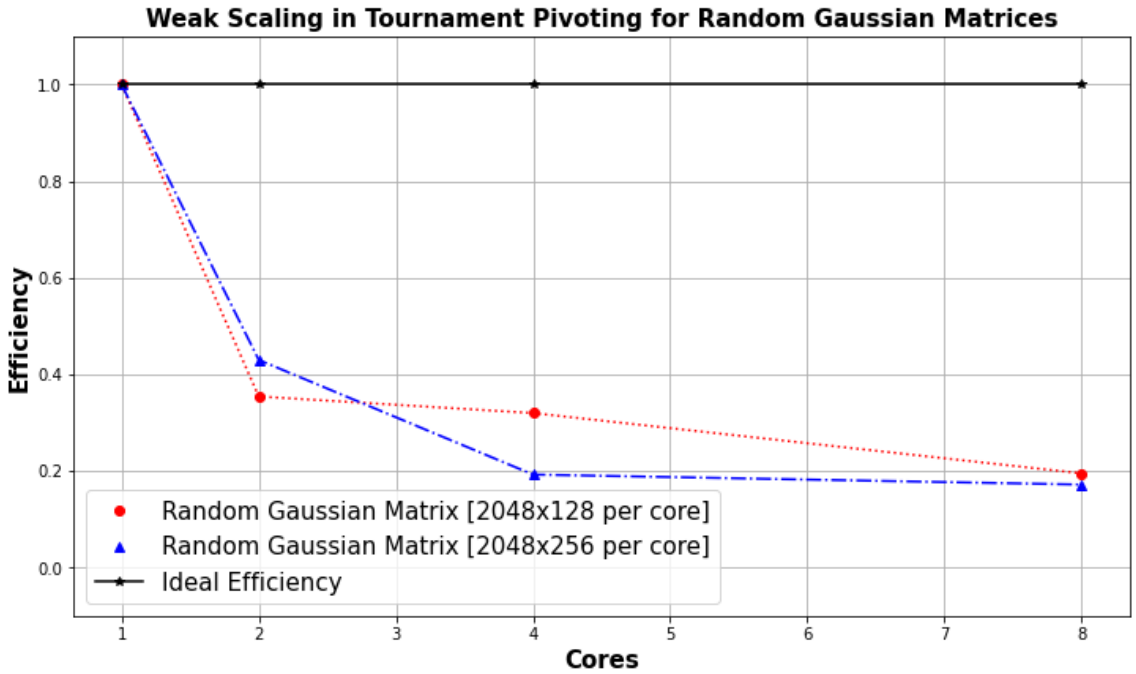
- All matrices experience a sharp drop-off in performance immediately, tapering off to nearly 0% comparative efficiency on 8 cores.

Weak Scaling - Pivoting

- The weak scaling for this parallelised segment of the code was much better than that of the overall weak scaling.

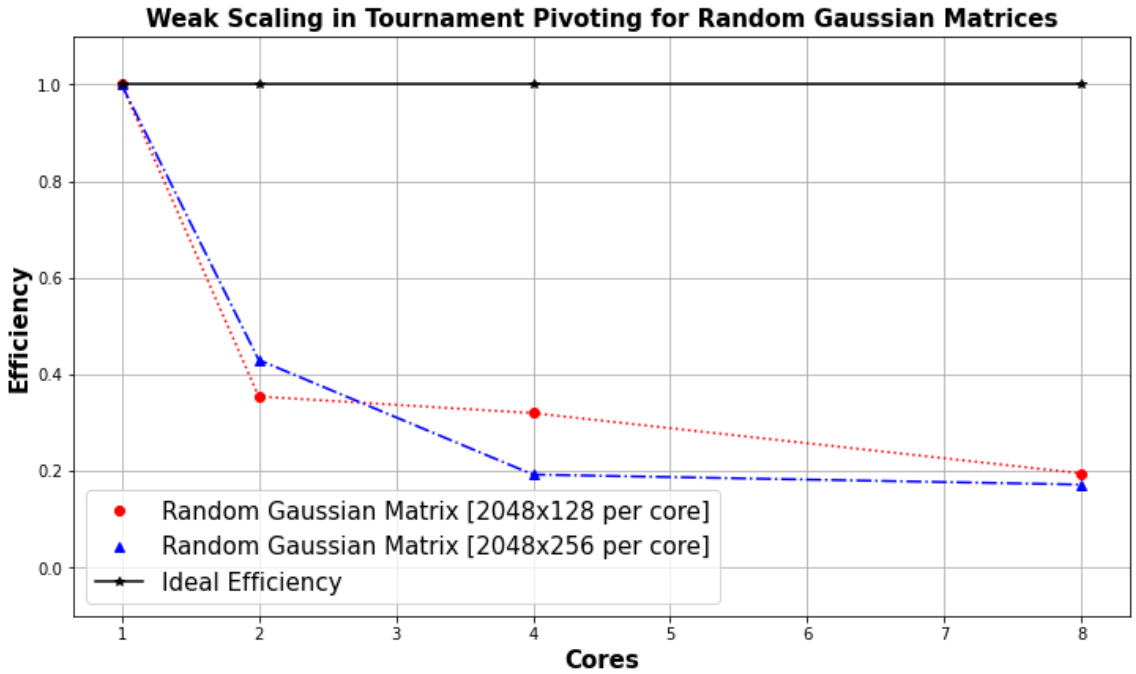
Weak Scaling - Pivoting

- The weak scaling for this parallelised segment of the code was much better than that of the overall weak scaling.
- Efficiency hovers about 20% on the 8 core tests as shown below:



Weak Scaling - Pivoting

- The weak scaling for this parallelised segment of the code was much better than that of the overall weak scaling.
- Efficiency hovers about 20% on the 8 core tests as shown below:



- The efficiency on two processors still dropped to below 50%, but as the behaviour on 4 and 8 cores appeared to level out, further investigation would be merited.

Table of Contents

Stability and Conditioning

- Replicating numerical arithmetic using floating point arithmetic requires that we understand how to treat the differences between the two systems.

Stability and Conditioning

- Replicating numerical arithmetic using floating point arithmetic requires that we understand how to treat the differences between the two systems.
- The conditioning of a problem describes its sensitivity to perturbations. This is intrinsic to the problem, and can be described using a condition number, k.

Stability and Conditioning

- Replicating numerical arithmetic using floating point arithmetic requires that we understand how to treat the differences between the two systems.
- The conditioning of a problem describes its sensitivity to perturbations. This is intrinsic to the problem, and can be described using a condition number, k.
- The stability of an algorithm describes its sensitivity to perturbations. A single problem can be treated by different algorithms, and finding an algorithm that is stable for a given regime is of great importance for ensuring accuracy. The stability of an algorithm can be described by its growth factor, g.

Growth Factor

- The growth factor g is used to measure the stability of the TSLU factorisation, comparing the product of the L and U matrices to the permuted matrix PA to check for consistency.

Growth Factor

- The growth factor g is used to measure the stability of the TSLU factorisation, comparing the product of the L and U matrices to the permuted matrix PA to check for consistency.
- In this implementation, growth factor was computed using the following formula:

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

Growth Factor

- The growth factor g is used to measure the stability of the TSLU factorisation, comparing the product of the L and U matrices to the permuted matrix PA to check for consistency.
- In this implementation, growth factor was computed using the following formula:

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

- The infinity norm was calculated by looping over all rows, summing together the absolute values of the entries on a given row, and then returning the largest sum produced by adding those absolute values together.

Stability under Strong Scaling Tests

Matrix Type	k	4096×256 (1 core)	4096×256 (2 cores)	4096×256 (4 cores)	4096×256 (8 cores)
Circulant	$4.10 \cdot 10^{03}$	1	1	1	1
Diagonally Dominant	$1.73 \cdot 10^{11}$	1	1	1	1
Jordan Block	$5.26 \cdot 10^{3}$	1	1	1	1
KMS	9.0	1	1	1	1
Neumann	$2.99 \cdot 10^{17}$	1	1	1	1
Gaussian	$4.10 \cdot 10^{3}$	1	1	1	1

Table: Values for g calculated during strong scaling tests.

- As shown in Table 1, all tests run on matrices of a suitable width (defined by the limit $b \leq \frac{m/2}{\text{nprocs}}$) returned a growth factor of $g = 1$. This indicated that the regeneration of the matrix PA from LU was accurate, with no error in either the Gaussian elimination or the matrix multiplication.

Stability under Weak Scaling

Matrix Type	k	512 × 256 (1 core)	1024×2562 (2 cores)	2048 × 256 (4 cores)	4096 × 256 (8 cores)
Circulant	$4.10 \cdot 10^{03}$	1	1	1	1
Diagonally Dominant	$1.73 \cdot 10^{11}$	1	1	1	1
Jordan Block	$5.26 \cdot 10^{3}$	1	1	1	1
KMS	9.0	1	1	1	1
Neumann	$2.99 \cdot 10^{17}$	1	1	1	1
Gaussian	$4.10 \cdot 10^{3}$	1	1	1	1

Table: Values for g calculated during weak scaling tests.

- As shown in Table 2, all tests run on matrices of a suitable width returned a growth factor of $g = 1$. This indicated that the regeneration of the matrix PA from LU was accurate also.
- Note that even though some matrices possess very high condition numbers, they still return a stable result.

Table of Contents

Points of Improvement

- Avoiding the Gaussian elimination bottleneck:
  - The Gaussian elimination step, taking the longest amount of time, needs to be parallelised in future implementations.

Points of Improvement

- Avoiding the Gaussian elimination bottleneck:
  - The Gaussian elimination step, taking the longest amount of time, needs to be parallelised in future implementations.
- More rigorous comparison to existing implementations:
  - This implementation could have been compared in performance and stability to ScaLAPACK's PDGETRF function to highlight the benefits of communication avoiding techniques.

Points of Improvement

- Avoiding the Gaussian elimination bottleneck:
  - The Gaussian elimination step, taking the longest amount of time, needs to be parallelised in future implementations.
- More rigorous comparison to existing implementations:
  - This implementation could have been compared in performance and stability to ScaLAPACK's PDGETRF function to highlight the benefits of communication avoiding techniques.
- Scaling into full CALU:
  - The TSLU 2021 algorithm developed as the focus of this dissertation is ready to be used in the performance of full CALU.

Limitations & Problems

- Tests could not be run on Kelvin as the code failed to build against MKL despite all efforts - this limited the number of cores the code could run on, as well of the size of the RGMs generated at runtime.

Limitations & Problems

- Tests could not be run on Kelvin as the code failed to build against MKL despite all efforts - this limited the number of cores the code could run on, as well of the size of the RGMs generated at runtime.
- Because of GIT restraints on filesize, only certain kinds of readily compressed matrices could be stored and loaded when it came to testing the special matrices. This meant known pathological matrices like the Wilkinson or Foster matrices could not be investigated to really push the stability tests to the limit.

Limitations & Problems

- Tests could not be run on Kelvin as the code failed to build against MKL despite all efforts - this limited the number of cores the code could run on, as well of the size of the RGMs generated at runtime.
- Because of GIT restraints on filesize, only certain kinds of readily compressed matrices could be stored and loaded when it came to testing the special matrices. This meant known pathological matrices like the Wilkinson or Foster matrices could not be investigated to really push the stability tests to the limit.
- As the Gaussian elimination bottleneck had been identified quite late in the project, it was not within scope to implement a fully parallelised GE routine to resolve the overall performance issue. That said, it is well understood where this particular limitation now lies.

Final Conclusions

- Ultimately, the project to develop and implement a TSLU algorithm was successful.

Final Conclusions

- Ultimately, the project to develop and implement a TSLU algorithm was successful.
- Whilst hindered by the Gaussian elimination bottleneck, it could be shown that the tournament pivoting and then the application of the pivots themselves performed to the desired specification in terms of strong scaling, and performed well for the weak scaling.

Final Conclusions

- Ultimately, the project to develop and implement a TSLU algorithm was successful.
- Whilst hindered by the Gaussian elimination bottleneck, it could be shown that the tournament pivoting and then the application of the pivots themselves performed to the desired specification in terms of strong scaling, and performed well for the weak scaling.
- Additionally, the pivoting was deemed stable as the Gaussian elimination did not produce any errors over the investigated matrices. This meant that the pivots adequately preconditioned the panels, with a communication avoiding framework in place.

Final Conclusions

- Ultimately, the project to develop and implement a TSLU algorithm was successful.
- Whilst hindered by the Gaussian elimination bottleneck, it could be shown that the tournament pivoting and then the application of the pivots themselves performed to the desired specification in terms of strong scaling, and performed well for the weak scaling.
- Additionally, the pivoting was deemed stable as the Gaussian elimination did not produce any errors over the investigated matrices. This meant that the pivots adequately preconditioned the panels, with a communication avoiding framework in place.
- Satisfaction can be taken in knowing that the algorithmic development and implementation was stable and sound, meeting the requirement that these algorithms should not just be fast, but accurate as well.

Questions

Thank you for your time!