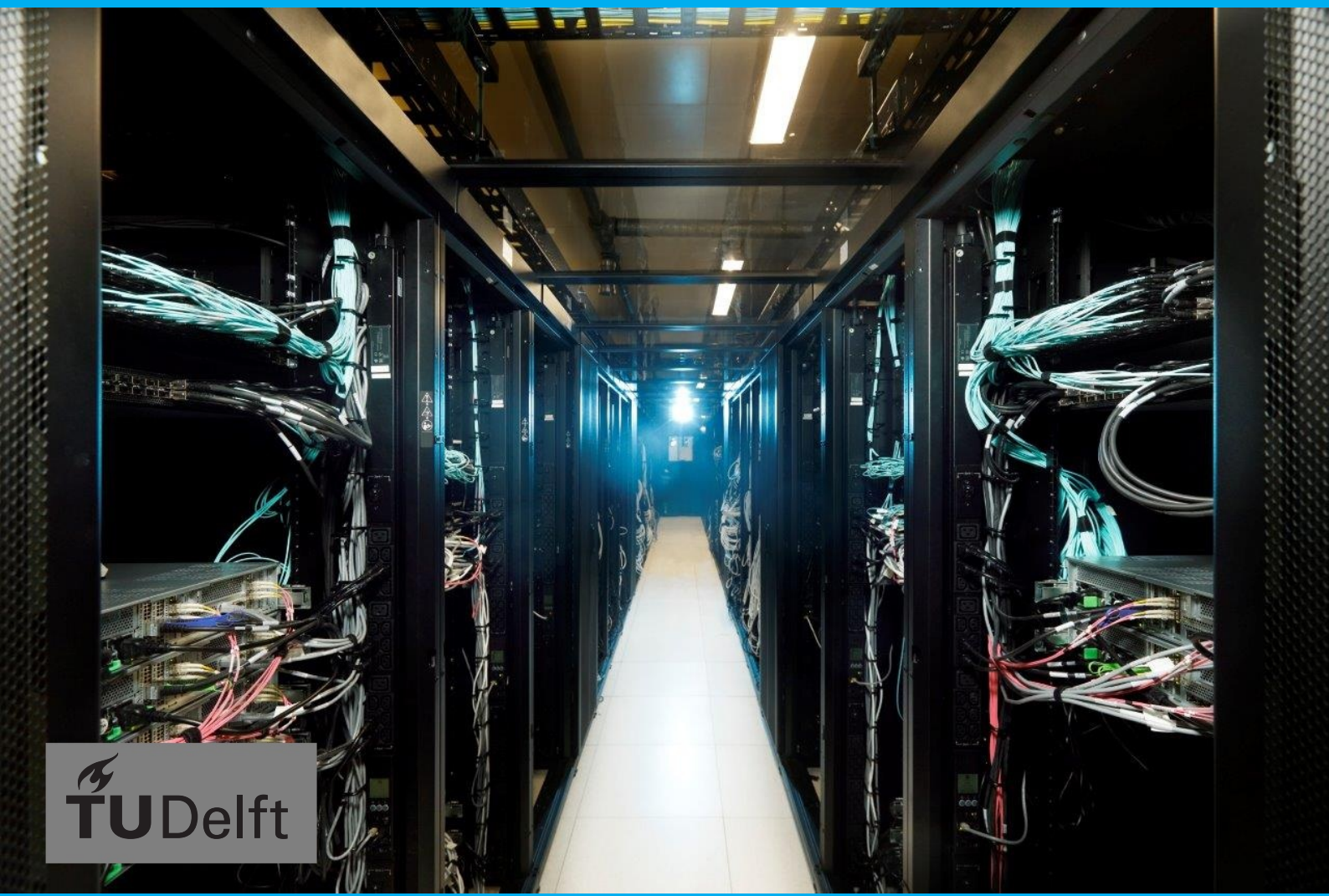
Improving time efficiency by parallelising the LU decomposition

A.R.T.Y. Bartelink



Improving time efficiency by parallelising the LU decomposition

by

A.R.T.Y. Bartelink

to obtain the degree of Bachelor of Science
at the Delft University of Technology,

Student number: 5634040
Project duration: April 22, 2025 – July 9, 2025
Thesis committee: Prof. M. B. van Gijzen, TU Delft, supervisor
Dr. H. N. Kekkonen, TU Delft

Cover image: (Students, 2022)

TU Delft

Laymen's Summary

Krylov subspace methods are methods to find solutions to high-dimensional linear systems efficiently. One of those methods is the Induced Dimension Reduction method, a method that has been implemented in a parallel Fortran package. To ensure the efficiency of this package, it is important that low-level computations go fast, like creating the LU decomposition. In this paper, a parallel algorithm for the LU decomposition is developed and improved. Later, the algorithm is extended to work efficiently for matrices with a special structure, band matrices. From this, it follows that the algorithms created do show an increase in efficiency and a decrease in computational time. Furthermore, initial testing after integration in the IDR package also shows an improvement in computational time.

Summary

Shifted Krylov subspace methods are methods to efficiently find solutions to high-dimensional linear systems of the form $(A - \sigma I)\mathbf{x} = \mathbf{b}$ by iteratively computing the Krylov subspace:

$$\mathcal{K}_m(A - \sigma I, \mathbf{b}) = \text{span}\{\mathbf{b}, (A - \sigma I)\mathbf{b}, (A - \sigma I)^2\mathbf{b}, \dots, (A - \sigma I)^{m-1}\mathbf{b}\}. \quad (1)$$

One method that uses Krylov subspaces is the Induced Dimension Reduction (IDR) method, a method that only stores parts of the Krylov subspaces, thereby decreasing the memory needed by the algorithm. To enhance the efficiency of the implementation of the parallel IDR(s) method, low-level operations such as using the LU decomposition for the preconditioning can be improved.

In this paper, a sequential algorithm for the LU decomposition is first implemented to find a basis for the parallel algorithm. Then, using a block-wise distribution of the matrix, the sequential algorithm is updated to a parallel algorithm and improved for greater efficiency by reducing sending operations. The next step was to update the algorithm to also work efficiently on banded matrices. This was first done by changing the original algorithm to also keep in mind the bandwidth of the matrix. Later, the matrix structure was changed to only store the diagonals inside the band. The LU decomposition algorithm was changed accordingly so that the right indices were used for the updates during the algorithm.

From a set of tests, it follows that creating a parallel implementation of the LU decomposition does improve the efficiency and computational time of the algorithm. The improved algorithm is the fastest algorithm for dense matrices. For banded matrices, the full matrix algorithm is better suited if the speed of the computation has to be high and the bandwidth is large. If the memory is more of a concern or the bandwidth is small, the memory-efficient algorithm is the better choice. On top of that, early implementations in the IDR (s) Fortran package also show a decrease in computational time. So, all in all, while a sequential algorithm might be easier to create, the parallel algorithms do speed up the process.

Contents

1 Introduction

Krylov subspace methods are widely known as one of the most significant classes of numerical algorithms in scientific computing. Their ability to efficiently approximate solutions to high-dimensional linear systems has made them an essential tool in various fields such as computational physics, engineering, and applied mathematics (Dongarra and Sullivan, 2000). As computational problems continue to grow in size and complexity, the need for robust and high-performing iterative solvers becomes increasingly important.

Given a linear system of the form

$$A\mathbf{x} = \mathbf{b}$$

where $A \in \mathbb{C}^{n\times n}$, $\mathbf{x} \in \mathbb{C}^n$ and $\mathbf{b} \in \mathbb{C}^n$, Krylov subspace methods generate a series of approximations to the solution $\mathbf{x}$ by iteratively constructing a Krylov subspace of order m, defined as

$$\mathcal{K}_m(A, \mathbf{b}) = \text{span}\{\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, ..., A^{m-1}\mathbf{b}\} \tag{1.1}$$

These subspaces form a nested sequence where

$$\mathcal{K}_m \subseteq \mathcal{K}_{m+1} \subseteq \cdots \subseteq \mathcal{K}_M \tag{1.2}$$

which serves as the foundation for a variety of iterative solvers (Gutknecht, 2007).

Krylov subspaces can be adapted to solve a larger set of problems, like shifted Krylov problems. In many applications, the linear systems involve a shift in the matrix A. The resulting system is typically of the form

$$(A - \sigma I)\mathbf{x} = \mathbf{b} \tag{1.3}$$

where σ is a scalar shift and I is the identity matrix. This transformation leads to a shifted Krylov subspace, where the subspaces are created by

$$\mathcal{K}_m(A - \sigma I, \mathbf{b}) = \text{span}\{\mathbf{b}, (A - \sigma I)\mathbf{b}, (A - \sigma I)^2\mathbf{b}, \dots, (A - \sigma I)^{m-1}\mathbf{b}\}. \tag{1.4}$$

The most prominent Krylov subspace methods are the Conjugate Gradient (CG) method and the Generalized Minimal Residual (GMRES) method. The CG method is especially effective for solving symmetric positive definite systems, where it minimises the quadratic function

$$\frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{b}^T\mathbf{x} \tag{1.5}$$

(Shewchuk, 1994). GMRES, on the other hand, applies to all non-singular matrices. It creates an orthonormal basis of the Krylov subspace using the Arnoldi process and solves a least-squares problem at every iteration. While GMRES is highly stable and flexible, it requires a lot of memory to store all previous Krylov vectors (Saad and Schultz, 1986). Additionally, although the number of iterations for GMRES and CG is approximately the same, the time required per iteration for GMRES is longer, making it a slower method.

To address the performance issue of GMRES and the limited applicability of CG, other Krylov solvers have been implemented in high-performance computing environments. One method is the Induced Dimension Reduction (IDR) method. Instead of storing all Krylov vectors, IDR creates subspaces that decrease in dimension with each iteration, thereby increasing memory efficiency. The basic IDR(1) uses a one-dimensional subspace, and IDR(s) generalises this to an s-dimensional space. On top of that, IDR does not require full orthogonalisation, making it computationally cheaper for large matrices (Gutknecht, 2011).

A recent Fortran package implements several IDR algorithms designed for parallel execution (van Gijzen and Collignon, 2011). Particularly for solving sequences of shifted systems (Baumann and van Gijzen, 2015), (van Gijzen et al., 2014). Enhancing the performance of this package, especially for high-dimensional problems, often requires improvements to low-level operations. In particular, an efficient parallel implementation of the LU decomposition can greatly improve the efficiency of the preconditioning for sequences of problems.

Research Question:

How can we design an efficient parallel LU decomposition for integration into the IDR Fortran package?

Objectives

1. **Development of a Sequential LU Decomposition Algorithm:** Research reliable and efficient sequential LU Decomposition algorithms. This algorithm will be the baseline for the parallel algorithm. The focus will be on minimising memory usage and reducing the overall computational time.

2. **Parallelisation of the LU Decomposition Algorithm:** Develop a parallel algorithm using the sequential algorithm. Decide on the software used for the parallelisation and compare different matrix distributions. Then optimise the algorithm by targeted broadcasting.

3. **LU Decomposition for Banded Matrices:** Improve the parallel algorithm to work more efficiently with banded matrices and optimise its memory efficiency for matrices of that form.

4. **Testing the algorithms:** Use a series of tests to gain insight into the parallel efficiency of the various algorithms for the LU decomposition using Amdahl's law.

This paper is structured across several chapters, each focusing on different aspects. Firstly, Chapter 2 focuses on a sequential algorithm for the LU decomposition, which will be the basis for the parallel algorithm. The introduction of the matrix distribution scheme and the parallel algorithms for dense matrices is done in Chapter 3. In Chapter 4, the parallel algorithms are extended to work efficiently on banded matrices. First, updating sending operations and

later changing to a different storage scheme for banded matrices. Chapter 5 analyses the performance of the algorithms using Amdahl's law. The paper concludes in Chapter 6, the conclusion, and Chapter 7, the discussion.

2

Implementing the LU decomposition

To accelerate the convergence of the IDR(s) algorithm, it is common to solve the equivalent preconditioned system

$$AP^{-1}\mathbf{y} = \mathbf{b}, \quad \mathbf{x} = P^{-1}\mathbf{y} \tag{2.1}$$

where $P$is the preconditioner. The role of the preconditioner is to improve the system in order to achieve faster convergence. When solving sequences of shifted systems, a frequently used preconditioner is given by:

$$P = A - \sigma I \tag{2.2}$$

where σ is the shift parameter.

While direct methods such as Gauss-Jordan elimination can be used to invert or factor such a matrix P, these approaches are generally too computationally expensive for high-dimensional problems. Fortunately, in many practical applications, the matrix A remains fixed while only the right-hand side vector b changes. In such cases, it is advantageous to decompose the matrix A into a unit lower triangular matrix L and an upper triangular matrix U, also known as the LU decomposition. This factorisation allows for fast and repeated solutions of the system using forward and backward substitution.

This chapter introduces the concept of the LU decomposition and explores two algorithms for constructing it. It also outlines an efficient implementation of the forward and backward substitution to solve the resulting system. This will provide a basis for the development of the later parallel algorithms.

2.1. The LU decomposition

An LU decomposition of a matrix A represents the matrix as the product of a unit lower triangular matrix L and an upper triangular matrix U.

$$A = LU \tag{2.3}$$

The matrices L and U have the following forms:

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n,1} & \cdots & l_{n,n-1} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} u_{1,1} & \cdots & \cdots & u_{1,n} \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{n,n} \end{bmatrix} \tag{2.4}$$

The entries of the LU decomposition can be determined using the Gauss-Jordan elimination method. This process starts with the matrix A and an identity matrix I of the same dimensions. By applying row reduction to A, we obtain the upper triangular matrix U. To create the lower triangular matrix L, the same sequence of row operations is performed on the matrix I (Friedberg et al., 2014).

Once the LU decomposition is found, solving the linear system $A\mathbf{x} = \mathbf{b}$ becomes a lot easier for the computer:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ LU\mathbf{x} &= \mathbf{b} \\ \mathbf{x} &= U^{-1}(L^{-1}\mathbf{b}) \end{aligned} \tag{2.5}$$

Here $L^{-1}\mathbf{b} = \mathbf{y}$ is solved using forward substitution, and $U^{-1}\mathbf{y} = \mathbf{x}$ is solved using backward substitution.

2.1.1. Pivoting in the LU decomposition

The LU decomposition can also be written in the form

$$PA = LDU \tag{2.6}$$

where A is the original matrix, L is a unit lower triangular matrix, U is an unit upper triangular matrix and D the diagonal matrix. The permutation matrix P represents the row swaps applied to A during the LU decomposition, also known as pivoting.

Pivoting is an important step when the pivot element, the value used to eliminate the entries below the diagonal, is zero or close to zero. If the value is zero or close to zero, it can lead to division errors or rounding errors. To avoid this, rows are swapped so that the largest available element (in absolute value) in the current column below the diagonal becomes the pivot. These row swaps can be found in the matrix P

Although pivoting improves numerical stability and accuracy, it comes with additional computational time. In this paper, the LU decomposition is used for preconditioning rather than solving the system directly. So, numerical accuracy is less important than computational efficiency. For that reason, pivoting is not implemented in any of the algorithms presented in this paper.

2.2. Two algorithms to create the LU decomposition

As discussed in Section 2.1, an LU decomposition involves factoring a matrix A into the product of a unit lower triangular matrix L and an upper triangular matrix U. While one method for computing this decomposition has already been introduced, several alternative approaches exist. This section presents two sequential algorithms for computing the LU decomposition. The derivations and corresponding algorithms are based on the work of Rob Bisseling (2010) in *Parallel Scientific Computations: A Structured Approach Using BSP*. 2020

2.2.1. The sequential algorithm

The derivation of the algorithm begins by expanding the equation (2.3) as follows:

$$a_{ij} = \sum_{r=1}^{n} l_{ir} u_{rj} \tag{2.7}$$

This derivation uses Fortran-style indexing, so indices start at 1. Given that $l_{ir} = 0$ for $i < r$ and $u_{rj} = 0$ for $r > j$, the sum simplifies to:

$$\sum_{r=1}^{n} l_{ir} u_{rj} = \sum_{r=1}^{\min(i,j)} l_{ir} u_{rj} \tag{2.8}$$

In the case where $i \leq j$ and using that $l_{ii} = 1$, it follows that

$$\begin{aligned} a_{ij} = \sum_{r=1}^{\min(i,j)} l_{ir} u_{rj} &= \sum_{r=1}^{i} l_{ir} u_{rj} \\ &= l_{i1} u_{1j} + l_{i2} u_{2j} + \cdots + l_{i,i-1} u_{i-1,j} + l_{ii} u_{ij} \\ &= l_{i1} u_{1j} + l_{i2} u_{2j} + \cdots + l_{i,i-1} u_{i-1,j} + u_{ij} \\ &= \sum_{r=1}^{i-1} l_{ir} u_{rj} + u_{ij} \end{aligned} \tag{2.9}$$

Rearranging the terms gives an expression for the entries of the upper triangular matrix U.:

$$u_{ij} = a_{ij} - \sum_{r=1}^{i-1} l_{ir} u_{rj} \tag{2.10}$$

Similarly, from the case that $j \leq i$ follows that

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{r=1}^{j-1} l_{ir} u_{rj} \right) \tag{2.11}$$

Defining the matrix A^k with $0 \leq k \leq n$ as

$$\begin{aligned} a_{ij}^{k} &= a_{ij} - \sum_{r=1}^{k-1} l_{ir} u_{rj} \\ &= a_{ij}^{k-1} - l_{i,k-1} \cdot u_{k-1,j} \end{aligned} \tag{2.12}$$

It follows that $A^0 = A$ and $A^n = \mathbf{0}$. Based on this definition, the updates for L and U at each step are given by

$$\begin{aligned} u_{ij} &= a_{ij}^{k}, \quad \text{for } 1 \leq i < j \leq n \\ l_{ij} &= \frac{a_{ij}^{j}}{u_{jj}}, \quad \text{for } 1 \leq j < i \leq n \end{aligned} \tag{2.13}$$

Using the equations (2.12) and (2.13), Algorithm 2.1 can be created. This algorithm takes the square matrix A as input and has as output the unit lower triangular matrix L and upper triangular matrix U, such that $A = LU$.

Algorithm 2.1: Sequential LU decomposition

$input:$ $A : n \times n$ matrix
$output:$ $L : n \times n$ lower triangular matrix with ones on the main diagonal
$U : n \times n$ upper triangular matrix

$A^0 := A$
for *k := 1* ***to*** *n* **do**
  for *j := k* ***to*** *n* **do**
    $u_{kj} = a^k_{kj}$
  end
  for *i := k+1* ***to*** *n* **do**
    $l_{ik} = a^k_{ik} / u_{kk}$
  end
  for *i = k+1* ***to*** *n* **do**
    for *j = k+1* ***to*** *n* **do**
      $a^{k+1}_{ij} = a^k_{ij} - l_{ik} \cdot u_{kj}$
    end
  end
end

2.2.2. The memory-efficient sequential algorithm

Algorithm 2.1, as described in the previous section, constructs the LU decomposition by generating two new $n \times n$ matrices, one for L and one for U. Together with the original matrix A, this algorithm needs to store three $n \times n$ matrices. While this is fine for small values of n, it quickly becomes impractical when the dimensions become higher.

A more memory-efficient approach is to store the matrices L and U within a single matrix. This is possible because L is a lower triangular matrix with ones on the main diagonal, and U is an upper triangular matrix. Since the diagonal elements of L are known to be ones, they do not need to be stored explicitly. Therefore, the lower diagonal part of the matrix can store the entries of the matrix L, and the upper diagonal part, including the main diagonal, can store the entries of the matrix U.

Moreover, all the information needed for step k can be stored in the same matrix as well. Figure 2.1 provides an example of this combined storage structure during step $k = 3$ of the decomposition. As shown, the computed entries of L and U are written in place, and the remaining part of the matrix holds the values needed to complete the decomposition. This approach, referred to as the in-place LU decomposition, only requires storage for one $n \times n$ matrix, making it significantly more memory-efficient.

To implement this in-place approach, the update equations (2.12) and (2.13) have to be changed. These become:

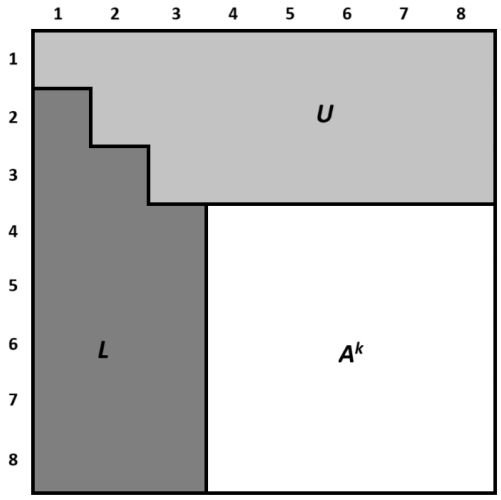


Figure 2.1: The LU decomposition of a 8×8 matrix at step $k = 3$. It shows the entries of L and U that have been computed (Bisseling, 2020)

$$
\begin{aligned}
u_{ij} = a_{ij}^{k} \quad &\Longrightarrow \quad a_{ij} = a_{ij} \quad \text{for } 1 \leq i < j \leq n \\
l_{ij} = \frac{a_{ij}^{j}}{u_{jj}} \quad &\Longrightarrow \quad a_{ij} = \frac{a_{ij}}{a_{jj}} \quad \text{for } 1 \leq j < i \leq n \\
a_{ij}^{k} = a_{ij}^{k-1} - l_{i,k-1} \cdot u_{k-1,j} \quad &\Longrightarrow \quad a_{ij} = a_{ij} - a_{ik} \cdot a_{kj}
\end{aligned}
\tag{2.14}
$$

These modified equations ensure the algorithm performs all computations directly in the original matrix and therefore minimising the memory needed. Based on these equations, the memory-efficient algorithm, Algorithm 2.2, can be implemented.

Algorithm 2.2: Memory-efficient sequential LU decomposition

$input:$ $A : n \times n$ matrix
$output:$ $A : n \times n$ matrix, $A = L - I_n + U$ with
$L : n \times n$ lower triangular matrix
$U : n \times n$ upper triangular matrix

```
for k := 1 to n do
    for i := k+1 to n do
        a_ik = a_ik / a_kk
    end
    for i := k+1 to n do
        for j := k+1 to n do
            a_ij = a_ij − a_ik · a_kj
        end
    end
end
```

Matrix size n	Average computation time Algorithm 2.1 (s)	Average computation time Algorithm 2.2 (s)
10	$3.300 \cdot 10^{-6}$	$2.699 \cdot 10^{-6}$
100	$1.195 \cdot 10^{-3}$	$9.834 \cdot 10^{-4}$
250	$1.798 \cdot 10^{-2}$	$1.500 \cdot 10^{-2}$
500	0.171	0.141
1000	2.316	1.310
1500	11.399	7.348
2000	31.036	23.503

Table 2.1: The average computation time of Algorithms 2.1 and 2.2 for varying sizes of A

2.2.3. Comparison of the two algorithms

To evaluate whether Algorithm 2.2 performs better than Algorithm 2.1, a series of tests were performed on a personal laptop. Both algorithms were implemented in Fortran. The experiment involved computing the LU decompositions of square matrices of varying sizes using each algorithm.

For each matrix size, the CPU time required to complete the decomposition was measured. To ensure reliability, each test was repeated 100 times, and the average of these results was taken. This method ensures a fairer comparison between the two algorithms.

Table 2.1 presents the average computation times for both sequential algorithms across a range of matrix sizes. The results indicate that Algorithm 2.2 consistently outperforms Algorithm 2.1 in terms of speed, regardless of n. This performance gap becomes especially pronounced as the matrix size increases to $n = 1500$, where the difference in execution time becomes substantial. Because of its efficiency, Algorithm 2.2, the memory-efficient variant, will become the foundation for developing the parallel versions of the LU decomposition.

From a theoretical standpoint, one expects the average computation time of both algorithms to scale with n^3, since the LU decomposition has a time complexity of $\mathcal{O}(n^3)$. This growth is observed for the smaller matrices in Table 2.1. However, as n increases beyond 500, the scale does not work anymore. This increase in computation time is primarily due to memory limitations on the laptop. Once the matrix becomes larger, the data can no longer be entirely stored in the cache and must instead be accessed from the main memory.

2.3. Backward and Forward substitution

Once the LU decomposition of the matrix A is created, the linear system $A\mathbf{x} = \mathbf{b}$ can be solved using forward and backward substitution.

The first step involves solving the system

$$L\mathbf{y} = \mathbf{b} \tag{2.15}$$

using forward substitution. This method computes the values of $\mathbf{y}$ sequentially, starting with the first equation. The first equation is straightforward, because the matrix L is a lower triangular matrix with ones on its main diagonal:

$$y_1 = b_1 \tag{2.16}$$

The following values y_i are computed using the values of the previously computed y_i's. Specifically using the formula:

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j \tag{2.17}$$

The second step involves solving the system

$$U\mathbf{x} = \mathbf{y} \tag{2.18}$$

using backward substitution. This method, like the forward substitution, also computes the vector values sequentially. However, instead of starting at the first equation, backward substitution starts at the last equation. The first computation is still straightforward because the matrix U is an upper triangular matrix. However, the main diagonal does not necessarily have ones, so to get the value of x_n, it also has to be divided by u_{ii}.

$$x_n = y_n / u_{ii} \tag{2.19}$$

The other vector values are computed using

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii} \tag{2.20}$$

Algorithm 2.3 shows an implementation of the forward and backward substitution.

Algorithm 2.3: Forward and Backward substitution

$input:$ $L : n \times n$ unit lower triangular matrix
$U : n \times n$ upper triangular matrix
$\mathbf{b} : n \times 1$ vector
$output:$ $\mathbf{x} : n \times 1$ vector

$x = b$
for *i := 1, n* **do**
  for *j := 1, i-1* **do**
    $x_i = x_i - l_{i,j} \cdot x_j$
  end
end
for *i := n, 1, -1* **do**
  for *j := i+1, n* **do**
    $x_i = x_i - u_{i,j} \cdot x_j$
  end
  $x_i = x_i / u_{i,i}$
end

3

Parallelising the LU decomposition

While Chapter 2 explored two sequential algorithms for computing the LU decomposition, high-performance implementations often require higher efficiency than the sequential algorithms can provide. By using parallel computation, the LU decomposition can be performed significantly faster, especially for high-dimensional matrices.

This chapter introduces the foundations for parallelising LU decomposition, beginning with a discussing of the software. It then discusses different ways to distribute matrices across different processors or images. Then it presents a parallel implementation using Fortran Coarrays. After the first algorithm is created, the improvements that can be made to the algorithm are discussed before finally ending the chapter with a parallel implementation of the forward and backward substitution.

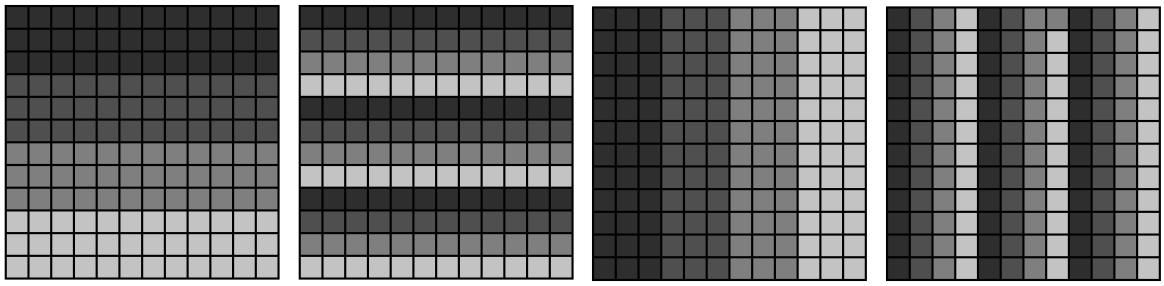
3.1. Fortran Coarray

The first step in creating a parallel implementation of the LU decomposition is selecting an appropriate parallel programming method. Several options exist for parallel programming, including OpenMP and MPI. However, since the goal is to accelerate the LU decomposition in a Fortran-based package, the algorithms make use of Fortran Coarrays, a parallel programming feature built directly into the Fortran language (Fanfarillo et al., 2014).

Created in 2008 by Robert Numrich and John Reid, Fortran Coarrays were designed to simplify parallel programming by integrating it into the language. Coarrays enable parallel execution by introducing images that can run at the same time and exchange data through coarray variables. The synchronisation between the images is handled using statements such as `sync all`, which synchronises all images, or `sync images`, which only synchronises a specified set of images.

3.2. Distributing the matrix over images

During the LU decomposition, each image must have access to the data required for its computations. Because images in Fortran Coarrays operate with separate local memory, data stored on one image is not automatically accessible to other images.
A naive approach would be to replicate the entire matrix on every image, which ensures that every image has all the data it will need. However, this method is highly inefficient in terms of memory usage, especially for high-dimensional matrices. Instead, the matrix should be dis-



(a) A block distribution over the rows (b) A cyclic distribution over the rows (c) A block distribution over the columns (d) A cyclic distribution over the columns

Figure 3.1: A visualisation of the block and cyclic distributions over the rows and columns

tributed, with each image only storing a portion of the data. The distribution of the matrix can be done in several ways.

A common and straightforward way is to distribute the matrix across its rows. In doing so, it must be decided how to assign specific rows to individual images. One method is the block distribution, shown in Figure 3.1a. Here, each image is allocated a block of consecutive rows. For example, given 12 rows and four images, image 1 would receive rows 1-3, image two would receive rows 4-6, and so on.

Alternatively, the rows can be distributed in a cyclic manner, as shown in Figure 3.1b. In this distribution scheme, the rows are assigned to images in a cyclic fashion. Continuing with the example of the 12×12 matrix, image 1 would hold rows 1, 5, and 9, image 2 would hold rows 2, 6, and 10, and so on.

The same applies if the matrix is distributed over its columns rather than rows. Both block and cyclic distributions can be used in the column-wise schemes, as shown in Figures 3.1c and 3.1d, respectively.

A more advanced distribution is the Cartesian distribution, as described by Rob Bisseling (Bisseling, 2020). In this distribution scheme, the matrix is mapped onto a two-dimensional grid of images, each responsible for a rectangular sub-block of the matrix. This method allows for a balanced distribution of both the rows and columns across the images, which minimises the communication between the images and improves performance.

The images are arranged in an $M \times N$ grid, where $M \cdot N$ equals the total number of images:

$$I(s,t), \quad with\ 0 \leq s \leq M\ and\ 0 \leq t \leq N \quad M \cdot N = \#images \tag{3.1}$$

In this scheme, each image is defined by a pair of coordinates (s,t) that corresponds to its position in the grid. The matrix entries are then mapped to these coordinates using a distribution function:

$$\phi : \{(i,j) : 0 \leq i,j < n\} \rightarrow \{(s,t) : 0 \leq s < M \ \wedge \ 0 \leq t < N\} \tag{3.2}$$

A key part of the Cartesian distribution is that the assigning of the row and columns do not depend on each other, which means that the mapping function $\phi(i,j)$ can be written as two independent functions ϕ_0 and ϕ_1:

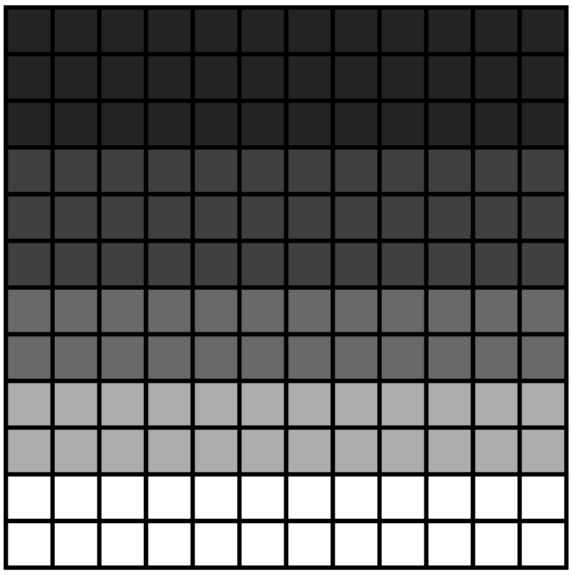


Figure 3.2: A block-wise row distribution of a 12×12 matrix over 5 images

$$\phi(i,j) = (\phi_0(i), \phi_1(j)), \quad 0 \leq i,j < n \tag{3.3}$$

This structure implies that the distribution for ϕ_0 depends on the row number i and ϕ_j depends on the column number j. This means that the rows could be distributed using a cyclic distribution, and the columns distributed using a block distribution.

3.2.1. The distribution used in the algorithm

As described in Section 3.2, there are several possible schemes for distributing the matrix across the images: block and cyclic distributions for rows or columns, and the Cartesian distribution. In this implementation, a row-wise distribution is chosen. This decision is driven by the fact that, in the Fortran package the algorithm is designed for, the matrices are stored in a row-wise structure. This makes a row-wise distribution a natural fit. Additionally, a block distribution is chosen over the cyclic one to reduce the communication between the images during the LU decomposition.

However, a block-wise distribution comes with its own difficulties when the number of rows is not evenly divisible by the number of images. A naive approach to address this would be to add zero rows to the matrix until the total number of rows becomes divisible by the number of images. This, however, will lead to inefficiencies, particularly for the last image. This image might be assigned a large number of zero rows and therefore do little of the work.

To address this, a more even distribution is chosen. The maximal number of rows assigned to each image is determined as follows:

$$max\_chunk\_size = \left\lceil \frac{\#rows}{\#images} \right\rceil \tag{3.4}$$

and the remainder is computed as

$$remainder = \#rows \mod \#images \tag{3.5}$$

The first `remainder` images are assigned chunks of size `max_chunk_size` and remaining images receive chunks that are one row smaller. This ensures a more evenly distributed load across the images.

Figure 3.2 shows an example of this approach. It is a distribution of a 12×12 matrix across 5 images. The result is an almost even division of the rows with at most a one-row difference between any two images.

3.3. Basic parallel algorithm

Once the matrix has been distributed across the images, the LU decomposition can be computed. The parallel algorithm presented here is based on Algorithm 2.2 form Section 2.2.2 and computes the decomposition in place as well to reduce the memory needed. The input to the algorithm is the part of the matrix owned by the current image, referred to as `my_chunk`. The output is the modified version of the chunk, which contains the corresponding part of the LU decomposition.

In the LU decomposition, the main computation at each step involves selecting the pivot row, specifically row k at step k, and using it to eliminate the entries below the pivot. In the sequential case, this is a very straightforward operation because all the rows are in a shared memory and their locations are immediately known. However, in the parallel case, the matrix is distributed across the images and each image only has access to its local part. Since images cannot directly access each other's memory, the location of the pivot row needs to be determined and it has to be shared when necessary.

The first step in each iteration is to determine which image owns the pivot row. Because the matrix is block-distributed, the location of a row depends on the total number of rows, the number of images and the remainder from dividing the number of rows by the number of images. This can be implemented as follows:

```
if (k<= remainder *(n/nimg +1)) then
    owner = (k-1)/(n/nimg +1) +1
else
    owner = remainder + (k-1 - remainder *(n/nimg+1))/(n/nimg) +1
end if
```

This function determines the image index, which starts at 1 in Fortran, that holds row k.

Once the owner of the pivot row is found, that image must share the pivot row with all the other images. This is done using the `broadcast` function build into Fortran. This sends the row from the owning image to all the other images and also includes a synchronisation function to ensure all the images reach the same point of computation before continuing.

After the pivot row has been received by the other images, each image can update its own local rows. To determine if a row needs to be updated, the global row number corresponding to each local row index is calculated as follows:

```
    global_row = start_row + i - 1
```

Only rows with a global index greater than k, those below the pivot row, have to do the computation steps. Rows above or equal to the pivot row remain unchanged in this iteration.

This sequence of steps leads to the development of Algorithm 3.1, which shows the basic structure of the parallel LU decomposition using Fortran Coarrays.

Algorithm 3.1: Parallel LU decomposition

$input:$ $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image
$output:$ $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image updated so it contains the entries of the LU decomposition

```
me = this_image()
for k := 1 to n do
    owner = owner(row_k)
    pivot_row = row_k

    broadcast(pivot_row)

    for i := 1 to chunk_size do
        global_row = global_row(i)        # Finds the global row number of row i

        if global_row > k then
            my_chunk(i,k) = my_chunk(i,k)/pivot_row(k)
            my_chunk(i,k+1:n) = my_chunk(i,k+1:n)
                                −my_chunk(i,k) · pivot_row(k+1:n)
        end
    end
end
```

3.4. Improving the algorithm

Intermediary tests confirm that the basic parallel LU decomposition functions properly. However, several optimisation can still be made to improve its performance and efficiency.

In the original implementation, certain variables, such as the starting row index and the chunk size, were recomputed in each iteration. This was eliminated by computing these values once at the beginning of the algorithm and reusing them throughout the whole algorithm. This reduces unnecessary computations and could lead to a more efficient execution, especially for large matrices.

An additional condition was introduced to further minimise unnecessary work. The previous algorithm only checked if a row's global index was greater than the current pivot index k before performing the computations. However, in cases where the pivot entry of a given row is already zero, the subtraction operation has no effect. For this, the following condition was added:

```
if (global_row>k .and. my_chunk(i,k)/=0) then
```

This check can lead to noticeable performance improvements if the matrix is sparse. For dense matrices, the effect on the runtime will be minimal, but might still get rid of some unnec-

essary computations.

Another improvement involves replacing the standard do loop with a do concurrent loop at each iteration. This informs the compiler, and the GPU, that the iterations of the loop are independent of each other and can be executed in parallel. The benefit of this will be limited on CPUs, but it can significantly increase efficiency if the algorithm is run on a GPU.

In each iteration k, the first $k-1$ entries of the pivot row are zero. This makes them irrelevant for the computations, because they will want to do work on positions that are already eliminated. Therefore, instead of broadcasting the entire row of length n, it is more efficient to share only part of the vector, starting at k and ending at n. This reduces the amount of data that has to be transferred between the images, especially in the later iterations, which can lead to a significant increase in performance for large matrices.

The last improvement has to do with the `broadcast` function. This function sends the pivot row to all image, even if the images do not need the information anymore. In many cases, images holding only rows above the current pivot row have already completed their work and do not require toe pivot row. To avoid the unnecessary sharing of information, a custom communication loop using the Fortran Coarrays was implemented.

```
do i = owner+1, number_images
    pivot_row(k:n)[i] = pivot_row(k:n)[owner]
end do
```

This loop shares the pivot row only with the images that hold rows below the pivot. While the improvement is small for lower-dimensional matrices, it becomes significant when scaling up to many images.

These optimisations come together in Algorithm 3.2, the improved parallel LU decomposition algorithm. By reducing communication costs and eliminating redundant computations, this version is more efficient and is also more scalable than its counterpart.

Algorithm 3.2: Efficient parallel LU decomposition

$input$: $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image

$output$: $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image updated so it contains the entries of the LU decomposition

```
me = this_image()
for k := 1 to n do
    owner = owner(row_k)
    pivot_row = row_k

    for i:=owner+1 to number_images do
        pivot_row(k:n)[i] = pivot_row(k:n)[owner]
    end

    for i := 1 to chunk_size .and. my_chunk(i,k) ≠0 do
        global_row = global_row(i)          # Finds the global row number of row i

        if global_row > k then
            my_chunk(i,k) = my_chunk(i,k)/pivot_row(k)
            my_chunk(i,k+1:n) = my_chunk(i,k+1:n)
                                −my_chunk(i,k) · pivot_row(k+1:n)
        end
    end
end
```

3.5. Parallel forward and backward substitution

As in Chapter 2, once the LU decomposition has been computed, the next step is solving the system

$$LU\mathbf{x} = \mathbf{y} \tag{3.6}$$

through forward and backward substitution. Since the matrix remains distributed across multiple images, these substitution steps must also be parallelised.

To solve the equation

$$L\mathbf{y} = \mathbf{b} \tag{3.7}$$

each value y_i requires access to the corresponding right-hand side entry b_i, row i of the lower triangular matrix L, and the previously computed entries of the vector $\mathbf{y}$, as shown in equation (2.17). To minimise communication between images, the algorithm assigns the computation of y_i to the image that owns row i of L. This ensures that only the necessary previously computed values of $\mathbf{y}$ have to be shared across the images.

The algorithm starts by setting $\mathbf{y} = \mathbf{b}$. It then proceeds to loop over all the row indices k, where each image checks if it owns the k-th row. If so, this image computes the value y_k. This computed value is then shared with all other images so they can use it in their computations. The synchronisation and communication steps of this algorithm are very similar to those of the LU decomposition.

Once $\mathbf{y}$ has been computed, the next step is to solve

$$U\mathbf{x} = \mathbf{y} \tag{3.8}$$

using backward substitution, as described by equation (2.20). Similar to the forward substitution, the value x_i depends on y_i, row i of the upper triangular matrix U and the already computed values of $\mathbf{x}$. Therefore, the parallel implementation can follow the same logic as used in the algorithm of the forward substitution.

The primary difference lies in the order of the outer loop. Backward substitution proceeds from the last row to the first. In other words, the loop starts at row index n and ends at index 1. As before, the image that owns row k is responsible for computing x_k, and the result is shared with the other images. Initially, the algorithm sets $\mathbf{x} = \mathbf{y}$. At each step, the relevant value of x_k is computed and then communicated to the other images for their local updates.

Together, these algorithms solve the system $LU\mathbf{x} = \mathbf{b}$. To optimise memory usage, the substitution steps can be performed in-place. However, overwriting the input vector $\mathbf{b}$ might not be wanted, because it might be needed later in the program. Therefore, the algorithm performs the forward substitution as described, storing the result in $\mathbf{y}$. The algorithm then performs the backward substitution in-place, updating $\mathbf{y}$ directly instead of copying it over to $\mathbf{x}$.

This process results in Algorithm 3.3, the parallel forward and backward substitution algorithm, which computes the solution of the linear system when the original matrix is distributed.

Algorithm 3.3: Parallel Forward and Backward substitution

$input:$ $my\_chunk : max\_chunk\_size \times n$: the block of the LU decomposed matrix owned by the image
$b\_chunk : max\_chunk\_size \times 1$: the block of the right-hand side vector **b** owned by the image

$output:$ $y\_chunk : max\_chunk\_size \times 1$: the block of the solution vector owned by the image

$y\_chunk = b\_chunk$

for *k:= 1* ***to*** *n* **do**
  owner = `owner`(row_k)
  $pivot\_y = y_k$

  for *i:=owner+1* ***to*** *number_images* **do**
    $pivot\_y[i] = pivot\_y[owner]$
  end

  for *i:=1* ***to*** *chunk_size* **do**
    global_row = global_row(i) # Finds the global row number of row i

    if global_row $> k$ **then**
      $y\_chunk(i) = y\_chunk(i) - pivot\_y \cdot my\_chunk(i,k)$
    end
  end
end
`sync all`
for *k:= n* ***to*** *1* **do**
  owner = `owner`(row_k)
  $pivot\_y = y_k / my\_chunk(k,k)$

  for *i:=1* ***to*** *owner-1* **do**
    $pivot\_y[i] = pivot\_y[owner]$
  end

  for *i:=chunk_size* ***to*** *1* **do**
    global_row = global_row(i) # Finds the global row number of row i

    if global_row $< k$ **then**
      $y\_chunk(i) = y\_chunk(i) - pivot\_y \cdot my\_chunk(i,k)$ **end**
  end
end

4

Creating the LU decomposition for banded matrices

The previous chapters described the development of the parallel LU decomposition algorithm for dense matrices. However, in real-world and physical applications, matrices are often sparse. Some of these sparse matrices also have specific structures, such as banded matrices, where the non-zero entries are concentrated around the main diagonal. This chapter explores some differences between dense and banded matrices, presents a parallel algorithm for fully stored banded matrices, and then introduces a more efficient approach for compactly stored banded matrices.

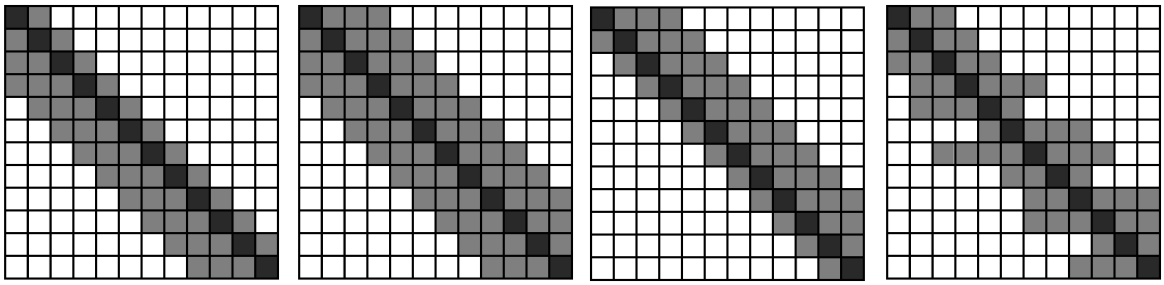
4.1. Differences between banded matrices and dense matrices

The first step in adapting the LU decomposition for banded matrices is understanding how the band matrix differs from a dense matrix. A dense matrix is a matrix in which most elements are non-zero and do not have a pattern of sparsity. Looking at the storage of a dense matrix, all entries of the matrix have to be stored, resulting in n^2 entries for an $n \times n$ matrix. On top of that, operations like the LU decomposition involve updating the entire matrix.

On the other hand, a banded matrix is a type of sparse matrix. The non-zero elements of the matrix are concentrated around the main diagonal. This structure appears in many scientific problems, especially in the discretisation of partial differential equations using the finite difference method. In the LU decomposition, the different iterations only involve updating certain neighbouring elements of the matrix, instead of updating the entire matrix. For a banded matrix, the result of the in-place LU decomposition is also a banded matrix.

While many band matrices are structurally symmetric, meaning that they have the same number of non-zero diagonals above and below the main diagonal, see Figure 4.1b, this is not always the case. In practice, band matrices can be structurally asymmetric, with a different number of lower and upper diagonals. For example, a matrix might have more non-zero diagonals above the main diagonal than below, see Figure 4.1c, which depends on the physical problem. On top of that, the bandwidth does not have to be constant across all the rows. Some rows may have more non-zero entries, and others may have fewer. This leads to a variable or irregular bandwidth, which can appear in problems with non-uniform grids.

A key concept when working with banded matrices is their bandwidth. The lower bandwidth k_1 is the number of sub-diagonals below the main diagonal containing non-zero values. The



(a) A band matrix with more diagonals below the main diagonal

(b) A structurally symmetric band matrix

(c) A band matrix with more diagonals above the main diagonal

(d) A band matrix with irregular bandwidth

Figure 4.1: A visualisation of the different kinds of band matrices. The dark diagonal is the main diagonal, the light gray are non-zero elements and the white blocks are zero elements.

upper bandwidth k_2 is the number of super-diagonals above it. In structurally symmetrically banded matrices, the lower bandwidth is equal to the upper bandwidth, $k_1 = k_2$. The total bandwidth is defined as $k_1 + k_2 + 1$, and the bandwidth is defined as the maximum of the upper and lower bandwidth.

$$bandwidth = \max(k_1, k_2) \tag{4.1}$$

For irregular band matrices, like the example shown in Figure 4.1d, the bandwidth is the maximum of the maximum lower bandwidth and the maximum upper bandwidth, which would make the bandwidth of that matrix 4.

4.2. The maximal_bandwidth function

Determining the bandwidth of a matrix plays a crucial role in optimising the parallel algorithm, especially for sharing the pivot row during the LU decomposition. The bandwidth informs us of the influence each pivot row has, allowing for more efficient data sharing between images by not sending the data to images holding rows outside of the band.

In the cases shown in the Figures 4.1a, 4.1b, and 4.1c, computing the bandwidth is very straightforward because the number of diagonals below and above the main diagonal is constant. However, finding the bandwidth is more complex when it is not constant, which is the case in Figure 4.1d.

To find the bandwidth of such a matrix, a function is implemented that computes the bandwidth of any square matrix. It does not matter if it is a sparse or a dense matrix. This function iterates over each row of the matrix, finds the furthest non-zero elements to the left and right of the main diagonal, and then computes the corresponding lower and upper bandwidths for that row. After iterating through all the rows, the function returns the maximum values it found for the lower and upper bandwidth. The overall bandwidth is the maximum of those two values.

A pseudo-code of this function is presented in Algorithm 4.1, the maximal bandwidth function.

Algorithm 4.1: The maximal_bandwidth function

$input:$ $A : n \times n$: the full matrix
$output:$ $maximal$: the bandwidth of the matrix A

for *i:= 1 to n* **do**
    $first\_nonzero = i$
    $last\_nonzero = i$

    for *j:=i+1 to n* **do**
        If $a_{ij} \neq 0$ **then**
            $last\_nonzero = j$
        end
    end

    for *j:=i-1 to 1* **do**
        If $a_{ij} \neq 0$ **then**
            $first\_nonzero = j$
        end
    end
    $first\_nonzero = i - first\_nonzero$
    $last\_nonzero = last\_nonzero - i$

    $lower\_bandwidth = \max(lower\_bandwidth, first\_nonzero)$
    $upper\_bandwidth = \max(upper\_bandwidth, last\_nonzero)$
end
$maximal = \max(lower\_bandwidth, upper\_bandwidth)$

4.3. Updating the algorithm for full matrices with bands

Once the bandwidth of the matrix has been identified, the LU decomposition of the banded matrix can be performed using Algorithm 3.2. However, the banded nature of the matrix introduces sparsity that can be used to optimise the algorithm.

The first improvement looks at the size of the pivot row that is communicated between images. In Algorithm 3.2, the entire tail of the pivot row, from column k to n, is shared. In the banded matrices, a big part of the tail will be zero entries. To improve this, the upper bandwidth of row k is computed during each iteration. In matrices with a regular bandwidth, this value remains constant. However, to maintain robustness, the check is applied at each step for all banded matrices. By limiting the communication to only the necessary entries, only a vector of size $upper\_bandwidth + 1$ is sent instead of a vector of size $n - k + 1$. This reduces communication and increases the efficiency of the algorithm, especially if the matrix has a small bandwidth.

The second optimisation looks at which images receive the pivot row. In the algorithm for dense matrices, the pivot row is broadcast to all images that own rows below the pivot row. This is unnecessary for banded matrices, since only rows within the bandwidth below the pivot can be affected. So, instead of sending the pivot row to all images, the result of the `maximal_bandwidth` function can be used to find how far the pivot row might influence.

The corresponding image that owns the final row is identified, and the pivot row is only sent to images up to that point. This reduces the communication between images and improves the efficiency, especially if one uses a large number of images.

The last improvement looks at the update of the rows. In Algorithm 3.2, every image loops over all the rows it owns, even when none fall within the band that is affected by the pivot row. To avoid this, a check is added before the loop is entered. If the image does not own any rows that fall within the region that is influenced by the pivot row, the update step is skipped entirely.

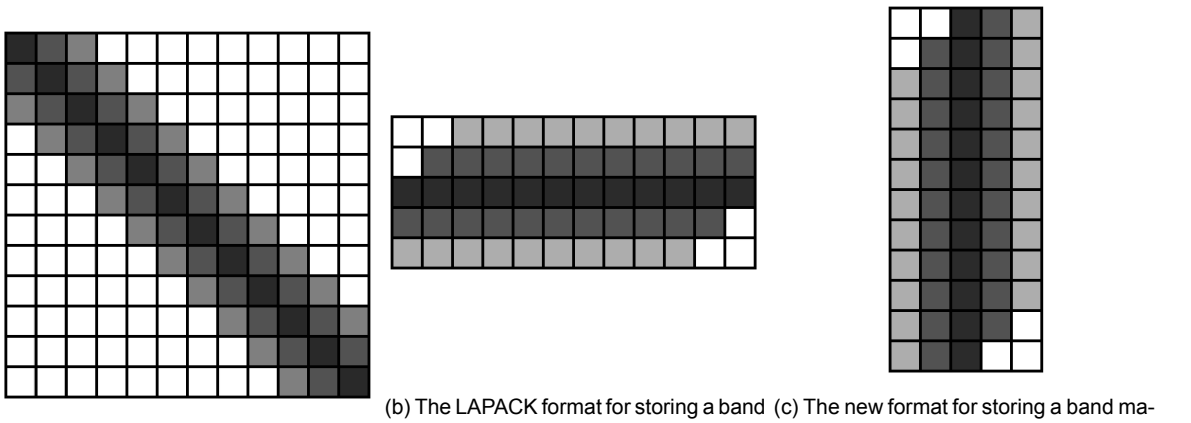
These improvements result in a more efficient LU decomposition. The updated algorithm is presented in Algorithm 4.2, which is the Parallel LU decomposition for banded matrices.

Algorithm 4.2: Parallel LU decomposition for full banded matrices

$input:$ $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image
$bandwidth$: the bandwidth of the matrix A

$output:$ $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image updated so it contains the entries of the LU decomposition

```
me = this_image()
for k := 1 to n do
    owner = owner(row_k)
    last_owner = owner(row_{k+bandwidth})

    for i:= k+1 to n do
        If pivot_row(i)≠ 0
            end_row = i
        end
    end

    pivot_row(k : end_row) = row_k(k : end_row)

    for i:=owner+1 to last_owner do
        pivot_row(k:end_row)[i] = pivot_row(k:end_row)[owner]
    end

    If image ≥ owner .and. image ≤ owner_last
        for i := 1 to chunk_size .and. my_chunk(i,k) ≠0 do
            global_row = global_row(i)        # Finds the global row number of row i

            if global_row > k then
                my_chunk(i,k) = my_chunk(i,k)/pivot_row(k)
                my_chunk(i,k+1 : n) = my_chunk(i,k+1 : n)
                                      −my_chunk(i,k) · pivot_row(k+1 : n)
            end
        end
    end
end
```



(a) A band matrix with five bands.

(b) The LAPACK format for storing a band matrix

(c) The new format for storing a band matrix

Figure 4.2: A visualisation of the different kinds of band matrices. The dark diagonal is the main diagonal, the light gray are non-zero elements, and the white blocks are zero elements.

4.4. Memory-efficient band matrices

Since a band matrix is a sparse matrix with a specific structure, it can be stored more efficiently than a full dense matrix. Instead of allocating memory for all the $n \times n$ entries of the matrix, only the diagonals containing non-zero values need to be stored. This section introduces an efficient storage method for banded matrices and presents adaptations for the LU decomposition and the forward and backward substitution.

4.4.1. Storing the band matrix

A common method for storing banded matrices is the LAPACK (Linear Algebra Package) band storage method. In this method, only the diagonals that are within the band are stored, which reduces the memory usage. To be specific, for a matrix of size $n \times n$ with a bandwidth of $k = \max(k_1, k_2)$, the matrix is stored in a matrix of size $(2k + 1) \times n$. Each row of the reduced matrix corresponds to one of the diagonals of the original matrix (Anderson et al., 1999). A visual representation of the standard LAPACK band storage method is shown in Figure 4.2b.

However, while the LAPACK format is space-efficient and used in a variety of packages, it is not suited to the structure of the parallel LU decomposition algorithm described in this paper. The main problem is that LAPACK stores the diagonals as rows, which means the algorithms will need to traverse over the columns instead of the rows. The algorithms described in the previous chapters use a row-wise distribution model, where each image contains a block of rows and shares the pivot rows when needed. Using the LAPACK format for these algorithms would require fully redoing the data distribution and key parts of the algorithms, making the implementation very complex.

To maintain the efficiency of the LAPACK format but also have compatibility with the earlier algorithms created, a modified banded storage method is used: the Compressed Diagonal Storage (CDS) (Stathis, 2004). In this method, the diagonals of the matrix are stored as columns instead of rows. This is equivalent to a transposed version of the LAPACK format, but has the advantage of being compatible with the row-wise distribution method. The storage requirements remain the same; the only thing that changes is the orientation of the matrix so it better fits with the algorithms' structure. A visualisation of this storage scheme is shown in Figure 4.2c.

4.4.2. Memory-efficient algorithm

Once the storage format of the banded matrix has been decided, the LU decomposition algorithm can be adapted to the storage format. Due to the differences in structure between the compact banded storage and the full matrix used in the earlier algorithms, a few modifications are necessary. The most significant change occurs in the update step of the algorithm. If the banded matrix is stored in a full matrix, elements a_{ij} and $a_{i+1,j}$ are elements in the same column and are adjacent. In the compact storage format, however, the elements in the same column are entries from the same diagonal, not the same column.

To address this, a `shift` variable is introduced. This variable calculates the offset between the update row i and the pivot row k using the global row indices:

```
shift = global_row - k
```

The shift determines how far the update row has to be adjusted to do its updates correctly. It is also used to see if the row has to be updated by adding it to a check. If the shift is larger than the bandwidth of the matrix, the update row is outside of the bandwidth and does not need to be updated.

On top of that, a `centre` variable is used to identify the column corresponding to the main diagonal. Combining these two variables ensures that the updates are applied correctly when using the compact banded storage.

With these changes, the parallel LU decomposition algorithm can be used when a banded matrix is stored efficiently. This updated method is further described in Algorithm 4.3, the memory-efficient parallel LU decomposition for banded matrices.

4.4.3. Forward and backward substitution

As described in Chapter 3, once the LU decomposition is computed, the next step is solving the linear system using forward and backward substitution. For the basic parallel LU decomposition applied to banded matrices that are stored in a full matrix, no changes had to be made. However, when working with the compactly stored banded matrices, adaptations are needed because of the different matrix structure.

The adaptations to the forward and backward substitution algorithms are similar to the changes made to the LU decomposition. A `shift` variable is added to compute the offset between the current update row and the pivot row. On top of that, a `centre` variable is used to find the column in the storage format that corresponds to the main diagonal. During the forward substitution, the `shift` variable is subtracted from the `centre` variable to find the correct entry in the matrix. In the backward substitution, the `shift` is added instead.

An additional check is included to ensure that the update row lies within the bandwidth. This check prevents unnecessary computations by comparing the absolute value of the `shift` variable to the known bandwidth of the matrix.

These adaptations result in a forward and backward substitution algorithm that can be used with compactly stored banded matrices. This algorithm is presented in Algorithm 4.4, ”Parallel Forward and Backward Substitution for Memory-Efficient Banded Matrices.”

Algorithm 4.3: Memory efficient parallel LU decomposition for banded matrices

$input$: $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image
$bandwidth$: the bandwidth of the matrix A

$output$: $my\_chunk : max\_chunk\_size \times n$: the block of the matrix owned by the image updated so it contains the entries of the LU decomposition

me = `this_image()`
for *k := 1* ***to*** *n* **do**
  owner = `owner`(row_k)
  last_owner = `owner`$(row_{k+bandwidth})$
  centre = column(main_diagonal)

  $pivot\_row(1 : band\_width)$ = $row_k(centre : 2 \cdot band\_width + 1)$

  for *i:=owner+1* ***to*** *last_owner* **do**
    pivot_row(1:band_width)[i] = pivot_row(1:band_width)[owner]
  end

  If image ≥ owner **.and.** image ≤ owner_last
    for *i := 1* ***to*** *chunk_size* ***.and.*** *my_chunk(i,k) ≠0* **do**
      global_row = global_row(i) # Finds the global row number of row i
      shift = global_row - k
      if global_row > k **.and.** shift ≤ bandwidth **then**

$$my\_chunk(i, centre - shift) = my\_chunk(i, centre - shift)/pivot\_row(1)$$
$$start = centre - shift + 1$$
$$ending = start + bandwidth$$
$$my\_chunk(i, start : ending) = my\_chunk(i, start : ending) - my\_chunk(i, centre - shift) \cdot pivot\_row(2, 2 \cdot bandwidth + 1)$$

      end
  end
end

Algorithm 4.4: Parallel Forward and Backward substitution for memory-efficient banded matrices

$input:$ $my\_chunk : max\_chunk\_size \times n$: the block of the LU decomposed matrix owned by the image
$b\_chunk : max\_chunk\_size \times 1$: the block of the right-hand side vector **b** owned by the image

$output:$ $y\_chunk : max\_chunk\_size \times 1$: the block of the solution vector owned by the image

$y\_chunk = b\_chunk$
centre = column(main_diagonal)

for *k:= 1* ***to*** *n* **do**
  owner = $\texttt{owner}(row_k)$
  $pivot\_y$ = y_k

  for *i:=owner+1* ***to*** *number_images* **do**
    $pivot\_y[i] = pivot\_y[owner]$
  end

  for *i:=1* ***to*** *chunk_size* **do**
    global_row = global_row(i) # Finds the global row number of row i
    shift = global_row - k

    if global_row $> k$ **then**
      $y\_chunk(i) = y\_chunk(i) - pivot\_y \cdot my\_chunk(i, centre - shift)$
    end
  end
end
`sync all`
for *k:= n* ***to*** *1* **do**
  owner = $\texttt{owner}(row_k)$
  $pivot\_y$ = $y_k / my\_chunk(k,k)$

  for *i:=1* ***to*** *owner-1* **do**
    $pivot\_y[i] = pivot\_y[owner]$
  end

  for *i:=chunk_size* ***to*** *1* **do**
    global_row = global_row(i) # Finds the global row number of row i
    shift = global_row - k

    if global_row $< k$ **then**
      $y\_chunk(i) = y\_chunk(i) - pivot\_y \cdot my\_chunk(i, centre + shift)$ **end**
  end
end

5

Results

This chapter analyses the performance of the different LU decomposition algorithms developed in the previous chapters. The focus lies on evaluating the computational efficiency of the four algorithms: the basic parallel algorithm, the efficient parallel variant, the full banded version, and the memory-efficient algorithm for banded matrices. Each of these algorithms is tested and analysed using Amdahl's Law to find their parallel performance and scalability.

The experiments are performed on the DelftBlue supercomputer compute type-a nodes from phase 1 ((DHPC), 2024). While a singular node was not required during the computations, all tests were performed on a singular node, reducing the communication time. The matrices used are synthetic matrices generated using Fortran's `RANDOM_NUMBER` subroutine. The shift σ on the main diagonal is set to 2 for all test cases, and the bandwidth of the banded matrices is set to 5000 for one set of tests and 100 in the other set of tests. A variety of image numbers are used to assess the performance of the algorithms.

The results presented in this chapter show the strengths and limitations of each LU decomposition method. Comparison might help determine which algorithm is best suited for particular matrix characteristics, such as size and sparsity.

5.1. Amdahl's law

To understand how programs scale when parallel computing is used, Amdahl's law was developed. This law gives insights into the limitations of parallel algorithms by examining the relationship between the serial fraction S and the parallel fraction P of the algorithm. These two fractions should be constant and together add up to one:

$$S + P = 1$$

.

Amdahl's law describes the theoretical limit of speed-up achievable in a parallel system of fixed size. It says that the overall performance gain is limited by the fraction of the algorithm that cannot be parallelised. This has significant implications when creating high-performance algorithms. This theoretical speed-up is defined as:

$$Speedup_T(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{5.1}$$

Number of images	CPU time in seconds	Speed-up compared to one image	Speed-up compared to previous # images	Computed P (Amdahl's law)
1	2982.04	1.00	-	-
2	2586.26	1.15	1.15	0.261
4	1449.79	2.05	1.78	0.683
8	947.43	3.14	1.53	0.778
16	487.44	6.11	1.94	0.891

Table 5.1: Speed-up for Algorithm 3.1 for a problem of fixed size with $n = 10000$

where P is the parallel fraction of the task and N is the number of processors or images used. As the number of images goes towards infinity, the speed-up will be bounded by the fraction $\frac{1}{1-P}$, which shows that even a small sequential fraction will become a bottleneck when scaling the algorithm up. This makes Amdahl's law an important tool for evaluating the efficiency and scalability of parallel algorithms, such as the LU decomposition (Amdahl, 1967).

To compute the measured speed-up of an algorithm, the following function is used

$$Speedup_M(N) = \frac{T_1}{T_N} \tag{5.2}$$

where T_N is the measured computational time of the algorithm using N images and T_1 is the measured computational time of the algorithm using 1 image.

5.2. Performance of the basic parallel algorithm

The performance of the basic parallel LU decomposition algorithm is evaluated using Amdahl's law. The matrix size is fixed at 10000×10000, and the CPU times were noted down for 1, 2, 4, 8, and 16 images, as shown in Table 5.1. The second column shows that the CPU time consistently decreases as the number of images increases. However, the speed-up is not linear. This is to be expected, since the algorithm does have a non-zero sequential fraction.

Column 3 of the table shows the measured speed-up using equation 5.2. The initial speed-up, especially between 1 and 2 images, is relatively small. This is because of the absence of communication between different images in the serial version of the program. At 1 image, no data sharing is needed, and the `broadcast` function is not used. As soon as the algorithm is parallelised over two images, the function is used, which reduces the efficiency and lowers the speed-up of the algorithm.

This pattern is also visible in the computed parallel fraction P, which is computed using equation 5.1. At lower image counts, P is relatively small due to the communication penalty, but as the number of images increases, P improves significantly. At 16 images, the parallel fraction is set around $P = 0.891$, which means nearly 89.1% of the work benefits from the parallelisation.

To use Amdahl's law, the sequential and parallel fractions of the algorithm should remain the same over a variety of different images. This is not the case for this algorithm, as can be seen in the fifth column of Table 5.1. The parallel fraction goes from $P = 0.261$ to $P = 0.891$, which is a significant difference. Therefore, Amdahl's law cannot be used to analyse the maximum speed-up this algorithm can achieve.

Number of images	CPU time in seconds	Speed-up compared to one image	Speed-up compared to previous # images	Computed P (Amdahl's law)
1	1468.00	1.00	-	-
2	1622.10	0.91	0.91	-0.21
4	820.03	1.79	1.98	0.588
8	462.17	3.18	1.77	0.785
16	264.61	5.55	1.75	0.875

Table 5.2: Speed-up for Algorithm 3.2 for a problem of fixed size with $n = 10000$

5.3. Performance of the efficient parallel algorithm

For Algorithm 3.2, there is a notable increase in CPU time when moving from 1 to 2 images, as shown in Table 5.2. This increase is mostly caused by the fact that, at 1 image, no data sharing is necessary, and all computations are done on one image. In contrast, once 2 or more images are involved, the algorithm must communicate data between images. Since Algorithm 3.2 uses a custom implementation for the sharing of the data, the penalty caused by the communication is significant.

After the initial penalty, the CPU time decreases significantly as the number of images increases. The performance improves almost proportionally with each doubling of images. This indicates that the parallel parts of the algorithm scale well and are effective.

Based on the observed speed-ups, the parallel fraction P is computed to be between $P = -0.21$ and $P = 0.875$. Since this is a very big difference, and one of the measured parallel fractions is not even a possible fraction, $S, P \geq 0$, Amdahl's law cannot be applied here to find the maximum theoretical speed-up.

5.4. Performance of the parallel algorithm for banded matrices

Algorithm 4.2 uses the same custom data-sharing function as Algorithm 3.2. The second column in Table 5.3 shows the cost of the communication as the number of images increases from 1 to 2, when the matrix has a bandwidth of 5000. When only a single image is used, no communication between images is needed, and the algorithm executes without any penalty because of it. However, at 2 or more images, this penalty will apply, leading to an increase in the CPU time between 1 and 2 images.

Despite this initial slowdown, the second column shows that the CPU time starts to decrease as the number of images increases beyond 2. This implies that the parallel components of the algorithm begin to dominate the total runtime as the number of images increases.

The increase in computation time when moving from 1 to 2 images is less pronounced in the tests with a bandwidth of 100. As shown in Table 5.4, the CPU time continues to decrease as the number of images increases. This means that the communication penalty is significantly smaller for matrices with narrower bandwidths, which makes sense, as the rows that need to be shared between images are much shorter.

Another important observation is the difference in final speed-ups achieved by the algorithm. For matrices with a bandwidth of 5000, the speed-up at 16 images reaches 3.41, while for those with a bandwidth of 100, the speed-up is only 1.16, which is almost negligible.

Number of images	CPU time in seconds	Speed-up compared to one image	Speed-up compared to previous # images	Computed P (Amdahl)
1	602.01	1.00	-	-
2	1079.89	0.56	0.56	-1.00
4	560.31	1.07	1.93	0.023
8	342.78	1.76	1.63	0.582
16	176.34	3.41	1.94	0.782

Table 5.3: Speed-up for Algorithm 4.2 for a problem of fixed size with $n = 10000$ and $bandwidth = 5000$

Number of images	CPU time in seconds	Speed-up compared to one image	Speed-up compared to previous # images	Computed P (Amdahl)
1	0.888	1.00	-	-
2	0.854	1.04	1.04	0.038
4	0.826	1.08	1.03	0.026
8	0.807	1.10	1.02	0.014
16	0.765	1.16	1.05	0.011

Table 5.4: Speed-up for Algorithm 4.2 for a problem of fixed size with $n = 10000$ and $bandwidth = 100$

Using the measured speed-ups in Tables 5.3 and 5.4, the parallel fraction P can be estimated using equation (5.1). For the bandwidth 5000 case, the parallel fraction ranges from -1.00 to 0.782, and for the bandwidth 100 case, it ranges from 0.011 to 0.038. In both scenarios, the parallel fraction is not constant, which should be the case when using Amdahl's law. As a result, Amdahl's law cannot be used to predict the maximum theoretical speed-up of the algorithm.

5.5. Performance of the memory-efficient algorithm for banded matrices

For the memory-efficient algorithm for banded matrices, the same performance tests are conducted as in the previous cases. The matrix size is fixed at $n = 10000$, and, since the algorithm is designed for banded matrices, the bandwidth is first fixed at 5000 and then fixed at 100. The tests are executed for 1, 2, 4, 8, and 16 images. The results of these tests are presented in Table 5.5 and Table 5.6. As seen in Algorithms 3.2 and 4.2, for the bandwidth 5000 case, there is once again an increase in CPU time due to the penalty caused by the communication of the data.

Once more than 2 images are used, the total computation time begins to decrease. However, the improvement is less dramatic compared to earlier algorithms. The third column of the table shows that the speed-up from 1 to 16 images is only 3.12, which indicates that the scalability for this implementation is limited.

The increase of the CPU time when moving from 1 image to 2 images is not as visible in the bandwidth 100 case, as seen in Table 5.6. The CPU time continues to decease as the number of images increases, so the communication penalty is significantly smaller in this algorithm if matrices have a smaller bandwidth.

Number of images	CPU time in seconds	Speed-up compared to one image	Speed-up compared to previous # images	Computed P (Amdahl)
1	703.78	1.00	-	-
2	1283.91	0.55	0.55	-1.00
4	676.96	1.04	1.90	0.013
8	395.69	1.78	1.71	0.598
16	225.30	3.12	1.76	0.757

Table 5.5: Speed-up for Algorithm 4.3 for a problem of fixed size with $n = 10000$ and $bandwidth = 5000$

Number of images	CPU time in seconds	Speed-up compared to one image	Speed-up compared to previous # images	Computed P (Amdahl)
1	0.354	1.00	-	-
2	0.261	1.36	1.36	0.267
4	0.236	1.50	1.11	0.167
8	0.232	1.53	1.02	0.074
16	0.230	1.54	1.01	0.036

Table 5.6: Speed-up for Algorithm 4.3 for a problem of fixed size with $n = 10000$ and $bandwidth = 100$

Using equation 5.1, the parallel fractions P are computed. This gives column 5 in Tables 5.5 and 5.6. These columns show that the parallel fraction of the algorithm is somewhere between $P = -1.00$ and $P = 0.757$ if the bandwidth is 5000 and between $P = 0.036$ and $P = 0.267$ if the bandwidth is 100. In both cases, the difference between those numbers is significant enough to conclude that the parallel fraction is not constant. On top of that, it is not possible that the parallel fraction is negative. So, it can be concluded that Amdahl's law cannot be used to find the maximum theoretical speed-up of this algorithm.

6

Conclusion

The primary goal of this paper was to develop an efficient parallel LU decomposition algorithm that can be used in the preconditioning of the IDR Fortran package. A systematic approach was used, beginning with the study and implementation of two sequential LU decomposition algorithms. The memory-efficient algorithm laid the foundations for the development of the basic parallel algorithm, which was refined for greater efficiency into the efficient algorithm for dense matrices. The work finalised in the adaptation of this algorithm for banded matrices, one of which focuses on memory efficiency.

The performance results of the dense matrix algorithms, Algorithm 3.1 and Algorithm 3.2, clearly demonstrate the benefits of the optimisations introduced in the latter. Although an increase in CPU time can be observed when moving from 1 to 2 images, due to the communication penalty, the efficient algorithm consistently outperforms the basic version across all tested image counts. This shows that the communication penalty is overshadowed by the overall efficiency of Algorithm 3.2.

For banded matrices, two approaches were analysed: a full banded LU decomposition (Algorithm 4.2) and a memory-efficient banded variant (Algorithm 4.3). When the bandwidth is high, Algorithm 4.2 tends to perform better in terms of CPU time. This is mostly due to Algorithm 4.3 requiring extra computations to preserve correctness during the matrix update phase. These extra computations decrease its performance efficiency. However, the performance dynamic shifts when the bandwidth is smaller. In that case, the additional computations in Algorithm 4.3 become less disadvantageous, and its requirement of less memory, $n \times (2 \cdot bandwidth + 1)$ instead of $n \times n$, makes it the more efficient option overall.

This leads to a trade-off between memory usage and computational speed. For problems with limited available memory or small bandwidths, Algorithm 3.2 is preferred. However, when the memory is not a constraint and the matrix has a large bandwidth, Algorithm 3.2 will achieve faster results.

Another important observation is the relationship between the bandwidth and parallel efficiency. The maximum speed-up achieved by the banded algorithms decreases significantly as the matrix bandwidth decreases. This is because the number of rows requiring updates in each step is directly tied to the bandwidth. A wider bandwidth means more rows can be updated at the same time, leading to better usage of the parallel resources. However, a narrow bandwidth limits the parallel workload per step, resulting in images that do less or no work

during certain steps and lower speed-up values. This explains the decrease in measured speed-up when going from bandwidth 5000 to 100.

Overall, this paper shows that the use of Fortran Coarrays enables the construction of scalable and efficient parallel LU decomposition algorithms. The created methods enhance the potential performance of the IDR package for solving large systems of shifted linear equations. For dense matrices, the efficient parallel algorithm is recommended due to its speed and scalability. For banded matrices, the choice between full and memory-efficient algorithms should be based on the requirements of the application.

The algorithms developed in this paper are currently being implemented in the IDR(s) Fortran package. Initial integration and testing have shown promising initial results, with notable improvements in computational efficiency. These early outcomes suggest that the parallel LU decomposition methods could significantly improve the performance of the IDR(s) package, especially for solving high-dimensional linear systems.

7

Discussion

The results presented in this paper show the performance of several LU decomposition algorithms in a parallel computing environment using Fortran Coarrays. While the memory-efficient and banded matrix variants were all designed with specific structural advantages in mind, the performance analysis shows that there is still room for improvement.

One aspect that may enhance scalability is changing the distribution of the matrix. Instead of choosing the block-wise distribution, a cyclic distribution of the matrix rows might enhance the efficiency. In the block-wise distribution, consecutive rows are stored on the same image, which leads to load imbalance the further the algorithm progresses. A cyclic distribution strategy, where rows are distributed in a round-robin fashion across images, could offer better load balancing and reduce the idle time of images. Future implementations should investigate how the cyclic distribution affects the computational time and the communication time during the LU decomposition.

Furthermore, while Amdahl's Law was used to interpret the theoretical limits of the speed-up based on the serial fraction of the algorithm, it assumes the problem size is fixed. This may not fully show the scalability potential when working with large matrices. In this case, Gustafson's Law provides a more realistic perspective. It says that as the number of processors increases, the problem size can be scaled proportionally while maintaining efficiency. Applying Gustafson's law could give a better understanding of how the algorithms might perform on high-dimensional problems.

The memory-efficient banded algorithm (Algorithm 4.3) offers significant storage benefits, but is limited in its performance due to additional computations caused by the altered storage scheme. One direction for improvement is optimising these additional computations or restructuring the storage scheme for better updates.

Finally, numerical accuracy is another area that could be revisited. Since pivoting was intentionally excluded to improve performance, the algorithms may suffer from instability in certain cases. While pivoting does decrease performance, implementing pivoting or partial pivoting might be worth exploring for scenarios where preconditioning quality is more important than the runtime.

Bibliography

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, 483–485.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammerling, S., McKenney, A., & Sorensen, D. (1999). *Lapack User's Guide* (Third). SIAM.

Baumann, M., & van Gijzen, M. B. (2015). Nested Krylov methods for shifted linear systems. *SIAM Journal on Scientific Computing*, *37*, S90–S112.

Bisseling, R. H. (2020). *Parallel Scientific Computation; A Structured Approach Using BSP* (Second). Oxford University Press.

(DHPC), D. H. P. C. C. (2024). DelftBlue Supercomputer (Phase 2).

Dongarra, J., & Sullivan, F. (2000). Guest Editors Introduction to the top 10 algorithms. *Computing in Science & Engineering*, *2*(1), 22–23. https://doi.org/10.1109/MCISE.2000.814652

Fanfarillo, A., Burnus, T., Cardellini, V., Filippone, S., Nagle, D., & Rouson, D. (2014). OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers. *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. https://doi.org/10.1145/2676870.2676876

Friedberg, S., Insel, J., & Spence, L. (2014). *Linear Algebra* (Third). Pearson.

Gutknecht, M. H. (2007). A Brief Introduction to Krylov Space Methods for Solving Linear Systems. Frontiers of Computational Science. https://doi.org/10.1007/978-3-540-46375-7_5

Gutknecht, M. H. (2011). IDR explained. *ETNA: Electronic Transactions on Numerical Analysis*, *36*, 126–148.

Saad, Y., & Schultz, M. H. (1986). GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, *7*(3), 856–869. https://doi.org/10.1137/0907058

Shewchuk, J. R. (1994). *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain* (tech. rep.). USA, Carnegie Mellon University.

Stathis, P. T. (2004). *Sparse Matrix Vector Processing Formats* [Master's thesis, Technical University of Delft].

Students, N. (2022). DelftBlue Supercomputer [https://tunews.weblog.tudelft.nl/files/2022/09/DHPC_DelftBlue_Sklein.jpg].

van Gijzen, M. B., & Collignon, T. P. (2011). Minimizing synchronization in IDR (s). *Numerical Linear Algebra with Applications*, *18*, 805–825. https://doi.org/https://doi.org/10.1002/nla.764

van Gijzen, M. B., G., S. G. L., & Zemke, J.-P. M. (2014). Flexible and multi-shift induced dimension reduction algorithms for solving large spars linear systems. *Numerical Linear Algebra with Applications*, *22*, 1–25.