mathematics

MDPI

Article

Coarse-Grained Column Agglomeration Parallel Algorithm for LU Factorization Using Multi-Threaded MATLAB

Osama Sabir [1] and Reza Alebrahim [2,*]

[1] Head of R&D, Euler X Coding, Istanbul, Türkiye
[2] Engineering Faculty, Università degli Studi Niccolò Cusano, 00166 Rome, Italy
* Correspondence: reza.alebrahim@uniroma3.it or reza.alebrahim@unicusano.it

Abstract: MATLAB programing language is one of the most popular scientific computing tools, especially for solving linear algebra problems. LU factorization is an essential component for the direct solution of linear equations systems. This paper studied a coarse-grained column agglomeration parallel algorithm in MATLAB to analyze the implementation performance among all the available computation resources. In this paper, we focus on parallelizing the LU decomposition without pivoting algorithm using Gaussian elimination under MATLAB R2020b platform. Numerical experiments were provided to demonstrate the efficiency of CPU parallelization. Performances of the present methods were assessed by comparing the speed and accuracy of different coarse-grained column agglomeration algorithms using different sizes of matrices. Different algorithms were implemented in a four-core Xeon E3-1220 v3 @ 3.10 GHz CPU with 16 GB RAM memory.

Keywords: coarse-grained column agglomeration; high-level languages; parallel MATLAB

MSC: 68W10

Academic Editor: Theodore E. Simos

Received: 25 December 2024
Revised: 12 January 2025
Accepted: 16 January 2025
Published: 17 January 2025

Citation: Sabir, O.; Alebrahim, R. Coarse-Grained Column Agglomeration Parallel Algorithm for LU Factorization Using Multi-Threaded MATLAB. *Mathematics* **2025**, *13*, 298. https://doi.org/10.3390/math13020298

1. Introduction

Matrix analysis and decomposition are critically important in various computational fields, including engineering and physics. These methods provide powerful tools for breaking down matrices into simpler components, facilitating the solution of systems of equations and matrix inversion. Common matrix factorization techniques include Lower-Upper (LU), QR, Cholesky, Eigenvalue, and Singular Value Decomposition (SVD).

LU decomposition is particularly prominent for solving linear systems and often incorporates row pivoting to enhance numerical stability [1–7]. It is essential for addressing dense linear systems and serves as a fundamental step in their direct solution. Linear algebra systems can be solved using two primary approaches: direct and iterative methods. Direct methods use specific algorithms to compute exact solutions for linear systems. These methods are typically employed for dense matrices, such as those encountered in solving continuous systems using the Finite Element Method (FEM). Gaussian elimination, a widely recognized direct method, simplifies complex matrix computations into more manageable operations, enabling the solution of linear systems, determinant computation, and matrix inversion.

With modern computational advancements, it has become feasible to solve most dense matrices using direct methods. To fully leverage the capabilities of computer memory and multiprocessor systems, agglomeration parallel algorithms are often employed. The development and analysis of agglomeration LU factorizations have been extensively studied by various researchers [8,9]. A broader discussion on clustering LU factorizations and solving

linear systems on parallel computing platforms can be found in [3,10–12]. Higham et al. [13] have proposed a class of random dense matrices where LU factorization with pivoting leads to large growth factors, significantly exceeding those of typical random matrices. They were generated these matrices via a MATLAB function, exhibit unique properties, and GMRES-based iterative refinement is shown to stabilize solutions effectively, even under large growth conditions in low-precision LU factors. In another study [14], two algorithms, pre-pivoted LU decomposition and mixed-precision panel factorization, were introduced. Authors explored the use of half-precision arithmetic on GPUs to accelerate LU decomposition in double precision.

MATLAB is one of the most popular scientific computing tools since the advent of numerical methods. Numerous studies have explored LU factorization using MATLAB [15,16]. Its primary advantage lies in the efficiency of its code; a single command in MATLAB often replaces multiple lines of code in languages like C or FORTRAN. However, transforming MATLAB code to execute in parallel mode and fully utilize advanced computing power remains a challenging task. A critical question is whether an algorithm can be implemented for parallel computing in MATLAB in a manner similar to lower-level languages such as C or FORTRAN. Historically, key challenges in developing a parallel MATLAB platform included limitations in memory size, computational granularity, and business considerations, as discussed in the 1995 article, "Why There Isn't a Parallel MATLAB" [17]. Recent releases of MATLAB have addressed these challenges by providing easier access to multi-threading and enhanced computational capabilities, supporting large-scale projects. The MathWorks Parallel Computing Toolbox extends MATLAB's libraries and modifies the language itself to enable parallel computing [1,18,19]. Additional features include message passing, implicit multi-threading for computations on multicore or multiprocessor machines, and support for both CPU and GPU computing [20,21]. Furthermore, MATLAB offers robust visualization and debugging tools, which significantly benefit the programming process.

The Single Program Multiple Data (SPMD) scheme is a widely used approach for executing parallel LU factorization. SPMD operates by processing multiple coarse-grained data blocks in parallel across multiple core processors, all controlled by a single code. MATLAB supports this methodology through the SPMD command, which enables users to distribute data blocks among the desired processors. This technique facilitates LU factorization by discretizing a large array into multiple data blocks, which are then processed concurrently by a single program. Parallel computation is essential for efficiently processing large-scale data, particularly in solving linear algebra systems where matrix operations dominate. MATLAB's high-level abstraction simplifies development but often struggles with parallel execution compared to low-level programming languages such as C++ or FORTRAN. While MATLAB's Parallel Computing Toolbox provides tools for multi-threading, it lacks the control and efficiency of languages optimized for parallel algorithms. Low-level languages such as C++ and CUDA allow fine-grained control over hardware resources, including memory allocation, thread scheduling, and synchronization. For example, LU factorization implemented in CUDA on NVIDIA GPUs achieves speedups of over $10\times$ for matrices exceeding 1000×1000 in size, as reported in recent benchmarks [22–27]. In contrast, MATLAB struggles to surpass a speedup of $2\times$ for similar matrix sizes due to significant communication and synchronization overheads. Additionally, C++ with OpenMP demonstrates efficient thread utilization, achieving parallel efficiencies exceeding 80%, compared to MATLAB's suboptimal performance in scenarios requiring frequent inter-thread communication.

Despite its potential, relatively few studies have implemented parallel algorithms using the MATLAB language [28]. Most MATLAB-based parallelization efforts have focused on built-in library functions. To the best of the authors' knowledge, no prior study

has specifically addressed the implementation of coarse-grained column agglomeration (CGCA) parallel algorithms for LU factorization in MATLAB.

This study aims to address these gaps by implementing and evaluating a coarse-grained column agglomeration (CGCA) parallel algorithm for LU factorization. Unlike previous studies focusing solely on MATLAB's built-in functions, this research benchmarks custom parallel algorithms to highlight performance differences, communication overhead, and limitations inherent to MATLAB. The primary objectives are to analyze the parallel efficiency across available computational resources and establish a benchmark for MATLAB-based CPU coarse-grain parallelization. Numerical experiments are conducted to evaluate the efficiency of CPU parallelization. The performance of the proposed method is assessed by comparing the speed and accuracy of different CGCA algorithms across varying matrix sizes. The algorithms were implemented on a four-core Xeon E3-1220 v3 @ 3.10 GHz CPU with 16 GB of RAM.

2. Methods

The algebraic process of the LU factorization begins by transforming a matrix $\boldsymbol{A}$ into a product of a lower triangular matrix $\boldsymbol{L}$, and an upper triangular matrix $\boldsymbol{U}$ whose elements are only on the diagonal and above. Equation (1) shows the system of linear equations in matrix form, where array $\boldsymbol{A}$ is factorized to $\boldsymbol{L}$ and $\boldsymbol{U}$ as it is demonstrated in Equation (2),

$$\boldsymbol{A}\,\overline{x} = \overline{b} \tag{1}$$

$$\boldsymbol{A} = \boldsymbol{L} \times \boldsymbol{U} \tag{2}$$

considering the matrices $\boldsymbol{A}, \boldsymbol{L}$ and $\boldsymbol{U}$ as a_{ij}, l_{ij} and u_{ij}, respectively, where $i, j \in \{1, 2, 3, \ldots, n\}$, substituting Equation (2) into Equation (1) leads to the following statement:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \Rightarrow \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & 0 & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \tag{3}$$

where the $l_{ii} = 1$ for $i = 1, 2, 3, \ldots, n$. The Gaussian elimination algorithm without pivoting of the basic LU factorization of a matrix $\boldsymbol{A}$ is described in Figure 1. The computational complexity of the Gaussian elimination algorithm is of the order $O(n^3)$ corresponding to $n^3/3$ paired additions and multiplications in a serial model. The communication cost is of the order $O(n^2)$ equivalent to $n^2/2$ divisions. The matrix $\boldsymbol{A}$ is separated to matrices $\boldsymbol{L}$ and $\boldsymbol{U}$ after the factorization is accomplished.

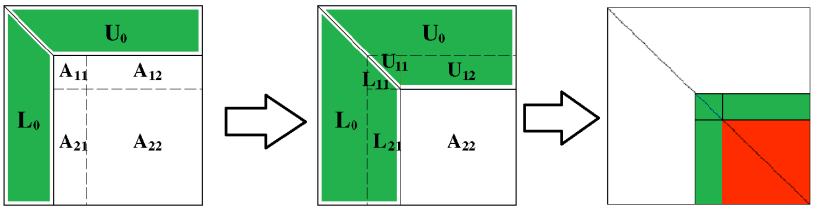

Figure 1. Right blocking LU algorithm.

For each k^{th} step, the algorithm is divided into two loops. The i^{th} loop computes the k^{th} column starting from second row to the n^{th} row, with a computation complexity of $O(n^2)$. The j^{th} loop calculates the k^{th} lower sub-matrices, where the computational complexity of this operation is $O(n^3)$. As shown in Algorithm 1, this algorithm exhibits

high data dependences since i^{th} loop is used to update the j^{th} loop of sub-matrix. The j^{th} loop can only be executed after all updating in the i^{th} loop are completed. Since the matrix for decomposition is typically large, the i^{th} loop must be read from memory during the execution of the j^{th} loop, resulting in a significant number of memory access operations. The main disadvantage of this algorithm is that the two loops are executed sequentially. Additionally, all intermediate matrices are written back to memory, further increasing computational overhead.

Algorithm 1: The sequential LU decomposition algorithm

1. **For** $k \in n-1$ **do**
2. **For** $k+1 \in n$ **do**
3. $A_{i,k} = A_{i,k} / A_{k,k}$
4. **End for**
5. **For** $k+1 \in n$ **do**
6. $A_{k+1,j} = A_{k+1,j} - A_{k+1,k} \times A_{k,j}$
7. **End for**
8. **End for**

The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently. Granularity refers to the amount of computational cost performed by a task. In coarse-grain column parallelism, an algorithm is divided into a large number of small groups of vertical tasks, which are individually assigned to multiple processors. Each processor handles an evenly distributed workload associated with the parallel tasks. The tasking model is based on the group topology of computational tasks that arise during block-wise LU factorization. As shown in Algorithm 2, this algorithm is particularly well-suited for multi-core processors and demonstrates significantly improved parallel scaling behavior compared to a naive parallel-for-based LU factorization. In the algorithm shown in Algorithm 2, processor calls are executed using SPMD MATLAB commands, which enable specific distribution of CGCA tasks among the desired core processors. In general, row interchanges (pivoting) may be required to ensure the existence of LU factorization and the numerical stability of the Gaussian elimination algorithm. However, for simplicity, this aspect is omitted in the present study. Additionally, all the data used in this research consist of dense, diagonally dominant, and random matrices, which eliminates the need for pivoting.

Message Passing Functions

Computational time and communication time are key factors regulating execution time. High communication time can result in some parallel algorithms being slower than their serial counterparts. The Message Passing Interface (MPI) protocol is commonly used in parallel computing for data transfer between multiple processors. In parallel algorithms, low-level programming languages such as C and FORTRAN often utilize the MPI library for point-to-point communication. Many developers have efficiently employed MPI in these low-level programming platforms for various parallel applications. Although MPI specifications are not directly implemented in MATLAB, the MATLAB message-passing library is built on the MPI-2 standard, adapting it for this high-level platform. According to the MPI protocol, data must be identified by size and type before transmission. However, in MATLAB, pre-allocating the size of arrays that will be used later in the program is not mandatory.

Algorithm 2: The fine-grained LU decomposition algorithm

For $k \in n-1$ **do**
  For $k+1 \in n$ **do**
    $A_{i,k} = A_{i,k} / A_{k,k}$
  End for
  Broadcast A to N processors
  For $k+1 \in n$ **do**
    Call processor 1
    $A_{k+1,j} = A_{k+1,j} - A_{k+1,k} \times A_{k,j}$
    Call processor 2
    $A_{k+1,j+1} = A_{k+1,j+1} - A_{k+1,k} \times A_{k,j+1}$
    .....
    Call processor N
    $A_{k+1,j+N-1} = A_{k+1,j+N-1} - A_{k+1,k} \times A_{k,j+N-1}$
  End for
End for

In low-level programming languages, it is mandatory to declare the type and size of arrays in advance. In contrast, MATLAB allows the transmission of various data array types, such as double-precision values, integers, characters, strings, structures, booleans, and cells, without requiring prior declaration. Before sending data, MATLAB first transmits a small header message to pre-allocate memory. This header contains information about the size and type of the data, eliminating the need for extra time to search for larger contiguous memory allocations. MATLAB provides several commands for point-to-point communication, including labSend, labReceive, and labSendReceive. These commands allow data exchange without prior declaration. The labSend command utilizes non-blocking MPI sending, enabling immediate execution for small messages (less than 256 KB). However, for larger messages, labSend must wait for the MPI protocol to process the data on the sending processor and for the receiving processor to be ready to execute labReceive [5]. Additionally, MATLAB offers the labBroadcast command for broadcasting data. However, using labSend and labReceive provides greater control, allowing users to detect cyclic deadlocks and miscommunications during parallel execution.

3. Results

To analyze the parallel performance on the MATLAB platform, different codes were developed for the numerical experiments. In addition to the built-in function, four sequential and three parallel codes were created. The sequential code runs on a single thread, with each algorithm implementing a different CGCA. In contrast, the parallel codes execute the coarse-grained columns across two, three, or four threads. In this study, LU factorization of matrix $\boldsymbol{A}$ was performed using both sequential and parallel codes based on the Gaussian elimination approach. Matrix $\boldsymbol{A}$ is a diagonally dominant, dense array created from random double-precision numbers. The matrix sizes selected for testing ranged from 100×100 to $15{,}000 \times 15{,}000$ elements. All computational analyses were carried out on a Dell Precision T1700 Workstation running Ubuntu 16.04 LTS 64-bit operating system [29].

The execution time for the sequential and parallel algorithm codes was obtained using the timeit MATLAB function. Figure 2 compares the execution time of the sequential method with MATLAB's built-in LU function for different matrix sizes. It can be seen that the execution time of the MATLAB LU function is faster than the sequential code for all array sizes, except for some cases where the matrix size is smaller than 100×100 elements.

As shown by the trend of the curves in Figure 2, the gap between the MATLAB built-in function and the sequential code remains constant after the matrix size reaches 5000×5000. It is also worth noting that the fluctuation in the graphs decreases after the matrix size exceeds 1000×1000 in the semi-logarithmic plot.

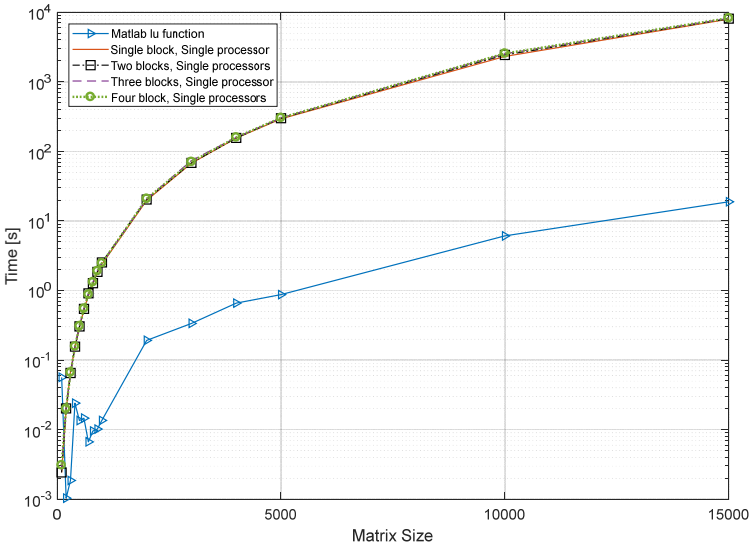

Figure 2. Execution time vs. matrix size in a semi-logarithmic plot for a single processor using various CGCA parallelization blocks. The blue color diagram shows MATLAB's built-in LU function execution time for different matrix sizes.

Furthermore, it was observed that there are no significant differences between the various CGCA codes when executed on a single thread. Since a single processor handles all calculations without shared data, the execution time for all applied methods is nearly the same.

The mean absolute error of the single thread results was illustrated in Figure 3. The error of the MATLAB built-in function is less than CGCA codes when running in single thread. The trend of the error shows more consistency for large matrix sizes.

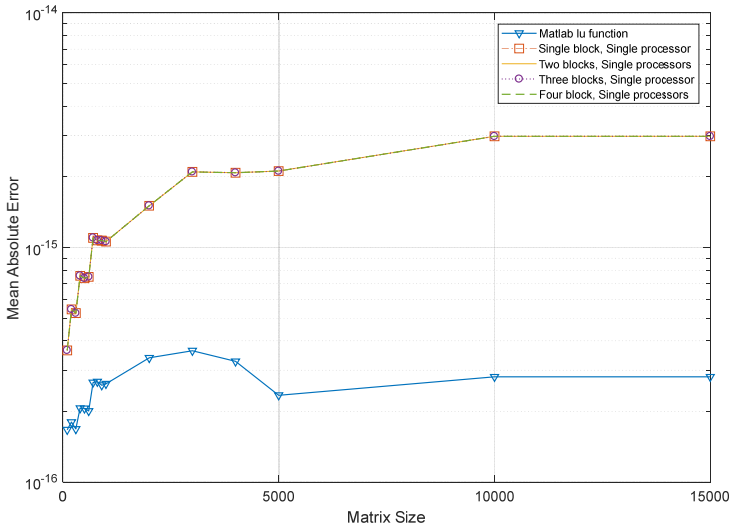

Figure 3. Mean absolute error vs. matrix size in a semi-logarithmic plot for a single processor using various CGCA parallelization blocks. The blue color diagram shows MATLAB's built-in LU function mean absolute error for different matrix sizes.

Figure 4 shows the execution time of CGCA codes in comparison with MATLAB's built-in LU function for different matrix sizes. The CGCA codes were executed using both single and multiple threads. The curve for the MATLAB built-in function shows similar results to the single-thread algorithm for small matrix sizes (below 300×300). As shown, the execution time for LU factorization and the CGCA code using a single thread is much faster than the parallel multi-thread algorithms. This is due to the high communication cost between threads in MATLAB. Since MATLAB is a high-level language, it does not allow full control over data distribution among threads.

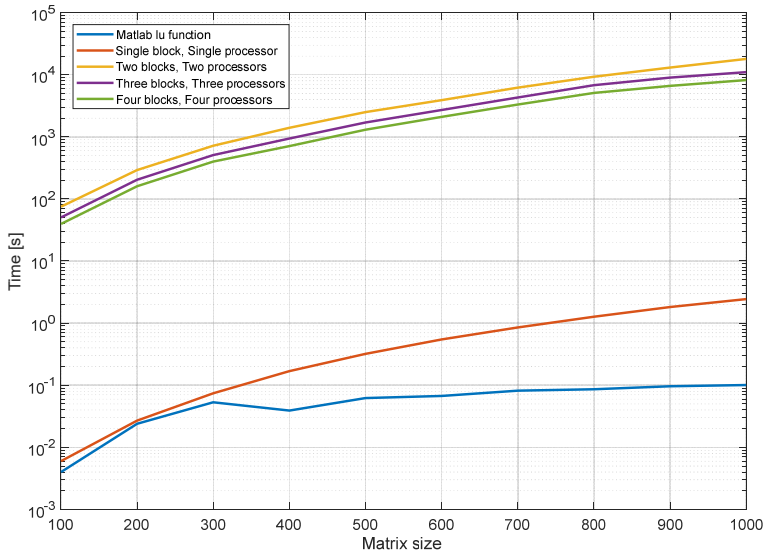

Figure 4. Execution time vs. matrix size in a semi-logarithmic plot for a multi-processor system using various CGCA parallelization blocks. The blue plot represents the execution time of MATLAB's built-in LU function for different matrix sizes.

However, it can be observed that increasing the number of threads from two to four reduces the execution time. This is because the computational cost is divided among more threads using MATLAB's Parallel Computing Toolbox command (SPMD). However, there is still a significant gap between the execution times of the single-thread and multi-thread algorithms. The differences in execution time between two and three threads are more pronounced than the differences between three and four threads. If this trend continues, it can be concluded that after a certain number of threads, there will be no significant change in the execution time curves.

Table 1 presents the results for the execution time of different matrix sizes. From Table 1, the speedup and parallel efficiency were calculated and are presented in Tables 2 and 3, respectively. The speedup is calculated by dividing the sequential execution time by the parallel execution time. The values in Table 2 indicate a slowdown rather than a speedup when parallel algorithms are implemented. Parallel efficiency is calculated by dividing the speedup values by the number of processors involved. It can be observed that the parallel efficiency values are significantly low, highlighting the poor efficiency of the CGCA algorithm implementation in MATLAB.

Table 1. Comparing the execution time for different CGCA algorithms of different matrix sizes.

Methods	Size of the Matrices									
	100	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**
MATLAB LU built-in function	0.004	0.024	0.053	0.039	0.062	0.067	0.082	0.086	0.096	0.101
Single block, Single processor	0.006	0.027	0.074	0.169	0.320	0.545	0.850	1.266	1.820	2.434
Two blocks, Single processors	0.006	0.026	0.076	0.173	0.328	0.557	0.888	1.300	1.855	2.504
Three blocks, Single processors	0.006	0.026	0.077	0.174	0.331	0.565	0.886	1.320	1.878	2.536
Four block, Single processors	0.006	0.027	0.078	0.175	0.332	0.568	0.899	1.330	1.892	2.554
Two blocks, Two processors	74.86	292.3	718.5	1.4×10^3	2.5×10^3	3.9×10^3	6.2×10^3	9.3×10^3	1.3×10^4	1.8×10^4
Three blocks, Three processors	50.40	203.7	507.8	937.9	1.7×10^3	2.7×10^3	4.3×10^3	6.8×10^3	9.0×10^3	1.1×10^4
Four blocks, Four processors	39.51	160.2	399.4	711.0	1.3×10^3	2.1×10^3	3.3×10^3	5.1×10^3	6.6×10^3	8.2×10^3

Table 2. Comparing the speedup value for different CGCA algorithms of different matrix sizes.

Methods	Size of the Matrices									
	100	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**
2 B, 2 P	8.01×10^{-5}	8.89×10^{-5}	1.06×10^{-4}	1.24×10^{-4}	1.31×10^{-4}	1.43×10^{-4}	1.43×10^{-4}	1.40×10^{-4}	1.43×10^{-4}	1.39×10^{-4}
3 B, 3 P	1.19×10^{-4}	1.28×10^{-4}	1.52×10^{-4}	1.86×10^{-4}	1.95×10^{-4}	2.09×10^{-4}	2.06×10^{-4}	1.94×10^{-4}	2.09×10^{-4}	2.31×10^{-4}
4 B, 4 P	1.52×10^{-4}	1.69×10^{-4}	1.95×10^{-4}	2.46×10^{-4}	2.55×10^{-4}	2.70×10^{-4}	2.72×10^{-4}	2.61×10^{-4}	2.87×10^{-4}	3.11×10^{-4}

Table 3. Comparing the Parallel efficiency for different CGCA algorithms of different matrix sizes.

Methods	Size of the Matrices									
	100	**200**	**300**	**400**	**500**	**600**	**700**	**800**	**900**	**1000**
2 B, 2 P	4.01×10^{-5}	4.45×10^{-5}	5.29×10^{-5}	6.18×10^{-5}	6.56×10^{-5}	7.14×10^{-5}	7.16×10^{-5}	6.99×10^{-5}	7.13×10^{-5}	6.96×10^{-5}
3 B, 3 P	3.97×10^{-5}	4.25×10^{-5}	5.05×10^{-5}	6.18×10^{-5}	6.49×10^{-5}	6.98×10^{-5}	6.87×10^{-5}	6.47×10^{-5}	6.96×10^{-5}	7.68×10^{-5}
4 B, 4 P	3.80×10^{-5}	4.21×10^{-5}	4.88×10^{-5}	6.15×10^{-5}	6.38×10^{-5}	6.76×10^{-5}	6.81×10^{-5}	6.52×10^{-5}	7.17×10^{-5}	7.79×10^{-5}

4. Conclusions and Contributions

MATLAB strategies focus on improving single-processor computations. Its parallelism emphasizes MATLAB functions and readily available parallel templates, making it accessible to both novices and experts. This study analyzes the efficiency of coarse-grained column agglomeration (CGCA) parallel algorithms in MATLAB. The results demonstrate the implementation of one-, two-, three-, and four-column agglomeration parallel algorithms.

It was observed that applying the CGCA parallelization block for a single-thread core yields results similar to MATLAB's built-in functions for small matrix sizes. However, as the matrix size increases, the performance of the CGCA parallelization blocks deteriorates, leading to a reduction in the speed of the algorithms. Parallel computation in MATLAB faces two major challenges: communication time and the overhead associated with calling the parallel toolbox (SPMD). Communication time refers to the duration required to transmit data between different threads. Parallelizing algorithms for multi-threaded execution increases communication and synchronization overhead between core processors in MATLAB. Sending and receiving large arrays across multiple threads in any parallel structure leads to longer allocation times in MATLAB's shared and distributed memory.

It was also observed that the parallelism of CGCA algorithms in MATLAB can be slower than the original sequential approach. This is because the responsibility for optimizing CGCA parallelism lies with the compiler, not the program itself. As a result, CGCA algorithms perform better in low-level languages that support faster communication. Additionally, shared memory systems with low communication times are more suitable for CGCA parallelism. Based on these findings, it is recommended that MATLAB is not ideal for programming parallel algorithms, except when using its built-in functions.

As the structure of the PYTHON programming language is very similar to that of MATLAB, its parallel computing capabilities using multiple processors remain largely unexplored. Since PYTHON does not have a direct equivalent to MATLAB's SPMD, multiprocessing is required to simulate parallel computation. This investigation can be conducted in future studies.

Author Contributions: Conceptualization, R.A. and O.S.; methodology, O.S.; software, O.S.; validation, R.A. and O.S.; formal analysis, R.A.; investigation, O.S.; resources, O.S.; data curation, O.S.; writing—original draft preparation, O.S.; writing—review and editing, R.A.; visualization, O.S.; supervision, R.A.; project administration, R.A.; funding acquisition, R.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: There no more data in this study. All data are already added to the manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Huang, S.; Sha, J.; Shi, L. An Analytical Model for Domain-Specific Accelerator Deploying Sparse LU Factorization. In Proceedings of the 2023 IEEE 15th International Conference on ASIC (ASICON), Nanjing, China, 24–27 October 2023; pp. 1–4.
2. Lindquist, N.; Luszczek, P.; Dongarra, J. Using Additive Modifications in LU Factorization Instead of Pivoting. In Proceedings of the 37th International Conference on Supercomputing, Orlando, FL, USA, 21–23 June 2023; pp. 14–24.
3. Gallivan, K.A.; Plemmons, R.J.; Sameh, A.H. Parallel algorithms for dense linear algebra computations. *SIAM Rev.* **1990**, *32*, 54–135. [CrossRef]
4. Szabó, A.; Paál, G. Reused LU factorization as a preconditioner for efficient solution of the parabolized stability equations. *Comput. Fluids* **2024**, *269*, 106115. [CrossRef]
5. Zhou, J.; Yang, W.; Dai, M.; Cai, Q.; Wang, H.; Li, K. Parallel Sparse LU Factorization with Machine-Learning Method on Multi-core Processors. In Proceedings of the 2021 7th International Conference on Systems and Informatics (ICSAI), Chongqing, China, 13–15 November 2021; pp. 1–11.
6. Gates, M.; Abdelfattah, A.; Akbudak, K.; Al Farhan, M.; Alomairy, R.; Bielich, D.; Burgess, T.; Cayrols, S.; Lindquist, N.; Sukkari, D.; et al. Evolution of the SLATE linear algebra library. *Int. J. High Perform. Comput. Appl.* **2025**, *39*, 3–17. [CrossRef]
7. Alebrahim, R.; Thamburaja, P.; Srinivasa, A.; Reddy, J.N. A robust Moore–Penrose pseudoinverse-based static finite-element solver for simulating non-local fracture in solids. *Comput. Methods Appl. Mech. Eng.* **2023**, *403*, 115727. [CrossRef]
8. Demmel, J.W.; Higham, N.J.; Schreiber, R.S. Stability of block *LU* factorization. *Numer. Linear Algebra Appl.* **1995**, *2*, 173–190. [CrossRef]
9. Mattheij, R.M.M. Stability of Block LU-Decompositions of Matrices Arising from BVP. *SIAM J. Algebr. Discret. Methods* **1984**, *5*, 314–331. [CrossRef]
10. Dackland, K.; Elmroth, E.; Kågström, B.; Loan, C.V. Design and evaluation of parallel block algorithms: LU factorization on an IBM 3090 VF/600J. In Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia, PA, USA, 25–27 March 1991; pp. 3–10.
11. Dongarra, J.J.; Duff, I.S.; Sorensen, D.C.; Van der Vorst, H.A. *Solving Linear Systems on Vector and Shared Memory Computers*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1991.
12. Vannieuwenhoven, N.; Meerbergen, K. IMF: An incomplete multifrontal LU-factorization for element-structured sparse linear systems. *SIAM J. Sci. Comput.* **2013**, *35*, A270–A293. [CrossRef]
13. Higham, D.J.; Higham, N.J.; Pranesh, S. Random matrices generating large growth in LU factorization with pivoting. *SIAM J. Matrix Anal. Appl.* **2021**, *42*, 185–201. [CrossRef]

14. Sahraneshinsamani, N.; Catalán, S.; Herrero, J.R. Mixed-precision pre-pivoting strategy for the LU factorization. *J. Supercomput.* **2025**, *81*, 87. [CrossRef]
15. Nikabdullah, N.; Azizi, M.A.; Alebrahim, R.; Singh, S.S.K.; Khidir, E.A. The application of peridynamic method on prediction of viscoelastic materials behaviour. In Proceedings of the 3rd International Conference on Mathematical Sciences, Kuala Lumpur, Malaysia, 17–19 December 2013; American Institute of Physics: College Park, MD, USA, 2014; Volume 1602, pp. 357–363.
16. Aldlemy, M.S.; Al-Jumaili, S.A.K.; Al-Mamoori, R.A.M.; Ya, T.; Alebrahim, R. Composite patch reinforcement of a cracked simply-supported beam traversed by moving mass. *J. Mech. Eng. Sci.* **2020**, *14*, 6403–6415. [CrossRef]
17. Moler, C. *Why Isn't There a Parallel MATLAB*; The MathWorks Newsletter; Spring: Berlin/Heidelberg, Germany, 1995.
18. Moler, C. *Parallel MATLAB: Multiple Processors and Multiple Cores*; The MathWorks News & Notes; MathWorks, Inc.: Natick, MA, USA, 2007.
19. Sharma, G.; Martin, J. MATLAB®: A Language for Parallel Computing. *Int. J. Parallel Program.* **2009**, *37*, 3–36. [CrossRef]
20. Park, B. Computational Enhancement of Sparse Tableau via Block LU Factorization for Power Flow Studies. *IEEE Trans. Power Syst.* **2023**, *38*, 4974–4977. [CrossRef]
21. He, K.; Tan, S.X.-D.; Wang, H.; Shi, G. GPU-accelerated parallel sparse LU factorization method for fast circuit analysis. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2015**, *24*, 1140–1150. [CrossRef]
22. Chen, X.; Ren, L.; Wang, Y.; Yang, H. GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *26*, 786–795. [CrossRef]
23. Lee, W.-K.; Achar, R.; Nakhla, M.S. Dynamic GPU parallel sparse LU factorization for fast circuit simulation. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2018**, *26*, 2518–2529. [CrossRef]
24. Jia, Y.; Luszczek, P.; Dongarra, J. Multi-GPU implementation of LU factorization. *Procedia Comput. Sci.* **2012**, *9*, 106–115. [CrossRef]
25. D'azevedo, E.; Hill, J.C. Parallel LU factorization on GPU cluster. *Procedia Comput. Sci.* **2012**, *9*, 67–75. [CrossRef]
26. Volkov, V.; Demmel, J. *LU, QR and Cholesky Factorizations Using Vector Capabilities of GPUs*; Technical Report No. UCB/EECS-2008; University of California at Berkeley: Berkeley, CA, USA, 2008.
27. Faverge, M.; Herrmann, J.; Langou, J.; Lowery, B.; Robert, Y.; Dongarra, J. Mixing LU and QR factorization algorithms to design high-performance dense linear algebra solvers. *J. Parallel Distrib. Comput.* **2015**, *85*, 32–46. [CrossRef]
28. Heath, M.T. *Parallel Numerical Algorithms*; Lecture Notes; Springer: Berlin/Heidelberg, Germany, 2011; pp. 1–42.
29. Petersen, R. *Ubuntu 16.04 LTS Desktop: Applications and Administration*; Surfing Turtle Press: Online, 2016.