IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 15 November 2023, accepted 28 November 2023, date of publication 30 November 2023,
date of current version 5 December 2023.

*Digital Object Identifier 10.1109/ACCESS.2023.3338544*

RESEARCH ARTICLE

# A Parallel Computing Method for the Computation of the Moore–Penrose Generalized Inverse for Shared-Memory Architectures

**ELKIN GELVEZ-ALMEIDA[1,3], RICARDO J. BARRIENTOS[2,4], KARINA VILCHES-PONCE[2,5], AND MARCO MORA[2,4]**

[1]Doctorado en Modelamiento Matemático Aplicado, Universidad Católica del Maule, Talca 3480112, Chile
[2]Laboratory of Technological Research in Pattern Recognition (LITRP), Facultad de Ciencias de la Ingeniería, Universidad Católica del Maule, Talca 3480112, Chile
[3]Facultad de Ciencias Básicas y Biomédicas, Universidad Simón Bolívar, San José de Cúcuta 540006, Colombia
[4]Departamento de Computación e Industrias, Facultad de Ciencias de la Ingeniería, Universidad Católica del Maule, Talca 3480112, Chile
[5]Departamento de Matemáticas, Física y Estadística, Facultad de Ciencias Básicas, Universidad Católica del Maule, Talca 3480112, Chile

Corresponding author: Ricardo J. Barrientos (rbarrientos@ucm.cl)

This work was supported in part by the National Agency for Research and Development (ANID)/Scholarship Program/BECAS DOCTORADO NACIONAL/2020 under Grant 21201000; in part by the ANID Subdirección de Investigación Aplicada/Concurso IDeA I+D 2023, under Grant ID23i10242; and in part by the Research Project ANID FONDECYT REGULAR 2020 "Very Large Fingerprint Classification Based on a Fast and Distributed Extreme Learning Machine," Government of Chile, under Grant 1200810.

**ABSTRACT** The computation of the Moore–Penrose generalized inverse is a commonly used operation in various fields such as the training of neural networks based on random weights. Therefore, a fast computation of this inverse is important for problems where such neural networks provide a solution. However, due to the growth of databases, the matrices involved have large dimensions, thus requiring a significant amount of processing and execution time. In this paper, we propose a parallel computing method for the computation of the Moore–Penrose generalized inverse of large-size full-rank rectangular matrices. The proposed method employs the Strassen algorithm to compute the inverse of a nonsingular matrix and is implemented on a shared-memory architecture. The results show a significant reduction in computation time, especially for high-rank matrices. Furthermore, in a sequential computing scenario (using a single execution thread), our method achieves a reduced computation time compared with other previously reported algorithms. Consequently, our approach provides a promising solution for the efficient computation of the Moore–Penrose generalized inverse of large-size matrices employed in practical scenarios.

**INDEX TERMS** High-performance computing, Moore–Penrose generalized inverse matrix, neural networks with random weights, parallel computing, Strassen algorithm.

## I. INTRODUCTION

The Moore–Penrose generalized inverse is a valuable tool in various science and engineering fields [1]. Neural networks are a powerful approach in machine learning. Their effectiveness has been demonstrated in a wide range of applications, including pattern recognition, decision making, and complex process optimization [2]. The problems of overdetermination or underdetermination are encountered in the assignation and adjustment of neural network random weights and neurons in the hidden layer, making it challenging to obtain optimal solutions using conventional methods [3]. In this context, the Moore–Penrose generalized inverse is valuable because it provides a robust and effective method for finding solutions in large-scale database environments [2].

In the realm of numerical computing and the computation of the Moore–Penrose generalized inverse, several algorithms have been proposed to improve the efficiency, accuracy,

The associate editor coordinating the review of this manuscript and approving it for publication was Tomas F. Pena.

and computation time. Courrieu [4] introduced an algorithm based on a full-rank Cholesky factorization, demonstrating significantly faster results than those obtained from previously proposed algorithms, especially for large-size matrices. Baksalary and Baksalary [5] developed a specific formula for the computation of the Moore–Penrose generalized inverse of column-partitioned matrices. Petković and Stanimirović [6], [7] introduced a comprehensive recursive method for calculating outer generalized inverses of a given square matrix. The authors integrated the efficient Strassen matrix multiplication and inversion algorithm into their method and developed a partially recursive algorithm that can compute various classes of generalized inverses.

Toutounian and Ataei [8] proposed an algorithm based on the conjugate Gram-Schmidt process for the computation of the Moore-Penrose generalized inverse of arbitrary matrices, including symmetric and rectangular matrices. Their numerical experiments demonstrated that the resulting pseudoinverse is accurate, and its computation time is significantly lower than that of other methods, especially for large sparse matrices. Katsikis and Pappas [9] introduced a computational method for computing the Moore–Penrose generalized inverse. Their method is applicable to both square and full-rank $m \times n$ matrices. Later, in collaboration with Petralias [10], they extended their findings to encompass all types of matrices using QR factorization, thereby enhancing the accuracy of the method and enabling its application to both dense and sparse matrices. That same year, Marco and Martínez [11] presented an algorithm based on QR factorization for several classes of totally positive matrices that are ill-conditioned.

Li and Li [12] introduced a family of iterative methods for calculating an approximate inverse of a square matrix using quadratic convergence. Chen and Wang [13] expanded these iterative methods and demonstrated that the resulting sequence converges at a superior rate toward the Moore–Penrose generalized inverse of a matrix. Stanimirović et al. [14] presented iterative approaches for computing the outer inverse of a matrix that originates from the second Penrose condition and exhibits linear or quadratic convergence. Behera et al. [15] studied various generalized inverses of tensors within the contexts of commutative and noncommutative rings. The authors also proposed algorithms for computing the inner inverse, Moore–Penrose generalized inverse, and weighted Moore–Penrose generalized inverse for tensors in a noncommutative ring setting.

More recently, Stanojevi'c et al. [16] proposed an algorithm based on the generalized Cholesky factorization and the Strassen matrix inversion algorithm, which has been specifically designed for parallel computing architectures, particularly for using graphics processing units (GPUs) in the compute unified device architecture. Their results showed significant advantages when GPUs are employed for large-size matrix computations. Similarly, Ma et al. [17], Stanimirovć et al. [18], [19], [20], Chen and Ji [21], and Aldhafeeri et al. [22] worked on the computation of the Moore–Penrose and other generalized inverses.

In this paper, we introduce a novel approach for the computation of the Moore–Penrose generalized inverse of full-rank matrices for shared-memory architectures. Our methodology aims to overcome the problems posed by large-size matrices by exploiting the advantages of parallel computing and the recursive Strassen algorithm and improve the performance and accuracy in matrix inverse computations. The main contributions of our work are as follows:

- Development of an efficient algorithm for the computation of the Moore–Penrose generalized inverse matrix: We introduce a novel algorithm specifically designed to compute the Moore–Penrose generalized inverse of full-rank matrices of size $m \times n$, where $m \neq n$. Our methodology is based on the well-known Strassen algorithm, which is used to compute the inverse of a nonsingular matrix, and can be used in practical applications in various scientific fields.
- Algorithm optimization for shared-memory architectures: We address the issue of computational efficiency by developing an algorithm optimized for shared-memory architectures. This allows us to efficiently exploit the resources available in these architectures, resulting in significantly reduced computation times and better hardware utilization.
- Application to large-size matrices: An important contribution of this study is the successful implementation of the algorithm for the computation of the Moore–Penrose generalized inverse of large-size matrices. Numerical experiments demonstrate that the proposed algorithm can efficiently handle large-size matrices, making it a valuable tool for applications involving massive data and complex problems.

These contributions signify a substantial advancement in the computation of the Moore–Penrose generalized inverse by providing an efficient and suitable solution for dealing with large-scale problems in shared-memory architectures. This work paves the way for new opportunities in practical applications across science and engineering, especially those that need to implement neural networks with randomly initialized weights, where the efficient handling of full-rank matrices with varying dimensions is crucial for achieving accurate and swift results.

The remainder of the paper is organized as follows. In Section II, we introduce the definition of the Moore–Penrose generalized inverse and provide a detailed description of the Strassen algorithm, which is used for the computation of the Moore–Penrose generalized inverse of a nonsingular matrix. In Section III, we present our parallel computing method for the computation of the Moore-Penrose generalized inverse. In Section IV, we present and analyze the results obtained from our numerical experiments. Finally, in Section V, we present the conclusions and highlight the main contributions of our work.

## II. NOTATION AND PRELIMINARIES

In this section, we introduce the notation for the Moore–Penrose generalized inverse and the Strassen algorithm, which is used to compute the inverse of a nonsingular matrix. Using this theoretical foundation, we present our method for the computation of the Moore–Penrose generalized inverse of large-dimension full-rank matrices, thereby addressing the computational challenges encountered in real-world applications.

### A. FULL-RANK TRIANGULAR MATRICES

A matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is considered to be a full-rank matrix if its rank, denoted in the literature as rank($A$), is equal to the minimum value between $m$ and $n$, that is:

$$\text{rank}(\mathbf{A}) = \min(m, n). \tag{1}$$

This implies that in a full-rank matrix, all columns (or rows) are linearly independent, and there is no linear combination of columns (or rows) that can yield a null column (or row) [2].

### B. MOORE–PENROSE GENERALIZED INVERSE

A nonsingular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ has a unique inverse $\mathbf{X} \in \mathbb{R}^{n\times n}$ that satisfies the following condition:

$$\mathbf{AX} = \mathbf{XA} = \mathbf{I}_n, \tag{2}$$

where, $\mathbf{I}_n$ represents the identity matrix of order $n$. However, various applications in applied mathematics necessitate an inverse for singular or rectangular matrices. In this context, the *generalized inverse* or *pseudoinverse* of $\mathbf{A}$ has been introduced as a matrix $\mathbf{X}$, also denoted as $\mathbf{X} = \mathbf{A}^{-1}$, which satisfies the following criteria [23]:

(i) It exists for a broader class of matrices than that of nonsingular matrices.
(ii) It exhibits some of the properties of the traditional inverse.
(iii) It reduces to the traditional inverse when matrix $\mathbf{A}$ is nonsingular.

The Moore–Penrose generalized inverse is a type of generalized inverse [24]. For $\mathbf{A} \in \mathbb{R}^{m\times n}$, there exists a unique matrix $\mathbf{X} \in \mathbb{R}^{n\times m}$ that satisfies the following conditions, known as Penrose conditions [25]:

$$\mathbf{AXA} = \mathbf{A} \tag{3}$$
$$\mathbf{XAX} = \mathbf{X} \tag{4}$$
$$\mathbf{XA} = (\mathbf{XA})^T \tag{5}$$
$$\mathbf{AX} = (\mathbf{AX})^T \tag{6}$$

where $(\mathbf{XA})^T$ and $(\mathbf{AX})^T$ are the transpose matrices of $\mathbf{XA}$ and $\mathbf{AX}$, respectively. $\mathbf{X}$ is commonly known as the Moore–Penrose generalized inverse and denoted as $\mathbf{A}^\dagger$ [1], [24].

### C. STRASSEN ALGORITHM

In 1968, Strassen [26] introduced an innovative algorithm for matrix multiplication. This algorithm surpasses the traditional approach of Naïve algorithm for matrix multiplication by performing only 7 multiplication operations instead of the 8 required [27]. This approach results in a significant improvement in computational efficiency. Let $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n\times n}$ be two block-partitioned matrices. The number of scalar operations required to compute the matrix product $\mathbf{C} = \mathbf{AB}$ using the standard multiplication method is $2n^3 - n^2 = O(n^3)$. However, the matrix multiplication algorithm introduced by Strassen has a complexity of $O(n^{\log_2 7}) \approx O(n^{2.807})$ [6].

Furthermore, the Strassen algorithm can compute the inverse of a nonsingular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ with the same complexity as the standard matrix multiplication. This method is based on the block decomposition of matrix $\mathbf{A}$ and the corresponding decomposition of its ordinary inverse [6]. Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a block-partitioned matrix as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \quad \mathbf{A}_{11} \in \mathbb{R}^{k\times k}, \tag{7}$$

where $\mathbf{A}$ and $\mathbf{A}_{11}$ are nonsingular matrices, and $k$ represents the integer quotient of $n/2$. Its inverse is given as:

$$\mathbf{A}^{-1} = \begin{bmatrix} \mathbf{X}_{11} & \mathbf{X}_{12} \\ \mathbf{X}_{21} & \mathbf{X}_{22} \end{bmatrix}, \quad \mathbf{X}_{11} \in \mathbb{R}^{k\times k}. \tag{8}$$

Matrices $\mathbf{X}_{11}$, $\mathbf{X}_{12}$, $\mathbf{X}_{21}$, and $\mathbf{X}_{22}$ are computed as follows [26]:

$$\begin{array}{ll} 1. \ \mathbf{R}_1 = \mathbf{A}_{11}^{-1} & 7. \ \mathbf{X}_{12} = \mathbf{R}_3\mathbf{R}_6 \\ 2. \ \mathbf{R}_2 = \mathbf{A}_{21}\mathbf{R}_1 & 8. \ \mathbf{X}_{21} = \mathbf{R}_6\mathbf{R}_2 \\ 3. \ \mathbf{R}_3 = \mathbf{R}_1\mathbf{A}_{12} & 9. \ \mathbf{R}_7 = \mathbf{R}_3\mathbf{X}_{21} \\ 4. \ \mathbf{R}_4 = \mathbf{A}_{21}\mathbf{R}_3 & 10. \ \mathbf{X}_{11} = \mathbf{R}_1 - \mathbf{R}_7 \\ 5. \ \mathbf{R}_5 = \mathbf{R}_4 - \mathbf{A}_{22} & 11. \ \mathbf{X}_{22} = -\mathbf{R}_6 \\ 6. \ \mathbf{R}_6 = \mathbf{R}_5^{-1} & \end{array} \tag{9}$$

where $\mathbf{R}_1, \ldots, \mathbf{R}_7$ are temporary matrices. Here, $\mathbf{R}_5 = -(\mathbf{A}_{12} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}) = -(\mathbf{A}/\mathbf{A}_{11})$; that is, $\mathbf{R}_5$ is the minus Schur complement of $\mathbf{A}_{11}$ [6]. If $\mathbf{A}$ and $\mathbf{A}_{11}$ are nonsingular matrices, then $\mathbf{R}_5 = -(\mathbf{A}/\mathbf{A}_{11})$ is also a nonsingular matrix [28]. Equation (9) is applicable if $\mathbf{A}_{11}$ and the Shur complement $(\mathbf{A}/\mathbf{A}_{11})$ are nonsingular. Because temporary matrices $\mathbf{R}_1$ and $\mathbf{R}_6$ are inverses of other matrices, they are computed recursively until the dimensions of the new matrix become $1 \times 1$.

A notable limitation of this method for calculating the inverse of a nonsingular matrix is its high memory consumption. This challenge arises due to its necessity of storing temporary matrices during algorithm execution (Equations (7) and (9)). The effect of memory consumption depends on the size of the involved matrices and recursion depth. It becomes particularly pronounced in the case of large-scale matrices. The algorithm divides the original matrices into smaller submatrices, thereby substantially increasing the amount of data that needs to be stored in memory [6], [29]. As the recursion depth increases, the numbers of generated sub and auxiliary matrices increase, thereby raising the demand for memory. This increases memory usage can be a concern in resource-limited systems.

## III. PROPOSED METHOD

In this section, we introduce our parallel computing method for the computation of the Moore–Penrose generalized inverse. As mentioned in Section I, various computational methods have been developed for the computation of the Moore–Penrose generalized inverse [4], [5], [8], [9], [10], [16]. In this paper, we develop a parallel computing algorithm for shared-memory architectures. This algorithm employs the method presented by Katsikis and Pappas [9]. The authors proposed a method for computing the Moore–Penrose generalized inverse of a tensor-product matrix, which can also be applied to full-rank rectangular matrices. In this method, the corresponding Gram matrix comprising two linearly independent vectors is computed and then the square linear system is solved, specifically using the mldivide (\) operator in MATLAB. However, in our approach, this linear system is solved using the Strassen algorithm, which is designed for computing the inverse of nonsingular matrices.

In this regard, let $\mathbf{A} \in \mathbb{R}^{n\times m}$ be a full-rank matrix. The Moore–Penrose generalized inverse is defined as follows [23]:

$$\mathbf{A}^{\dagger} = \begin{cases} \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T, & \text{si} \quad rank(\mathbf{A}) = n, \\ \mathbf{A}^T\left(\mathbf{A}\mathbf{A}^T\right)^{-1}, & \text{si} \quad rank(\mathbf{A}) = m. \end{cases} \tag{10}$$

Equation (10) describes two scenarios based on the rank of matrix $\mathbf{A}$. If its rank equals the number of columns $n$, the left inverse is obtained by multiplying the transpose of $\mathbf{A}$ by itself, resulting in a full-rank square matrix. If the rank of $\mathbf{A}$ matches equals the number of rows $m$, the right inverse is obtained by multiplying $\mathbf{A}$ by its transpose, also resulting in a full-rank square matrix. In both cases, the full-rank property ensures that the resulting matrices obtained from the product and inverse multiplication operations are invertible and have unique solutions. This is essential for the Moore–Penrose generalized inverse to preserve its outstanding properties such as the projection capability and minimization of the Euclidean norm of the solution [25].

To compute the inverses of matrices $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}\mathbf{A}^T$, we apply the Strassen algorithm described in Section II. Hereafter, we outline the proposed strategy of applying parallel computing to the Strassen algorithm. Subsequently, we present our parallel computing methodology for the computation of (10), which is based on the reuse of the parallel algorithms designed for the operations involved in the Strassen algorithm.

### A. PARALLEL METHOD FOR COMPUTING THE INVERSE OF A NONSINGULAR MATRIX

To calculate the inverse of a nonsingular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, we employ the Strassen algorithm presented in the previous section. The computation process of the Strassen algorithm can be divided into three steps: 1) block partitioning of the original matrix, as described in (7); 2) performing the operations described in (9); 3) assembling matrices $\mathbf{X}_{11}$, $\mathbf{X}_{12}$, $\mathbf{X}_{21}$, and $\mathbf{X}_{22}$ as described in (8). Hereafter, we introduce the proposed parallel computing algorithm for executing the three steps required by the Strassen algorithm.

**Algorithm 1** Algorithm for partitioning a matrix $A \in \mathbb{R}^{n\times n}$ (Equation (7)).

**INPUT:**
$\mathbf{A} \in \mathbb{R}^{n\times n}$ ← Matrix to be partitioned into blocks.
*initialRow* ← Starting row of the new matrix.
*finalRow* ← Ending row of the new matrix.
*initialCol* ← Ending column of the new matrix.
*finalCol* ← Starting column of the new matrix.
*numThreads* ← Threads for parallel execution.
**OUTPUT:**
$\mathbf{A}_{pq} \in \mathbb{R}^{k\times k}$ ← Submatrix extracted from the original matrix.
$k = finalRow - initialRow = finalCol - initialCol$.
$p$ and $q$ represent the position of the submatrix.

```
#pragma omp parallel for num_threads (numThreads)
for i = initialRow to finalRow do
    for j = initialCol to finalCol do
        A_pq(i − intialRow, j − initialCol) ← A(i, j);
    end for
end for
return A_pq ∈ R^{k×k};
```

#### 1) ALGORITHM FOR MATRIX BLOCK PARTITIONING

The first step in executing the Strassen algorithm is to divide a nonsingular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ into four submatrices $\mathbf{A}_{11}$, $\mathbf{A}_{12}$, $\mathbf{A}_{21}$, and $\mathbf{A}_{22}$, where $\mathbf{A}_{11} \in \mathbb{R}^{k\times k}$ is a nonsingular matrix and $k$ represents the integer quotient of $n/2$. Algorithm 1 obtains a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ as input; it also obtains the rows and columns where the partition begins and ends as well as the threads that will execute the operation. Subsequently, each thread independently and in parallel extracts the values of each row, considering the start and final positions. Finally, the extracted rows are assembled to obtain submatrix $\mathbf{A}_{ij} \in \mathbb{R}^{k\times k}$. At this point, $k$ is defined as the dimension of the submatrix and corresponds to the number of rows and columns extracted from the original matrix.

#### 2) ALGORITHM FOR MATRIX ADDITION, SUBTRACTION, AND MULTIPLICATION

Once the original matrix is partitioned into blocks using Algorithm 1, it is necessary to perform the operations described in (9). These operations involve matrix subtraction and multiplication as well as the recursive computation of the inverse of a nonsingular matrix. In this regard, we now present the proposed parallel computing algorithms that efficiently perform the additions, subtraction, and multiplication operations, which will be used later in the parallel computing algorithm for the computation of the inverse of a nonsingular matrix.

Algorithm 2 is used for both matrix subtraction and addition. This methodology is applied similarly to both cases. The algorithm obtains two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m\times n}$, where $m$ and $n$ can be equal, as an input; it also obtains the threads that will execute the operation. Since the values of the resulting matrix are independent, we assign the operations of each row

**Algorithm 2** Algorithm for matrix addition and subtraction.

**INPUT:**
$\mathbf{A} \in \mathbb{R}^{m\times n}$ ← First matrix in the addition or subtraction.
$\mathbf{B} \in \mathbb{R}^{m\times n}$ ← Second matrix in the addition or subtraction.
*numThreads* ← Threads for parallel execution.
**OUTPUT:**
$\mathbf{C} \in \mathbb{R}^{m\times n}$ ← Resultant matrix of the addition or subtraction.

1: *#pragma omp parallel for num_threads (numThreads)*
2: **for** $i = 1$ to $m$ **do**
3: **for** $j = 1$ to $n$ **do**
4: $\mathbf{C}(i, j) \leftarrow \mathbf{A}(i, j) \pm \mathbf{B}(i, j)$;
5: **end for**
6: **end for**
7: **return** $\mathbf{C} \in \mathbb{R}^{m\times n}$;

**Algorithm 3** Algorithm for matrix multiplication.

**INPUT:**
$\mathbf{A} \in \mathbb{R}^{m\times n}$ ← First matrix in the multiplication.
$\mathbf{B} \in \mathbb{R}^{n\times p}$ ← Second matrix in the multiplication.
*numThreads* ← Threads for parallel execution.
**OUTPUT:**
$\mathbf{C} \in \mathbb{R}^{m\times p}$ ← Resultant matrix of the multiplication.

1: *#pragma omp parallel for num_threads (numThreads)*
2: **for** $i = 1$ to $m$ **do**
3: **for** $j = 1$ to $p$ **do**
4: double $sum = 0.0$;
5: **for** $k = 1$ to $n$ **do**
6: $sum+ = \mathbf{A}(i, k) * \mathbf{B}(k, j)$;
7: **end for**
8: $\mathbf{C}(i, j) \leftarrow sum$;
9: **end for**
10: **end for**
11: **return** $\mathbf{C} \in \mathbb{R}^{m\times p}$;

to a thread for parallel execution. In this way, each thread is responsible for adding or subtracting the values of a row to calculate the corresponding values of the same row in the resulting matrix. Finally, the values are assembled to return a matrix $\mathbf{C} \in \mathbb{R}^{m\times n}$, which is the result of the addition or subtraction.

Algorithm 3 is used for matrix multiplication. The algorithm obtains two matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times p}$ as an input; it also obtains the threads that will execute the operation. In this case, each thread is responsible for multiplying the elements of a row from the first matrix by all the elements of the second matrix. These operations are performed by each thread independently and in parallel. Finally, the values obtained from each thread are assembled to obtain matrix $\mathbf{C} \in \mathbb{R}^{m\times p}$, which is the result of multiplication.

In addition to the matrix subtraction and multiplication operations described in (9), we need to perform a sign change in the values of the temporary matrix $\mathbf{R}_6$. To perform this operation, we use a methodology similar to that implemented in Algorithm 2. In this case, the sign-change algorithm obtains a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ as an input; it also obtains the threads that will execute the operation. Subsequently, each thread is responsible for performing the sign change on the values of a row by multiplying each value by –1. Finally, the results obtained from each thread are assembled to return a matrix $\mathbf{B} \in \mathbb{R}^{m\times n}$, which is the result of the sign-change operation.

**Algorithm 4** Algorithm for assembling the resulting matrices.

**INPUT:**
$\mathbf{X}_{11} \in \mathbb{R}^{k\times k}$
$\mathbf{X}_{12} \in \mathbb{R}^{k\times n-k}$
$\mathbf{X}_{21} \in \mathbb{R}^{n-k\times k}$
$\mathbf{X}_{22} \in \mathbb{R}^{n-k\times n-k}$
*numThreads* ← Threads for parallel execution.
**OUTPUT:**
$\mathbf{X} \in \mathbb{R}^{n\times n}$ ← Resultant matrix of the assembly.

1: *#pragma omp parallel for num_threads (numThreads)*
2: **for** $i = 1$ to $n$ **do**
3: **for** $j = 1$ to $n$ **do**
4: **if** $i < k$ **then**
5: **if** $j < k$ **then**
6: $\mathbf{X}(i, j) \leftarrow \mathbf{X}_{11}(i, j)$;
7: **else**
8: $\mathbf{X}(i, j) \leftarrow \mathbf{X}_{12}(i, j - k)$;
9: **end if**
10: **else**
11: **if** $j < k$ **then**
12: $\mathbf{X}(i, j) \leftarrow \mathbf{X}_{21}(i - k, j)$;
13: **else**
14: $\mathbf{X}(i, j) \leftarrow \mathbf{X}_{22}(i - k, j - k)$;
15: **end if**
16: **end if**
17: **end for**
18: **end for**
19: **return** $\mathbf{X} \in \mathbb{R}^{n\times n}$;

### 3) ALGORITHM FOR ASSEMBLING THE RESULTING MATRICES

After obtaining submatrices $\mathbf{X}_{11}$, $\mathbf{X}_{12}$, $\mathbf{X}_{21}$, and $\mathbf{X}_{22}$ via the operations described in (9), it is necessary to assemble them to obtain the inverse of the nonsingular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$. Algorithm 4 obtains submatrices $\mathbf{X}_{11} \in \mathbb{R}^{k\times k}$, $\mathbf{X}_{12} \in \mathbb{R}^{k\times n-k}$, $\mathbf{X}_{21} \in \mathbb{R}^{n-k\times k}$, and $\mathbf{X}_{22} \in \mathbb{R}^{n-k\times n-k}$ as an input; it also obtains the threads that will execute the operation. Here, $n$ represents the dimension of the assembled matrix, and $k$ is the integer quotient of $n/2$. Following the same methodology as in the previous algorithms, each thread is responsible for assembling one row of the resulting matrix. Upon completion, the algorithm returns matrix $\mathbf{X} \in \mathbb{R}^{n\times n}$, which is the result of assembling the four submatrices.

In general, in Algorithms 1 through 4, we observe that the values in each row of the output matrices are obtained independently and in parallel using a specifically assigned thread. In line 1 of each algorithm, the program assigns a thread to execute the loop in line 2. The number of simultaneous executions is constrained by the number of threads allocated for the algorithm execution. Once a thread completes one loop, it is assigned a new task from those waiting in the queue. In this way, multiple tasks can be executed in parallel. Below, we present the implementation of Algorithms 1 through 4 for the calculation of the inverse of a nonsingular matrix.

### 4) INVERSE OF A NONSINGULAR MATRIX USING THE STRASSEN ALGORITHM

A three-step process must be followed to compute the inverse of a nonsingular matrix using the Strassen algorithm. First, we divide the original matrix into four submatrices,

**Algorithm 5** Algorithm for computing the inverse of a nonsingular matrix using the Strassen algorithm.

**INPUT:**
$\mathbf{A} \in \mathbb{R}^{n\times n}$ ← Nonsingular matrix.
*numThreads* ← Threads for parallel execution.
**OUTPUT:**
$\mathbf{X} \in \mathbb{R}^{n\times n}$ ← Inverse of the original matrix.

1: **if** $n = 1$ **then**
2: $\mathbf{X}(1,1) \leftarrow 1.0/\mathbf{A}(1,1)$;
3: **return** $\mathbf{X} \in \mathbb{R}^{n\times n}$;
4: **else**
5: int $k = n/2$;
6: $\mathbf{A}_{11} \leftarrow subMatrix(\mathbf{A}, 1, k, 1, k, numThreads)$;
7: $\mathbf{A}_{12} \leftarrow subMatrix(\mathbf{A}, 1, k, k+1, n, numThreads)$;
8: $\mathbf{A}_{21} \leftarrow subMatrix(\mathbf{A}, k+1, n, 1, k, numThreads)$;
9: $\mathbf{A}_{22} \leftarrow subMatrix(\mathbf{A}, k+1, n, k+1, n, numThreads)$;
10: $\mathbf{R}_1 \leftarrow invStrassen(\mathbf{A}_{11}, numThreads)$;
11: $\mathbf{R}_2 \leftarrow multiplyMatrices(\mathbf{A}_{21}, \mathbf{R}_1, numThreads)$;
12: $\mathbf{R}_3 \leftarrow multiplyMatrices(\mathbf{R}_1, \mathbf{A}_{12}, numThreads)$;
13: $\mathbf{R}_4 \leftarrow multiplyMatrices(\mathbf{A}_{21}, \mathbf{R}_3, numThreads)$;
14: $\mathbf{R}_5 \leftarrow subtractMatrices(\mathbf{R}_4, \mathbf{A}_{22}, numThreads)$;
15: $\mathbf{R}_6 \leftarrow invStrassen(\mathbf{R}_5, numThreads)$;
16: $\mathbf{X}_{12} \leftarrow multiplyMatrices(\mathbf{R}_3, \mathbf{R}_6, numThreads)$;
17: $\mathbf{X}_{21} \leftarrow multiplyMatrices(\mathbf{R}_6, \mathbf{R}_2, numThreads)$;
18: $\mathbf{R}_7 \leftarrow multiplyMatrices(\mathbf{R}_3, \mathbf{X}_{21}, numThreads)$;
19: $\mathbf{X}_{11} \leftarrow subtractMatrices(\mathbf{R}_1, \mathbf{R}_7, numThreads)$;
20: $\mathbf{X}_{22} \leftarrow changeSign(\mathbf{R}_6, numThreads)$;
21: $\mathbf{X} \leftarrow assemblyMatrices(\mathbf{X}_{11}, \mathbf{X}_{12}, \mathbf{X}_{21}, \mathbf{X}_{22}, numThreads)$;
22: **return** $\mathbf{X} \in \mathbb{R}^{n\times n}$;
23: **end if**

as described in (7). Next, we perform the operations described in (9). Finally, we assemble the results obtained in the previous stage, following the procedure described in (8). To perform these calculations in parallel, we employ Algorithms 1 to 4.

Algorithm 5 operates on a nonsingular matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ and obtains the threads that will execute the process in parallel as an input. Where $n = 1$, the algorithm returns the inverse value of a single element of $\mathbf{A}$ (lines 1 to 3). When $n > 1$, the algorithm proceeds to calculate $k = n/2$, where $k$ represents the integer quotient of $n/2$ (line 5). Subsequently, by applying Algorithm 1, the input matrix is divided into four submatrices, as described in (7) (lines 6 to 9). Temporary matrices $\mathbf{R}_1$ and $\mathbf{R}_6$ (lines 10 and 15, respectively) are calculated recursively. In turn, temporary matrix $\mathbf{R}_5$ and submatrix $\mathbf{X}_{11}$ (lines 14 and 19, respectively) are obtained using Algorithm 2. $\mathbf{R}_2$, $\mathbf{R}_3$, $\mathbf{R}_4$, $\mathbf{X}_{12}$, $\mathbf{X}_{21}$, and $\mathbf{R}_7$ (lines 11, 12, 13, 16, 17, and 18, respectively) are computed using Algorithm 3. For submatrix $\mathbf{X}_{22}$ (line 20), an approach similar to that of Algorithms 2 and 3 is employed, where each thread performs the sign change in the rows assigned by the program. Once submatrices $\mathbf{X}_{11}$, $\mathbf{X}_{12}$, $\mathbf{X}_{21}$, and $\mathbf{X}_{22}$ are available, they are assembled using Algorithm 4 (line 21). Finally, Algorithm 5 returns $\mathbf{X} \in \mathbb{R}^{n\times n}$, where $\mathbf{X}$ is the inverse of the original input matrix.

A flowchart of Algorithm 5 is illustrated in Fig. 1. The processes shown in blue are executed using a single thread, and the processes shown in green are performed using $p$ threads, which are specified in the algorithm input. The process marked with an asterisk (*) indicates a recursive call in the algorithm.

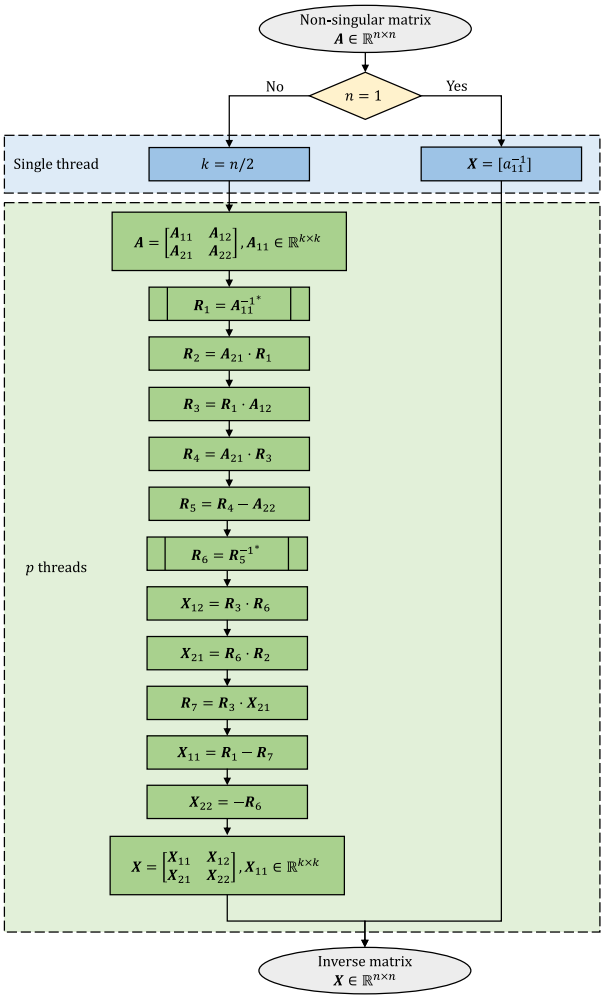


**FIGURE 1.** Computation of the inverse of a nonsingular matrix using the Strassen algorithm. The processes shown in blue are executed using a single thread, and the processes shown in greed are performed using $p$ threads, which are specified in the algorithm input. The process marked with * indicates a recursive call of the algorithm.

### B. PARALLEL COMPUTING METHOD FOR THE COMPUTATION OF THE MOORE–PENROSE GENERALIZED INVERSE

Using the above-described algorithms, we present our parallel computing method, which is used for the computation of the Moore–Penrose generalized inverse of a full-rank matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$. Algorithm 6, obtains the input matrix and as an input; it also obtains the threads that will execute the operation. Initially, the rank of the matrix is checked (lines 1 and 8); then, the operations described in (10) are performed according to each case. To transpose a matrix (lines 2 and 9), we follow an approach similar to that used in Algorithms 2 and 3, where each thread is responsible for transposing the rows assigned by the program. Next, we perform the operations described in (10) using Algorithms 3 and 5. Finally, Algorithm 6 returns the Moore–Penrose generalized inverse of a full-rank matrix.

A flowchart of Algorithm 6 is illustrated in Fig.2. The processes shown in green are executed using $p$ threads,

**Algorithm 6** Algorithm for computing the Moore–Penrose generalized inverse.

**INPUT:**
$\mathbf{A} \in \mathbb{R}^{m\times n} \leftarrow$ Rectangular matrix of full rank.
*numThreads* $\leftarrow$ Threads for parallel execution.

**OUTPUT:**
$\mathbf{A}^{\dagger} \in \mathbb{R}^{n\times m} \leftarrow$ Moore–Penrose generalized inverse of the original matrix.

**if** $m > n$ **then**
  $\mathbf{A}^T \leftarrow transposeMatrix(\mathbf{A}, numThreads)$;
  $\mathbf{A}^T\mathbf{A} \leftarrow multiplyMatrices(\mathbf{A}^T, \mathbf{A}, numThreads)$;
  $(\mathbf{A}^T\mathbf{A})^{-1} \leftarrow invStrassen(\mathbf{A}^T\mathbf{A}, numThreads)$;
  $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T \leftarrow multiplyMatrices((\mathbf{A}^T\mathbf{A})^{-1}, \mathbf{A}^T, numThreads)$;

  **return** $\mathbf{A}^{\dagger} \in \mathbb{R}^{n\times m}$;
**else**
  $\mathbf{A}^T \leftarrow transposeMatrix(\mathbf{A}, numThreads)$;
  $\mathbf{A}\mathbf{A}^T \leftarrow multiplyMatrices(\mathbf{A}, \mathbf{A}^T, numThreads)$;
  $(\mathbf{A}\mathbf{A}^T)^{-1} \leftarrow invStrassen(\mathbf{A}\mathbf{A}^T, numThreads)$;
  $\mathbf{A}^T(\mathbf{A}^T\mathbf{A})^{-1} \leftarrow multiplyMatrices(\mathbf{A}^T, (\mathbf{A}^T\mathbf{A})^{-1}, numThreads)$;

  **return** $\mathbf{A}^{\dagger} \in \mathbb{R}^{n\times m}$;
**end if**

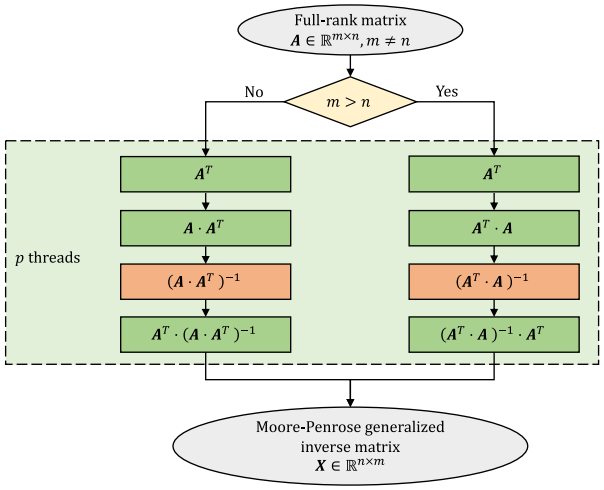


**FIGURE 2.** Computation of the Moore–Penrose generalized inverse of full-rank rectangular matrices. The processes shown in green are executed using $p$ threads, which are defined in the algorithm input. The process shown in orange represents the computation of the inverse of nonsingular matrices using the Strassen algorithm.

which are defined in the algorithm parameters, and the process shown in orange represents the computation of the inverses of nonsingular matrices using the Strassen algorithm.

## IV. NUMERICAL EXPERIMENTS

In this section, we present the numerical experiments conducted to validate the proposed method. All experiments were implemented on a server equipped with 2×Intel(R) Xeon(R) Gold 6238R CPUs @2.20GHz, 56 physical cores, and a 128-GB of RAM in a Debian GNU/Linux 5.10.0-19-amd64 ×86_64 Operating System. The code was written in C++ programming language using OpenMP to enable parallel processing with shared memory. The `#pragma omp parallel for` sentence in OpenMP enables the execution of each iteration of the following `for` using a different thread. The threads are then executed on the available core according to the criteria of the operating system.

**TABLE 1.** Proposed method and other sequential algorithms reported in the literature used for the computation of the Moore-Penrose generalized inverse.

| Algorithm | Method |
|---|---|
| StrassenGinv | Equation (10) using Strassen algorithm |
| Ginv [9] | LU factorization |
| SVDginv [24] | Singular Value Decomposition |
| Geninv [4] | Full-rank Cholesky factorization |
| BlockGinv [5] | Column-wise partitioned matrix |
| QRginv [10] | QR factorization |
| QRgeninv [30] | QR factorization and Geninv |

The matrix coefficients that were double-precision floating-point numbers were randomly selected from the real interval $[-1, 1]$. The random matrices were subsequently tested to ensure that $\mathbf{A}$ and $\mathbf{A}_{11}$ are nonsingular matrices, as presented in (7). The algorithm accuracy was evaluated using the 2-norm of error matrices for the four Penrose conditions (Equations (3)–(6)), and the execution time was recorded in seconds.

### A. SEQUENTIAL COMPUTATION OF THE MOORE–PENROSE GENERALIZED INVERSE

We conducted sequential computing experiments utilizing some of the most widely reported methods in the literature. These methods include the tensor product of two vectors [9] based on LU factorization, singular value decomposition (SVD) [24], full-rank Cholesky factorization [4], column-partitioned matrix [5], and QR factorization [30]. Furthermore, we conducted experiments based on the proposed method, which employed the Strassen algorithm using a single thread. The comparison of various algorithms is listed in Table 1. In the experiments, we used full-rank matrices with a size of $2^k$, where $k = 8, \ldots, 12$. Furthermore, similar to previous report, we used full-rank matrices of size $m \times n$, where $m = 2n$ (overdetermined matrices, where $n = 2^k$) and $n = 2m$ (underdetermined matrices, where $m = 2^k$) [4], [8].

Tables 2 and 3 present the results for matrices with sizes $m = 2n$ and $n = 2m$, respectively. The results are in the $10^{-15}$–$10^{-10}$ range for all cases, demonstrating a notable accuracy for all algorithms. However, the execution time varies significantly among the various methods. The differences in execution time are generally small for small-sized matrices but become significant as the matrix size increases. Using the proposed method, the computation of the Moore–Penrose generalized inverse is achieved in the shortest possible time. This is important not only in the real-time analysis of large datasets but also in the optimization of available computational resources.

The variation in computation time using different algorithms when $m = 2n$ is shown in Fig. 3. The differences in computation times become more pronounced with increasing value of $k$. The computation time of the SVD-based algorithm is notably longer than those of other algorithms.

**TABLE 2.** Sequential computation time for obtaining the Moore–Penrose generalized inverse and 2-norm of the four Penrose conditions in full-rank matrices. The size of matrices is $m \times n$, where $m = 2n$ and $Rank = n$.

| Rank | Method | Time (sec) | $\|\|\mathbf{A}\mathbf{A}^{\dagger}\mathbf{A}-\mathbf{A}\|\|_2$ | $\|\|\mathbf{A}^{\dagger}\mathbf{A}\mathbf{A}^{\dagger}-\mathbf{A}^{\dagger}\|\|_2$ | $\|\|\mathbf{A}\mathbf{A}^{\dagger}-(\mathbf{A}\mathbf{A}^{\dagger})^{\mathrm{T}}\|\|_2$ | $\|\|\mathbf{A}^{\dagger}\mathbf{A}-(\mathbf{A}^{\dagger}\mathbf{A})^{\mathrm{T}}\|\|_2$ |
|---|---|---|---|---|---|---|
| $2^8$ | StrassenGinv | 0.012 | 3.1535e-13 | 3.3268e-15 | 3.1079e-14 | 4.0296e-14 |
| | Ginv | 0.014 | 3.0742e-13 | 3.2178e-15 | 3.0604e-14 | 4.0143e-14 |
| | SVDginv | 0.524 | 1.2130e-12 | 1.0487e-14 | 1.3280e-13 | 1.0932e-13 |
| | Geninv | 0.024 | 9.5912e-13 | 7.3078e-15 | 6.8581e-14 | 1.2347e-13 |
| | BlockGinv | 0.170 | 8.6342e-13 | 6.7604e-15 | 1.0501e-13 | 7.5861e-14 |
| | QRginv | 0.057 | 3.6715e-13 | 2.7091e-15 | 3.5894e-14 | 4.1362e-14 |
| | QRgeninv | 0.067 | 9.7149e-13 | 6.9570e-15 | 7.1503e-14 | 1.2205e-13 |
| $2^9$ | StrassenGinv | 0.093 | 7.6924e-13 | 3.7088e-15 | 5.4736e-14 | 6.9161e-14 |
| | Ginv | 0.105 | 6.8818e-13 | 3.4530e-15 | 5.2145e-14 | 6.3507e-14 |
| | SVDginv | 6.188 | 3.4764e-12 | 1.4993e-14 | 2.7763e-13 | 2.3746e-13 |
| | Geninv | 0.197 | 2.0710e-12 | 8.3398e-15 | 1.0783e-13 | 1.9478e-13 |
| | BlockGinv | 1.718 | 2.4916e-12 | 1.0152e-14 | 2.2335e-13 | 1.5872e-13 |
| | QRginv | 0.442 | 8.5603e-13 | 3.1975e-15 | 6.2844e-14 | 7.1462e-14 |
| | QRgeninv | 0.537 | 2.1214e-12 | 8.1080e-15 | 1.1298e-13 | 1.9657e-13 |
| $2^{10}$ | StrassenGinv | 0.734 | 1.7109e-12 | 4.0931e-15 | 9.1983e-14 | 1.1022e-13 |
| | Ginv | 0.807 | 1.5686e-12 | 3.7328e-15 | 8.8016e-14 | 1.0091e-13 |
| | SVDginv | 56.766 | 1.1941e-11 | 2.5182e-14 | 6.1025e-13 | 5.1364e-13 |
| | Geninv | 1.594 | 4.7469e-12 | 9.6192e-15 | 1.7513e-13 | 3.1508e-13 |
| | BlockGinv | 18.110 | 7.5492e-12 | 1.5001e-14 | 4.5822e-13 | 3.2650e-13 |
| | QRginv | 3.643 | 2.0971e-12 | 3.8641e-15 | 1.0980e-13 | 1.2313e-13 |
| | QRgeninv | 4.414 | 4.8403e-12 | 9.4327e-15 | 1.8572e-13 | 3.1843e-13 |
| $2^{11}$ | StrassenGinv | 5.657 | 3.8169e-12 | 4.4145e-15 | 1.4733e-13 | 1.7282e-13 |
| | Ginv | 6.431 | 3.4100e-12 | 3.9488e-15 | 1.4072e-13 | 1.5422e-13 |
| | SVDginv | 707.629 | 4.1125e-11 | 4.1795e-14 | 1.4395e-12 | 1.1482e-12 |
| | Geninv | 13.008 | 1.0133e-11 | 1.0142e-14 | 2.7150e-13 | 4.7425e-13 |
| | BlockGinv | 193.089 | 2.4172e-11 | 2.3734e-14 | 1.0075e-12 | 7.0068e-13 |
| | QRginv | 38.186 | 4.6782e-12 | 4.4812e-15 | 1.9024e-13 | 2.0544e-13 |
| | QRgeninv | 44.748 | 1.0569e-11 | 1.0336e-14 | 2.9895e-13 | 4.9116e-13 |
| $2^{12}$ | StrassenGinv | 45.176 | 8.2409e-12 | 4.6891e-15 | 2.2619e-13 | 2.6483e-13 |
| | Ginv | 50.608 | 7.2675e-12 | 4.1018e-15 | 2.1555e-13 | 2.3101e-13 |
| | SVDginv | 11973.700 | 1.5738e-10 | 7.8339e-14 | 3.5611e-12 | 2.7643e-12 |
| | Geninv | 125.929 | 2.1206e-11 | 1.0714e-14 | 4.0605e-13 | 7.0488e-13 |
| | BlockGinv | 2104.230 | 8.8585e-11 | 4.1889e-14 | 2.3733e-12 | 1.5884E-12 |
| | QRginv | 347.340 | 1.1169e-11 | 5.4348e-15 | 3.3258e-13 | 3.5586e-13 |
| | QRgeninv | 420.991 | 2.3046e-11 | 1.1332e-14 | 4.7844e-13 | 7.5678e-13 |

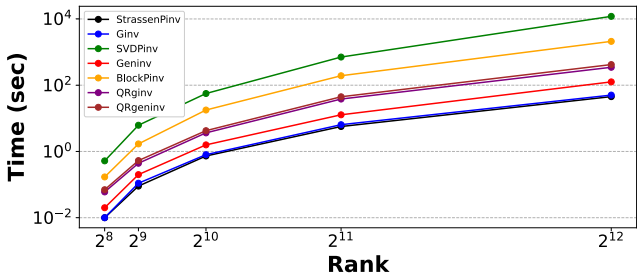


**FIGURE 3.** Computation time for obtaining the Moore–Penrose generalized inverse of full-rank $m \times n$ matrices with $m = 2n$ and $Rank = n$. All methods employ $2^k$-rank matrices, where $k = 8, \ldots, 12$.

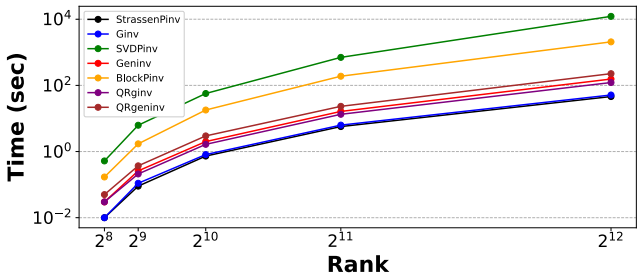


**FIGURE 4.** Computation time for obtaining the Moore–Penrose generalized inverse for full-rank $m \times n$ matrices with $n = 2m$ and $Rank = m$. All methods employ $2^k$-rank matrices, where $k = 8, \ldots, 12$.

This substantial difference in computation time is attributed to the inherent complexity of the SVD procedure. Furthermore, the BlockGinv algorithm exhibits an increase in computation time, although not as significant as that observed in the SVD-based algorithm. By contrast, the method using the tensor product of two vectors and our method, which is based on the Strassen algorithm, are the fastest. These results demonstrate that the algorithm based on LU factorization and our method can calculate the Moore–Penrose generalized inverse with lesser complexity than other algorithms.

Similar results were obtained for $n = 2m$ (see Fig. 4). However, in this scenario, algorithms based on QR factorization exhibit shorter computation times than other algorithms. This difference in performance arises from the inherent properties of the two categories of matrices. For $n = 2m$, the matrices have more columns than rows, which increases the efficiency of the QR factorization process. The specific characteristics of these matrices reduce the number of operations required for factorization and, consequently, for the computation of the Moore–Penrose generalized

**TABLE 3. Sequential computation time for obtaining the Moore–Penrose generalized inverse and 2-norm of the four Penrose conditions in full-rank matrices. The size of matrices is $m \times n$, where $n = 2m$ and $Rank = m$.**

| Rank | Method | Time (sec) | $\|\|\mathbf{A}\mathbf{A}^{\dagger}\mathbf{A}-\mathbf{A}\|\|_2$ | $\|\|\mathbf{A}^{\dagger}\mathbf{A}\mathbf{A}^{\dagger}-\mathbf{A}^{\dagger}\|\|_2$ | $\|\|\mathbf{A}\mathbf{A}^{\dagger}-(\mathbf{A}\mathbf{A}^{\dagger})^{\mathrm{T}}\|\|_2$ | $\|\|\mathbf{A}^{\dagger}\mathbf{A}-(\mathbf{A}^{\dagger}\mathbf{A})^{\mathrm{T}}\|\|_2$ |
|---|---|---|---|---|---|---|
| $2^8$ | StrassenGinv | 0.012 | 3.1140e-13 | 3.1658e-15 | 3.6432e 14 | 3.0491e.14 |
| | Ginv | 0.014 | 2.9619e-13 | 3.0755e-15 | 3.4474e-14 | 3.0463e-14 |
| | SVDginv | 0.533 | 1.2082e-12 | 1.0395e-14 | 1.1407e-13 | 1.3527e-13 |
| | Geninv | 0.032 | 6.0509e-13 | 6.9802e-15 | 9.6027e-14 | 5.7516e-14 |
| | BlockGinv | 0.171 | 8.4012e-13 | 6.6721e-15 | 7.6361e-14 | 1.0599e-13 |
| | QRginv | 0.028 | 3.1164e-13 | 3.1555e-15 | 3.5407e-14 | 3.9868e-14 |
| | QRgeninv | 0.047 | 4.8493e-13 | 4.7166e-15 | 5.8932e-14 | 4.9600e-14 |
| $2^9$ | StrassenGinv | 0.093 | 7.5549e-13 | 3.6470e-15 | 6.4005e-14 | 5.5334e-14 |
| | Ginv | 0.107 | 6.5613e-13 | 3.2782e-15 | 5.3412e-14 | 5.1755e-14 |
| | SVDginv | 6.333 | 3.5550e-12 | 1.5408e-14 | 2.3639e-13 | 2.7610e-13 |
| | Geninv | 0.256 | 1.2738e-12 | 7.2824e-15 | 1.4280e-13 | 9.1078e-14 |
| | BlockGinv | 1.733 | 2.5091e-12 | 1.0303e-14 | 1.5812e-13 | 2.2397e-13 |
| | QRginv | 0.217 | 7.2006e-13 | 3.5794e-15 | 5.8785e-14 | 6.8930e-14 |
| | QRgeninv | 0.377 | 1.1043e-12 | 5.7272e-15 | 9.9434e-14 | 8.3876e-14 |
| $2^{10}$ | StrassenGinv | 0.733 | 1.6710e-12 | 3.9997e-15 | 1.0053e-14 | 9.1517e-14 |
| | Ginv | 0.809 | 1.4990e-12 | 3.6090e-15 | 8.6785e-14 | 8.8288e-14 |
| | SVDginv | 57.046 | 1.1796e-11 | 2.4447e-14 | 5.1025e-13 | 6.0068e-13 |
| | Geninv | 2.034 | 3.0237e-12 | 8.7119e-15 | 2.4130e-13 | 1.5331e-13 |
| | BlockGinv | 18.271 | 7.6135e-12 | 1.4917e-14 | 3.2681e-13 | 4.5971e-13 |
| | QRginv | 1.669 | 1.7424e-12 | 4.2267e-15 | 1.0119e-13 | 1.2447e-13 |
| | QRgeninv | 2.967 | 2.5765e-12 | 6.8131e-15 | 1.6832e-13 | 1.4754e-13 |
| $2^{11}$ | StrassenGinv | 5.689 | 3.6962e-12 | 4.3422e-15 | 1.5765e-13 | 1.47213-13 |
| | Ginv | 6.439 | 3.2355e-12 | 3.8423e-15 | 1.3306e-13 | 1.4073e-13 |
| | SVDginv | 711.256 | 3.9555e-11 | 4.1734e-14 | 1.1460e-12 | 1.4018e-12 |
| | Geninv | 16.582 | 6.5136e-12 | 9.4246e-15 | 3.7128e-13 | 2.4065e-13 |
| | BlockGinv | 189.970 | 2.4146e-11 | 2.3989e-14 | 7.0144e-13 | 1.0110e-12 |
| | QRginv | 13.475 | 4.2127e-12 | 5.0972e-15 | 1.7351e-13 | 2.2367e-13 |
| | QRgeninv | 23.886 | 5.9329e-12 | 7.8849e-15 | 2.7435e-13 | 2.5591e-13 |
| $2^{12}$ | StrassenGinv | 45.936 | 7.9101e-12 | 4.6136e-15 | 2.3986e-13 | 2.2501e-13 |
| | Ginv | 51.196 | 6.8391e-12 | 4.0177e-15 | 1.9954e-13 | 2.1556e-13 |
| | SVDginv | 12203.900 | 1.5909e-10 | 7.8725e-14 | 2.7615e-12 | 3.4351e-12 |
| | Geninv | 155.854 | 1.4014e-11 | 1.0265e-14 | 5.7139e-13 | 3.6920e-13 |
| | BlockGinv | 2076.670 | 8.8742e-11 | 4.1707e-14 | 1.5901e-12 | 2.3675e-12 |
| | QRginv | 121.825 | 1.0480e-11 | 6.3418e-15 | 3.0522e-13 | 4.0939e-13 |
| | QRgeninv | 233.104 | 1.3717e-11 | 9.0017e-15 | 4.4483e-13 | 4.5086e-13 |

inverse. In contrast, for $m = 2n$, the matrices have more rows than columns, which affects the efficiency of QR factorization in terms of necessary operations. Consequently, the computation times of the Moore–Penrose generalized inverse increase.

In general, the accuracy achieved for all algorithms is quite high, considering that the errors in the four Penrose conditions are $10^{-15}$ to $10^{-10}$. However, regarding the computation time, our method, which employs the Strassen algorithm, is the fastest in both scenarios (overdetermined and underdetermined matrices). This is very significant in real-time applications.

### B. PARALLEL COMPUTATION OF THE MOORE–PENROSE GENERALIZED INVERSE

Considering the results obtained from sequential computing experiments for various matrix dimensions (both overdetermined and underdetermined), we conducted our parallel computing experiments for $m = 2n$, and hence $rank = n$. In this context, we conducted initial numerical experiments using the proposed parallel computing method to determine the threshold of matrix dimensions at which our algorithm, which employed threads $p = 3, 6, 12, 24, 48$, surpasses the efficiency of the single-threaded case. To achieve this, we computed the Moore–Penrose generalized inverses of matrices with random ranks by varying the rank value in the range 50–400 range using increments of 50 in each iteration. Apart from the computation time, we conducted an accuracy analysis of the result when the program was executed using $p$ threads.

Table 4 presents the 2-norm of the first Penrose condition, which is the most susceptible to errors among the four conditions. A fluctuation in the $10^{-14}$ – $10^{-13}$ range is observed, which is consistent with the values obtained from the sequential computing experiments. It is worth noting that the 2-norm results obtained using a single thread are identical to those obtained using $p$ threads. These results demonstrate that in our method, the accuracy is not compromised when the number of execution threads increases. Thus, a level of accuracy comparable to that of sequential computing algorithms previously reported in the literature is maintained.

Regarding the computation time, there is variability, which depends on the matrix rank and the execution threads used. In matrices with a rank of $n = 50$, our method, which employs execution threads, is slower than its corresponding sequential computing method (1 execution thread). However, our algorithm becomes faster as the matrix rank increases.

**TABLE 4.** Computation time required for obtaining the Moore–Penrose generalized inverse and 2-norm of the four Penrose conditions in full-rank matrices using $p$ threads. The size of matrices is $m \times n$, where $m = 2n$ and $Rank = n$. The 2-norm corresponds to the first (most error-sensitive) Penrose condition.

| Rank | Time (sec) | | | | | | 2-norm |
|---|---|---|---|---|---|---|---|
| | 1 thread | 3 threads | 6 threads | 12 threads | 24 threads | 48 threads | |
| 50 | 0.0012 | 0.0014 | 0.0021 | 0.0037 | 0.0059 | 0.0165 | 3.3153e-14 |
| 100 | 0.0071 | 0.0043 | 0.0051 | 0.0080 | 0.0142 | 0.0289 | 9.8162e-14 |
| 150 | 0.0197 | 0.0125 | 0.0121 | 0.0144 | 0.0195 | 0.0451 | 1.7417e-13 |
| 200 | 0.0442 | 0.0198 | 0.0149 | 0.0238 | 0.0337 | 0.0669 | 2.6516e-13 |
| 250 | 0.0893 | 0.0541 | 0.0329 | 0.0328 | 0.0392 | 0.0636 | 3.7667e-13 |
| 300 | 0.1433 | 0.0574 | 0.0372 | 0.0409 | 0.0494 | 0.0955 | 4.8124e-13 |
| 350 | 0.2251 | 0.0881 | 0.0541 | 0.0509 | 0.0608 | 0.1059 | 6.0700e-13 |
| 400 | 0.3361 | 0.1285 | 0.0771 | 0.0628 | 0.0645 | 0.1122 | 7.4548e-13 |

**TABLE 5.** Computation time for obtaining the Moore–Penrose generalized inverse for large-sized full-rank matrices using $p$ threads. The size of matrices is $m \times n$, where $m = 2n$ and $Rank = n$. Each thread is executed exclusively on one core.

| Rank | Matrix size | Time (sec) | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 thread | 3 threads | 6 threads | 12 threads | 24 threads | 48 threads |
| 5,000 | 381.45MB | 795 | 256 | 137 | 70 | 36 | 27 |
| 10,000 | 1.49GB | 17,950 | 8,200 | 3,979 | 2,037 | 1,115 | 636 |
| 15,000 | 3.35GB | 49,493 | 22,283 | 10,985 | 5,517 | 2,946 | 1,793 |
| 20,000 | 5.96GB | 324,690 | 86,264 | 42,558 | 19,853 | 16,932 | 13,586 |
| 25,000 | 9,31GB | 509,379 | 173,753 | 73,592 | 49,143 | 30,143 | 24,767 |

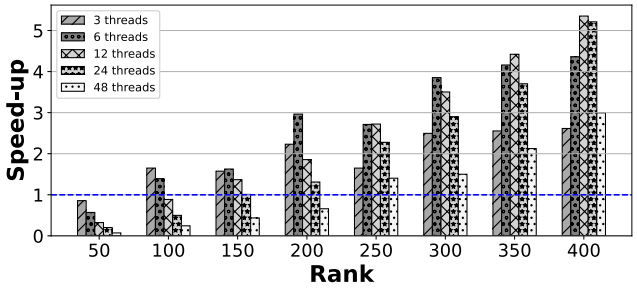


**FIGURE 5.** Speed-up of the Moore–Penrose generalized inverse for full-rank matrices. The size of matrices is $m \times n$, where $m = 2n$ and $Rank = n$. The blue line indicates that the computation time is the same when using a single thread. The speed-up is defined as $T(1)/T(p)$, where $T(1)$ is the running time for a sequential algorithm (executed on one core with one thread) and $T(p)$ is the running time for a parallel algorithm.

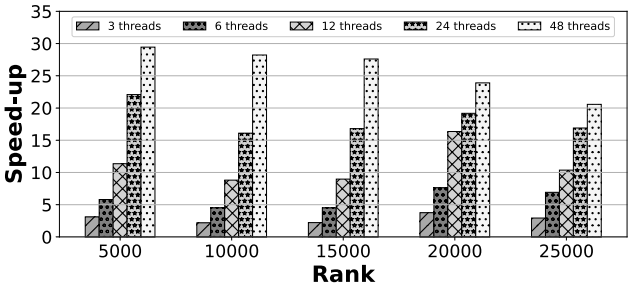


**FIGURE 6.** Speed-up of the Moore–Penrose generalized inverse for large-size full-rank matrices. The size of matrices is $m \times n$, where $m = 2n$ and $Rank = n$. The speed-up is defined as $T(1)/T(p)$, where $T(1)$ is the running time for a sequential algorithm (executed on one core with one thread) and $T(p)$ is the running time for a parallel algorithm.

For example, for a matrix rank of $n = 100$, our algorithm is faster when using 3 and 6 execution threads compared to that using a single thread but becomes slower when using 12 or more threads. When using 12 and 24 execution threads, our algorithm surpasses its sequential computing version in terms of speed when the matrix rank is $n \geq 200$. Ultimately, for our algorithm employing 48 execution threads to be efficient compared with its sequential computing version, the matrix rank should be $n \geq 250$. This is clearly shown in Fig. 5. The blue line indicates that the computation time is the same with that when using a single thread.

### C. PARALLEL COMPUTING WITH LARGE-SIZE FULL-RANK MATRICES

To assess the performance of our algorithm, we conducted experiments with large matrices. Similar to the previous case, we conducted experiments using $p$ execution threads, where $p = 3, 6, 12, 24, 48$, and compared the results with those obtained from sequential computing (1 execution thread). Additionally, we used randomly generated full-rank matrices with dimensions $m \times n$, where $m = 2n$ and $Rank = n$. The matrix sizes ranged from $n = 5,000$ to $n = 25,000$ with increments of 5,000 columns. In this scenario, we analyzed the calculation time and the speed-up. Table 5 presents the calculation time results obtained using our approach with $p$ execution threads. Fig. 6 shows the speed-up of our approach compared to its sequential computing version.

The results presented in Table 5 exhibit a consistent pattern: as the matrix size increases, there is a noticeable decrease in computation time for all scenarios. The advantage of our approach is shown in Fig. 6. We observe that the speed-up steadily increases, meaning that, as the number of execution threads increases, the speed-up also increases, resulting in a significant improvement in execution efficiency. These results are largely attributed to the Strassen algorithm

(Algorithm 5). The Strassen algorithm steps are executed sequentially, but each step is performed using a set of parallel threads. The use of parallel computing in these steps is the primary reason for the high speed-up observed in our experiments.

These results indicate a significant advancement in the computing efficiency of the Moore–Penrose generalized inverse by exploiting the power of parallel computing in a shared-memory architecture. The clear improvement in computation time and its speed-up, which is proportional to the number of execution threads, emphasize the potential of this strategy in applications requiring a fast and accurate calculation of the Moore–Penrose generalized inverse of full-rank large matrices.

## V. CONCLUSION

The computation of the Moore–Penrose generalized inverse is a significant operation in the field of artificial intelligence, particularly for training neural networks with randomly initialized weights, given the computational cost it entails.

In this work, we proposed and developed a parallel computing algorithm, based on the Strassen algorithm. Our algorithm computes the Moore–Penrose generalized inverse of full-rank rectangular matrices in a shared-memory hardware architecture. Initially, we assessed the performance of our method in a sequential computing environment (using a single core) by comparing it against that of other algorithms reported in the literature. In this environment, our method demonstrated its superiority in terms of computation time for the Moore–Penrose generalized inverse compared with previously reported algorithms. To assess the accuracy of our method, we evaluated the four Penrose conditions using the 2-norm. The results showed that our algorithm maintains a level of accuracy comparable to that of the other algorithms employed in the study.

In a parallel computing environment, the results demonstrated the efficiency of our algorithm when employing multiple execution threads, especially when the matrix size equals or exceeds 250 rows or columns, depending on the type of matrix (overdetermined or underdetermined). Additionally, our experiments showed an important aspect: the inherent accuracy of our algorithm remains intact as the number of execution threads increases, providing significant performance stability. In the case of large matrices, our algorithm can improve the speed-up by increasing the number of execution threads. These results not only validate the efficiency of our algorithm in large-scale parallel computing scenarios but also indicate its suitability for maintaining a robust accuracy, further enhancing its applicability to practical applications and theoretical analyses in the context of the Moore–Penrose generalized inverse and full-rank matrices.

Our proposal is inspired by the operation of neural networks with random weights and is thus a valuable reference to applications utilizing these neural networks. However, the proposed method may not suit applications dealing with target matrices exhibiting different characteristics, such as singular matrices. Moreover, our approach involves the generation of temporary matrices, necessitating an architecture with sufficient memory capacity.

In the future, a promising direction will be to integrate our method into random weight-based neural networks. This is particularly important as numerous architectures of these neural networks incorporate Moore–Penrose generalized inverse calculations, which often involve substantial matrix dimensions. Considering the current scale of databases and the increasing complexity of learning models, the efficient optimization of these calculations is essential. By integrating our methodology into this context, we could potentially enhance the efficiency and performance of large-scale matrix computation operations, contributing to the scalability and practical feasibility of these cutting-edge applications.

## ACKNOWLEDGMENT

The author Elkin Gelvez-Almeida is appreciative for the licenses of Ph.D. studies for the ''Fund for Teacher and Professional Development'' of Universidad Simón Bolívar, Colombia.

## REFERENCES

[1] J. C. A. Barata and M. S. Hussein, ''The Moore–Penrose pseudoinverse: A tutorial review of the theory,'' *Brazilian J. Phys.*, vol. 42, nos. 1–2, pp. 146–165, Apr. 2012.

[2] X.-D. Zhang, *A Matrix Algebra Approach to Artificial Intelligence*. Singapore: Springer, 2020.

[3] A. K. Malik, R. Gao, M. A. Ganaie, M. Tanveer, and P. N. Suganthan, ''Random vector functional link network: Recent developments, applications, and future directions,'' *Appl. Soft Comput.*, vol. 143, Aug. 2023, Art. no. 110377.

[4] P. Courrieu, ''Fast computation of Moore–Penrose inverse matrices,'' *Neural Inf. Process.-Lett. Rev.*, vol. 8, no. 2, pp. 25–29, Aug. 2005.

[5] J. K. Baksalary and O. M. Baksalary, ''Particular formulae for the Moore–Penrose inverse of a columnwise partitioned matrix,'' *Linear Algebra Appl.*, vol. 421, no. 1, pp. 16–23, Feb. 2007.

[6] M. D. Petković and P. S. Stanimirović, ''Generalized matrix inversion is not harder than matrix multiplication,'' *J. Comput. Appl. Math.*, vol. 230, no. 1, pp. 270–282, Aug. 2009.

[7] M. Petkovic and P. Stanimirovic, ''Block recursive computation of generalized inverses,'' *Electron. J. Linear Algebra*, vol. 26, pp. 394–405, Jan. 2013.

[8] F. Toutounian and A. Ataei, ''A new method for computing Moore–Penrose inverse matrices,'' *J. Comput. Appl. Math.*, vol. 228, no. 1, pp. 412–417, Jun. 2009.

[9] V. Katsikis and D. Pappas, ''Fast computing of the Moore–Penrose inverse matrix,'' *Electron. J. Linear Algebra*, vol. 17, pp. 637–650, Jan. 2008.

[10] V. N. Katsikis, D. Pappas, and A. Petralias, ''An improved method for the computation of the Moore–Penrose inverse matrix,'' *Appl. Math. Comput.*, vol. 217, no. 23, pp. 9828–9834, Aug. 2011.

[11] A. Marco and J.-J. Martínez, ''Accurate computation of the Moore–Penrose inverse of strictly totally positive matrices,'' *J. Comput. Appl. Math.*, vol. 350, pp. 299–308, Apr. 2019.

[12] W. Li and Z. Li, ''A family of iterative methods for computing the approximate inverse of a square matrix and inner inverse of a non-square matrix,'' *Appl. Math. Comput.*, vol. 215, no. 9, pp. 3433–3442, Jan. 2010.

[13] H. Chen and Y. Wang, ''A family of higher-order convergent iterative methods for computing the Moore–Penrose inverse,'' *Appl. Math. Comput.*, vol. 218, no. 8, pp. 4012–4016, Dec. 2011.

[14] P. S. Stanimirović, V. N. Katsikis, S. Srivastava, and D. Pappas, ''A class of quadratically convergent iterative methods,'' *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales. Serie A. Matemáticas*, vol. 113, no. 4, pp. 3125–3146, May 2019.

[15] R. Behera, J. K. Sahoo, R. N. Mohapatra, and M. Z. Nashed, "Computation of generalized inverses of tensors via t-product," *Numer. Linear Algebra With Appl.*, vol. 29, no. 2, p. e2416, 2022.
[16] V. Stanojević, L. Kazakovtsev, P. S. Stanimirović, N. Rezova, and G. Shkaberina, "Calculating the Moore–Penrose generalized inverse on massively parallel systems," *Algorithms*, vol. 15, no. 10, p. 348, Sep. 2022.
[17] J. Ma, F. Gao, and Y. Li, "An efficient method to compute different types of generalized inverses based on linear transformation," *Appl. Math. Comput.*, vol. 349, pp. 367–380, May 2019.
[18] P. S. Stanimirović, A. Kumar, and V. N. Katsikis, "Further efficient hyperpower iterative methods for the computation of generalized inverses $A_T, S^{(}2),$" *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales. Serie A. Matemáticas*, vol. 113, no. 4, pp. 3323–3339, May 2019.
[19] P. S. Stanimirović, V. N. Katsikis, and D. Kolundžija, "Inversion and pseudoinversion of block arrowhead matrices," *Appl. Math. Comput.*, vol. 341, pp. 379–401, Jan. 2019.
[20] P. S. Stanimirović, F. Roy, D. K. Gupta, and S. Srivastava, "Computing the Moore–Penrose inverse using its error bounds," *Appl. Math. Comput.*, vol. 371, Apr. 2020, Art. no. 124957.
[21] X. Chen and J. Ji, "A divide-and-conquer approach for the computation of the Moore–Penrose inverses," *Appl. Math. Comput.*, vol. 379, Aug. 2020, Art. no. 125265.
[22] N. Aldhafeeri, D. Pappas, I. P. Stanimirović, and M. Tasić, "Representations of generalized inverses via full-rank QDR decomposition," *Numer. Algorithms*, vol. 86, no. 3, pp. 1327–1337, Mar. 2021.
[23] A. Ben-Israel and T. N. E. Greville, *Generalized Inverses: Theory and Applications*, 2nd ed. New York, NY, USA: Springer, 2003.
[24] G. Wang, Y. Wei, S. Qiao, P. Lin, and Y. Chen, *Generalized Inverses: Theory and Computations*, vol. 53. Singapore: Springer, 2018.
[25] R. Penrose, "A generalized inverse for matrices," *Math. Proc. Cambridge Philos. Soc.*, vol. 51, no. 3, pp. 406–413, Jul. 1955.
[26] V. Strassen, "Gaussian elimination is not optimal," *Numerische Math.*, vol. 13, no. 4, pp. 354–356, Aug. 1969.
[27] H. D. Macedo, "Gaussian elimination is not optimal, revisited," *J. Log. Algebr. Methods Program.*, vol. 85, no. 5, pp. 999–1010, Aug. 2016.
[28] R. A. Horn and F. Zhang, "Basic properties of the Schur complement," in *The Schur Complement and Its Applications*, F. Zhang, Ed. New York, NY, USA: Springer, 2005, pp. 17–46.
[29] D. H. Bailey, K. Lee, and H. D. Simon, "Using Strassen's algorithm to accelerate the solution of linear systems," *J. Supercomput.*, vol. 4, no. 4, pp. 357–371, Jan. 1991.
[30] S. Lu, X. Wang, G. Zhang, and X. Zhou, "Effective algorithms of the Moore–Penrose inverse matrices for extreme learning machine," *Intell. Data Anal.*, vol. 19, no. 4, pp. 743–760, Jul. 2015.

**ELKIN GELVEZ-ALMEIDA** received the B.S. degree in mathematics and computer science from Universidad Francisco de Paula Santander, Colombia, in 2011, and the M.S. degree in basic sciences teaching-learning: mathematics from Universidad Nacional Experimental del Táchira, Venezuela, in 2016. He is currently pursuing the Ph.D. degree in applied mathematical modeling with Universidad Católica del Maule, Chile. He is an Assistant Professor with the Faculty of Basic and Biomedical Sciences, Universidad Simón Bolívar, Colombia, and an Associate Researcher with the Administrative Department of Science, Technology and Innovation, Colombia. His research interests include pattern recognition and mathematical modeling.

**RICARDO J. BARRIENTOS** received the B.S. degree in computer engineering from Universidad de Magallanes, Chile, the first M.Sc. degree in computer science from Universidad de Chile, Chile, and the second M.Sc. and Ph.D. degrees in computer science from Universidad Complutense de Madrid, Spain. He is currently an Assistant Professor with Universidad Católica del Maule, Chile, and the Director of the master's program in computer science. He is also a Co-Investigator of a Fondecyt Regular project, funded by the Government of Chile, and a Researcher with the LITRP Laboratory (www.litrp.cl). His main research interests include high-performance computing and biometrics.

**KARINA VILCHES-PONCE** received the B.S. and M.Sc. degrees in mathematics from Pontificia Universidad Católica de Chile, in 2007 and 2009, respectively, and the dual Ph.D. degree in mathematical modeling from Universidad de Chile and mathematical sciences from Sorbonne University, in 2014. She is currently an Associate Professor with the Faculty of Basic Sciences, Universidad Católica del Maule, Chile, where she develops research in interdisciplinary contexts. She is also a Researcher with the Laboratory for Technological Research in Pattern Recognition (LITRP, www.litrp.cl). Her research interests include the application of mathematical background to real problems in science, computation, and biology.

**MARCO MORA** received the B.S. degree in electronic engineering and the M.Sc. degree in electrical engineering from the Department of Electrical Engineering, Universidad de Concepción, Concepción, Chile, in 1998 and 2004, respectively, and the Ph.D. degree in computer science from Institut National Polytechnique de Toulouse (INPT), Université de Toulouse, Toulouse, France, in 2008. He is currently a full-time Professor with the Department of Computer Science and Industry, Universidad Católica del Maule, Talca, Chile. He is also the Head and a Senior Researcher with the LITRP Laboratory (www.litrp.cl), Universidad Católica del Maule. In addition, he teaches image processing and pattern recognition courses in the M.Sc. and Ph.D. programs at Universidad Católica del Maule. His research interests include digital image processing, neural networks, biometrics, and industrial applications of pattern recognition.