Parallelization of the Single Value Decomposition algorithm on MPI cluster

Leonardo Rigotti, Lorenzo Masè

January 2026

Disclaimer: Generative AI Usage

Generative Artificial Intelligence tools were utilized during the development of this project in compliance with the University of Trento Guidelines. It was used in particular for LaTeX formatting, code refactor and help in debugging. All the generated material has been revised by the authors and final responsibility on the developed material rests with the authors.

1 Introduction

In this document, we will discuss the parallelization of the Singular Value Decomposition (SVD) algorithm on an MPI cluster. Starting from a brief mathematical and logical definition of what SVD is and what its purposes are, we will then present two different approaches to perform the decomposition, exploring their parallelization strategies and performance.

2 Single Value Decomposition

Singular Value Decomposition can be seen as the generalization of eigen-decomposition for rectangular matrices [1], and as for eigen-decomposition, its main goal is to decompose a rectangular matrix into three simpler matrices, two orthogonal and one diagonal.

Based on this, the canonical mathematical definition of SVD is the following: Let $A \in \mathbb{R}^{m \times n}$. Then there exist two orthogonal matrices $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ such that:

$$A = U \Sigma V^T \tag{1}$$

where Σ is the diagonal matrix containing the singular values of A. The matrix Σ can be represented in block form as:

$$\Sigma = \begin{pmatrix} S & 0 \\ 0 & 0 \end{pmatrix} \tag{2}$$

where $S = \text{diag}(\sigma_1, \ldots, \sigma_r)$ and r is the rank of A. In this formulation, the matrix U contains the left singular vectors of A as its columns, which correspond to the eigenvectors of the symmetric matrix AA^T. Similarly, the columns of V (or the rows of V^T) represent the right singular vectors of A, which are the eigenvectors of A^TA [1]. A rigorous derivation of this theorem is available in [6].

Singular Value Decomposition is utilized across a wide range of fields [12]; as to Principal Component Analysis (PCA), its primary purpose is to identify the principal axes along which the data exhibits the most variance.

Thus, the computational core of SVD relies on determining the eigenvectors and eigenvalues of the square matrices AA^T and A^TA. Consequently, our parallelization strategy focuses on proper strategies for solving these eigenproblems. We have identified two algorithms suitable for this objective, which are presented below.

2.1 QR Decomposition

Let $A \in \mathbb{R}^{m \times n}$, with $m > n$, rank(A) = n. A can be decomposed in the product of an orthogonal $m \times n$ matrix Q, and an upper triangular $n \times n$ matrix R

$$A = QR \tag{3}$$

There are many algorithms to compute the QR decomposition [4], starting from the Classical Gram-Schmidt (CGS) and the Modified Gram-Schmidt (MGS), which offers a better stability to rounding errors [4]. The actual standard is the Householder Reflections (or Transformations), nevertheless we chose the CGS since is the one that offers the better parallelization opportunity, as the operations can be done all at the same time for a column. In [7] can be seen that the CGS always performs better in terms of execution time and efficiency when parallelized with respect to MGS and other algorithms, while it does not perform so well in terms of convergence and stability.

2.1.1 Classical Gram-Schmidt

In this brief explication of the algorithm, c_{ik} represents the i-th row and k-th column element of a matrix C, while $\mathbf{c}_k$ is the k-th column of a matrix C.

Given the matrix A as defined previously, the Classical Gram-Schmidt (CGS) algorithm consists of n steps. In each step, a column of A is transformed to form a corresponding orthonormal column of Q.

At the k-th step, we first compute the projection coefficients r_{ik} for $i = 1, \ldots, k-1$ by projecting the current column $\mathbf{a}_k$ onto the previously computed orthogonal basis vectors $\mathbf{q}_i$:

$$r_{ik} = \mathbf{q}_i^\top \mathbf{a}_k \tag{4}$$

Next, we subtract these projections from $\mathbf{a}_k$ to obtain the residual vector $\mathbf{u}_k$:

$$\mathbf{u}_k = \mathbf{a}_k - \sum_{i=1}^{k-1} r_{ik}\mathbf{q}_i \tag{5}$$

The resulting vector $\mathbf{u}_k$ is orthogonal to $\{\mathbf{q}_1, \ldots, \mathbf{q}_{k-1}\}$. Finally, we compute the diagonal element r_{kk} as the Euclidean norm of $\mathbf{u}_k$ and normalize to obtain the new Q column vector $\mathbf{q}_k$:

$$r_{kk} = \sqrt{\|\mathbf{u}_k\|}, \quad \mathbf{q}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|} \tag{6}$$

2.1.2 QR Algorithm

The QR decomposition serves as the foundation of the QR algorithm, which is the actual point of our project, since it is utilized to compute the eigenvalues and eigenvectors of a square matrix. The algorithm proceeds iteratively: given a square matrix A, we compute its decomposition $A = QR$ with CGS and subsequently form $A_1 = RQ$. This procedure is repeated until the sequence converges to an upper triangular matrix A_k. At this stage, the diagonal elements of A_k represent the eigenvalues of A. Furthermore, the eigenvectors of A correspond to the columns of the matrix obtained by accumulating (multiplying) all the orthogonal matrices Q generated during the iterations [14], the algorithm is represented in Algorithm 1.
Since our goal was not precision and perfect convergence of the algorithm, we chose a fixed number of steps for which repeat the orthogonalization instead of verifying the upper triangularity of A.

Algorithm 1 QR Algorithm for Eigenvalues and Eigenvectors (using CGS)

Require: Square matrix $A \in \mathbb{R}^{n \times n}$, max iterations K_{max}
Ensure: Eigenvalues λ, Eigenvectors V
$A^{(0)} \leftarrow A$
$V \leftarrow I$ ▷ Initialize eigenvector matrix as Identity
$k \leftarrow 0$
while $k < K_{max}$ **do**
    Step 1: Decomposition
    $(Q, R) \leftarrow$ CGS_Decomposition$(A^{(k)})$ ▷ Using Classical Gram-Schmidt
    Step 2: Update
    $A^{(k+1)} \leftarrow R \times Q$ ▷ Compute next iterate
    Step 3: Accumulate Eigenvectors
    $V \leftarrow V \times Q$ ▷ Update eigenvector basis

    $k \leftarrow k + 1$
end while
return $\lambda = \text{diag}(A^{(k)})$, V

2.2 Power method

The second method proposed is the power method. The goal of this iterative algorithm is to find the single dominant eigenvalue from a square matrix A, and the corresponding eigenvector; specifically, it is one of the oldest algorithms to solve this problem[8].

The Power Method by itself would be used only to find a single dominant eigenvalue and the related eigenvector, but different algorithms based on blocks have been discovered to find the most significant eigenvectors of matrices, reducing the complexity [2].

For our purpose, the Classical Power Method was used together with the Deflation to obtain multiple eigenvalues and eigenvectors, being one of the most used methods for this problem[11].

2.2.1 Classical Power Method

Starting from a non zero random vector $b_0 \in \mathbb{R}^n$ and a square matrix $A \in \mathbb{R}^{n\times n}$ the Power Method iteration is described as follows:

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|} \tag{7}$$

Therefore, the b vector is multiplied by the matrix A at each step and then normalized. Assuming that A has a unique eigenvalue of higher modulus

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|,$$

the vector b_k converges to the corresponding eigenvector of $|\lambda_1|$[2].

The core step of the algorithm is the *power iteration*, in which the initial vector is repeatedly multiplied by A, while the variation between successive iterates is monitored as an indicator of convergence toward the dominant eigenvector.

2.2.2 Deflation method

Once the dominant eigenvalue λ_1 and associated eigenvector v_1 of A are found. The Deflation method permits us to see further eigenvalues and eigenvectors.

The core of this method is the deflation step:

$$A_1 = A - \sigma_1 v_1 v_1^T \tag{8}$$

A_1 represents the new matrix used to find the next dominant. Once this step is done, a new Power Iteration can begin, with a new random vector b_0 and the matrix A_1.

A quite important matter that needs to be discussed is the floating-point round-off problem. An issue related to these iterative algorithms is that errors are propagated between iterations; these errors primarily come from the representation of variables in computer memory. As it is possible to notice in the equation 8, the new matrix obtained comes from the calculations made in the

previous iteration, meaning that, along with iterations, if there are small errors, these are propagated further.

Further solutions may apply Gram-Schmidt to re-orthogonalize the resulting matrix or block algorithms cited before, which increase the stability, but for our purpose, the normalisation and the deflation are already sufficient methods to test the parallelization strategies.

3 Parallelization strategies

3.1 Related Work

The parallelization of the Singular Value Decomposition (SVD) on MPI clusters is fundamentally tied to the effective parallelization of eigenvalue algorithms, as the SVD is intrinsically linked to finding the eigenvectors of AA^T and A^TA. Since the core algorithms for these problems have remained stable for decades, the relevant literature primarily spans from the late 20th century to the early 2000s.

Berry et al. (2005) [3] conducted a comprehensive study focusing largely on the Jacobi method. They introduced the QJAC algorithm , a hybrid approach combining QR Decomposition with the Jacobi method. In this scheme, the QR decomposition is applied to the initial matrix, after which the SVD is computed solely on the resulting upper-triangular matrix R using the Jacobi method. The final SVD is recovered by multiplying the resulting unitary matrices by Q. While they employed Householder transformations for the QR step, they noted that the Classical Gram-Schmidt (CGS) algorithm—the method selected for our work—offers superior parallelization potential, at the cost of numerical stability.

Wang et al. (2020) [13] utilized the Hestenes-Jacobi method and proposed a Maximum Data Sharing (MDS) ordering strategy to optimize data reuse on an FPGA-based engine. Their approach recognizes that a column processed by a specific Processing Unit (PU) can either be reused by that same PU or transferred to another. The MDS algorithm designates one of the two active columns as "private" (retained by the current PU for the next iteration) and the other as "public" (sent to a different PU). By essentially "wiring" the communication path, this method significantly reduces off-chip bandwidth usage.

Higham et al. (1993) [5] proposed computing the SVD via the Polar Decomposition . Their method first decomposes the matrix into a unitary factor U_p and a Hermitian factor H, computes the eigendecomposition of H, and finally reconstructs the SVD by combining the results, similarly to [3]. The critical contribution of this work is the use of Newton's iteration to compute the initial Polar Decomposition. Unlike standard approaches, Newton's iteration relies heavily on matrix multiplication and inversion which are Basic Linear Algebra Subroutine of Level 3. These operations operate on blocks of data, making them highly parallelizable and efficient on distributed systems.

3.2 Dataset

To generate the test data, we developed a Python script that produces `.csv` file containing the matrix elements. We created a dataset of 30 matrices with dimensions ranging from 500×500 to 1000×1000 (corresponding to 2.5×10^5 up to 10^6 total elements). The matrix entries are initialized with uniformly distributed random floating-point values in the interval $[-10.0, 10.0]$. In the C implementation, these files are read in the main function, where the first two values defines the number of rows and columns, respectively.

3.3 Matrix Multiplication

For the matrix multiplication step, we adopted a straightforward "Scatter-Compute-Gather" strategy based on 1D row-wise decomposition. It is important to note that standard MPI collective operations, such as `MPI_Bcast` and `MPI_Scatterv`, operate most efficiently on contiguous memory blocks. Therefore, prior to communication, we flatten the 2D matrices into single, contiguous 1D arrays. This linearization ensures that the data layout is compatible with MPI transmission buffers, avoiding the overhead of creating derived datatypes or sending rows individually.

The procedure then operates as follows:

1. **Data Partitioning (Scatter):** The left matrix A (of size $m \times n$) is split into horizontal bands. Each process receives a contiguous block of rows roughly equal to m/P, with the final process handling any remainder rows to account for indivisibility.

2. **Broadcast:** The right matrix B is broadcast in its entirety to all participating processes, ensuring that every node possesses the full column space required for the multiplication.

3. **Local Computation:** Each process independently calculates its corresponding slice of the result matrix, C_{local}, by multiplying its assigned rows of A by the full matrix B:

$$C_{local} = A_{local} \times B$$

   The resulting local block has the same vertical dimension as the local slice of A.

4. **Result Collection (Gather):** Finally, the local result slices C_{local} are gathered from all processes and concatenated to reconstruct the complete global matrix C.

3.4 QR Decomposition

Recalling our earlier discussion on the QR Algorithm for solving eigenproblems, the procedure iteratively decomposes the matrix A into Q and R, followed by

matrix multiplications to generate the next iterate and accumulate the eigenvectors. From this structure, it is evident that the core challenge in parallelization, aside from matrix multiplication, for which the literature is extensive, lies in effectively parallelizing the Classical Gram-Schmidt (CGS) algorithm. Thus, by breaking down the entire process, we have identified that the primary bottleneck requiring a specialized parallel solution is the CGS algorithm itself.

Since this algorithm works on one column at a time, we adopted a row wise partition of work, meaning that a number of elements of each columns get assigned to each process. In this scheme, the matrices A and Q are partitioned horizontally.

Let n be the dimension of the matrix and P be the total number of processes. We define the basic block size as $S = \lfloor n/P \rfloor$. Each process with rank p (where $0 \leq p < P$) is assigned a contiguous block of rows indexed from $start_p$ to $end_p - 1$, defined as:

$$start_p = p \times S$$

$$end_p = \begin{cases} start_p + S & \text{if } p < P - 1 \\ n & \text{if } p = P - 1 \end{cases}$$

Consequently, each process is responsible for computing and storing only the corresponding rows of the orthogonal basis Q and the upper triangular matrix R. The parallel algorithm for the i-th iteration proceeds as follows:

1. **Local Projections (Batch Computation):** Instead of computing the projection coefficients r_{ji} sequentially (which would require i separate communication events), each process computes the *partial* inner products for all previous columns $j < i$ simultaneously using its local data:

$$\text{local\_dot}_j = \sum_{k=start}^{end} Q_{kj} A_{ki} \quad \text{for } j = 0, \ldots, i-1$$

2. **Global Reduction:** A single `MPI_Allreduce` operation sums these partial results across all processes to produce the global projection coefficients stored in the i-th column of R:

$$r_{ji} = \sum_{p=0}^{P-1} (\text{local\_dot}_j)_p$$

   This "batching" of inner products significantly reduces network latency, addressing the primary bottleneck of distributed CGS.

3. **Local Update:** With the global coefficients r_{ji} now available to all processes, each rank updates its local segment of the working vector $\mathbf{u}$:

$$u_k = A_{ki} - \sum_{j=0}^{i-1} r_{ji} Q_{kj} \quad \text{for } k \in [start, end)$$

4. **Normalization:** To normalize the vector, a local squared norm is computed and then summed globally using a second `MPI_Allreduce`. The square root of this sum yields the diagonal element r_{ii}, which is then used to scale the local segment of $\mathbf{u}$ to obtain the new orthogonal segments of $\mathbf{q}_i$.

5. **Synchronization:** Finally, an `MPI_Allgatherv` operation is performed to gather the distributed segments of the newly computed column $\mathbf{q}_i$. This ensures that, at the end of the step, every process possesses the full column, facilitating simpler data management for subsequent operations.

To avoid the communication overhead due to the distribution of the matrices A and Q at each iteration, A gets distributed in the main function as soon as it is read-as it is explained in the dataset section-while Q is kept updated on each node at each iteration.

3.4.1 QR Algorithm for SVD

To perform the Singular Value Decomposition (SVD), we first compute the matrices A^TA and AA^T, then apply the QR Algorithm to determine their eigenvectors. Specifically, the eigenvectors of A^TA correspond to the right singular vectors (V in Equation 1), while the eigenvectors of AA^T correspond to the left singular vectors (U in Equation 1).

The singular values (Σ) are derived from the diagonal elements of the final matrix A_k produced by the QR Algorithm after k iterations. It is important to note that our implementation executes a fixed, pre-determined number of QR iterations rather than iterating until strict convergence (when A becomes strictly upper triangular).

Regarding the parallel strategy, the accumulation of the orthogonal matrices Q—which yields the final eigenvector matrix—is gathered and computed exclusively at Rank 0. When decomposing A^TA, this accumulated result is V (or V^T), whereas for AA^T, it represents U. Apart from the QR decomposition itself and the matrix multiplications (which are parallelized as detailed in Section 3.3), the remainder of the algorithm is executed serially and is therefore not a primary focus of this parallelization analysis. The complete code, properly documented and commented on, can be seen in [9].

3.4.2 Data dependencies

Regarding the Classical Gram-Schmidt (CGS) procedure—the core of our parallelization strategy—several variables require synchronization via MPI communication. The process begins with the computation of two local variables, `local_dots` and `local_norm`, which store the results of matrix multiplications performed independently on each node's data partition. These partial results are subsequently aggregated across all processes to form the global variables `global_dots` and `global_norm`, corresponding to the entries of the matrix R and the normalization factors for Q. Crucially, the collection of these global

variables is a blocking operation, requiring all nodes to reach the synchronization point before the algorithm can proceed.

Table 1: Data Dependencies in Parallel CGS Algorithm

Variable	**Scope**	**Dependency Type**	**Blocking?**
`local_dots`	Local	None (Computed independently)	No
`global_dots` (R_{ji})	Global	All-to-All Reduction (Sum)	Yes
`local_norm`	Local	None (Computed independently)	No
`global_norm` (R_{ii})	Global	All-to-All Reduction (Sum)	Yes

The communication overhead introduced by these collective operations significantly influences the overall performance of the algorithm. This trade-off between parallel computation and synchronization costs makes our solution particularly effective with a sufficient problem size and number of nodes. These performance characteristics are analyzed in detail in the following section, where we present our experimental results and discuss the scalability of the implementation.

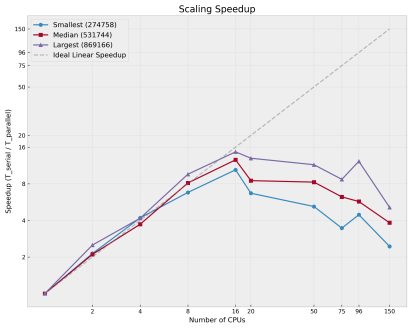
3.4.3 Results

We evaluated the performance of our algorithm on the High Performance Computing (HPC) cluster at the University of Trento by varying both the number of MPI processes and the matrix dimensions. To ensure robust results, the algorithm was executed on all 30 matrices from the dataset described in Section 3.2. For each test case, the matrix size (number of elements) and the corresponding total execution time were recorded in a text file.

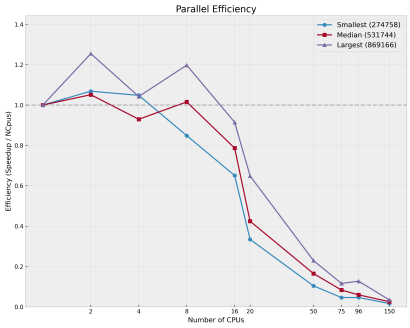
The experiments were managed using the Portable Batch System (PBS) scheduler. To establish a baseline for performance metrics, serial execution times were obtained by submitting jobs with a single process (`mpirun -n 1`). Subsequently, we scaled the number of processes to assess parallel performance. We analyzed the scalability of the implementation by plotting the speedup and efficiency for three representative cases: the smallest, the median, and the largest matrices in the dataset.

As illustrated in Figure 1a, the speedup peaks at 16 processes, after which performance begins to degrade, with only occasional fluctuations observed for the largest matrix. This trend highlights the limited scalability of the current implementation. The primary bottleneck stems from the interplay between the row-wise data distribution and the low computational intensity of the algorithm, which consists primarily of multiplications between double types.

Each process is assigned n/P rows, where n is the total number of rows and P is the number of processes. Scalability is limited by the trade-off between computation and communication: there is a threshold where the time saved by parallelizing the arithmetic operations on n/P elements is outweighed by the communication overhead required to distribute and gather data across the nodes. For our dataset, which has an average row count of 750 (ranging from



(a) Speedup analysis for small, medium, and large matrices.



(b) Efficiency trends as the number of processes increases.

Figure 1: Performance metrics on the HPC Cluster. The left plot (a) shows the speedup, while the right plot (b) illustrates the parallel efficiency.

500 to 1000), this saturation point occurs at approximately 16 processes. This suggests that the optimal workload granularity for this algorithm is approximately 44 elements (750/16) per process; below this threshold, communication costs dominate the execution time.

3.4.4 On nodes placing strategies

Accordingly with the analysis above, if communication overhead is the primary bottleneck, performance should improve by minimizing the physical distance between processes. This hypothesis can be verified by manipulating the node placement strategy via the PBS scheduler.

- **Pack / Pack Exclusive:** This strategy fills all available slots on a single physical node before allocating a new one. By grouping processes onto the fewest number of nodes possible, "packing" maximizes data locality. This is particularly beneficial for communication-bound applications, as it allows processes to communicate via high-speed shared memory rather than the network connections. The "exclusive" variant ensures that entire nodes are dedicated to the job, preventing interference from other users' tasks.
- **Scatter / Scatter Exclusive:** In contrast, this strategy distributes processes cyclically across as many distinct nodes as possible (e.g., placing only one process per node). While this maximizes the available memory bandwidth and cache capacity per process, it forces nearly all communication to travel over the network.

Consistent with these two definitions, the most suitable strategy to increase the performance of our solution is the Pack one, so fill all the CPUs on a single node. This should considerably decrease the communication overhead,

reducing the total execution time. On the other hand, the Scatter strategy is in contrast with our goal of reducing communication latency, so it should worsen the performance of our solution.

To empirically verify these hypotheses, we selected the optimal configuration identified in our initial experiments—specifically, the 16-process case—and evaluated it under all four node placing strategies.

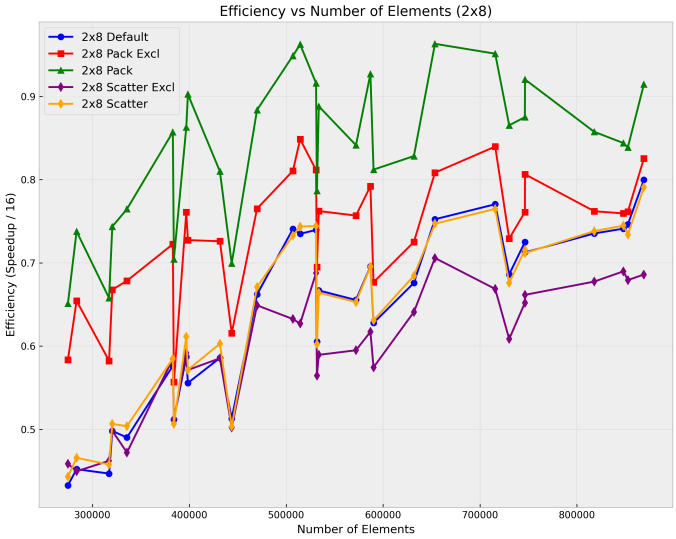


Figure 2: Efficiency versus matrix size (number of elements) for the 16-process configuration across four different node placement strategies.

As illustrated in Figure 2, the experimental results confirm our theoretical expectations. The Pack strategy consistently outperforms the others, demonstrating superior efficiency. This result reinforces the conclusion that the primary performance bottleneck is the trade-off between computational gain and communication latency.

3.5 Power method

In this section, the parallelized Power Method for SVD is explained together with the data dependencies related to the parallel approach. The code is developed in C with basic mathematical libraries and **mpi** for the parallelization and can be found in this directory [9]. As explained previously, the Power Method for SVD used is based on power iteration and deflation to compute multiple eigenvalues and eigenvectors with a single execution.

3.5.1 Power Method for SVD

The code is developed to run with a dataset that has the same structure as the one described in 3.2, as the execution is based dynamically on information that has to be inside the dataset, such as the number of matrices and for the single matrix, the number of columns and rows. Firstly, the data is read by a single process, *process 0*, and saved inside a dynamic array of **doubles**.

Then, the sizes of the current matrix are sent to all other processes with an **MPI_Bcast**, as these are required for boundaries. The matrix is then divided into rows for each process, and the data is sent using the **MPI_Scatterv** function, which permits sending a specific amount of data for each process. **MPI_Barrier** is used to ensure that the starting time is accurate for performance benchmarking across all the processes.

Once all of the processes have the data, each process starts with a shared random vector and starts the **Power Iteration**: during the power iteration, the following formula is implicitly computed by combining the single results of each process.

$$v_{k+1} = \frac{A^T A v_k}{||A^T A v_k||} \tag{9}$$

which is exactly the formula proposed in 2.2.1 with the square matrix $A^T A$. The Power Iteration, in our case, has an upper limit of 50 iterations. This choice made it so the results would still be correct while maintaining a decent time complexity.

Each local row's result is computed by calculating $w_{local} = A_{local} * v$, and then $z_{local} = A^T_{local} * w_{local}$. Then, with an **MPI_Allreduce** call, all the local results of the processes are combined and distributed to each process, as an array of doubles $y = A^T A v$, which permits us then to compute $\lambda = ||y||$.

Then, two checks are made to ensure that lambda still contains useful information (e.g., it is not zero, or we have already converged), and if that's the case, the Power Iteration is stopped. Otherwise, v is updated as $v = \frac{y}{\lambda}$.

Once the Power Iteration is finished, λ is the dominant eigenvalue, which permits us to compute the dominant Singular Value as:

$$\sigma = \sqrt{\lambda} \tag{10}$$

Then, a similar check to the one made before on λ is done over σ to check whether it still contains information; if not, it means we have found all the possible non-zero Singular Values.

Then, the function *compute_u* is parallelized to compute the dominant left singular vector as:

$$u = \frac{Av}{\sigma} \tag{11}$$

using the same parallel approach as before, so each process computes its local rows of the vector u. In this case, it is not required to gather the complete vector u across all processes, since the deflation step can be performed locally on each process.

At last, the Deflation method explained in 2.2.2 is executed, each process using their local variables u_{local}, A_{local}, $local\_rows$, and σ computes:

$$A_{local,k+1} = A_{local,k} - \sigma u_{local} v^T \tag{12}$$

which is the same formula explained in 2.2.2 for our matrix A. Once the deflation is complete, the next Singular Value can be calculated. This process is repeated k times, effectively finding all the non-zero Singular Values of the matrix A.

3.5.2 Data dependencies

For the Power Method for SVD developed, the single problematic data dependency is the $MPI\_Allreduce$ call inside the Power Iteration to compute y. As y cannot be computed if the z_{local} of each process is not computed, creating a dependency between the different processes.

In section 3.5.3, this dependency is checked and empirically proved using placing strategies of processes inside of nodes.

3.5.3 Results

The algorithm has then been tested in the High Performance Computing (HPC) cluster of the University of Trento. The main ideas for the testing section are based on two concepts: first, the number of processes running the code, and secondly, how the placement strategies of the processes would impact the parallelization with the communication overhead.

The dataset used for all the tests is the one described in 3.2, which contains 30 matrices with different dimensions up to 10^6 total elements.

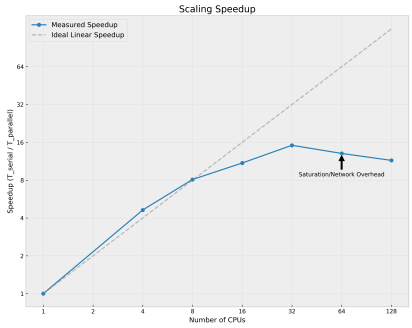
Firstly, the serial version of the algorithm was tested, which has been used as baseline knowledge to compute the speedup and the efficiency of the parallelized approach. Then, for each execution, the number of processes has been doubled to check for the best result without specifying the placement of the processes with PBS directives.

As shown in figure 3a and 3b, both the trend of the speedup of the algorithm and the parallel efficiency were computed.

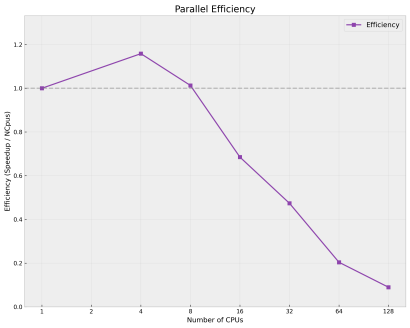
The results of the speedup show that our Power Method for SVD has the best speedup with 32 processes running, while at around 64 processes, the communication overhead slows down the execution, as the time required for the computation is greater than the previous one.

If this loss in performance is due to the communication required between the processes to share the z_{local} to compute y explained in 3.5.1, the results obtained can vary based on the placement of the processes; if the processes are physically closer to each other, the communication overhead would be lower, while if the physical distance between the processes is higher, the computation would require more time.

Other than the bottleneck, it is possible to notice a superlinear speedup up to 8 processes; this can happen as the I/O components may fasten the algorithm with specific operations, such as optimized cache memory retrievals[10].



(a) Speedup analysis for 2^1 to 2^7 processes.



(b) Efficiency trend with increasing number of processes.

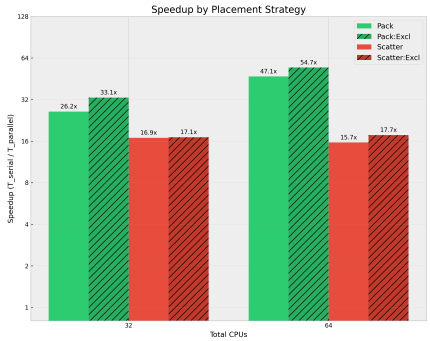
Then, graph 3b provides the trend for the efficiency of the algorithm, as it is shown that up to 4 processes, the efficiency is higher than 1, showing the super-linearity of the algorithm. While after 16 processes the efficiency collapses, this happens because of the bottleneck due to the synchronization between a larger number of processes.

At last, to analyze more deeply the possibilities of the algorithm, starting from the knowledge obtained from the previous testing, more specific tests were done, checking how the placement strategies would change the results.

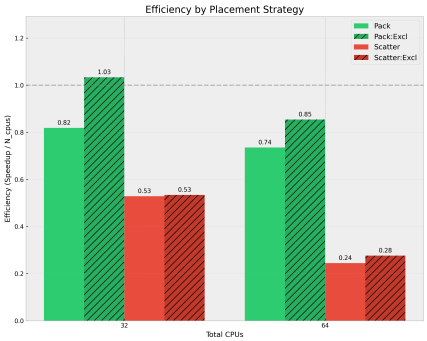
3.5.4 Placing strategies

The analysis focuses on executions with 32 (4*8) and 64 (8*8) processes, since the effect of placement becomes more important as the number of processes increases. As shown in 3.5.3, the best speedup was observed in the 32-process scenario, while the 64-process case likely suffers from communication saturation. Therefore, we consider these two cases the most interesting for analyzing placement strategies explained in 3.4.4.

As shown in 4a and 4b, the results represent what was expected, as the **pack** and **pack:excl** strategies solve the communication overhead issue related to the z_{local} values computed and shared between the processes. While **scatter** performs equally, as shown in 3a and 3b, and **scatter:excl** performs slightly better than its non-exclusive version.



(a) Speedup analysis for all the different placement strategies for 32 and 64 processes.



(b) Efficiency analysis based on the speedup obtained with the placing strategies.

Figure 4: Comparison of the different placing strategies for 32 and 64 processes running, (a) shows the new speedup obtained with each strategy, while (b) represents the efficiency based on the serial execution.

These results show that the communication overhead is the main issue related to this algorithm, as demonstrated with the placing strategies, as pack:excl outperforms for both 32 and 64 processes by keeping the processes inside the same nodes.

3.6 Comparison between the two strategies

Putting aside the speedup results, the performance of both algorithms is quite similar. In particular, they both start to recede from the ideal speedup line ($speedup = n_p$) after 16 processes, with the Power Method that peaks later at 32 processes. This indicates a slightly higher scalability for the latter algorithm, though overall the performances are fully comparable.

The main difference between the two parallelized approaches is the data dependencies explained in 3.4.2 and 3.5.2, due to the implementation of the Gram-Schmidt method. Multiple dependencies are created in this step, which makes the algorithm very reliant on the speed of the communication between the different processes.

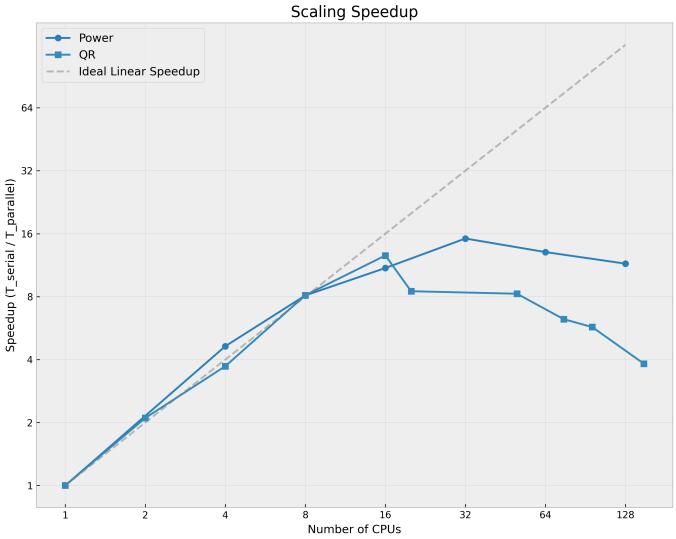


Figure 5: Speedup combined for the two algorithms.

4 Conclusion

This document implements two different parallelized algorithms for SVD, the QR method and the Power Method. Testing of the algorithms, and an analysis of the parallel techniques implemented, is provided together with the code and results in [9].

In conclusion, both methods are correctly parallelized, as the results provided show; the main problems related to the communication headers are proven and discussed through placement strategies.

For future works, both algorithms could be implemented differently, for example, the implementation of Gram-Schmidt for the Power Iteration, which brings more stability during the process, and the algorithm would be less prone to error propagation. Other algorithms as the ones discussed in 3.1, could be tested as well, to look for more robust results for the communication header problems.

Another possibility is the one of testing these algorithms on a different cluster than the one used by the University of Trento, as there would be a possibility to test with even more processes running, looking for possibly better results together with the placement strategies.

References

[1] Hervé Abdi. "Singular value decomposition (SVD) and generalized singular value decomposition". In: *Encyclopedia of measurement and statistics* 907.912 (2007), p. 44.

[2] Abdeslem Bentbib and A. Kanber. "Block Power Method for SVD Decomposition". In: *Analele Stiintifice ale Universitatii Ovidius Constanta, Seria Matematica* 23 (June 2015), pp. 45–58. DOI: 10.1515/auom-2015-0024.

[3] Michael W Berry et al. "Parallel algorithms for the singular value decomposition". In: *Handbook of parallel computing and statistics*. Chapman and Hall/CRC, 2005, pp. 133–180.

[4] Walter Gander. "Algorithms for the QR decomposition". In: *Res. Rep* 80.02 (1980), pp. 1251–1268.

[5] Nicholas J Higham and Pythagoras Papadimitriou. *Parallel singular value decomposition via the polar decomposition*. University of Manchester, Department of Mathematics, 1993.

[6] V. Klema and A. Laub. "The singular value decomposition: Its computation and some applications". In: *IEEE Transactions on Automatic Control* 25.2 (1980), pp. 164–176. DOI: 10.1109/TAC.1980.1102314.

[7] Frederik Jan Lingen. "Efficient Gram–Schmidt orthonormalisation on parallel computers". In: *Communications in numerical methods in engineering* 16.1 (2000), pp. 57–66.

[8] Richard von Mises and Hilda Pollaczek-Geiringer. "Praktische Verfahren der Gleichungsauflösung". In: *Zeitschrift für Angewandte Mathematik und Mechanik* 9 (1929), pp. 152–164.

[9] Leonardo Rigotti and Lorenzo Mase. *Official project repository, Singular Value Decomposition parallel*. URL: https://github.com/leorigo2/svd-parallel.

[10] Sasko Ristov et al. "Superlinear speedup in HPC systems: Why and when?" In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016, pp. 889–898.

[11] Weiya Shi and Dexian Zhang. "The power and deflation method based kernel principal component analysis". In: *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*. Vol. 1. 2010, pp. 828–832. DOI: 10.1109/ICICISYS.2010.5658803.

[12] Michael E Wall, Andreas Rechtsteiner, and Luis M Rocha. "Singular value decomposition and principal component analysis". In: *A practical approach to microarray data analysis*. Springer, 2003, pp. 91–109.

[13] Yu Wang et al. "A scalable FPGA engine for parallel acceleration of singular value decomposition". In: *2020 21st International Symposium on Quality Electronic Design (ISQED)*. IEEE. 2020, pp. 370–376.

[14] David S Watkins. “Understanding the QR algorithm”. In: *SIAM review* 24.4 (1982), pp. 427–440.