UNIVERSITÀ
DI PAVIA

Implementation of Optimized matrix multiplication and inversion with MPI

Shayan Kamalzadeh

University of Pavia, Department of Computer Engineering, Pavia, Italy

Date: October 08, 2024

Abstract:

This project focuses on developing a computer program that efficiently processes multiple rectangular number grids (matrices) and performs matrix inversion quickly. The program, written in C++ using MPI, leverages parallel computing to speed up operations by using multiple processors simultaneously. I converted a simple, sequential program into one that can perform tasks concurrently, significantly improving its performance. I identified key areas where parallel processing could be applied and tested the program using various data sizes and processor counts. These tests were conducted on the Google Cloud Platform, allowing for the use of numerous computers working together. The new approach not only increases speed but also makes the program scalable for more complex matrix operations.

Contents

1. WHAT IS CONCEPT OF ARTICLE

Multiple Matrix [1]:

A matrix is a rectangular array of numbers, symbols, or expressions organized into rows and columns. For example, a matrix with 2 rows and 3 columns would look like this:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

In many scientific and mathematical applications, it is common to work with multiple matrices simultaneously. Operations involving these matrices can include addition, subtraction, multiplication, or more complex tasks, depending on the application.

Matrix Inversion [2]:

Inverting a matrix is akin to finding the reciprocal of a number but in the context of matrices. It is a way of finding another matrix that, when multiplied with the original matrix, gives the identity matrix (a matrix with 1s on the diagonal and 0s elsewhere).

The inverse of a matrix A is usually denoted as A^{-1}, and it only exists if A is square (same number of rows and columns) and non-singular (it has a unique solution).

Method of Matrix Inversion:

I implemented Gaussian Elimination to solve matrix inversion, as this was an applicable method for the inversion of most matrix types and is simple to understand. To put it more precisely, I did the implementation based on **Gauss-Jordan elimination**, where code systematically goes through and applies necessary row operations to bring the matrix down to its reduced row echelon form while simultaneously transforming the identity matrix into the inverse. It provides an implementation for a range of different-sized square matrices, although it includes a time complexity of $O(n^3)$ since it must manipulate all its elements in the row operations against every other row and column, which becomes cumbersome with larger matrices.

Matrix Multiplication Implementation:

In my project, I created a function called "**multiply Matrices,"** which multiplies two square matrices. This function utilizes a conventional method of matrix multiplication with three nested loops to calculate the product of the matrices. The approach is straightforward and is a common practice in serial matrix multiplication. The computational complexity of this matrix multiplication is $O(n^3)$, where 'n' is the dimension of the matrices. This indicates that as the size of the matrices increases, the time taken to calculate the product escalates significantly. This is because each element in the resulting matrix requires 'n' multiplications, and there is a total of 'n^2' elements to compute.

2. Considerations for Speedup:

A key focus in my project is the potential for speeding up these computations, typically achieved through parallelization. This involves distributing the matrix operations across multiple processors or cores to enable simultaneous execution of multiple operations.

In the serial approach, as in my current implementation, operations are executed sequentially, one after the other. This method, while simple in concept, becomes increasingly time-intensive with larger matrices due to its cubic time complexity.

Transitioning to a parallelized approach can significantly reduce computation time, especially for large-scale matrix operations. However, it introduces additional complexities, such as managing dependencies between parallel tasks and ensuring synchronization. These aspects need to be carefully managed to ensure the parallel version offers a true speed advantage over the serial approach.

My project features a serial implementation of matrix operations, which is straightforward but can be slow for large matrices due to the $O(n^3)$ complexity. Exploring parallelization could offer substantial improvements in processing speed, particularly for large matrix operations.

In my project, I explored the potential performance gains from parallelizing the serial solution I developed for matrix multiplication and inversion. Initially, I analysed the serial code, setting aside less critical parts like parameter checks in the main function, library imports, and global variables. The only part of my code that posed challenges for efficient parallelization was the reading of datasets. All other functions were suitable for leveraging multiple CPU architectures.

To understand the possible speedup from parallelizing the serial version, I applied Amdahl's Law. This principle provides a formula to estimate the maximum improvement in performance achievable through parallelization. The law's formula is: $SpeedUp(\mathrm{N}) = \frac{1}{S+\frac{P}{N}}$

Here, N represents the number of CPUs used S is the fraction of the code that must remain serial, and P is the fraction that can be parallelized (with P+S=1)

My analysis revealed that the number of iterations for each task is closely tied to the dataset's size. The speedup depends not only on the serial and parallelizable portions of the code but also on the dataset's size. As the dataset grows, the serial portion of the code tends to diminish (approaching zero), suggesting that speedup should ideally increase linearly with the number of cores.

However, from a practical standpoint, the dataset's size can negatively impact execution time. This is due to the data transmission between nodes, which grows proportionally with the size of the training and testing samples. Therefore, while the theoretical perspective suggests a linear increase in speedup with additional cores, the actual performance gain might be less due to these data transmission overheads.

In my project demonstrates that while parallelization can significantly speed up matrix operations, especially for large datasets, the actual performance gains may vary based on factors like data size and transmission overheads.

In the context of my project, which involves parallelizing the operations of multiple matrix multiplication and inversion, the implementation using Open-MPI becomes crucial. Parallelizing tasks in computational fields, especially those that involve complex matrix operations, is essential to manage large-scale computations efficiently. Open-MPI offers a way to distribute these tasks across multiple nodes, enabling faster processing and handling of large datasets that would be challenging for a single CPU.

3. Implementation:

Parallelization Strategy: The primary goal is to divide the matrix multiplication and inversion tasks across different nodes. Unlike the serial approach where computations are done sequentially, the parallel approach leverages multiple CPUs to perform operations concurrently.

Open-MPI Implementation:

Using Open-MPI, I designed a distributed memory system where each node operates independently with its memory and computational power. This setup is crucial for large matrix operations as it allows for scalability and more efficient memory usage.

Master-Slave Nodes:

The master node in my implementation is responsible for distributing the matrix data among the slave nodes. For matrix multiplication, the master node divides the matrices into sub-matrices and distributes them to slave nodes. Each slave node then performs multiplication on its sub-matrices

and sends the results back to the master node, which assembles the final matrix.

Matrix Inversion [3]:

For matrix inversion using Gauss-Jordan elimination, the master node distributes rows or columns of the matrix to different slave nodes. Each slave node performs the necessary row operations and communicates with other nodes as required to progress the elimination process. The master node then combines the results to form the inverted matrix.

The code efficiently uses MPI to parallelize the matrix inversion process using Gauss-Jordan elimination. By leveraging collective operations like *MPI_Scatter*, *MPI_Bcast*, and *MPI_Gather*, it minimizes communication overhead and distributes computation effectively across multiple processes. This method ensures that the inversion is performed quickly, especially for large matrices, making it an ideal choice for parallel computing environments.

Matrix multiplication [4]:

In the implementation, I chose to use *MPI_Gather*, instead of implementing communication by hand, using *MPI_Send* and *MPI_Recv*, for example. The main reason for doing so is that *MPI_Gather* will automatically collect the partial results from all processes and assemble the final matrix in the root process. This collective communication function is more efficient and less error-prone compared to realizing point-to-point communication by using the functions *MPI_Send* and *MPI_Recv*, because in that way each process must take care of the transfer of its data. Moreover, *MPI_Gather* is optimized for such operations, keeping the communication overhead as low as possible and, hence, increasing overall performance. Thus, MPI_Gather is more effective in the context of this task on matrix multiplication.

4. Performance and Scalability:

By distributing the workload across multiple nodes, my project significantly reduces the time required for complex matrix operations. The scalability of this implementation allows processing speed to be maintained or improved as matrix sizes increase by adding more nodes. In the following sections of the article, you will find graphs and comparisons that illustrate the performance improvements and demonstrate how the implementation scales with varying matrix sizes and numbers of nodes.

5. Challenges in Parallelization:

1. Communication Overhead: As CPU count increases, MPI communication between processors adds delay, especially in matrix inversion, which requires more synchronization between processes.
2. Load Imbalance: In matrix inversion, the workload is not evenly distributed across CPUs, particularly with 6 CPUs, leading to worse performance than with fewer CPUs.
3. Memory Bandwidth: With larger matrices, multiple CPUs accessing shared memory can cause slowdowns, limiting the performance gains, especially in complex tasks like matrix inversion.
4. Scalability Issues: Adding more CPUs does not always result in better performance due to increased communication and synchronization, especially in inter-regional clusters with higher latency.
5. Task Granularity: Matrix multiplication benefits from simpler, independent tasks, while matrix inversion's more interdependent tasks make parallelization harder and less effective.

6. Cluster Setup for Testing Matrix Operations

To conduct the performance analysis of matrix inversion and multiplication, I created several clusters on the Google Cloud Platform. As shown in the figure below, these virtual machines (VMs) are distributed across different geographic zones and are configured with varying levels of computational resources. Some of the clusters are configured as **intra-region** and **infra-region** setups, allowing me to test the effects of geographic distribution on matrix operations. Additionally, I used different infrastructure types, including **fat** and **light** configurations, to evaluate how resource allocation impacts the performance of parallel processing tasks. By running the same matrix operations on these clusters, I was able to compare the performance of different setups and configurations, which will be analysed in the next section.

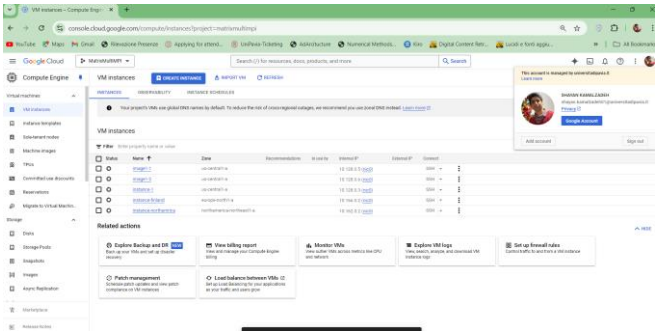

Fig. 2: Google Cloud Account for creating different Clusters.

UNIVERSITÀ
DI PAVIA

7. Comparing Speedup Across Different Clusters on Google Cloud

In this section, I compare the speedup achieved during matrix inversion and multiple tests on various cluster configurations using Google Cloud.

Performance on n1-fat-infra-Cluster

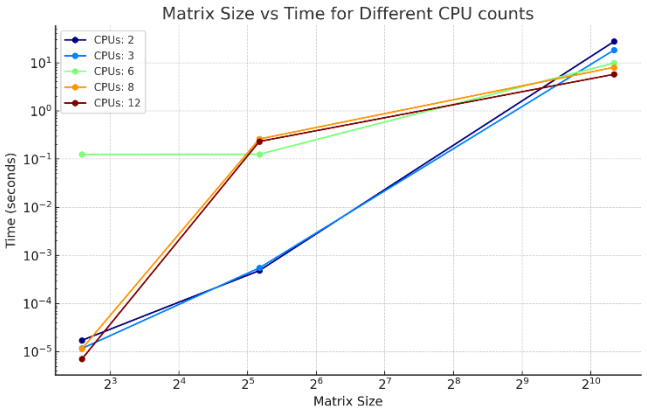

Fig. 3 Compare Multiplication **Mpi Code for fat infra cluster.**

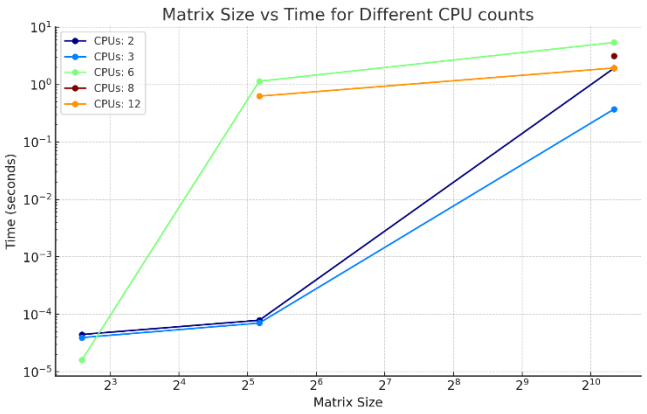

Fig. 4: **Compare** inversion **Mpi Code for fat infra cluster.**

Both matrix inversion and matrix multiplication tasks were tested on an N1-fat-infra cluster in Google Cloud, with MPI for parallel execution. The following observations were made based on the provided graphs:

Matrix Inversion (Fig. 4):

For small matrix sizes (2^3 to 2^5), the time to perform matrix inversion is minimal across all CPU configurations, regardless of the number of processors (ranging from 2 to 12 CPUs). This is likely due to the overhead associated with parallelizing smaller tasks not being significant enough to affect performance. As the matrix size grows (e.g., 2^8 and larger), the benefits of parallelization become more apparent. The time reduction is clear when moving from 2 CPUs to 12 CPUs. However, the system with 6 CPUs shows an unexpected increase in time for large matrix sizes, which may indicate inefficiencies in communication or load balancing when using MPI. For 12 CPUs, the time performance is the best overall, but this also comes with increased cloud resource consumption, which should be considered for large-scale deployments.

Matrix Multiplication (Fig. 3):

Similarly, for smaller matrix sizes, execution time remains low, with little variance across CPU counts. As matrix size increases, the system with 12 CPUs consistently shows better scalability, performing better than the configurations with fewer CPUs. Notably, the configuration with 2 CPUs experiences a significant increase in time for larger matrices, suggesting that multiplying large matrices benefits significantly from parallelization.

However, the performance of the system with 6 CPUs flattens out at matrix sizes around 2^8, implying potential bottlenecks or suboptimal thread utilization, possibly due to MPI's overhead when distributing work across this specific number of processors.

Comparative Analysis:

Scalability on N1-fat-infra: Both matrix inversion and multiplication show improved scalability as the number of CPUs increases on the N1-fat-infra cluster. However, matrix multiplication appears to scale more linearly compared to matrix inversion, where some configurations (notably 6 CPUs) encounter communication inefficiencies.

Efficiency of MPI:

While MPI is efficient in distributing tasks for both operations, matrix inversion has more intricate interdependencies between the matrix elements, which may cause additional communication overhead. This is reflected in the greater variance in execution time as matrix size increases.

Resource Utilization: Using a higher number of CPUs (8 or 12) reduces computation time significantly but also increases the cost on the cloud platform. For large-scale operations, matrix multiplication, with its more predictable scaling, is better suited for leveraging the full capacity of the N1-fat-infra cluster.

HIGHLIGHTS:

Testing these operations on the N1-fat-infra cluster with MPI shows that matrix multiplication benefits more consistently from parallel CPU allocation. Matrix inversion, while improved with additional CPUs, demonstrates potential MPI communication overhead at specific CPU counts.

UNIVERSITÀ
DI PAVIA

Performance on n1-Fat-Standard-8-core Cluster

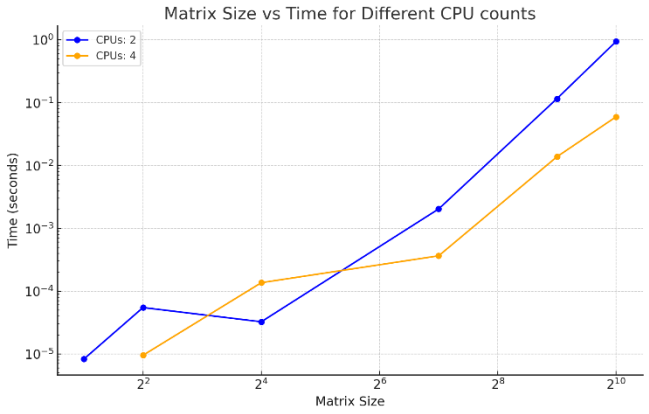

Fig. 5 Compare Inversion MPI Code for fat standard 8 core cluster.

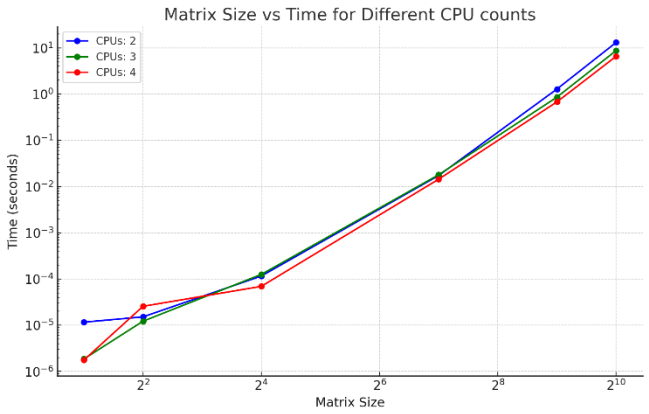

Fig. 6 **Compare** Multiplication **MPI Code for fat standard 8 core cluster.**

These results provide insight into how performance scales with different numbers of CPUs (2, 3, 4) for increasing matrix sizes.

Matrix Inversion (Fig. 5**)**

Scaling with Matrix Size:

At smaller matrix sizes (below 2^6), the time differences between using 2 and 4 CPUs are minimal, reflecting the low computational overhead of smaller matrices.

For larger matrices (e.g., 2^8 and beyond), we observe significant differences between the CPU configurations. As matrix size increases, the execution time rises steeply, especially for the 2 CPU configuration, indicating that inversion is highly compute-intensive.

The 4 CPU configuration shows much better performance than the 2 CPU setup for large matrices, reflecting the effectiveness of parallelization.

Efficiency of Parallelization:

The **4 CPU configuration** achieves better speed-up as the matrix size increases, although the improvement isn't perfectly linear, likely due to the communication overhead in distributing the tasks across CPUs.

For large matrices, the inversion process becomes bottlenecked due to the intrinsic complexity of the operation, and further CPUs hardly achieve increased performance.

Matrix Multiplication (Fig. 6**)**

Scaling with Matrix Size:

For smaller matrices, the time differences between 2, 3, and 4 CPUs are minimal, but as the matrix size grows (from 262^626 onward), the differences in performance between CPU counts become more noticeable.

The 4 CPU configuration consistently shows better performance, although the performance gain is small, suggesting matrix multiplication is well-parallelized across even a smaller number of CPUs.

Efficiency of Parallelization:

For matrix multiplication, the scaling with CPU count is more consistent across all matrix sizes compared to inversion. The performance gains are more predictable, with the 4 CPUs configuration performing slightly better than the 2 CPU configuration for large matrices.

Communication overhead: appears less pronounced in matrix multiplication than in inversion, as we see smoother, linear scaling as the matrix size increases.

Matrix Inversion vs. Multiplication:

Matrix inversion is more sensitive to CPU scaling, and its performance degrades more rapidly with fewer CPUs as the matrix size increases. This suggests that inversion is more complex and benefits more from parallelism, but only up to a point, where communication overhead becomes a limiting factor.

Matrix multiplication, on the other hand, scales more smoothly, indicating better parallel efficiency. It benefits from additional CPUs but shows less performance sensitivity than inversion, which indicates that matrix multiplication distributes computational work more evenly among the processors.

Expected Results with N1-fat-standard-8-core:

Both operations show performance improvements when using more CPUs, but matrix inversion has **higher communication overhead**, making it less efficient at scaling beyond a certain point, whereas matrix

multiplication consistently benefits from increased CPU power with predictable speedup.

Highlights

when running compute-heavy tasks like matrix inversion or multiplication on the N1-fat-standard-8-core cluster with MPI, matrix multiplication benefits more from parallel CPU execution with smoother scaling. Matrix inversion, while also benefiting from more CPUs, faces diminishing returns due to the complexity of the inversion algorithm and the communication overhead inherent in MPI.

Performance on n1-light-infraRegin-6core Cluster

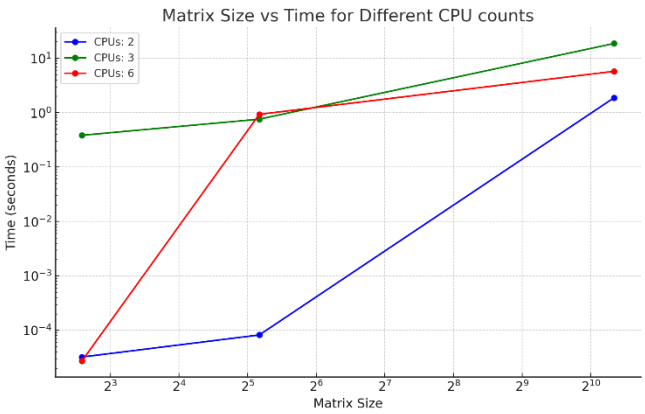

Fig. 7: **Compare Inversion Mpi Code for light infra region cluster.**

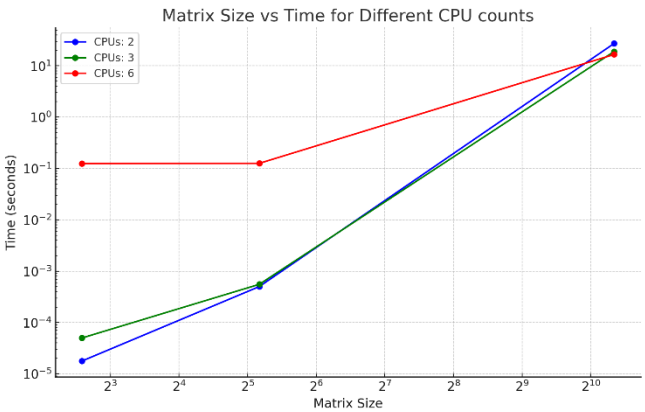

Fig. 8: Compare multiplication Mpi Code for light infra region cluster.

The following analysis is based on the results for matrix inversion and multiplication tasks executed using the "n1-light-infraRegion-6core" setup on Google Cloud. These results are based on configurations with 2, 3, and 6 CPUs for both operations, across increasing matrix sizes.

Matrix Inversion (Fig. 7)

Scaling with Matrix Size:

For smaller matrices (e.g., 2^3 to 2^5), the execution times are minimal across all CPU configurations, showing slight difference between 2, 3, and 6 CPUs.

Performance degradation is seriously noticed beyond a matrix size of 2^6, as we see that execution times for the 6 CPU configuration are greater compared to the other configurations tested - 2 CPUs and 3 CPUs. This is a counterintuitive result and could be due to some kind of communication overhead or inefficiency of the matrix inversion algorithm in dividing up the work across 6 CPUs, especially on the lightweight infrastructure n1-light.

Efficiency of Parallelization:

For larger matrices, the expected behaviour would be a reduction in time with increased CPU count, but the 6 CPU configuration exhibits an inefficiency, likely due to high inter-CPU communication costs or memory bandwidth limitations, which MPI may exacerbate in this scenario.

The 2 and 3 CPU configurations show more consistent scaling for matrix inversion, especially with larger matrix sizes, suggesting that in this case, fewer CPUs are handling the computational workload more efficiently without the overhead of communication across many CPUs.

Matrix Multiplication (Fig. 8)

Scaling with Matrix Size:

For small matrix sizes, the differences in performance are again minimal, as expected. However, as the matrix size increases (around 2^6 and beyond), the 6 CPU configuration shows worse performance than the configurations with 2 and 3 CPUs.

As with inversion, the 6 CPU configuration exhibits unexpected inefficiencies, particularly for larger matrices, where execution time increases more significantly than for the lower CPU counts.

Efficiency of Parallelization:

Like matrix inversion, matrix multiplication shows signs of inefficiency when using 6 CPUs, due to the overhead associated with parallelizing over more cores without the necessary infrastructure support (e.g., memory bandwidth).

The 2 and 3 CPU configurations show much more predictable scaling, with performance increasing steadily as the matrix size increases, and parallelization still providing benefits without the diminishing returns seen with 6 CPUs.

Comparative Analysis of Inversion and Multiplication:

Matrix Inversion vs. Multiplication:

In both tasks, the 6 CPU configuration demonstrates inefficiencies, due to MPI communication overhead and

UNIVERSITÀ
DI PAVIA

resource bottlenecks in the n1-light infrastructure. This is more evident in larger matrix sizes where inter-core communication becomes a significant factor.

Matrix inversion is more sensitive to CPU scaling, showing more pronounced performance issues with 6 CPUs, while matrix multiplication shows similar but less severe inefficiencies. N1-light-infraRegion-6core is not effective at distributing the load once more than 3 CPUs are utilized. Particularly this is so in the most computation-heavy tasks: matrix inversion and multiplication. That would seem to indicate a hypothesis that the lighter infrastructure types, such as "light" configurations, cannot be optimal for computation tasks that require heavy inter-CPU communication. In those cases, added overhead cancels out most performance gain through parallelization. For both operations, configurations with 2 or 3 CPUs perform more predictably and efficiently, likely because they avoid the high overhead of communication between a larger number of processors.

HIGHLIGHTS:

Both matrix inversion and multiplication tasks on the n1-light-infraRegin-6core infrastructure show diminishing returns with higher CPU counts, especially with 6 CPUs, where performance actually decreases for larger matrices. This indicates that the lightweight infrastructure and higher communication costs involved with MPI are bottlenecks that limit performance gains from parallelization. Using fewer CPUs (e.g., 2 or 3) provides more consistent and efficient scaling for these tasks, making it a better option for matrix operations on this infrastructure type.

Performance on n1-light-intraRegion-6core Cluster

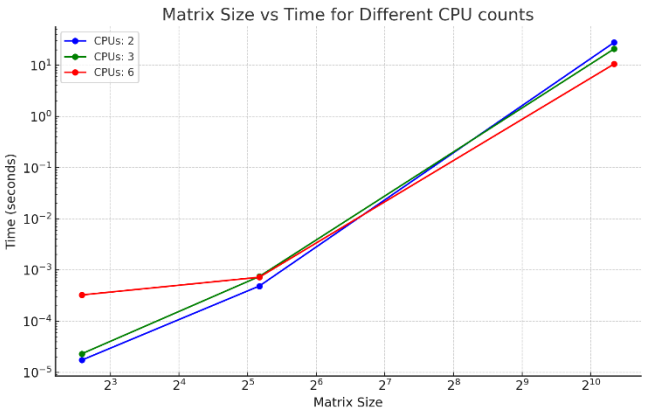

Fig. 9: Compare Multiplication MPI Code for n1-light-intraRegion Cluster.

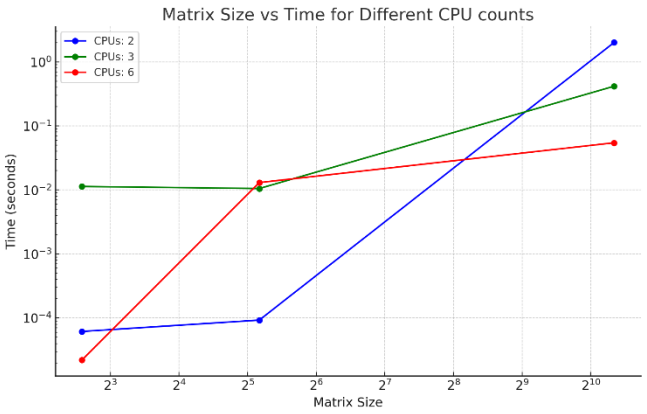

Fig. 10: Compare Inversion MPI Code for n1-light-intraRegion Cluster.

Analysis of Matrix Inversion and Multiplication on N1-light-intraRegion-6core Cluster. The graphs display how the execution time scales with increasing matrix sizes (from 2^3 to 2^10) across different CPU configurations (2, 3, and 6 CPUs).

Matrix Inversion (Fig. 10)

Scaling with Matrix Size:

For small matrices (below 2^5), the execution time remains constant across all CPU configurations. This is expected, as smaller matrices do not require significant computational power, and the overhead of parallelization in this case may be minimal.

For larger matrices (beyond 2^6), we observe that: - The 2 CPU configuration performs better for large matrix sizes compared to 3 and 6 CPUs.

6 CPUs show inefficient scaling. As the matrix size increases, performance worsens due to communication overhead or improper load distribution in the MPI framework, which is not ideal for this cluster configuration.

Efficiency of Parallelization:

While matrix inversion should benefit from parallelization, the 6 CPU configuration demonstrates significant overhead or inefficiency. This is due to the increased communication between the CPUs in the intra-regional setup, making matrix inversion performance degrade instead of improving with additional CPUs. The 3 CPU configuration provides better results than 6 CPUs but still does not scale well, especially for larger matrices, where 2 CPUs perform better.

Matrix Multiplication (Fig. 9)

Scaling with Matrix Size:

For smaller matrices (up to 2^5), all CPU configurations show similar performance. There's slight difference in the execution time between 2, 3, and 6 CPUs, indicating that the overhead for parallelization outweighs the benefits for small

matrix sizes. For larger matrices (e.g., 2^8 and above), the performance scaling becomes more distinct: The 6 CPU configuration starts to perform better compared to the smaller CPU configurations, unlike in matrix inversion. However, the increase in execution time for larger matrices is still more pronounced for 6 CPUs than 2 or 3 CPUs, showing that the scalability may be limited due to communication costs or load balancing issues.

Efficiency of Parallelization:

Matrix multiplication shows more predictable scaling, with 6 CPUs eventually performing better than the 2 CPU configuration, although the gain in performance is not linear due to the communication overhead in an intra-regional setup. The MPI-based parallelization shows better scalability in matrix multiplication than inversion, due to the more straightforward workload division in matrix multiplication.

Matrix Inversion vs. Multiplication:

Matrix inversion suffers more from the communication overhead between CPUs in the intra-regional cluster, especially with 6 CPUs. This is due to the more complex interdependence of matrix elements in the inversion process, leading to inefficient parallelization.

Matrix multiplication, on the other hand, scales better with 6 CPUs, particularly for larger matrices, as the operations are more independent and easier to parallelize.

Efficiency of Parallelization:

Both operations show performance degradation or inefficiency with 6 CPUs due to communication overhead in an intra-regional cluster. However, matrix multiplication manages to scale more effectively compared to inversion. This indicates that matrix inversion is less suitable for parallel execution on this specific cluster configuration due to the higher communication cost.

3 CPU configuration provides better performance for both operations than 6 CPUs, indicating that parallelization on fewer CPUs reduces the overhead while still benefiting from some level of parallelism.

Cluster Configuration and Performance:

The n1-light-intraRegion-6core cluster is not optimal for running matrix inversion at larger CPU counts due to communication and load balancing inefficiencies. Matrix multiplication benefits more from parallelization, but even then, the performance gains are limited due to the intra-regional communication overhead.

In this case, using fewer CPUs (such as 2 or 3) for matrix inversion might be a more effective approach to achieve better performance, while for matrix multiplication, 6 CPUs provide a slight advantage for larger matrices.

HIGHLIGHTS

- Matrix inversion on the n1-light-intraRegion-6core cluster demonstrates inefficient scaling beyond 3 CPUs, primarily due to communication overhead. It is recommended to use fewer CPUs for this operation on this cluster setup.

- Matrix multiplication shows better scalability with 6 CPUs but still encounters limitations due to the intra-regional communication overhead. While it performs better than inversion, the improvement is not linear.

- For both operations, the parallelization efficiency decreases as the number of CPUs increases, especially in the intra-regional setup. Therefore, optimizing CPU allocation and ensuring efficient workload distribution across CPUs are critical for improving performance on this cluster configuration.

Matrix Inversion Performance on the Standalone System (Intel i5-1135G7)

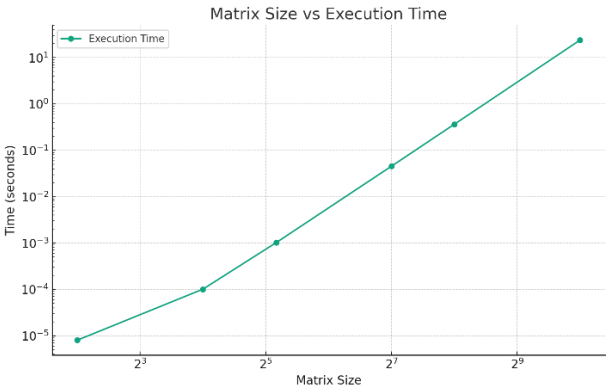

Fig. 11: Compare Inversion MPI Code Standalone System

Small Matrices (Size 2^2 to 2^4): Like the n1-fat-infra cluster, the execution time for small matrices remains low, and parallelization does not provide much of a speedup. The overhead of using multiple threads outweighs the benefits for small-scale computations on your standalone system.

Medium Matrices (Size 2^5 to 2^6): As the matrix size grows, the Intel i5-1135G7 demonstrates clear performance improvements with all 4 cores engaged. Although it performs well, the performance gains are limited compared to a setup with 6 or more cores, as found in the n1-fat-infra cluster.

Large Matrices (Size 2^7 and above): For large matrices, the standalone system with 4 cores sees a marked increase in inversion time. The performance benefit plateaus due to limited parallelism, unlike the n1-fat-infra cluster where 6 to 8 cores show better speedups.

UNIVERSITÀ
DI PAVIA

Highlights:

On 4-core standalone Intel i5-1135G7 system, matrix inversion benefit from parallelization, but the speedup is limited compared to the n1-fat-infra cluster, which has more cores available for parallel execution. For small matrices, both systems show minimal differences in performance, but for medium and large matrices, the n1-fat-infra cluster significantly outperforms my system, especially as it leverages 6 to 12 cores. The standalone system hits a performance ceiling due to its lower core count, leading to slower execution times for large matrices.

Conclusion

This project successfully demonstrated the optimization of matrix multiplication and inversion using parallel computing with MPI in C++. By transforming a serial implementation into a parallelized one, significant performance gains were achieved, particularly for medium to large matrices. The use of Open-MPI allowed the distribution of computational tasks across multiple processors, reducing execution times compared to the serial approach.

The implementation showed that, while parallelization offers substantial speedups, particularly with 6 to 8 CPU cores, diminishing returns can occur when adding more processors due to communication overhead and synchronization issues. Tests conducted on different clusters and standalone systems confirmed the scalability of the approach, making it suitable for large-scale matrix operations commonly required in scientific and engineering applications.

Overall, this project [5] highlights the power of parallel computing in improving matrix operation performance, particularly for larger datasets, and underscores the importance of balancing computational resources to maximize efficiency.

References:

[1] Explain Matrix Multiple

[2] Explain Matrix inversion. (2023). In Wikipedia.

[3] Implement Matrix Inversion MPI Source code: https://github.com/Shayankamalzadeh/ACA/blob/main/mpiInversion.cpp

[4] Implement Matrix Multiple MPI Source code: https://github.com/Shayankamalzadeh/ACA/blob/main/MpiMulti.cpp

[5] My GitHub Address for see all source code and folders of my test https://github.com/Shayankamalzadeh/ACA