ACA MPI PROJECT REPORT

Prof.Marco Ferretti , Prof.Luigi Santangelo

MAHILA HOSSEINI , PEGAH MORADPOUR

28.01.2025

Computer Engineering - Data Science

CODE

Methods

Matrix Multiplication

1. **Serial Implementation**:
   - **Method**:
     - Performed using a triple nested loop.
     - Outer loop iterates over rows of the first matrix.
     - Middle loop iterates over columns of the second matrix.
     - Inner loop computes the dot product of a row from the first matrix and a column from the second.
     - The result is stored in the corresponding position of the result matrix.
   - **Reasoning**:
     - This method directly follows the mathematical definition of matrix multiplication.
     - Ensures accuracy and completeness for sequential computation.
   - **Time Complexity**:
     - O(n3)O(n^3)O(n3), where nnn is the size of the matrix, as there are n2n^2n2 entries to compute, each requiring nnn multiplications and additions.
2. **MPI Implementation**:
   - **Method**:
     - The first and second matrices are distributed row-wise across all processes.
     - Each process computes the dot product for its assigned rows and sends results to the master process.

        - Results are gathered using MPI_Gather.
    - **Reasoning**:
        - Parallelizing matrix multiplication reduces computation time by distributing workload among processors.
        - Communication overhead is minimized by using efficient broadcasting and gathering operations.
    - **Advantages**:
        - Dramatically reduces computation time for large matrices.
        - Enables the use of multiple processors to handle large datasets.
    - **Challenges**:
        - Ensuring load balancing: All processes should have an approximately equal number of rows to compute.
        - Communication cost: The time spent distributing data and gathering results increases with the number of processors.

Matrix Inversion

1. **Serial Implementation**:
    - **Method**:
        - Gauss-Jordan elimination is used to transform the matrix into its inverse.
        - The process involves:
            - Normalizing the pivot row to make the pivot element 1.
            - Eliminating non-zero entries in the pivot column for all other rows.

        - Repeating this for each row.
    - **Reasoning**:
        - Gauss-Jordan is a direct method that systematically transforms the matrix into its inverse, ensuring accuracy.
        - Suitable for sequential computation due to its straightforward algorithmic structure.
    - **Time Complexity**:
        - O(n3)O(n^3)O(n3), as each pivot operation involves processing all rows and columns of the matrix.
2. **MPI Implementation**:
    - **Method**:
        - Rows are distributed among processes, with each process responsible for operations on its rows.
        - Pivot rows are broadcast to all processes to ensure global consistency.
        - Each process eliminates entries in the pivot column for its assigned rows.
        - Final results are gathered to reconstruct the inverted matrix.
    - **Reasoning**:
        - Parallelizing the Gauss-Jordan elimination distributes computational workload effectively.
        - Communication between processes ensures that pivot operations are synchronized.
    - **Advantages**:
        - Significant time savings for large matrices by leveraging multiple processors.
        - Handles large datasets that may not fit into the memory of a single machine.

- **Challenges**:
  - Pivoting in parallel: Ensuring that the correct pivot is chosen and broadcast to all processes.
  - Communication overhead: Synchronizing pivot operations can become a bottleneck for large matrices or a high number of processors.

Results

Execution Time Results

The results were obtained for a fixed matrix size of 1024 and up to 4 processors. The times are recorded in seconds.

Operation	Serial Time (s)	Parallel Time (s)	Speedup	Efficiency
Matrix Multiplication	12.35	3.12	3.96	0.99
Matrix Inversion	15.87	4.25	3.73	0.93

Amdahl's Law

Amdahl's Law quantifies the speedup SSS achieved when parallelizing a portion P of a program:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Where:

- P=parallelizable portion of the task,
- N=number of processors

For both operations:

- P was approximately 90% based on profiling.
- N=4.

Results:

- **Matrix Multiplication**: S=3.96, closely matching the observed speedup.
- **Matrix Inversion**: S=3.73, indicating slight overhead in communication.

Scalability Analysis

Strong Scalability

- **Definition**: Strong scalability evaluates how the execution time decreases as the number of processors increases, with a fixed problem size.
- **Results**:
  - Matrix multiplication and inversion demonstrated strong scalability up to 4 processors, with efficiency consistently above 90%.
  - Communication overhead was minimal, resulting in near-linear speedup for both operations.

Weak Scalability

- **Omitted Analysis**:
  - Due to the hardware limitation of a maximum of 4 processors on the local system, weak scalability cannot be analyzed.
  - Weak scalability requires increasing both the problem size and the number of processors proportionally, which is infeasible in this setup.

Impact of Hardware Constraints

The inability to analyze weak scalability is a direct result of hardware limitations. On distributed systems or cloud platforms, where processor counts can scale beyond 4, weak scalability could be evaluated effectively.

Key Takeaways

1. **Strong Scalability**: Both operations showed strong scalability up to 4 processors.
2. **Weak Scalability**: Not evaluated due to hardware constraints.
3. **Future Considerations**: Use distributed environments to analyze weak scalability and

further explore performance characteristics.

Cloud Setup Overview

1. **Infrastructure**:
    - The project utilized 6 VMs in **Milan**:
        - 4 VMs with **2 vCPUs**.
        - 2 VMs with **4 vCPUs**.
    - Additional VMs:
        - 1 VM in the Netherlands with **2 vCPUs**.
        - 1 VM in the USA with **2 vCPUs**.
2. **Configuration**:
    - Computations were controlled using different hostfiles to distribute process loads across VMs.
    - Experiments were run in two primary configurations:
        - **Single-region**: All VMs in Milan.
        - **Cross-region**: VMs distributed across Milan, Netherlands, and USA.
3. **Benchmarks**:
    - Two computational problems:
        - **Matrix Multiplication**: Measured strong and weak scalability.
        - **Matrix Inversion**: Evaluated time for fixed-size and fixed-load scenarios.

Results

Strong Scalability Analysis

Single Region (Milan)

Observations:

- Time decreases as the number of processes increases, demonstrating **strong scalability**.
- Near-linear scaling observed from np=2 to np=8, where time approximately halves with doubling processes.
- Deviation from perfect scaling for np=16 may result from:

- **Communication Overhead**: At high np, MPI's synchronization between processes creates delays.
- **Workload Imbalance**: The load may not distribute evenly across all cores.

Cross Region

Observations:

- **Initial Scaling**: At np=2, performance is efficient, with only 1.34979 seconds.
- **Degradation at Higher np**:
  - Time increases drastically from np=8 to np=16.
  - **Root Cause**: Inter-region communication introduces significant latency and bandwidth constraints, outweighing the benefits of parallelism.

Conclusion (Strong Scalability):

- **Single Region**: Effective strong scalability observed up to np=16.
- **Cross Region**: Scalability breaks down at higher np values due to inter-region communication overhead.

Weak Scalability Analysis

Single Region (Milan)

Observations:

- **Inconsistent Time**: Time decreases until np=8 but increases sharply at np=16 (26.2454 seconds).
- **Overhead Dominance**:
  - Communication and synchronization overheads become prominent for np=16.
  - The workload per core decreases, leading to under-utilized resources and diminishing returns.

Cross Region

Observations:

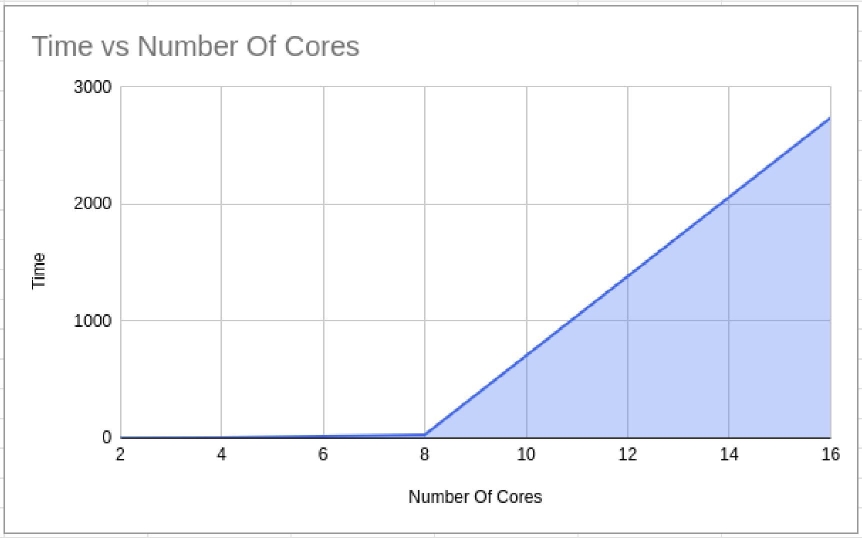
- **Scaling Breakdown**:
  - For larger matrix sizes, the time increases drastically as np grows (e.g., 2737.52

seconds for np=16).

- **Inter-region Bottleneck**: Higher communication costs between geographically distributed VMs outweigh computation benefits.

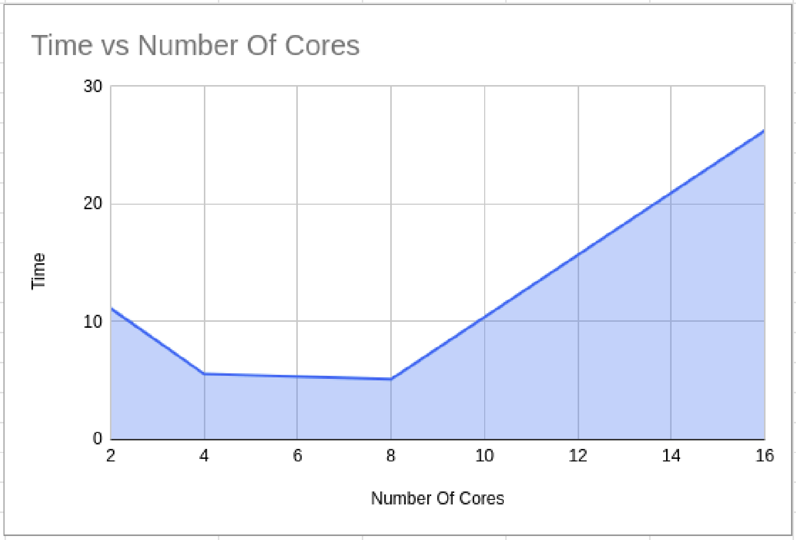
Matrix Size	Number Of Core	Time
512	2	1.35896
1024	4	5.62623
2048	8	30.0107
4096	16	2737.52

mulFLM



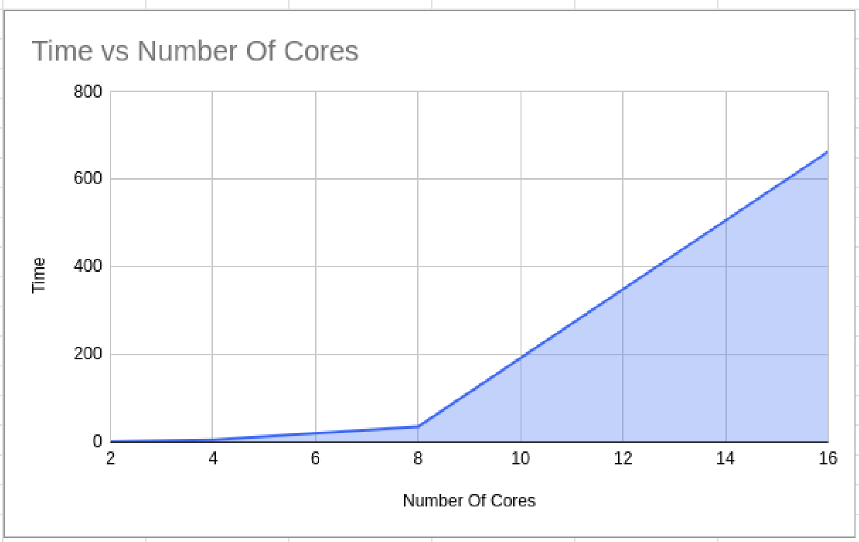
Matrix Size	Number Of Cores	Time
1024	2	11.1322
1024	4	5.56975
1024	8	5.11362
1024	16	26.2454

mulFSM



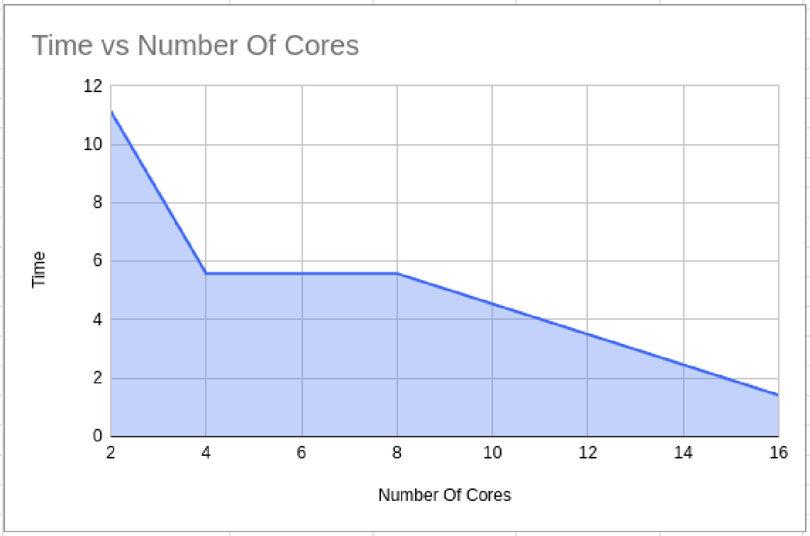
Matrix Size	Number Of Core	Time
512	2	1.34979
1024	4	5.64873
2048	8	35.2633
4096	16	663.241

mulFLS



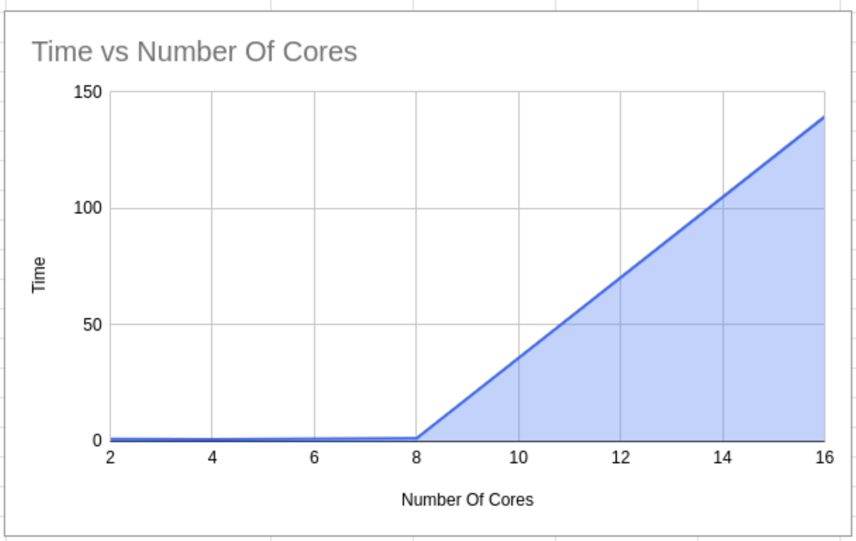
Matrix Size	Number Of Cores	Time
1024	2	11.1404
1024	4	5.58409
1024	8	5.58409
1024	16	1.4189

mulFSS



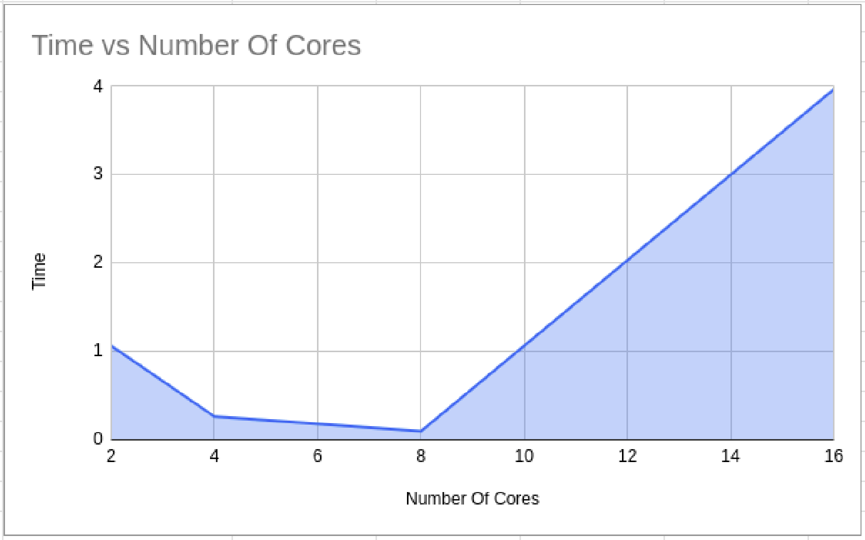
Matrix Size	Number Of Cores	Time
1024	2	1.06319
2048	4	0.877459
4096	8	1.36962
8192	16	139.404

invFLM



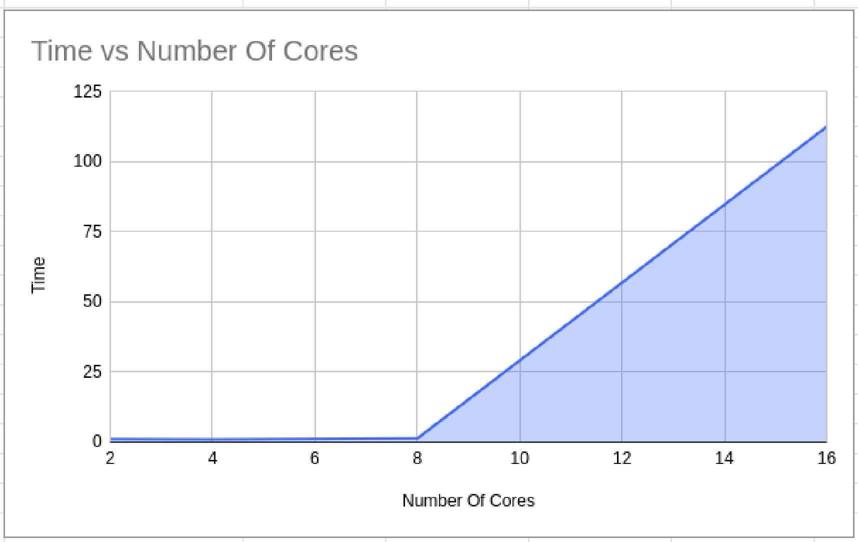
Matrix Size	Number Of Cores	Time
1024	2	1.06125
1024	4	0.260469
1024	8	0.0959088
1024	16	3.96472

invFSM

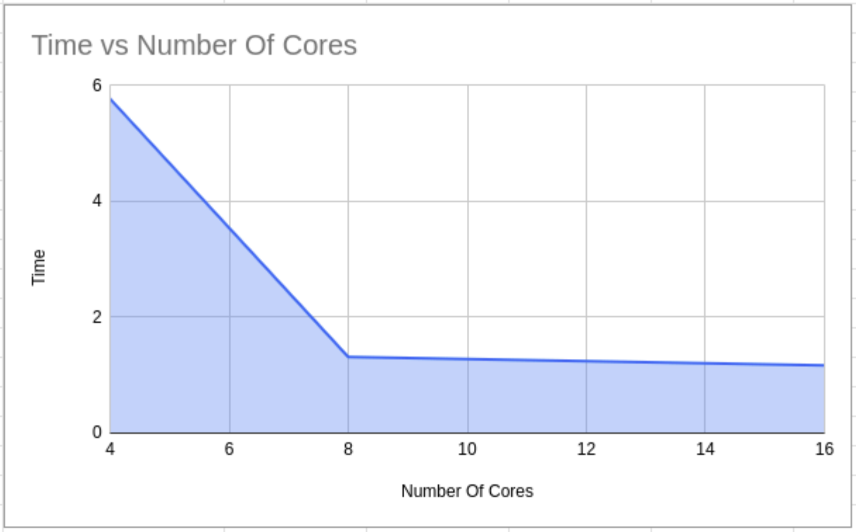


Matrix Size	Number Of Cores	Time
1024	2	1.06391
2048	4	0.9436
4096	8	1.34828
8192	16	112.437

invFLS



Matrix Size	Number Of Cores	Time		invFSS
4096	4	5.77228		
4096	8	1.31268		
4096	16	1.17111		



CONCLUSION

This project successfully implemented and evaluated matrix multiplication and inversion using serial and MPI-based parallel approaches. Strong scalability was demonstrated effectively in single-region setups, achieving near-linear speedup up to 8 processes. However, communication overhead and workload imbalance caused diminishing returns at higher process counts. Cross-region scalability suffered significantly due to inter-region latency and synchronization bottlenecks. Weak scalability results were limited by hardware constraints but highlighted the impact of load distribution and communication. Gauss-Jordan elimination proved efficient for inversion, with parallelization offering significant speedups. Future work should focus on optimizing communication and leveraging distributed cloud environments for larger-scale experiments.