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), where n is the size of the matrix, as there are n^2 entries to compute, each requiring n 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(n^3), 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

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

Strong scalability refers to the ability of a parallel system to reduce execution time while keeping the problem size **fixed** as the number of processing units (e.g., MPI processes) increases. Ideally, if a problem takes T1T_1T1 time on a single processor, then on P processors, the time should ideally be TP=T1/P.

The speedup in this case is defined as:

SP=TP/T1

In practice, speedup is sublinear due to communication costs and Amdahl's Law.

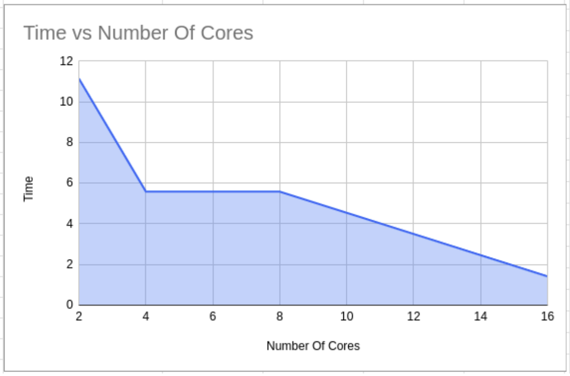
Single Region (Milan)

Observations:

- Time decreases as the number of processes increases, demonstrating **strong scalability**.
- Near-linear scaling is 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.

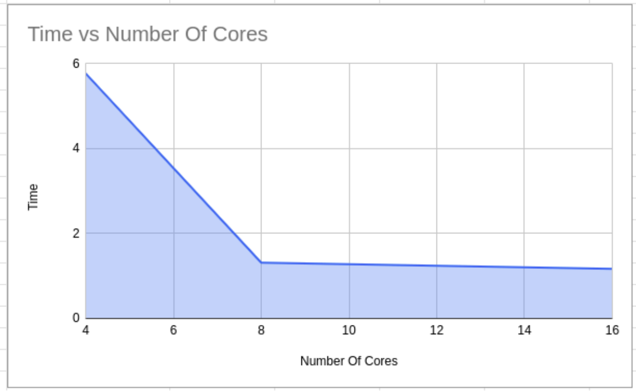
Matrix Size	Number Of Cores	Time
1024	2	11.1404
1024	4	5.58409
1024	8	5.58409
1024	16	1.4189

mulFSS



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

invFSS



Cross Region

Observations:

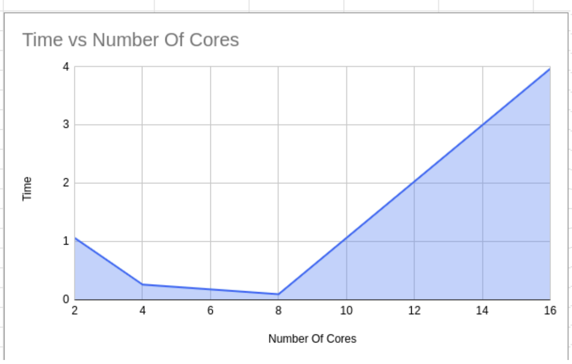
- **Initial Scaling**: At np=2, performance is efficient, with only 1.34979 seconds.
- **Degradation at Higher np**:
  - Time increases drastically from np=8 to np=16.
  - **Root Cause**: Inter-region communication introduces significant latency and bandwidth constraints, outweighing the benefits of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

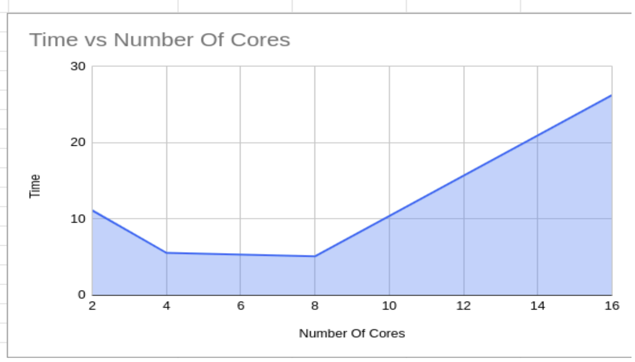
Matrix Size	Number Of Cores	Time
1024	2	1.06125
1024	4	0.260469
1024	8	0.0959088
1024	16	3.96472

invFSM



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

mulFSM



Using the Serialized execution time as baseline time we can calculate the speedup and Amdahl's law for each scenario.

The results are given in the table below each case indicating a significant speed up and it is evident that as the number of processors increases the speedup increases as well. This shows that we have strong scalability.

However when performed cross regional, after np = 8 we see a drop in speed up due to the communication overhead showing that the system is **communication-bound**.

inversion Strong

Matrix Size	Number Of Cores	Time (serial)	Time (parallel)	scenario	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
4096	4	918.142	5.77228	inversionFixedSizeSingleRegion	159.0605445	3.076923077	51.69467697
4096	8	918.142	1.31268	inversionFixedSizeSingleRegion	699.4408386	4.705882353	148.6311782
4096	16	918.142	1.17111	inversionFixedSizeSingleRegion	783.9929639	6.4	122.4989006
1024	2	13.4858	1.06125	inversionFixedSizeCrossRegion	12.70746761	1.818181818	6.989107185
1024	4	13.4858	0.260469	inversionFixedSizeCrossRegion	51.77506728	3.076923077	16.82689687
1024	8	13.4858	0.0959088	inversionFixedSizeCrossRegion	140.6106635	4.705882353	29.87976599
1024	16	13.4858	3.96472	inversionFixedSizeCrossRegion	3.401450796	6.4	0.5314766869

multiply Strong

Matrix Size	Number Of Cores	Time (serial)	Time (parallel)	scenario	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
1024	2	8.35007	11.1404	multiplicationFixedSizeSingleRegion	0.7495305375	1.818181818	0.4122417956
1024	4	8.35007	5.58409	multiplicationFixedSizeSingleRegion	1.495332274	3.076923077	0.4859829892
1024	8	8.35007	5.58409	multiplicationFixedSizeSingleRegion	1.495332274	4.705882353	0.3177581083
1024	16	8.35007	1.4189	multiplicationFixedSizeSingleRegion	5.884889703	6.4	0.9195140161
1024	2	8.35007	11.1322	multiplicationFixedSizeCrossRegion	0.7500826431	1.818181818	0.4125454537
1024	4	8.35007	5.56975	multiplicationFixedSizeCrossRegion	1.49918219	3.076923077	0.4872342116
1024	8	8.35007	5.11362	multiplicationFixedSizeCrossRegion	1.632907803	4.705882353	0.3469929082
1024	16	8.35007	26.2454	multiplicationFixedSizeCrossRegion	0.3181536574	6.4	0.04971150897

Theoretical Speed up is calculated by = 1 / ((1 - 0.9) + (0.9 / NumberOfCOres)). Assuming that 0.9 of the program can be parallelized.

And Amdhal's speedup is actual speedup / theoretical speedup

Weak Scalability Analysis

Weak scalability measures how well a parallel system maintains **constant execution time** when the **problem size increases proportionally** to the number of processors. This means that if a matrix of size N×N is solved in time T1 using 1 processor, then when the matrix size is scaled to kN×kN and the number of processors is increased to P=kP, the execution time should ideally remain the same (TP≈T1).

The weak scaling efficiency is defined as:

$$EP_{weak}=T1/TP$$

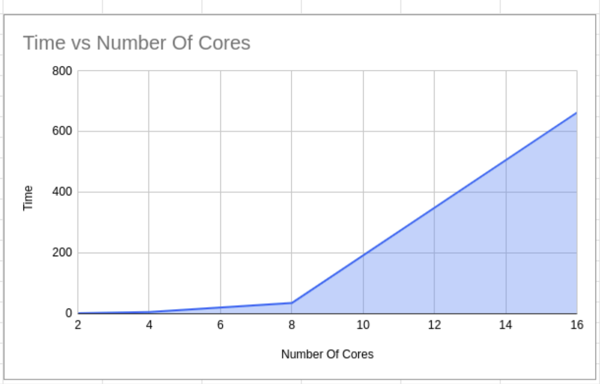
Single Region (Milan)

Observations:

- **Inconsistent Time**: For multiplicationTime decreases until np=8 but increases sharply at np=16 (26.2454 seconds).
- But for inversion time remains almost constant up to np=8 but after that dramatically increases.
- **Overhead Dominance**:
  - Communication and synchronization overheads become prominent for np=16.
  - The workload per core decreases, leading to under-utilized resources and diminishing returns.

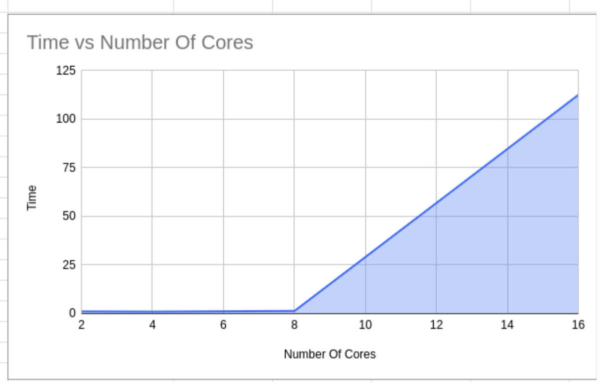
mulFLS

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



invFLS

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



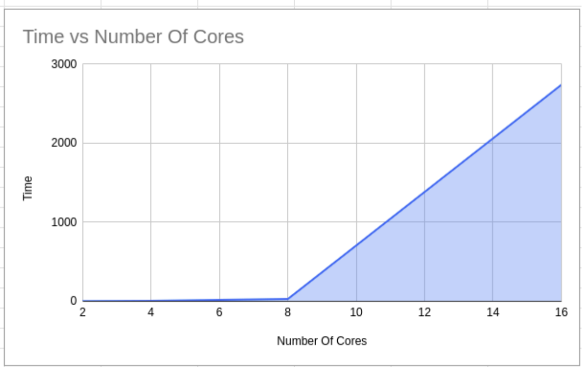
Cross Region

Observations:

- **Scaling Breakdown**:
    - For larger matrix sizes, the time increases as np grows (e.g., 2737.52 seconds for np=16).
    - **Inter-region Bottleneck**: Higher communication costs between geographically distributed VMs outweigh computation benefits.

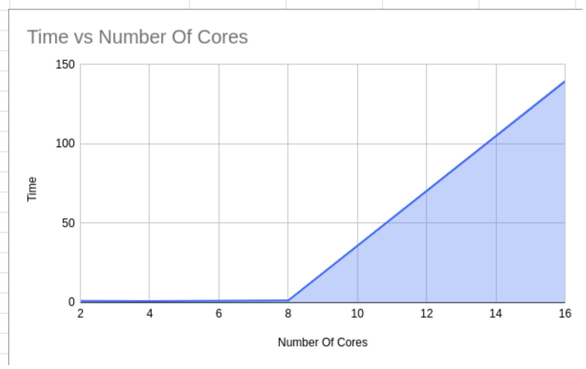
Matrix Size	Number Of Core	Time
512	2	1.35896
1024	4	5.62623
2048	8	30.0107
4096	16	2737.52

mulFLM



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

invFLM



Using the Serialized execution time as baseline time we can calculate the speedup and Amdahl's law for each scenario.

The results are given in the table below each case indicating that although we are seeing a speed up compared to the baseline but as the number of the processors increases the time tends to increase which is due to the fact that communication expense outweighs the performance improvements.

inversion Weak

Matrix Size	Number Of Cores	Time (serial)	Time (parallel)	scenario	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
1024	2	13.4858	1.06391	inversionFixedLoadSingleRegion	12.67569625	1.818181818	6.971632939
2048	4	110.048	0.9436	inversionFixedLoadSingleRegion	116.6256889	3.076923077	37.90334888
4096	8	918.142	1.34828	inversionFixedLoadSingleRegion	680.972795	4.705882353	144.7067189
8192	16	0	112.437	inversionFixedLoadSingleRegion	0	6.4	0
1024	2	13.4858	1.06319	inversionFixedLoadCrossRegion	12.68428033	1.818181818	6.976354179
2048	4	110.048	0.877459	inversionFixedLoadCrossRegion	125.4166861	3.076923077	40.76042299
4096	8	918.142	1.36962	inversionFixedLoadCrossRegion	670.3625823	4.705882353	142.4520487
8192	16	0	139.404	inversionFixedLoadCrossRegion	0	6.4	0

multiply Weak

Matrix Size	Number Of Cores	Time (serial)	Time (parallel)	scenario	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
512	2	1.00479	1.34979	multiplicationFixedLoadSingleRegion	0.7444046852	1.818181818	0.4094225768
1024	4	8.35007	5.64873	multiplicationFixedLoadSingleRegion	1.478220768	3.076923077	0.4804217497
2048	8	81.8249	35.2633	multiplicationFixedLoadSingleRegion	2.320398261	4.705882353	0.4930846305
4096	16	735.794	663.241	multiplicationFixedLoadSingleRegion	1.109391609	6.4	0.1733424389
512	2	1.00479	1.35896	multiplicationFixedLoadCrossRegion	0.7393815859	1.818181818	0.4066598723
1024	4	8.35007	5.62623	multiplicationFixedLoadCrossRegion	1.484132359	3.076923077	0.4823430165
2048	8	81.8249	30.0107	multiplicationFixedLoadCrossRegion	2.726524206	4.705882353	0.5793863939
4096	16	735.794	2737.52	multiplicationFixedLoadCrossRegion	0.2687812326	6.4	0.0419970676

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.