ACA MPI PROJECT REPORT

Prof.Marco Ferretti , Prof.Luigi Santangelo

MAHILA HOSSEINI , PEGAH MORADPOUR

28.02.2025

Computer Engineering - Data Science

Table of contents

Abstract

Sparse matrix multiplication is a fundamental operation in scientific computing, machine learning, and engineering applications. Due to its irregular memory access patterns and computational complexity, optimizing its execution is crucial for high-performance computing. In this study, we present an efficient parallel implementation of sparse matrix multiplication using OpenMPI in C++ and evaluate its performance on Google Cloud. We analyze strong and weak scalability to demonstrate the benefits and limitations of parallelization. Our experimental results confirm that parallel execution significantly reduces computation time, with optimal speedup achieved under balanced workload distribution.

Introduction

Sparse matrix multiplication is computationally expensive because:

1. Irregular Memory Access Patterns
    - Unlike dense matrices, where elements are stored in a contiguous block, sparse matrices use compressed storage formats (CSR, CSC, etc.).
    - This leads to random memory accesses, causing cache misses and slowing down performance.
2. High Number of Indirect Lookups
    - The algorithm requires multiple vector index lookups for row pointers, column indices, and values.
    - As seen in the profiling (gprof and perf reports), these lookups are a major bottleneck.
3. Computational Complexity
    - For a dense matrix, multiplication has $O(n^3)$ complexity.
    - Sparse matrix multiplication is more efficient, but in the worst case, each nonzero element must be multiplied with multiple values, making it still expensive.
4. Load Imbalance
    - Non-uniform sparsity distribution means some rows have more non-zero elements than others.
    - This creates workload imbalance, leading to inefficient CPU utilization.

How Parallelization Helps

Parallelizing sparse matrix multiplication can improve performance by:

1. Distributing Independent Computations Across Cores
    - Each row in the output matrix can be computed independently.

    - With multiple threads, each thread processes different rows → reducing total execution time.
2. Better CPU Utilization
    - Instead of a single core handling all operations, multiple cores can compute in parallel.
    - This takes advantage of modern multi-core processors.
3. Memory Bandwidth Utilization
    - By parallelizing, multiple threads can fetch different rows at once, reducing cache misses.

Challenges

- Synchronization overhead: If multiple threads modify shared data, locking can introduce overhead.
- Load balancing issues: Some rows may have significantly more nonzero values, leading to imbalanced work distribution.
- Theoretical Speedup (Amdahl's Law)
    - Since ~94% of the program's execution time is spent on matrix multiplication, parallelization can achieve significant speedup (~3.3x with 4 threads, ~5.5x with 8 threads, up to ~15x in ideal conditions).

Serial Implementation

Code

C++ implementation efficiently performs sparse matrix multiplication using the Compressed Sparse Row (CSR) format.
The generate.cpp code Randomly generates nonzero elements based on the sparsity factor (currently set to 20%). These matrices will be used as input to our csr_multiplication.cpp code to calculate the multiplication of 2 matrices.
The main computation is done in the multiplyCSR method.

Methodology

- The Compressed Sparse Row (CSR) format is used to store sparse matrices efficiently.
- The matrix multiplication follows the row-wise traversal approach:
    1. Iterate over each row in matrix A.
    2. For each nonzero element in A[i,j], retrieve the corresponding column index.
    3. Multiply this element with the matching row in matrix B.
    4. Accumulate the results into the corresponding row of the output matrix.

Reasoning

- Sparse matrix storage: Using CSR format reduces memory footprint by only storing nonzero values.
- Avoiding unnecessary computations: The algorithm skips zero multiplications, reducing the number of operations.
- Memory efficiency: The row-based access pattern improves cache locality compared to traditional dense multiplication.
- Index-based traversal: Using row_ptr and col_idx instead of an explicit 2D array allows for faster indirect lookups.

Time Complexity Analysis

Let:

- R_A = number of rows in matrix A
- C_B = number of columns in matrix B
- NNZ_A = number of nonzero elements in A
- NNZ_B = number of nonzero elements in B

The complexity is determined by:

- Outer loop (iterating over rows of A): O(R_A)
- Middle loop (iterating over nonzero elements in each row of A): O(NNZ_A)
- Inner loop (iterating over nonzero elements in corresponding row of B): O(NNZ_B/C_A)

Thus, the overall time complexity is:

$$O\left(NNZ_A \times \left(NNZ_B/C_A\right)\right)$$

which is significantly lower than dense matrix multiplication O(N^3), especially when $NNZ \ll N^2$.

Available Parallelism

Profiling

The profiling reports obtained from perf and gprof provide critical insights into the computational bottlenecks of the serial sparse matrix multiplication implementation.

Key Observations from perf Report

- multiplyCSR dominates execution time (~78% of total runtime).
- Significant time is spent accessing vectors:
  - std::vector<int>::operator[] (~9.58%)
  - std::vector<double>::operator[] (~8.23%)

Key Observations from gprof Report

- ~78% of execution time is spent in multiplyCSR.
- Vector index operations contribute to a significant portion of the execution time.

We can gather that the vast majority of execution time is spent on memory accesses rather than arithmetic operations.

Identification of Parallelizable Code Blocks

Based on the profiling analysis, the following insights help in deciding which parts of the algorithm can be parallelized.

Parallelizable Code Blocks

- Sparse Matrix Multiplication (multiplyCSR)
  - The outer loop iterating over rows of A is completely independent.
  - Each row of A can be processed independently since the result matrix rows do not overlap.
  - Ideal for domain decomposition: Each process handles a subset of rows.

Non-Parallelizable (or Less Efficient) Parts

- Sparse Matrix Reading
  - While this is technically parallelizable, it accounts for a small fraction of execution time so it is not a bottleneck.
  - Parallelizing this would have minimal impact on overall performance.
- Vector Indexing (std::vector::operator[])
  - While frequent, these are inherent to sparse matrix computations.
  - Optimization (e.g., using direct memory access) may help, but parallelizing alone won’t fix this bottleneck.

Communication & Synchronization Strategy

Since OpenMPI is used for parallelization, an efficient data distribution and communication strategy is essential.

Data Partitioning Strategy

- Row-wise partitioning of A (each MPI process handles a subset of rows).
- Matrix B is broadcast to all processes (since every row of A interacts with all columns of B).

Communication Overhead

- Each process computes partial results independently.
- Final result is gathered and put together.
- No need for synchronization within each process since rows are computed independently.

Theoretical Speedup Analysis (Amdahl's Law)

Amdahl's Law estimates maximum achievable speedup:

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

where:

- P = parallelizable portion of the program (~87% from profiling).
- N = number of processors.

$$S(4) = \frac{1}{(1-0.87) + \frac{0.87}{4}}$$

$$S(8) = \frac{1}{(1-0.87) + \frac{0.87}{8}}$$

$$S(16) = \frac{1}{(1-0.87) + \frac{0.87}{16}}$$

MPI Parallel Implementation

MPI implementation uses OpenMPI to distribute work across Google Cloud instances

- 5 VMs in Milan (NP = up to 10)
- 2 additional VMs in Netherlands (NP = up to 14)
- 1 additional VMs in the USA (NP = up to 16, extreme latency impact)

MPI-based sparse matrix multiplication is implemented using OpenMPI with a row-wise distribution strategy. The key components are

MPI Initialization and Setup

Each MPI process starts by initializing the environment

```
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

And then the master process (rank 0) reads the input sparse matrix and distributes it to worker processes. Both Matrices are broadcasted to all nodes as the CSR format makes it more memory efficient than normal matrix storage and it requires specific consideration when scattering

Then each node calculates the portion of the rows it has to calculate based on its rank and then extracts its local submatrix from A and performs the multiplication on those rows.

```
// Perform local multiplication (each process works on part of the result)
int rows_per_process = A.rows / size;
int remainder = A.rows % size;
int start_row = rank * rows_per_process + min(rank, remainder);
int end_row = start_row + rows_per_process + (rank < remainder);

CSRMatrix localA;
CSRMatrix localC;
localA.rows = end_row - start_row;
localA.cols = A.cols;
localA.row_pointers.push_back(0);
```

Each process then sends the local results to the master node

```
MPI_Send(&localC.row_pointers[0], localC.row_pointers.size(), MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&localC.col_indices[0], localC.col_indices.size(), MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&localC.values[0], localC.values.size(), MPI_INT, 0, 0, MPI_COMM_WORLD);
```

And on the master node the results are received and the final result is constructed in the CSR format

```
vector<int> values(values_size);

MPI_Recv(row_pointers.data(), row_pointers_size, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(col_indices.data(), col_indices_size, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(values.data(), values_size, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

The master node will update the row pointers

```
// Update rowPtrs for the final matrix (finalC)
int offset = finalC.row_pointers.back();
for (int j = 1; j < row_pointers.size(); j++) {
    finalC.row_pointers.push_back( row_pointers[j] + offset);
}
```

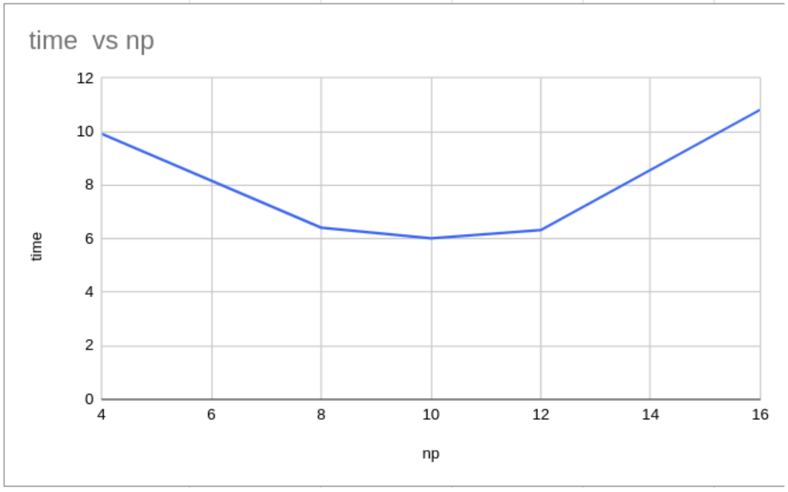
Performance and Scalability Analysis

In order to test the performance in different scenarios we have configured 8 VMs on google cloud with exactly the same configurations but in different regions.
Based on the theoretical calculation and the actual performance we have recorded the following results have been recorded.

Strong scalability

Keeping the problem size fixed and increasing the number of processors, we can see that the performance has started to get better but when we reached the number of nodes inside the Milan region and started to use other regions the performance drops due to the communication latency.

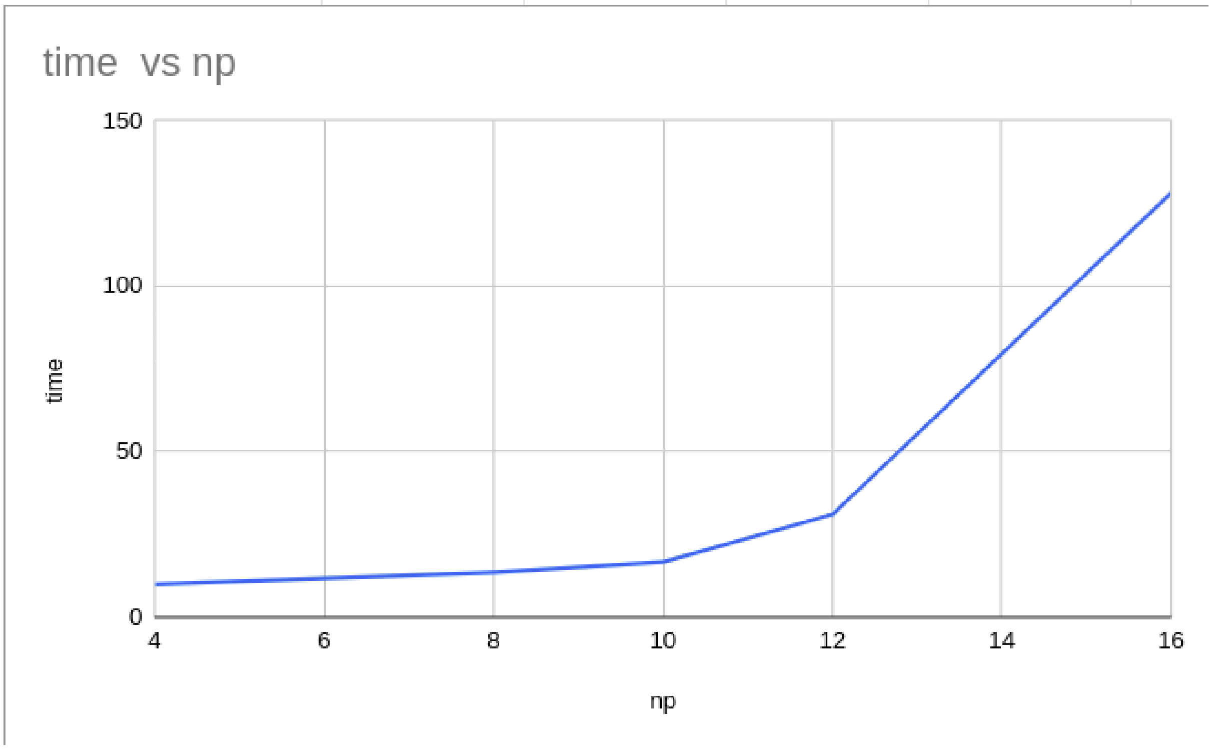
size	np	time
3000 * 4000 * 5000	4	9.926582353
3000 * 4000 * 5000	8	6.415225
3000 * 4000 * 5000	10	6.016582353
3000 * 4000 * 5000	12	6.326582353
3000 * 4000 * 5000	16	10.82022625



Weak scalability

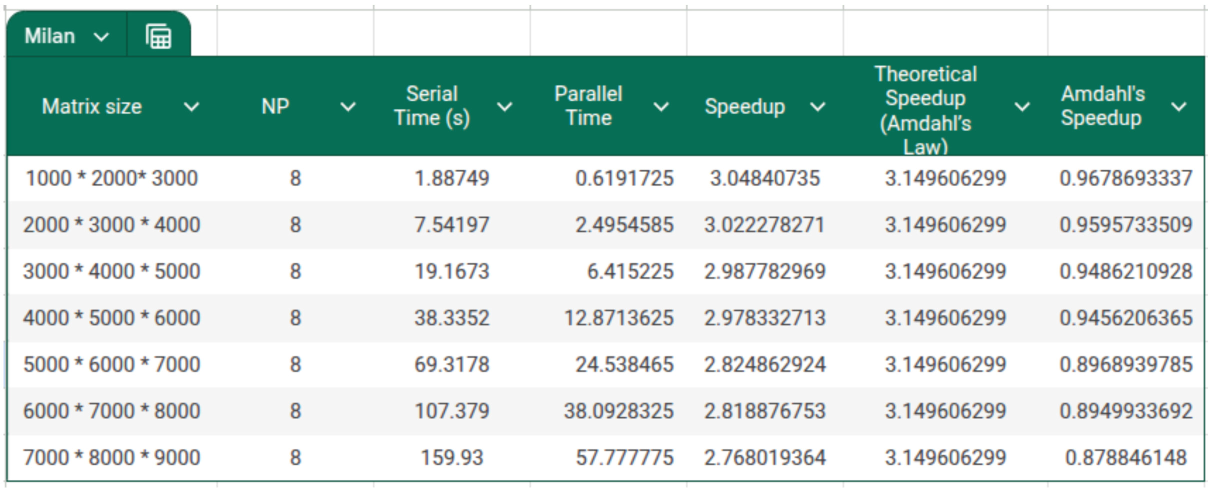
Keeping the load on each processor fixed (increasing the number of processors as well as increasing problem size), we can see that the performance falls due to the communication overhead the mpi introduces and when we reached the number of nodes inside the Milan region and started to use other regions the performance drops even more due to the additional communication latency.

size	np	time
3000 * 4000 * 5000	4	9.926582353
4000 * 6000 * 5000	8	13.548766
5000 * 6000 * 5000	10	16.642391
6000 * 6000 * 5000	12	31.09124
8000 * 6000 * 5000	16	128.146452



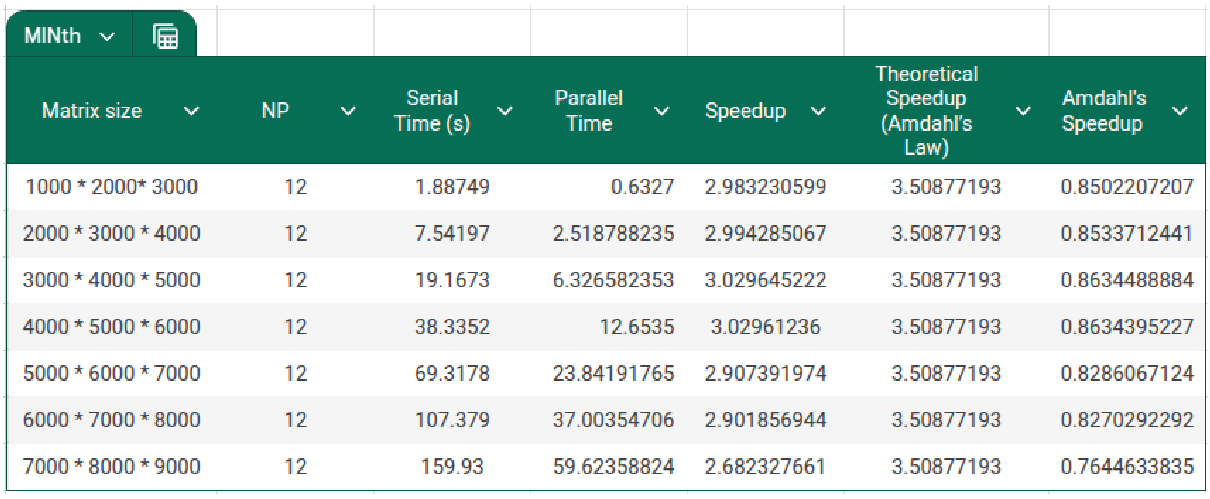
Overall analysis of Amdahl's Law and speedups

In the following tables we can see that the actual speed up is always lower than the theoretical one due the overhead and latency the communication introduces. But it is more evident as the problem size increases and the nodes are distributed intra regional.



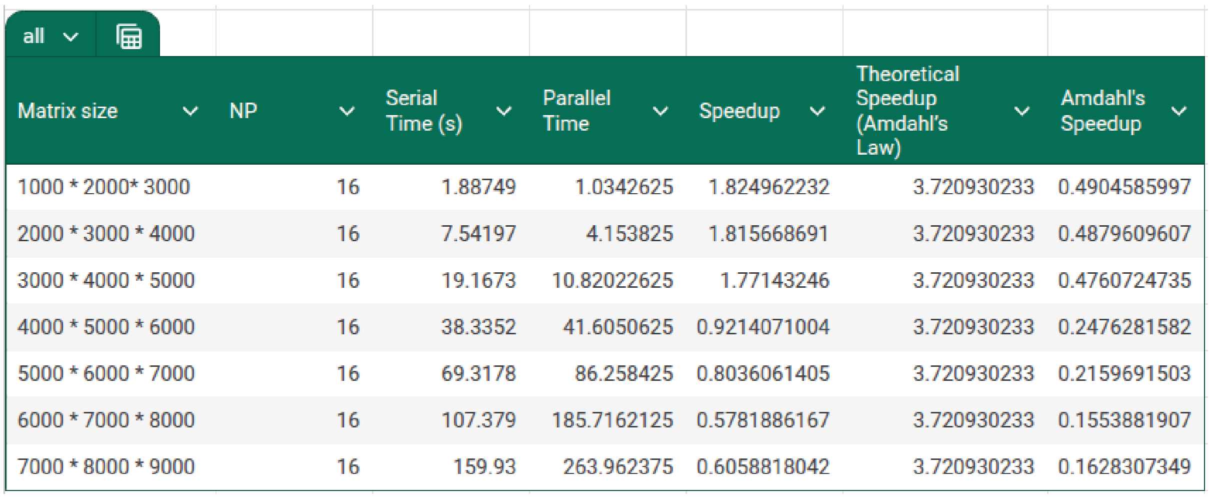
Matrix size	NP	Serial Time (s)	Parallel Time	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
1000 * 2000* 3000	8	1.88749	0.6191725	3.04840735	3.149606299	0.9678693337
2000 * 3000 * 4000	8	7.54197	2.4954585	3.022278271	3.149606299	0.9595733509
3000 * 4000 * 5000	8	19.1673	6.415225	2.987782969	3.149606299	0.9486210928
4000 * 5000 * 6000	8	38.3352	12.8713625	2.978332713	3.149606299	0.9456206365
5000 * 6000 * 7000	8	69.3178	24.538465	2.824862924	3.149606299	0.8968939785
6000 * 7000 * 8000	8	107.379	38.0928325	2.818876753	3.149606299	0.8949933692
7000 * 8000 * 9000	8	159.93	57.777775	2.768019364	3.149606299	0.878846148

While when using 8 nodes we stay in the Milan region we can see that the while we are having positive speedup and increase in performance but the speedup itself decreases as the matrix size increases which is because of the mpi communication overhead.



Matrix size	NP	Serial Time (s)	Parallel Time	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
1000 * 2000* 3000	12	1.88749	0.6327	2.983230599	3.50877193	0.8502207207
2000 * 3000 * 4000	12	7.54197	2.518788235	2.994285067	3.50877193	0.8533712441
3000 * 4000 * 5000	12	19.1673	6.326582353	3.029645222	3.50877193	0.8634488884
4000 * 5000 * 6000	12	38.3352	12.6535	3.02961236	3.50877193	0.8634395227
5000 * 6000 * 7000	12	69.3178	23.84191765	2.907391974	3.50877193	0.8286067124
6000 * 7000 * 8000	12	107.379	37.00354706	2.901856944	3.50877193	0.8270292292
7000 * 8000 * 9000	12	159.93	59.62358824	2.682327661	3.50877193	0.7644633835

But the decrease is more evident when using 12 nodes as in this scenario we are also using the Vm in the Netherlands and in addition to mpi communication overhead we have higher network latency. So however it is still beneficial to us but it is less efficient compared to inter-region distribution.



Matrix size	NP	Serial Time (s)	Parallel Time	Speedup	Theoretical Speedup (Amdahl's Law)	Amdahl's Speedup
1000 * 2000* 3000	16	1.88749	1.0342625	1.824962232	3.720930233	0.4904585997
2000 * 3000 * 4000	16	7.54197	4.153825	1.815668691	3.720930233	0.4879609607
3000 * 4000 * 5000	16	19.1673	10.82022625	1.77143246	3.720930233	0.4760724735
4000 * 5000 * 6000	16	38.3352	41.6050625	0.9214071004	3.720930233	0.2476281582
5000 * 6000 * 7000	16	69.3178	86.258425	0.8036061405	3.720930233	0.2159691503
6000 * 7000 * 8000	16	107.379	185.7162125	0.5781886167	3.720930233	0.1553881907
7000 * 8000 * 9000	16	159.93	263.962375	0.6058818042	3.720930233	0.1628307349

Increasing the np to 16 will make use of the Vm placed in the USA which introduces a huge network communication latency and as the matrix size increases it causes the performance drop all together. Meaning that not only we can see the reduction in speedup itself; but also after a certain size we do not benefit from any speedup.

Conclusion

Running the High performance computation using a parallel approach under the carefully considered environments and circumstances will benefit us a good deal in speed up and performance, however we should remember that the cost of communication may affect the performance and in some cases even outweigh the benefits.