CITS5507 Project 2: MPI Implementation and Performance Analysis Across Different Parallel Configurations

Dayu Liu (24188516)

Semester 2, 2024

Introduction

Matrix multiplication becomes increasingly expensive as the dimension size grows, particularly with sparse matrices, which contain mostly zero values and only a few non-zero elements. This results in unnecessary computations involving zeros, making the process inefficient in terms of both time and memory. To address these inefficiencies, we implemented a customized, simplified version of compressed matrix multiplication . In project 1, we have successfully explored how OpenMP enhances parallelization and improves the performance of compressed matrix multiplication by experimenting with different thread counts and parallelization strategies.

In this project, we would like to further integrate MPI on top of OpenMP, and conducted a series of experiments with different parallel approaches (Sequential, Pure OpenMP, Pure MPI, and MPI+OpenMP), along with various parameter combinations (processes per node and threads per process).

In the first part of the project, we successfully determined the maximum matrix size that a sequential task could handle under a 10-minute time constraint, considering different non-zero densities. However, due to resource limitations on the Pawsey supercomputer, we later shifted our objective to evaluate which parallel configurations could complete the task most efficiently in the second experiment.

> All code implementations are included in the submitted zip file. Please refer to the comments in the `project2.sh` for instructions on how to run the code.

Project Workflow

The project can be divided into two main parts:

1. MPI Code Implementation:
MPI enables us to break the task into smaller sub-tasks across multiple processes, each responsible for managing a certain number of threads. After generating and distributing the matrices to all allocated MPI processes, the computational workload for compressed matrix multiplication is evenly divided, with each MPI process responsible for a specific range of rows. Within each process, OpenMP is then used further to parallelize the operation across the designated threads. The overall execution time is measured once process 0 has successfully gathered the results from all other processes, allowing us to evaluate the performance at different levels of parallelization. Inter-process communications and necessary barriers are properly applied to ensure the correctness of the results.

2. Parallel Experiments:
In the second step, we conducted experiments using varying process counts, and thread counts. The performance was monitored using Sequential, MPI, OpenMP, and MPI+OpenMP parallelization on the Pawsey supercomputer. Performance was evaluated based on execution time across different configurations.

To conserve computing resources and address the usage shortage, our experiments focused on several signature configurations rather than running all possible permutations in bulk.

Code Implementation

Matrix Generation

The `generateMatrices()` function is responsible for creating matrices with a specified density of non-zero values. Each element in the matrix is assigned a value based on a random selection process that determines whether it will be a non-zero value, according to the specified density (percent).

This function generates two compressed matrices (values and indices) row by row. When a non-zero value is selected, a numerical value between 1 and 10 is assigned. These non-zero elements, along with their column indices, are stored in two corresponding compressed matrices.

Compressed Matrix Multiplication

The function `compressedMatrixMultiply()` leverages both MPI and OpenMP to parallelize the multiplication of compressed matrices. It divides the main loop into smaller sub-loops for each MPI process, assigning a specific range of rows between `start_row` and `end_row` to each process. These sub-loops are then parallelized using OpenMP's `parallel for` directive, which distributes the work among a given number of threads.

For each element in the compressed matrix, its value and corresponding index are extracted. These values are then used to compute the product, and the result is added directly to the appropriate position in the result matrix.

MPI and OpenMP Flags

To make sure we can apply different combinations (Sequential, Pure OpenMP, Pure MPI and MPI+OpenMP) in one unified code base, flags for MPI and OpenMP are used to check if either are enabled, this is done by adding `-fopenmp` and customized `-D_MPI` flags when compiling the program.

A hybrid combination would look like this: `mpicxx -fopenmp -D_MPI -o project2 project2.c`.

Our main program would then check MPI's `_MPI` flag or OpenMP's _OPENMP flag to decide whether it need to initialize or utilize either parallelization.

Message Passing and Barrier

Proper process communications and barriers are crucial to ensuring the correct behavior of the program. Process 0 is responsible for generating the compressed matrices, distributing necessary data and gathering the computed results. Once the matrix has been generated, the resulting matrices are flattened into 1-D arrays using the `flattenCompressedMatrix()` function, and the sizes of each row are recorded for later recovery. The data is then broadcasted to all other processes using `MPI_Bcast()`. Each process receives the flattened array and reconstructs the original matrix using the `reconstructCompressedMatrix()` function. This ensures that all processes are working with the same data for parallel computation.

At the end of `compressedMatrixMultiply()`, a MPI barrier is applied to ensure that all processes have completed their computations. Once this synchronization is achieved, each process uses `MPI_Send()` to send its computed result back to process 0. Process 0 then gathers these results using `MPI_Recv()`, consolidating the data to finalize the resulting matrix.

Experiments

Experiment Setup

The following parameters were explored during the experiments:

Experiment 1: Matrix Size Estimation based on Non-zero Density (Sequential Only)

- Maximum Time allowed: 10 minutes
- Percentage of non-zero elements: 1%, 2%, and 5%

In the first experiment, we would like to determine the impact of matrix size and non-zero element density (1%, 2%, and 5%) on execution time. Due to limited resources, the experiment was only conducted on the sequential setting, though the insights gained should be applicable to other parallel configurations as well.

Experiment 2: Performance Comparison Between Different Parallel Configurations

- Matrix dimension size: 100,000 x 100,000
- Parallel Combinations: Pure MPI, Pure OpenMP, MPI+OpenMP

In the second experiment, the matrix with 1% non-zero elements was used to explore the impact of different parallel combinations. The goal was to fully utilize all 128 physical cores per node for optimal performance. In the pure MPI experiment, 128 processes per node were created. In the pure OpenMP experiment, 128 threads per process were used. For the MPI+OpenMP experiment, we allocated 4 MPI processes per node, with 32 OpenMP threads per process.

Experiment 1

In the batch file (`project2.sh`), we started with a 10,000 x 10,000 matrix and doubled the size in each iteration to find a rough estimate of the largest matrix size that could be computed within 10 minutes. Once a rough estimate was obtained, we fine-tuned by incrementing the matrix size by 1000 in subsequent iterations to arrive at a more precise maximum size.

The executable is compiled once and reused across multiple iterations. An argument flag `--time=00:10:00` is applied to `srun` to ensure the program terminates if it exceeds the 10-minute time limit. Once the program has been terminated, we identified the last successful iteration and recorded its corresponding matrix size.

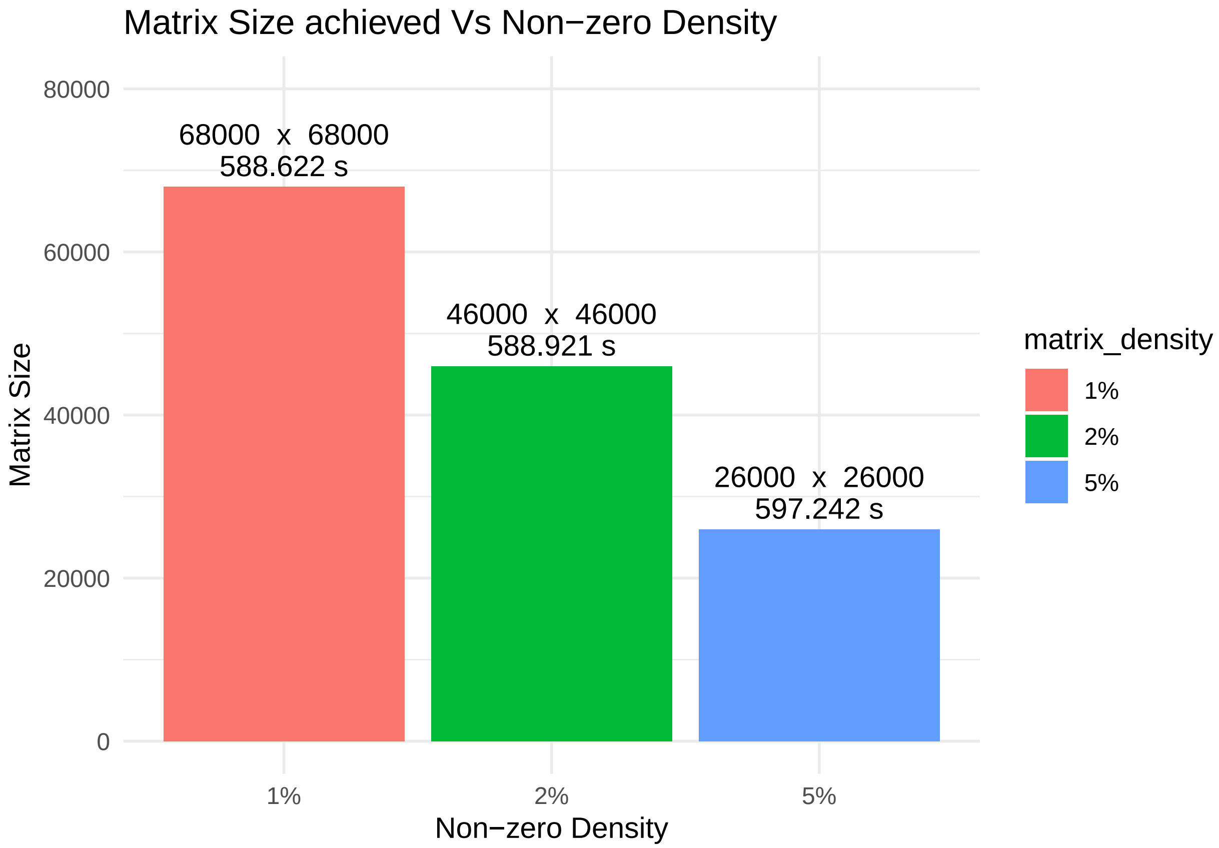
Experiment 1 Result

For the 1% non-zero matrix, the maximum matrix size computed within the 10-minute constraint was 68,000 x 68,000, taking 588.622 seconds. For the 2% non-zero matrix, the maximum size dropped significantly to 46,000 x 46,000, taking 588.921 seconds, which is roughly 1/2.18 of the 1% matrix size. For the 5% non-zero matrix, the maximum size was reduced further to 26,000 x 26,000 (1/6.84 compared to the 1% matrix), taking 597.242 seconds.

The significant decline in matrix size can be attributed to the increased computational workload as matrix density rises. At 1% density, the abundance of zero elements allows the algorithm to bypass many operations, focusing only on non-zero values. However, as density increases to 2% and 5%, the proportion of non-zero elements grows, leaving fewer opportunities to skip computations. This leads to a surge in the number of necessary multiplications and additions, which directly impacts performance, resulting in longer processing times for denser matrices.

Non-zero Density	Time Elapsed (s)	Matrix Size Achieved
1%	588.622	68,000 x 68,000
2%	588.921	46,000 x 46,000
5%	597.242	26,000 x 26,000

Non-zero Density	Time Elapsed (s)	Matrix Size Achieved



Experiment 2

Experiment 2 explored how different configurations of MPI processes and OpenMP threads affect the performance of matrix multiplication. The 1% non-zero density matrix was chosen for consistency and to conserve computational resources, as it had previously been shown to be the least computationally expensive compared to denser matrices.

In the main program (`project2.c`), command-line arguments divide the workload by specifying different numbers of processes per node and threads per process for parallelized compressed matrix multiplication. For example, the command `sbatch --nodes=1 project2.sh hybrid 100000 1 4 32` allocates 1 node, 4 processes per node and 32 threads per process, while applying both MPI and OpenMP to compute a matrix size of 100000 x 100000 with 1% non-zero elements.

Some combinations that are representative were selected for experimentation and comparison:

- Sequential: 1 Process/1 Thread
- Pure OpenMP: 32 OpenMP Threads
- Pure OpenMP: 128 OpenMP Threads
- Pure MPI: 128 MPI Processes
- Hybrid: 4 Processes/32 Threads

Experiment 2 result

In the pure OpenMP approach, we observed very similar performance, with a runtime of 75.5754 seconds using 32 threads and 71.84 seconds using 128 threads. This can be attributed to the relatively smaller computational workload, where the overhead of thread management and switching outweighs the benefit of utilizing more threads. When dealing with a smaller-size matrix like this, excessive threading can introduce inefficiencies, as the workload is too small to be effectively divided among many threads.

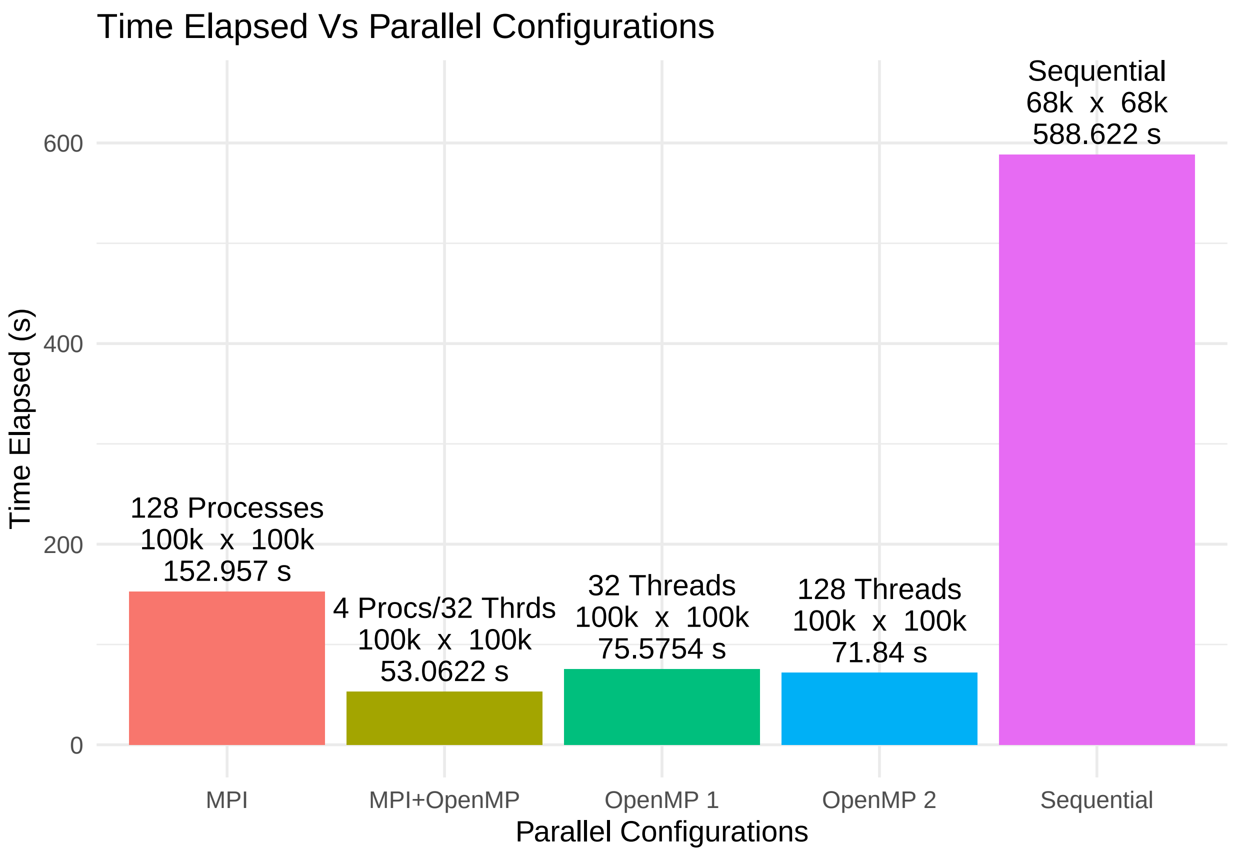
With the pure MPI approach using 128 processes, we witnessed the longest runtime of 152.957 seconds. A plausible explanation is that, while pure computation may benefit from such parallelism, the overhead introduced from communication and data distribution outweighed the benefits of dividing the workload among more processes. This resulted in a significantly higher runtime compared to a single-process OpenMP setup.

With the hybrid approach of 4 MPI processes and 32 OpenMP threads per process, we observed a runtime of 53.0622 seconds, which is remarkably close to our best observation in Project 1, where 60 threads yielded a runtime of 52.1606 seconds. This result indicates that this hybrid combination successfully balances parallelism at both the process and thread levels, minimizing overhead while efficiently dividing the workload.

Moreover, all of these parallel approaches significantly outperform the sequential approach. As a reference, the computation for a 68,000 x 68,000 matrix in the sequential approach alone took over 588.622 seconds.

Unfortunately, due to resource constraints, completing the sequential computation for a 100,000 x 100,000 matrix was not feasible, as the time required would far exceed available limits.

Approach	Parameters	Matrix Size	Time elapsed (s)
Pure OpenMP	32 OpenMP Threads	100,000 x 100,000	75.5754
Pure OpenMP	128 OpenMP Threads	100,000 x 100,000	71.84
Pure MPI	128 MPI Processes	100,000 x 100,000	152.957
Hybrid	4 Processes/32 Threads	100,000 x 100,000	53.0622
Sequential	1 Process/1 Thread	68,000 x 68,000	588.622


Time Elapsed Vs Parallel Configurations
Time Elapsed (s)
0
200
400
600
128 Processes
100k x 100k
152.957 s
4 Procs/32 Thrds
100k x 100k
53.0622 s
32 Threads
100k x 100k
75.5754 s
128 Threads
100k x 100k
71.84 s
Sequential
68k x 68k
588.622 s
MPI
MPI+OpenMP
OpenMP 1
OpenMP 2
Sequential
Parallel Configurations