# Matrix Multiplication Using Hybrid Parallelisation

## Student Information


- Imran Barton: 23409857
- Maxwell Slater: 23434844


## 1. Introduction

The goal of the project is to determine the largest array we can multiply within 10 minutes on the Pawsey supercomputer. In doing this, we look at different ways we can parallelise the ordinary sequential execution of this task. We compare the speed differences of shared and distributed memory systems in completing each required step of compressed matrix multiplication.

Our hardware environment was initially Setonix (4 nodes, 2x64 physical cores per node) but sudden and severe limitations caused us to change our environment to run on bare metal

We ran all tests on Imran's laptop, with 16gb of memory and 16 threads (Always limited to 10 for tests), Hyperthreading was disabled.

## 2. Program Implementation and Usage

### 2.1 Implementation Overview

Matrix row compression is used for sparse matrices where many cells may be equal to 0. We compress a matrix by first creating a complementary index matrix to store the original column of each non-zero value, and then we remove every zero from every row of the matrix until its size is equal to the index matrix.

For testing purposes we need a way to generate matrices. We initialise a random value with `srand()`, and then create matrices with densities varying from 1% to 5%, where the density is the likelihood of a non-zero value being present in any given cell.

## 2.2 Parallelisation Strategy

**Distributed memory (MPI) implementation**

- We distributed data evenly between the processes. The root process (rank 0) is the process that handles writing to logfiles, initialising data structures, and returning the final result.
- We relied on MPI_Scatterv() to share the matrices between the processes. Each process cuts their own “chunk” of the matrix to operate on based on their rank. The actual algorithm is the same as sequential, except each process returns its output to the root process with MPI_Recv() for compilation.
- Since the matrices are randomised, some processes may complete execution before others. We briefly looked at how we could perform some load-balancing for these “faster” processes, but given that the process count was so low it didn’t make sense to implement this feature.

**Shared memory (OpenMP) implementation**

- We employed OpenMP to parallelise the matrix multiplication algorithm, focusing on the outermost loop of the multiplication process. The parallelisation strategy includes:
    1. **Parallel For Loop**: The outermost loop, which iterates over the rows of matrix A, is parallelised using the `#pragma omp parallel for` directive. This allows multiple threads to process different rows of A simultaneously.
    2. **Static Scheduling**
        a. We used `schedule(static)` to distribute the iterations of the parallel loop evenly among the available threads.
        b. This is effective when the workload per row is relatively uniform, which is often the case for matrices with similar sparsity patterns across rows.
        c. But this will be tested when we vary the scheduling approach
    3. **Atomic Operations**:
        a. To prevent race conditions when updating the result matrix, we use `#pragma omp atomic` for the addition operation.
        b. This ensures that updates to the result matrix elements are performed atomically, avoiding conflicts between threads.
    4. **Memory Management**:
        a. The result matrix is allocated before the parallel region, ensuring that each thread can safely write to its designated portions of the matrix without conflicts.

This parallelisation strategy aims to balance the workload across available threads while maintaining the correctness of the multiplication operation. The effectiveness of this approach can vary depending on the sparsity pattern of the matrices and the hardware characteristics of the system.

- Hybrid parallelisation (MPI + OpenMP)
    - This feature was not implemented due to constraints involving a limited access to Setonix during the testing stages of our project.

# 3. Experimental Methodology

## 3.1 Test Environment

- 16GB Memory
- MPICH 4.2.3 and OpenMPI version 5.0
- We attempted running tests on other cluster systems like Amazon EC2 ParallelCluster, but due to cost / time limitations were unable to test performance in a meaningfully different way to testing on the laptop.

## 3.2 Test Parameters

- We tested matrices ranging in size from 5000^2 to 40,000^2 (limited by the available memory on our local testing machine), in increments of 5000.
- We tested at three densities: 0.01, 0.02, and 0.05
- The number of processes was always set to 10 and the number of threads set to 10

# 3.3 Performance Metrics

## 3.3.1 Time Measurement

- CPU time and Wall Clock time measured using MPI_Wtime() and omp_get_wtime()
- Timing begins at matrix multiplication initiation and ends after result computation
- Communication overhead tracked separately for MPI operations
- Multiple runs performed for each configuration to ensure statistical reliability

## 3.3.2 Data Collection

For each experimental run, we tracked:

- Matrix properties:
    - Dimension (N×N)
    - Density (0.01, 0.02, 0.05)
    - Number of non-zero elements

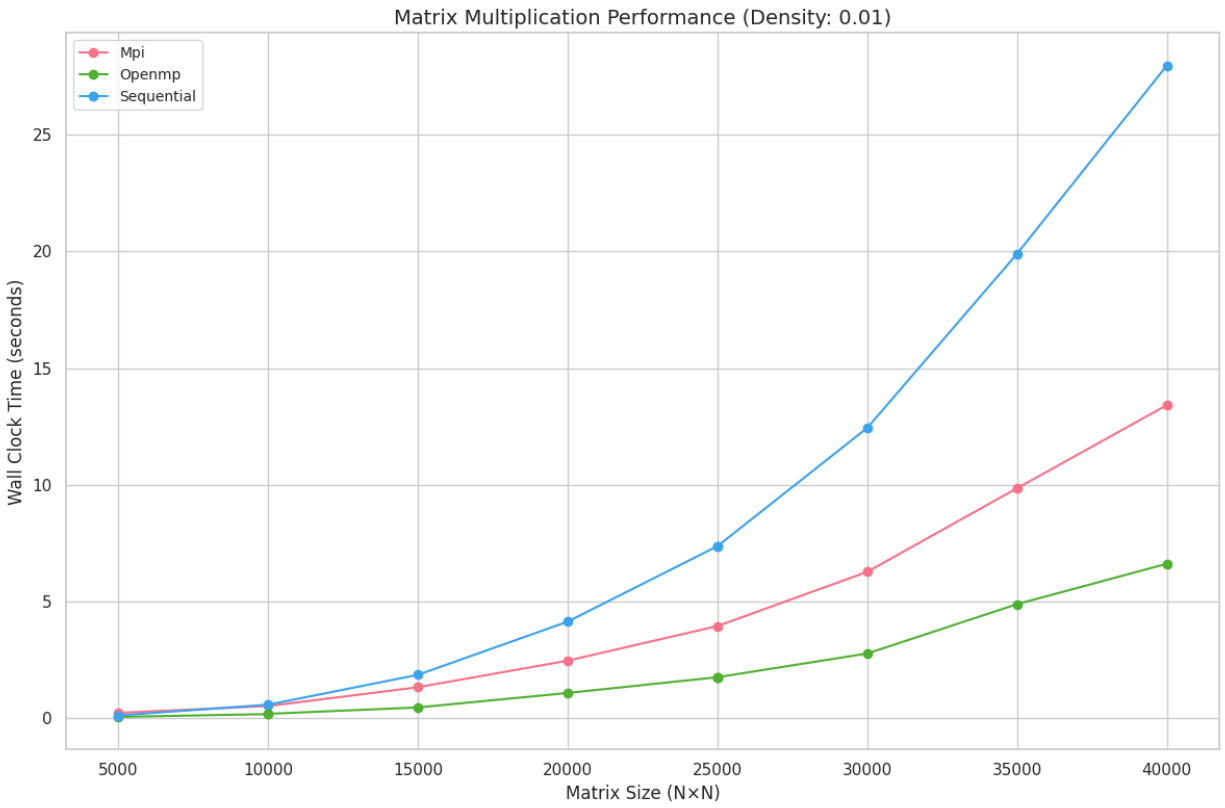
- Execution parameters:
    - Number of MPI processes
    - OpenMP threads per process
    - Node/core distribution
- Performance metrics:
    - Total execution time
    - Computation time
    - Memory utilisation

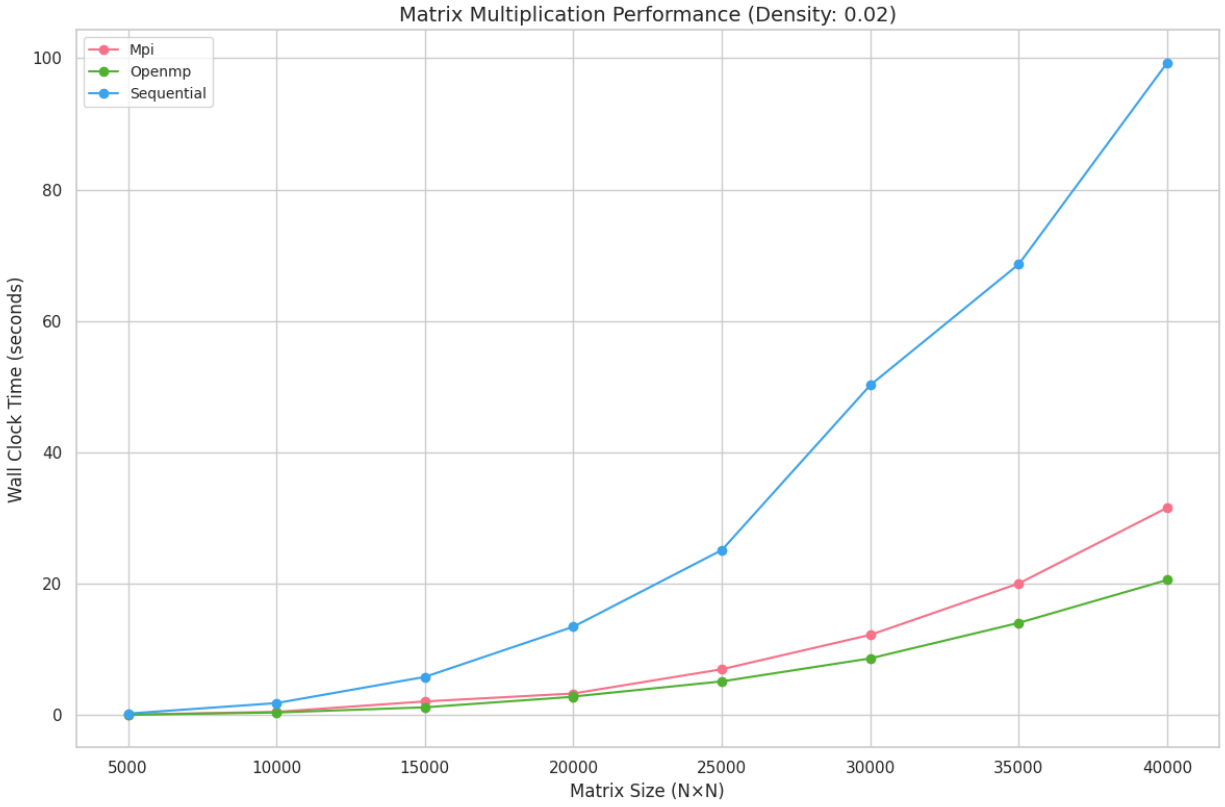
### 3.3.3 Data Management

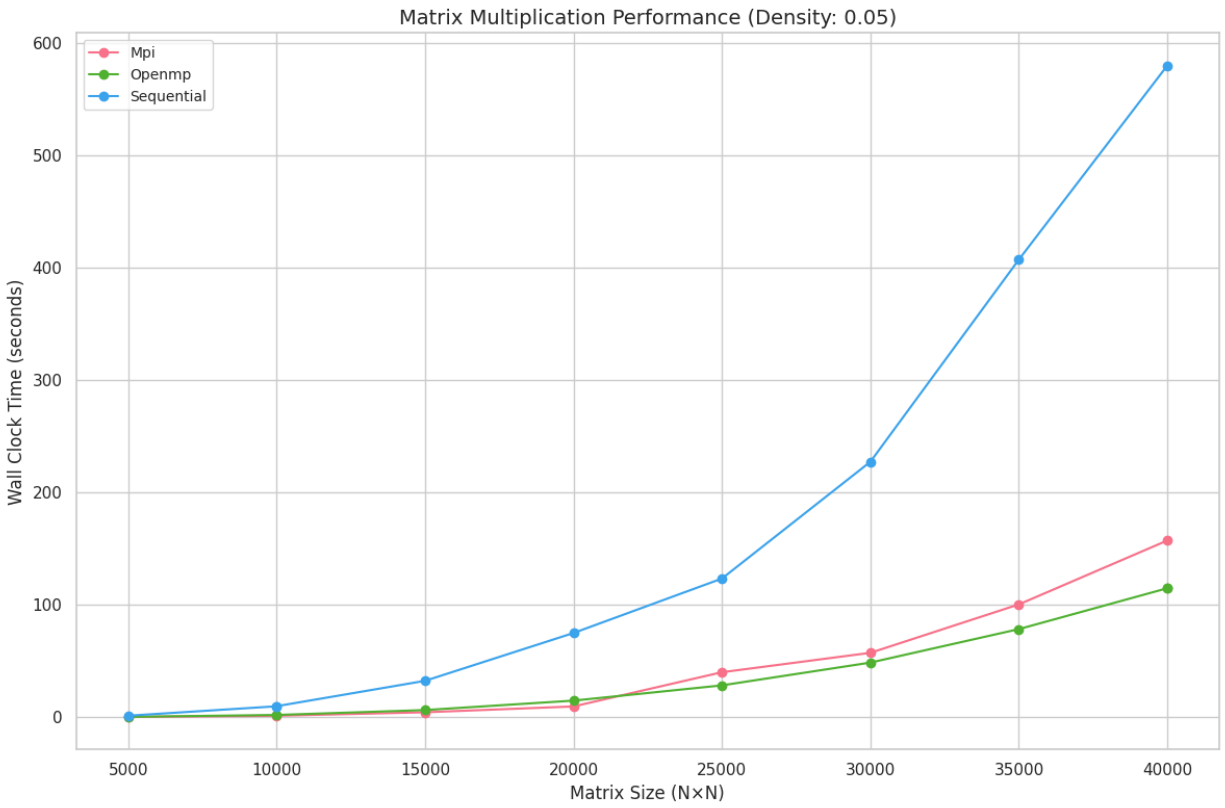
- Automated logging system stored results in structured log files, this is where you can find stored A and b matrices.
- Each run generates a unique, timestamp based identifier for traceability
- Log files contain complete execution context for reproducibility
- Performance data aggregated and processed using Python scripts for visualisation and analysis

# 4. Results and Analysis

The highest matrix size we could operate on was 40,000^2, we believe that this is due to a failing on our part with our implementation of our MPI row compression algorithm, and that had we improved it, we should be able to test up to ~25,000x25,000. This was the upper limit for sequential calculation, before running out of memory, so we would expect a similar limit for MPI. Although the difference is only 5000x5000, due to the memory use / performance curve as matrix size increases,  I believe this is substantial enough that if we were to conduct future investigations, we would be concerned about this discrepancy


Matrix Multiplication Performance (Density: 0.01)
Mpi
Openmp
Sequential
Wall Clock Time (seconds)
0
5
10
15
20
25
5000
10000
15000
20000
25000
30000
35000
40000
Matrix Size (N×N)


Matrix Multiplication Performance (Density: 0.02)
Mpi
Openmp
Sequential
Wall Clock Time (seconds)
0
20
40
60
80
100
5000
10000
15000
20000
25000
30000
35000
40000
Matrix Size (N×N)




## 4.1 Sequential Performance Baseline

- Sequential performance scaled according to the expected time complexity of O(n^3).

## 4.2 OpenMP Performance

- OpenMP outperformed all other parallelisation strategies, on our machine, as to be expected from a shared memory model

## 4.3 MPI Performance

- MPI performed marginally worse than OpenMP, due to the higher memory overhead demanded by a distributed memory model.
- The average time for MPI matrix multiplication was <time here>

## 4.5 Comparative Analysis

- From the figures above, we can see that our results almost perfectly reflected expected results. We believe that with greater memory we could see this even more clearly.

- As expected, increasing density massively increases the time taken, as the matrices are less sparse and row compression becomes less of a strategy to increase efficiency, but more of a burdensome calculation costing performance (to be clear we only timed multiplication timing, not generation / compression)
- Generally, as anticipated, Sequential is slowest, and MPI and OMP are comparable, with MPI typically behaving slower than OMP.
- At 0.01 density our results were surprising, as MPI consistently outperformed OMP. With greater matrices sizes, marginal outperformance was not present, and future investigation should be done as to why this occurred.

# 5. Discussion

Our results demonstrate the expected performance characteristics of parallel implementations on a single-node system. OpenMP consistently outperformed MPI in our local testing environment, which aligns with theoretical expectations for several key reasons:

1. Memory Access Model

- OpenMP operates with direct shared memory access
- MPI must replicate data across processes and manage explicit communication
- On a single node, MPI's communication overhead becomes pure performance penalty

2. Process Management Overhead

- OpenMP threads share the same address space and process context
- MPI processes require:
    - Full process initialisation
    - Memory allocation for each process
    - Context switching between processes
    - Message serialisation/deserialisation

These overheads are only justified when processes run on separate nodes like Setonix or our Amazon cluster (which didn’t work)

The inability to access Setonix significantly impacted our ability to demonstrate MPI's intended use case, as MPI is designed for  scaling across multiple physical machines and handling distributed memory.

Our attempted migration to Amazon EC2 (10 nodes, 1GB each) introduced additional challenges:

- Budget constraints limited testing and hardware we could access.
- Implementation complexity in a distributed environment unfamiliar, unlike Setonix

Several areas for potential improvement emerged from our analysis. For one, our current implementation separates compression and multiplication by a synchronisation barrier (for timing

purposes). For example, we could further optimise by overlapping compression and multiplication procedures. Another opportunity for optimisation arises in our communication constructs, the performance overhead of message passing is partially what caused our MPI solution to weaken in performance compared to OpenMP.

# 6. Conclusion

On our local machine, the fastest parallelisation method was accomplished using OpenMP's shared memory structure. Had this ran on Setonix, we could expect a real growth in processing speed with the combined potential of MPI's distributed memory model and OpenMP's shared memory model.