Project 2 Report

Gihad Coorey 23091788

Ethan Judge 23775697

Background

The aim of this project was to investigate the performance enhancements afforded through distributed-memory computation systems compared to shared-memory systems, and further compared to traditional sequential execution systems. Our initial strategy was to write a program that generated, compressed, and then multiplied square matrices of specified dimensions and test its time performance with various parallel computation paradigms such as multithreading and multiprocessing.

Given that Project 1 focused on multithreading with OpenMP, we decided to focus more on testing multiprocessing with distributed memory using MPI on multiple Setonix nodes. We would then give each configuration 10 minutes of wall time to run on Setonix and compare the largest size of matrices able to be generated, compressed, and multiplied within the allowed time for each configuration. We expected there to be a minimum size at which any matrices smaller than this size would experience a deterioration in performance compared to the single-process, single-thread configuration (or even indeed the single-process, multi-thread configuration), due to the overhead computation costs of establishing the MPI application. This was an extrapolation from our findings in Project 1, where the time taken to multiply smaller matrix sizes had a non-positive relationship with the thread count used to multiply them in parallel.

Unfortunately, our Setonix allocation as a class was exhausted before we could complete our tests, leading us to rely on local machine tests for the remainder of the project. To make the results comparable, this meant testing the configurations we had tested on Setonix again on our local machines. What we then found was that our local machine started using swap memory instead of RAM at quite small sizes (around 20 000), which made comparisons to smaller sizes invalid. Increasing sizes up to around 80 000 led to a significant increase in runtime, and sizes beyond 80 000 led to an exhaustion of system resources and caused a complete system crash in every case. As a result, it was impossible to test any meaningful increase in matrix size to distinguish the performance of the different configurations of process count and thread count.

We decided to alter our testing methodology entirely by holding constant the matrix dimensions being multiplied, and instead measure performance differences by the wall-time taken for each configuration to complete execution.

Method

We held our matrix size constant at 10 000 × 10 000.

We tested three matrix densities: 1%, 2%, and 5%. These densities simulated varying levels of efficiency for the row compression algorithm, allowing us to test different amounts of actual dot product calculations.

For each of these densities, we tested all possible combinations of 1, 2, 4, and 8 processes with 1, 2, 4, and 8 threads in each process. This yielded 16 possible testing configurations, each of which we tested 10 times to calculate an average runtime for each configuration.

Each test run had the program generate the sparse matrices from scratch, compress them according to the row compression algorithm from Project 1, multiply them, and print a simple `"DONE"` output to indicate completion. No other output was generated. Before testing, we ran the program with the `-c` flag to compare the result of the parallel computation to the sequential computation to ensure the matrices were correctly multiplied.

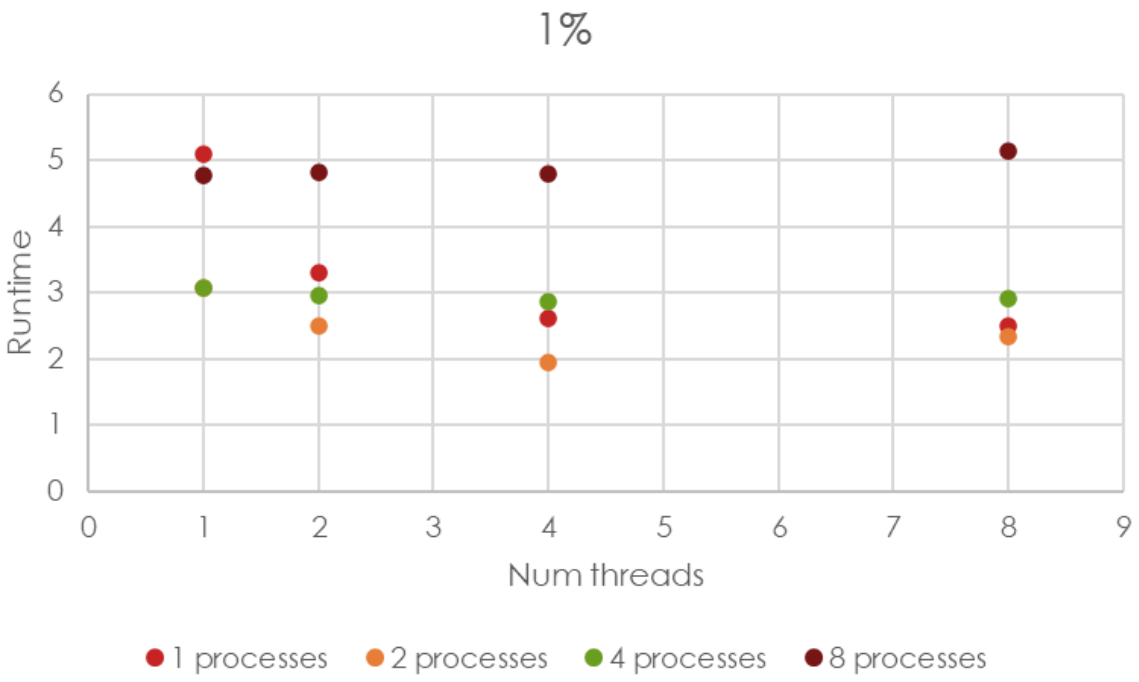
Runtime was measured using the `time` program available in the `bash` shell. From the output, the `real` time was recorded as this measured the wall time the entire program took to run, which is what we wanted to compare. The 1-process 1-thread configuration was taken to represent the sequential execution case, while the 1-process n-thread configuration was taken to represent the shared-memory-only case with multithreading.

Results

1%

	Num threads			
Num processes/nodes	**1**	**2**	4	8
1	5.099	3.304	2.614	2.512
2	3.084	2.504	1.961	2.348
4	3.077	2.956	2.877	2.915

	Num threads			
8	4.776	4.817	4.784	5.142

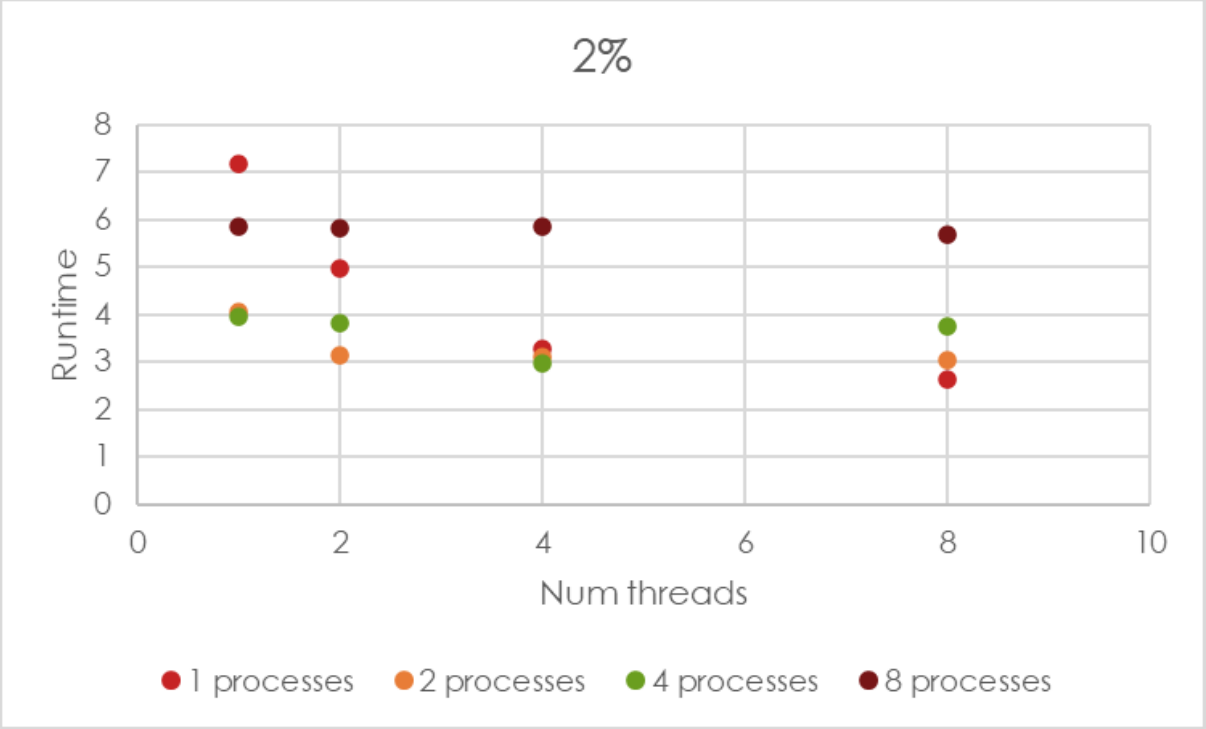


The tests with 1% density matrices show a clear distinction in performance across various thread and process counts. Configurations with 2 processes and 4 threads produced the shortest runtimes, suggesting a sweet spot between computational work and communication overhead. For this sparsely populated matrix, running more than 4 threads on higher process counts did not yield proportional gains, likely due to the minimal workload per thread and increasing overhead from inter-process communication. Configurations with the highest process counts (e.g., 8 processes) showed diminishing returns, and even performance degradation in some cases, as the cost of communication outpaced the computation time required to multiply the sparse matrices.

2%

	Num threads			
Num processes/nodes	1	2	4	8
1	7.192s	4.985	3.270	2.629
2	4.066	3.164	3.124	3.036
4	3.976	3.842	2.988	3.774

	Num threads			
8	5.869	5.823	5.851	5.707

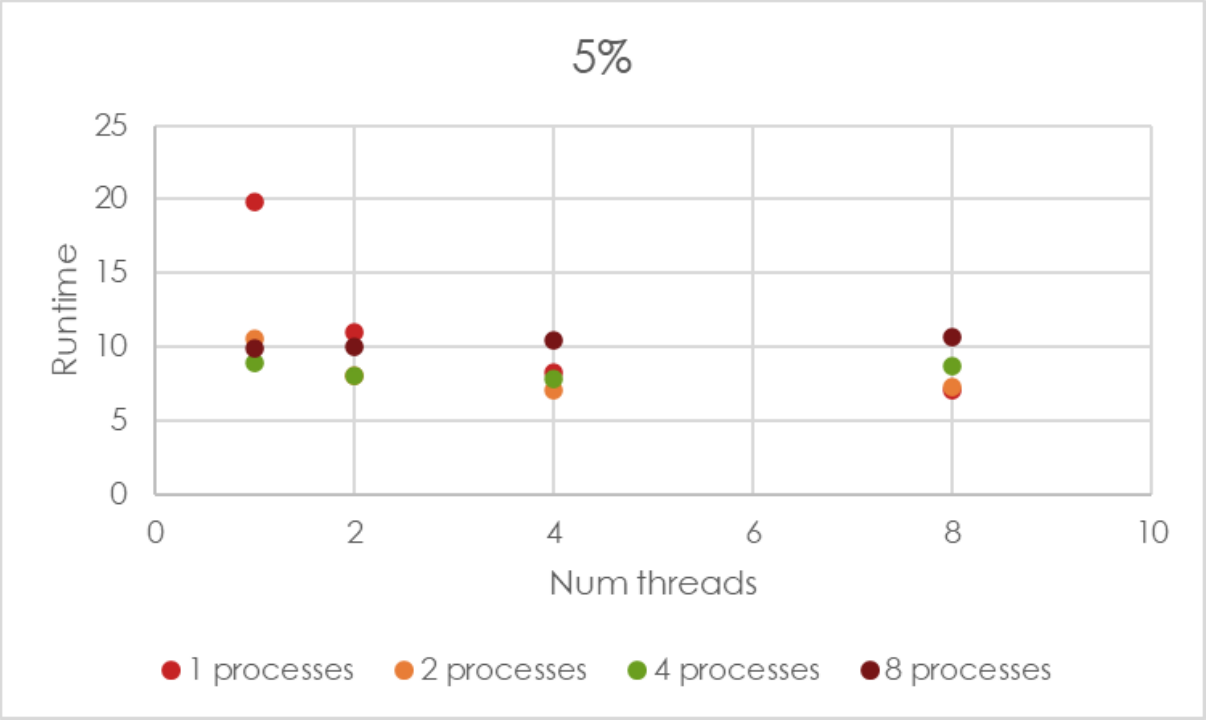


The 2% density matrix results revealed a more gradual improvement in runtime as thread counts increased up to 8 threads. However, unlike the 1% density matrix, performance gains from additional threads became less significant beyond 4 processes, indicating that communication costs start becoming a limiting factor with the increase in process count. Using 8 threads with 1 process yielded the lowest runtime, while configurations with 4 processes also performed well. There was a broader range of configurations that performed optimally with the 2% matrix density, as compared to the 1% density, suggesting that higher density matrices benefit more from increased thread counts.

5%

	Num threads			
Num processes/nodes	**1**	**2**	4	8
1	19.827	11.018	8.226	7.142
2	10.590	8.064	7.041	7.329

	Num threads			
4	8.893	8.013	7.812	8.751
8	9.944	10.055	10.480	10.618



For matrices with 5% density, performance varied less predictably with process and thread counts. As matrix density increased, so did the computational demand per thread, making it beneficial to use more threads and processes up to a point. However, with configurations beyond 2 processes and 4 threads, performance gains plateaued, and in some cases worsened, as the inter-process communication began to create overhead that outweighed computation time. The fastest execution time of 7.041 seconds was achieved with 2 processes and 4 threads in each.

Across all density configurations, we observed a few performance trends:

1. **Single Process/Thread**: The worst-performing setup in all cases was a single process with a single thread, as expected for its entirely sequential nature.
2. **Thread-Only (no MPI) and Process-Only (no OpenMP) Configurations**: Configurations with multiple threads under a single process and multiple processes with a single thread each showed varying degrees of

improvement. However, combining both threads and processes consistently performed better.

3. **Optimal Balance**: The ideal configuration varied by matrix density, generally favouring 2 processes for 1% and 5% density matrices but shifting slightly to 1 process for 2% density.
4. **Process Overhead with High Counts**: Using 8 processes generally resulted in reduced performance due to increased communication costs, particularly in configurations with lower matrix densities.

Our findings suggest that optimal performance is achieved by carefully balancing thread and process counts according to matrix density. Higher densities benefit more from threading, as each thread has sufficient workload to offset threading overhead, while lower densities can leverage fewer threads and processes to minimise inter-process communication time. We imagine that much larger matrix sizes may find more optimal configurations with a higher process count, as long as there are multiple computing nodes (or a more powerful single node) available to split the processes across.

Development Strategy

The big problem for Project 2 was to work out how to break the multiplication process down into independent parts that don't require shared memory access, so that we could distribute the work among processes. We realised that each row `i` in the matrix result `XY` depended only on the corresponding row `i` in matrix `X` and also on the entire matrix `Y`. This meant that each process could generate a part of matrix `XY` row-wise with its corresponding partial `X`, but would need a copy of the entire matrix `Y` in order to calculate the partial result rows. There were a few solutions to get around this.

1. Have the master process generate a queue of tasks, and all free processes consistently query the master process for a task (or chunk of tasks), similar to the OpenMP `schedule(dynamic)` paradigm. This was quickly dismissed since the task size was likely to be quite uniform, so it would be sufficient to divide them up statically in the code rather than at runtime using the Master-Worker model. It also required at least 2 processes to be running, which meant that sequential execution would have to be tested separately, which felt wasteful.
2. Task pipelining - have separate dedicated processes for generating, compressing, and multiplying. This was also quickly dismissed since the task

division would have been extremely uneven computation-wise, and processes would have been left idle frequently.

3. Have each process generate a partial `X` and an entire `Y`, ensuring that `Y` was the same across all processes by seeding all of the random matrix generations with the same value. Then have each process compress its partial `X` and whole `Y` and calculate a partial `XY` independently. Finally, use MPI to gather all partial `XY` matrices into a single whole `XY` matrix and return this as the result. This approach would have been quite simple to code and run but was deemed unideal since there would be a large amount of redundant work generating the same `Y` matrix across all the processes, wasting computing resources.
4. Generate both `X` and `Y` row-wise with each process generating a set number of rows in both. Then use MPI to broadcast each process's `Y` matrix to all other processes, so that each process can build its own whole `Y` matrix. Then each process multiplies its partial `X` matrix with the required rows in `Y` to generate a partial `XY` matrix. Finally, use MPI again to gather all partial `XY` results into an overall result. For larger matrix sizes, this was expected to perform much better than the first approach, as the computation of generating and compressing the `Y` matrix takes up more of the overall computing time compared to the MPI broadcasting overhead, making it much more efficient at scaling up compared to the first option.

To generate the sparse matrices `X` and `Y`, each process was made to generate a portion of the rows of the whole matrix by dividing the required matrix size by the number of processes running in the global communicator. This gave the number of rows required to be generated by each process. The random number generator used to populate the matrices was seeded by the rank of the process for reproducible results. Because each process's partial `X` and partial `Y` was being seeded by the same value, both partial matrices would have been equal. However, we treated them as logically different to avoid oversimplifying the problem.

Once all processes had generated and compressed their partial matrices, the next step was for each process to share its partial `Y` with all the other processes, while receiving the other processes' partial matrices in order to reconstruct the entire `Y` matrix in each process. There were a few options on how to do this.

1. Gather all the partial matrices into the master process using `MPI_Allgather`, compile them all into a single `Y` matrix in the master process, then broadcast

to all the other processes using `MPI_Broadcast`. This was an ok strategy, but risked bottlenecking in the master process when combining all the partial `Y` matrices and leaving the other processes idle.

2. Have each process broadcast its partial `Y` to all other processes, and immediately begin the multiplication process once the whole `Y` matrix had been reconstructed. While this process involved duplicating the reconstructing work in each of the processes, this approach was not obviously superior to the first alternative. However, the existence of the `MPI_Allgatherv` function of the MPI library for this exact operation meant that we wouldn't have to write a custom function to gather and broadcast the `Y` matrix, and also got to benefit from MPI's under-the-hood optimisations for the inter-process communication.

Presumably, there would be a minimum matrix size for which the gather/broadcast strategy resulted in too long of a bottleneck in the master process. Below this minimum size, the duplication of work in each process using the `Allgatherv` function would actually prove detrimental to performance due to the overhead of communicating to all processes instead of just the master. While we didn't have time to experiment and find this minimum size, we assumed that the matrices we were dealing with would be large enough that they would exceed this theoretical minimum.

The next problem to solve was how to transmit and receive the custom `CompressedMatrix` data structure as a 1D buffer, since the send and receive functions on MPI only allowed for primitive types to be transmitted, such as `MPI_Int`. One solution to this problem was to create custom `MPI_Datatype` which stored the metadata and data of the `CompressedMatrix` struct as an array of `MPI_Datatypes`. This would have required us to use the `MPI_Type_create_struct` function to convert our C-struct into an MPI-datatype, and then pass this datatype around using the various send and receive functions of the MPI library.

Another solution to this was to create serialising functions to *flatten* the matrices along with their metadata and then deserialise the received buffer using the known *flattened* structure. We serialised the compressed matrices by populating the send buffer one matrix row at a time, with each row element being preceded with its size. This way, the deserialising function would be able to infer where one row ended and the next started. We decided that this approach would be simpler to deal with due to the variable lengths of the rows in the `CompressedMatrix` structure.

Once all the processes had received the flattened partial matrices and reconstructed the complete compressed matrix `Y`, the compressed matrix multiplication algorithm from Project 1 was applied to multiply each process's partial `X` with the entire `Y`. These partial results were then gathered into the master process using `MPI_Gather` for output. The master process could then reconstruct the entire `XY` matrix result and write it to a file (or `stdout`).

As a way to check that our results were consistent with the results from the sequential compressed matrix multiplication algorithm, we added a `compareFlag` command-line option which added an additional instruction for the master process to `MPI_Gather` the compressed partial `X` matrices, reconstruct the whole compressed `X` matrix, multiply it with the whole compressed `Y` matrix sequentially, and compare the resulting `XY` matrix to the result yielded by first part of the program. We already knew that the compressed multiplication algorithm correctly generated the same output as the standard matrix multiplication algorithm because we tested it in Project 1. So comparing this result to the result generated by the parallel portion of our program ensured that the final result was correct.

To further parallelise the program using shared memory, we added `#pragma omp parallel for` directives to all the relevant matrix multiplication `for` loops as done in Project 1. Since we discovered that the `dynamic` schedule had the most consistent performance for larger matrices, we ran all our tests with this scheduling strategy kept constant. This allowed for a more direct observation of the effects of thread counts and process counts.

Usage instructions

The program is reasonably self-explaining when run with the `-h` command-line option. It must be compiled using `gcc` with `-fopenmp`. MPI provides a compiler called `mpicc` which wraps `gcc` or `clang` depending on the version and operating system. The compiler wrapped by `mpicc` can be found by running `mpicc --showme`. Since `gcc` has `openmp` defined in its include path, the program can simply be compiled using

`mpicc -fopenmp <filename.c> -o <filename.o>` .

For OSs where `mpicc` wraps `clang`, the compilation is a bit more complicated since you have to explicitly specify the location of the OpenMP header files. It would look something like

`mpicc -Xpreprocessor -fopenmp -L <path/to/libomp/lib> -I </path/to/libomp/include> -lomp <filename.c> -o <filename.o>`.

The program can then be run with

`mpiexec -np <num_processes> ./filename.o -h` for usage instructions.

The supported options are:
-t <num_threads> Number of threads to use for OpenMP. A warning will be printed if more threads are requested than the number of physical cores available on the node, but execution will continue.
-d <density> Density of the generated matrices, between 0 and 1.
-s <size> Size of the (square) matrices to multiply. Must be a whole multiple of `num_processes`.
-o [<output_file>] Output file to write compressed matrices. Pass `stdout` to print to stdout.
-c Flag to compare parallel result to sequential result.
-h Print help message.

The following command will run the program with 2 processes, 8 threads per process and 100×100 matrices of 2% density. It will write matrices to a file called `out.txt` and also compare the combined partial result matrices to the one generated by running the compressed multiplication algorithm sequentially in a single process.

```
mpiexec -np 2 ./obj.o -s 100 -t 8 -d 0.02 -o out.txt -c
```

The program will print an error to `stderr` if the requested thread-safety level of `MPI_THREAD_FUNNELED` is not provided by the given MPI implementation, or if any of the command-line arguments are invalid.

Depending on the OS, you may need to explicitly add executable `chmod +x` permissions to the object file for `mpiexec` to successfully run the program.