Parallel and Distributed Programming – Spring 2025 Report for Individual Project on Shear Sort

Oskar Perers

30th May 2025

1 Introduction

This individual project set out to implement a parallel solution to the shear-sort algorithm [1]. The main goal for the project was to achieve a speedup using a parallel solution for the sorting algorithm while creating an understanding of why the parallel version might outperform the sequential version and what the limitations are. In addition to this, the project set out to optimize the solution as far as possible within the time frame.

2 Algorithms

The algorithm used for shear-sort was given in the lecture on parallel sorting algorithms [1]. The goal is to sort a matrix in snake-like order. The main method of the sequential algorithm for a square matrix of size $n \times n$ is described as follows:

1. Sort even rows in ascending order - $\mathcal{O}(n)$
2. Sort odd rows in descending order - $\mathcal{O}(n)$
3. Sort each column in ascending order - $\mathcal{O}(n)$

Then repeat $\log(n) + 1$ times to sort $n \times n$ numbers. There was also a theorem presented that indicated that only the row steps require $\log(n) + 1$ times and that the column phases can be reduced to $\log(n)$ times.

A step-by-step execution of this algorithm can be seen in Figure 1.

2.1 Parallel Solution

To parallelize this algorithm, I decided to simply divide the rows/columns between the processes as evenly as possible and sort them locally. Between the row and column "phases", the matrix would be transposed and evenly divided again. By doing this, the sorting step could always be done on consecutive memory, utilizing more of the cache line.

The solution was implemented in `C++` to avoid some of the manual memory management associated with `C` but still take advantage of the speed of using a low-level language. To simplify debugging and improve readability and maintainability, the `matrix2D` class was implemented to encapsulate global and local matrix data and to support the reuse of functionality through abstraction. The main program was further divided into helper functions to improve readability. These were the following:

- `read_input()` which either reads the inputs from a provided file or generates a matrix with random elements based on a provided matrix order.

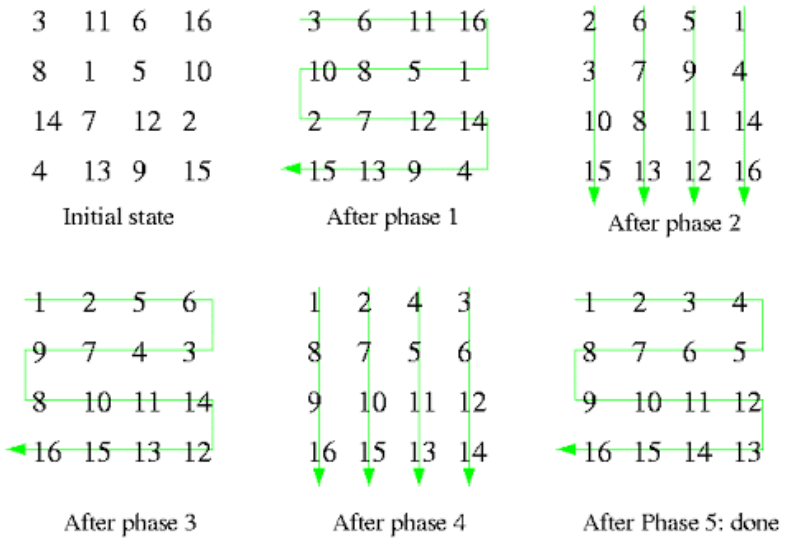

Figure 1: Example execution of the shear sort algorithm for a matrix of size 4×4 [1].

- `distribute_from_root()` which distributes the input data from the `ROOT` node to all processes such that each process has at most one more row than any other process to maximize load balancing. This was implemented using `MPI_Scatterv` to minimize communication overhead while allowing for uneven matrix sizes.
- `global_sort()` that performs the main part of the global sort method, sorting rows, exchanging data between processes, sorting columns and then exchanging data between the processes again. The implementation and optimizations here will be discussed further below.
- `gather_on_root()` which collects the sorted data from each process back to the `ROOT` node using `MPI_Gatherv` to also minimize communication overhead while allowing for uneven matrix sizes.
- `check_and_print()` which is used by `ROOT` node to check that the matrix has been sorted correctly as well as optionally writing the sorted matrix to a provided output file.

Now for the `global_sort()` implementation. When calling the method, you provide the local data stored in a $n \times (n/Nr\_of\_processes)$ matrix as well as the global index of the first row in the local data partition to know which rows to sort ascending/descending. The local sorting is done using `std::sort()` which uses iterators to sort a provided data partition.

After the initial set of rows are sorted the data is exchanged with all other processes in such a way that each process receives a set of columns stored in column-major order. This is done with `MPI_Alltoallw()` which is a version of `MPI_Alltoall()` that, similarly to `MPI_Alltoallv()` allows for process-specific send/receive counts as well as displacement but, in addition, allows the buffer elements to have different types [2]. This allowed data to be sent and received in place without having to rearrange the data in specific send/receive buffers, reducing memory usage. The reason this was so powerful was that, using derived data types [3], this method was able to account for both a differing number of rows/columns from each process and redistribution of the data such that each process received all data corresponding

to a column partition of the global matrix and using some clever tricks, it was also able to transpose the received data to store it in column-major order.

Before the call to `MPI_Alltoallw()` the data is stored sequentially in row-major order. The send types are thus defined as `MPI_Type_vector()`s with a count equal to the number of rows in the local data, a block size equal to the number of columns the receiving process is allocated and a stride equal to n to read the next row in the block. This datatype in collaboration with the send-displacement based on how many columns each process shall receive, divides the data correctly while still reading it straight from the local matrix. No extra buffer needed.

The receive types are even more complicated. For this, we first define a `MPI_Type_vector()` that shall store the incoming row as a column. This is how each block is transposed at arrival. For these datatypes, the count is still the number of rows in the local data partition and the stride is equal to n to access the next row in the block but the block size has been reduced to one, effectively transposing the received data.

To receive multiple of these column vectors next to each other such that the start of the next column vector is just one element after the last one, `MPI_Type_create_resized()` is used [4]. This allows us to redefine the "extent" of the datatype which is basically how far apart from each other they logically are stored when packed. By default, the extent of a datatype is the amount of bytes between the first and last element in the datatype. This is usually correct, as we expect each datatype to be stored back-to-back in memory but as we are trying to interleave the vectors, this will be incorrect. We therefore resize the extent of the datatype to `sizeof(int)` in order to store them back-to-back. Finally, the resized vector type is used to construct an `MPI_Type_contiguous()` for each process we receive from, with count equal to the number of rows we receive from each of those processes.

Now when `MPI_Alltoallw()` is called with these datatypes, displacements and counts equal to one of these corresponding types per destination/source. The received data is stored in a new matrix of equal size to the local data matrix but is now instead a set of columns thanks to the interaction between `MPI_Alltoallw()` and the derived datatypes.

Now the column phase is executed by sorting the columns locally in each process, again using `std::sort()`. Next, to exchange the data back to row partitions, we utilize the same datatypes as we used to transpose the data to column partitions using `MPI_Alltoallw()` just swapping the send and receive locations.

3 Experiments and results

The experiments conducted for this assignment were numerical experiments that tested the correctness of the algorithm and also performance experiments that tested the correctness of the algorithm and the runtime of the algorithm in order to understand the strong speedup and weak efficiency of the program.

The first experiments that were conducted tested the program on different input files. Since the program implemented contains a correctness checker, which checks if the numbers in the resulting matrix were sorted in ascending snaking order, the program only needed to be tested on different input files. These files were smaller, which wasn't really useful for the performance experiments.

For performance testing, a multitude of combinations were executed on the Snowy server on Uppmax, using job scripts to run the varying number of processes and input sizes. These jobs were designed to test either strong or weak scalability. Another parameter that varied was the number of nodes allocated on the server. This is to try and find out if adding more memory bandwidth would improve performance.

For strong scalability testing, the input size was fixed at $8000 \times 8000 = 64e6$. The number of processes varied and was, for one job set to 1, 2, 4, 8, 16, and 32, and in the other, made to match the number of processes used for weak scalability testing.

For weak scalability testing, the input size was fixed $2000 \times 2000 = 4e6$ per process. To calculate the order of the input matrix for a specific process count Equation 1 was used.

$$\begin{aligned} N &= k \times P \\ n^2 &= 2000^2 \times P \\ \sqrt{n^2} &= \sqrt{2000^2 \times P} \\ n &= 2000\sqrt{P} \end{aligned} \tag{1}$$

This implies that for this equation to result in a well-defined input, the number of processes must be a perfect square. Thus, the first five perfect squares were chosen, i.e. 1, 4, 9, 16 and 25, as the number of processes and the order of the input matrix was calculated using the Equation 1.

For timing, `MPI_Wtime()` was used to measure the execution time from just before distributing the elements from the root until the sorted matrix has been gathered back on the root. This gives us a good understanding of the scalability of the full parallel algorithm.

Each configuration was run 5 times and then the median time among the five times was used as the actual runtime of the configuration. This in turn reduces variance in the test results.

3.1 Results

The results can be seen in the tables and figures presented and will be further discussed in Section 4. The speedup for the strong scalability tests was calculated using Equation 2 and the efficiency was calculated using Equation 3.

$$S_p[S] = \frac{T_1}{T_p} \tag{2}$$

$$E_p[\%] = \frac{T_1}{T_p} \cdot 100 \tag{3}$$

Processes	Matrix order	Runtime (s)	Speedup
1	8000	73.8871	1.00
4	8000	19.1991	3.85
9	8000	9.0103	8.20
16	8000	7.5734	9.76
25	8000	9.6425	7.66

Table 1: Strong Scalability test ran on an input size of 8000×8000 elements on a resource allocation of 2 Nodes.

4 Discussion

Strong speedup seen in Table 1, Table 2 and Figure 2 is good up to about 8 processes. This is expected as we divide the main work evenly between processes. There also aren't any real

Processes	Matrix order	Runtime (s)	Speedup
1	8000	73.6798	1.00
2	8000	41.2932	1.78
4	8000	19.1554	3.85
8	8000	9.8541	7.48
16	8000	8.7318	8.44
32	8000	9.3646	7.87

Table 2: Strong Scalability test ran on an input size of 8000×8000 elements on a resource allocation of 3 Nodes.

Processes	Matrix order	Runtime (s)	Weak Efficiency (%)
1	2000	3.029	100.0
4	4000	3.8659	78.3
9	6000	4.6744	64.8
16	8000	7.5781	39.9
25	10000	12.7335	23.8

Table 3: Weak Scalability test ran on an input size equal to 2000×2000 elements per processes on a resource allocation of 2 Nodes.

bottlenecks in the algorithm. There are data dependencies between the processes in each iteration, but this does not really result in any decrease in performance. This is because sorting the rows/columns locally takes about equal time, as they are all of equal length and should, in addition, take way more time than the communication. Even if we account for variance in the time it takes to sort one specific row/column, it is unlikely to result in processes having to wait for each other for a significant time this variance will fall close to a normal distribution for a large number of sorting operations.

The reason we see such a drastic decrease in speedup as we increase the number of processes past 8 processes is likely due to Ahmdahl's law in addition to an increase in communication overhead as the number of processes increases. The collective MPI calls will take longer time proportional to the number of processes as we divide the data into more chunks. The initialization of MPI datatypes is included in the part of the execution that is timed which further decreases the potential speedup according to Ahmdahl's law. To definitively find what the reason is, more fine-grained benchmarks are needed, such as testing larger input files and timing smaller parts of the algorithm.

Weak scalability seen in Table 3, Table 4 and Figure 3 is quite bad. This is to be expected as when we increase the amount of elements proportional to the number of processes the work done by each process actually increases. The reason for this is that we are not just adding new rows/columns that can be evenly divided between the processes but in addition we are increasing the length of the rows/columns. This increases the complexity of the local sorting. Thus, we can't expect linear weak scalability.

A final conclusion is that we are unlikely to be memory bound for these results. This is because the differences in performance between the allocation of 2 vs. 3 nodes for execution were minimal and could just be caused by the margin of error.

There is a multitude of blocking MPI calls in this implementation. Swapping the scatter and gather operations used at the beginning and end of the program for their non-blocking counterparts could be experimented with but should ultimately be insignificant. This is because of the data dependence between the processes. The processes need the data from the scatter to continue the algorithm and the root needs the data from all processes before it can finish the

Processes	Matrix order	Runtime (s)	Weak Efficiency (%)
1	2000	3.0424	100.0
4	4000	3.8620	78.8
9	6000	4.6616	65.3
16	8000	9.0355	33.7
25	10000	13.3102	22.9

Table 4: Weak Scalability test ran on an input size equal to 2000^2 per processes on a resource allocation of 2 Nodes.

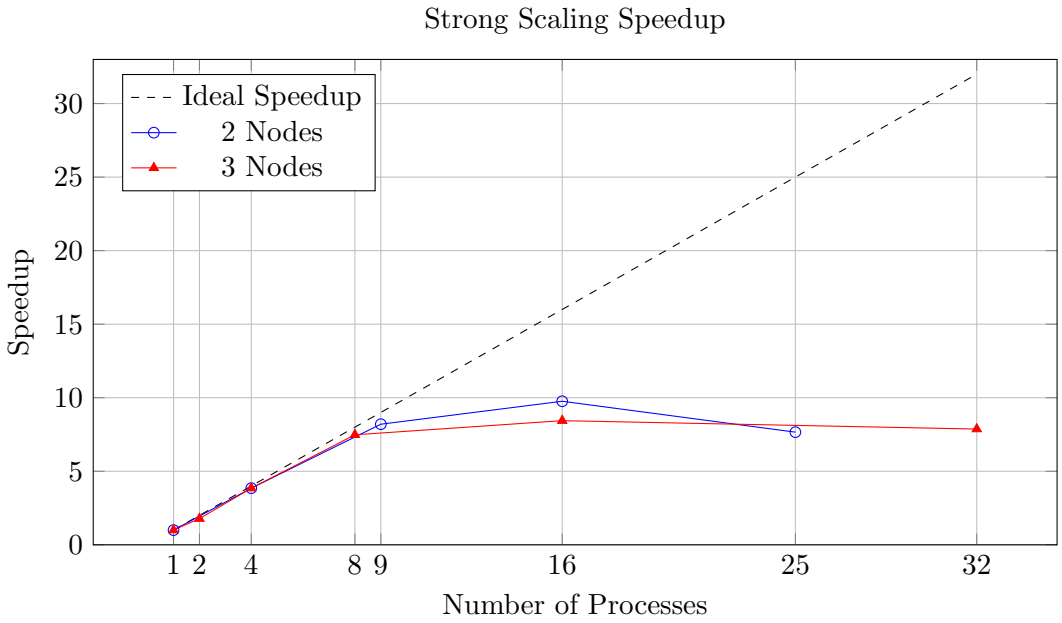

Figure 2: Plot for strong speedup based on results from Table 1 and Table 2.

program. Thus, we do need synchronization after these calls to ensure correct execution. The same can be said for the `MPI_Alltoallw()` calls in the main algorithm loop. The data from all processes are required to continue execution and thus some synchronization is necessary and thus using blocking communication shall not decrease performance significantly.

4.1 Possible improvements

I did try to use MPI_in_place but it failed as it did not transpose the data portion that stayed on the process but this could be solved by transposing that data portion manually. This would be particularly good if there existed a non-blocking version of `MPI_Alltoallw()` that could transpose the local partition while waiting for the data from the other processes but I could not find one. Ultimately I did not have time to experiment with this solution and I am also unsure how `MPI_Alltoallw()` would interact with MPI_in_place as it is generally used with matching send and receive types. The main advantage of this optimization would be to further reduce memory usage, which can be important if we can't fit two local data buffers in the cache.

Another parallel algorithm that would be interesting to look at and possibly implement would be to divide the matrix data into evenly sized square blocks that would use a method similar to the parallel quicksort algorithm implemented in Assignment 3 to sort the rows and columns. This would ultimately improve data locality but would in turn increase the amount

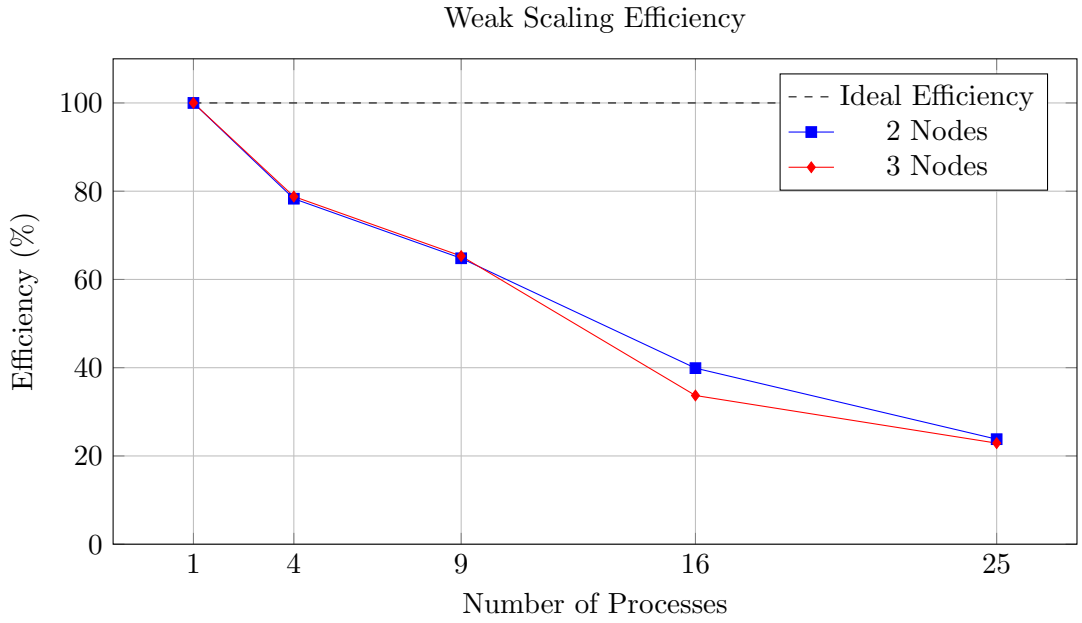

Figure 3: Plot for weak efficiency based on results from Table 3 and Table 4.

of communication overhead. Therefore, speculating on possible performance improvements of that solution is difficult.

5 Conclusion

Overall, I am very satisfied with my implementation and optimizations. The solution performs in a satisfying and expected way apart from the strong scalability at high process count. Despite this, I have come to reasonable theories of why this occurs. Finally, I have found a possible topic for my master's thesis in developing and testing a non-blocking version of `MPI_Alltoallw` which I will look into during the coming months.

6 Peer review

A review was conducted in collaboration with Adam Ross, in which we reviewed parts of each other's work. From this I received feedback on a multitude of things, including but not limited to test setup, result visualization, discussion of results, spelling and grammar. I further integrated this feedback into my work to enhance the outcome.

7 Declaration of AI use

AI has been used throughout the project to help with debugging, answering questions related to MPI documentation and better understand my solution by giving feedback on the proposed implementation. Though, to be honest, during the final stage of debugging, which took 12 hours, it was of little help, as it generally proposed solutions already tried or potential errors related to older versions of the documentation. AI was also used to create script skeletons for generating matrix input files as well as generating tables and plots from the raw results.

References

[1] Roman Iakymchuk. Parallel sorting algorithms. Lecture notes, 1TD070 Parallel and Distributed Programming, Uppsala University, 2025. Accessed: May 2025.

[2] RookieHPC. MPI_Alltoallw. `https://rookiehpc.org/mpi/docs/mpi_alltoallw/index.html`. Accessed: May 2025.

[3] Roman Iakymchuk. MPI: Derived Data Types. Lecture notes, 1TD070 Parallel and Distributed Programming, Uppsala University, 2025. Accessed: May 2025.

[4] RookieHPC. MPI_Type_create_resized. `https://rookiehpc.org/mpi/docs/mpi_type_create_resized/index.html`. Accessed: May 2025.