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

Oskar Perers

29th May 2025

1 Introduction

This individual project set out to implementing 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

- Describe the sequential algorithm that underlies the parallel algorithm.

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 rows 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

- Describe the parallel algorithm and your implementation using MPI.

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 ease debugging and readability the `matrix2D` class was developed to hold the matrix data, both globally and locally some as well implementing some reoccurring methods such as `resize()` and `tostring()`. The main program was further divided into helper functions to improve readability. These were the following:

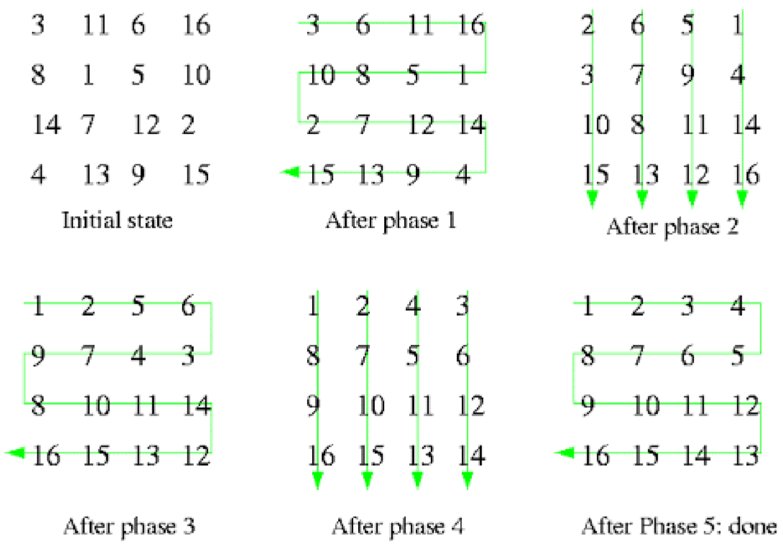

Figure 1: Example-execution of the shear sort algorithm for a matrix of size 4×4 [1].

- `read_input()` which either reads the inputs from a provided file or generates a matrix with random elements based on a provided matrix order.
- `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 it 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 is thus defined as a `MPI_Type_vector()` with a count equal to the number of rows in the local data, block-size equal to the number of columns the receiving process is allocated and stride equal to n to access 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 this type 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 between each other they logically are stored when packed. By default the extent of a data-type 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.

Finally `MPI_Alltoallw()` is called with these data-types, 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()`. Now 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

• presenting how you evaluate the performance of your solution along with your results, observations and comments.

The experiments conducted for this assignment were numerical experiments that tested the correctness of the algorithm and also performance experiments which tested both the correctness of the algorithm, runtime of the algorithm as well as strong-speedup and weak-efficiency.

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 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.

Further explanations on experiments to come!

For the performance experiments, the program was tested for both strong- and weak scalability, on both input files with random order and on files which was presented in descending order meaning that the entire input had to be flipped. These tests were run on the Rackham server on Uppmax, using job scripts to run with a processor amount of 2^k, where $k \in 0, 1, 2, 3, 4$.

3.1 Results

Processes	Runtime (s)	Speedup	Efficiency (%)
1	73.8871	1.00	100.0
4	19.1991	3.85	96.2
9	9.0103	8.20	91.2
16	7.5734	9.76	61.0
25	9.6425	7.66	30.6

Table 1: Strong Scalability (2 Nodes)

Processes	Runtime (s)	Speedup	Efficiency (%)
1	73.6798	1.00	100.0
2	41.2932	1.78	88.8
4	19.1554	3.85	96.3
8	9.8541	7.48	93.5
16	8.7318	8.44	52.7
32	9.3646	7.87	24.6

Table 2: Strong Scalability (3 Nodes)

Processes	Scenario Size	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 (2 Nodes)

The results can be seen in the tables and figures presented and will be further discussed in Section 4.

4 Discussion

With explanations of the results and with ideas for possible optimizations or improvements.

Strong speedup good as we divide the rows evenly between processes. What happens at the topend?

Weak speedup quite bad after 4-9 processes. This is expected as when we increase the amount of elements proportional to the number of processes we do not just add extra rows that can be evenly divided. We also increase the length of the rows which increases the complexity of the sort of each row. Thus we cant expect linear weak scalability

More Discussion on results to come!

Processes	Scenario Size	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 (3 Nodes)

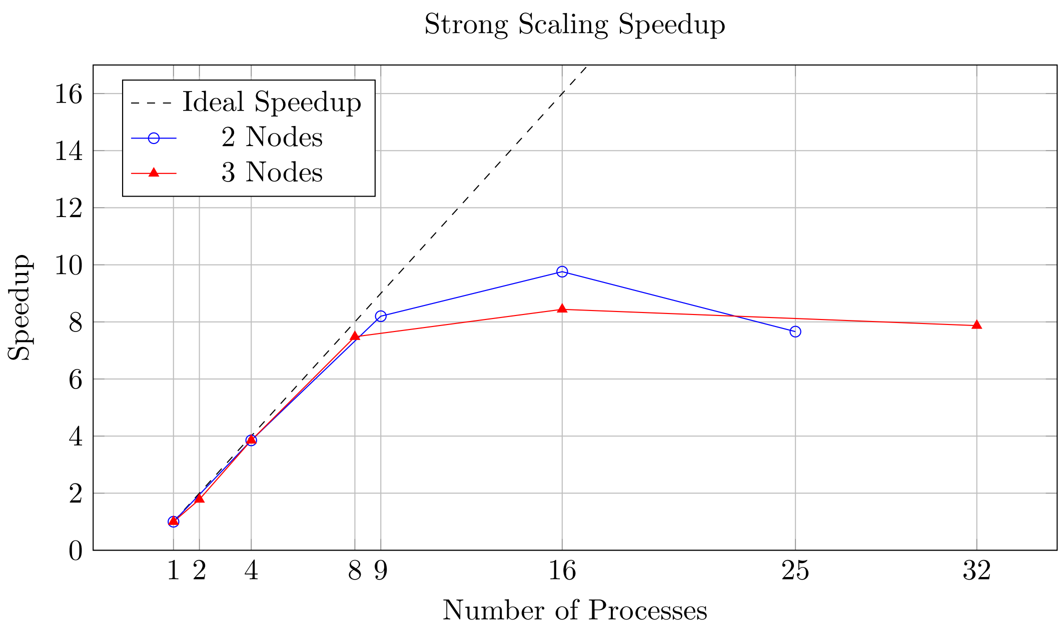

Figure 2: Strong Scaling Speedup with Ideal Line

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 might 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 of communication overhead. Therefore speculating in possible performance improvements of that solution is difficult.

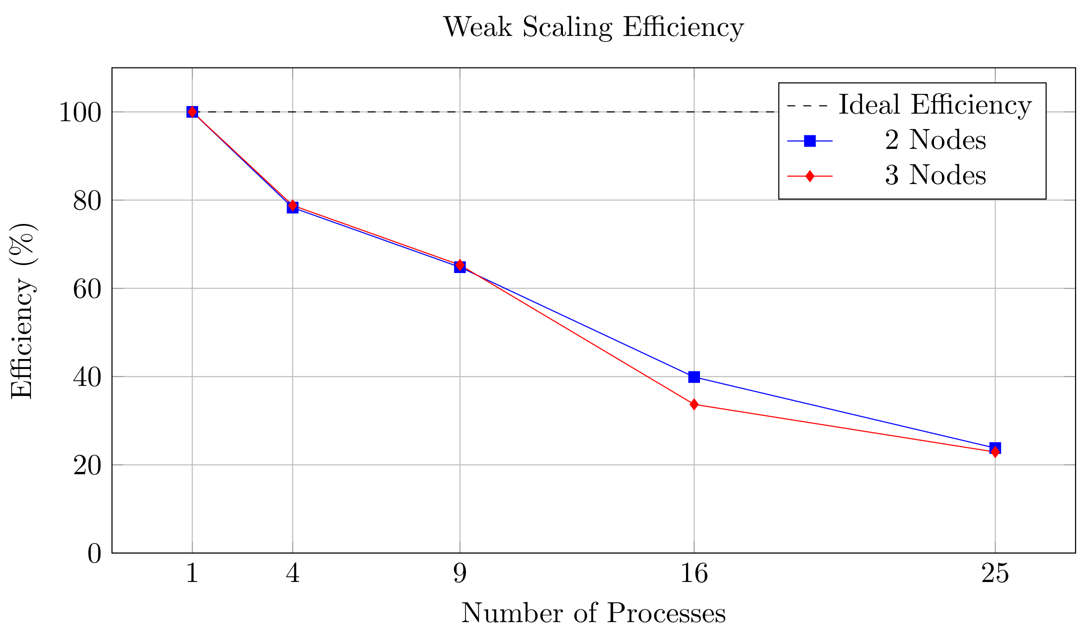

Figure 3: Weak Scaling Efficiency with Ideal Line

5 Conclusion

Overall I am very satisfied with my implementation and optimizations. The solution performs in a satisfying and expected way. Add more closing regards.

6 Deceleration of AI use

AI has been used throughout the project to help with debugging, answering questions related to MPI documentation and to better understand my solution by giving feedback on the proposed implementation. Though to be honest, during the final stage of debugging, that took 12h, it was to 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 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$_alltoallw/index.html.Accessed :$ May2025.

[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$_type_create_resized/index.html$ May2025.