Parallel and distributed programming

1TD062

Individual project
Shear sort

Author:
Cecilia Ahnlund

August 19, 2025

Contents

1 Introduction

The shear sort algorithm is a repetitive sorting algorithm which repeats three steps until a matrix is sorted. The object is to make the numbers of a matrix appear sorted in a snake like order. The algorithm, containing a lot of independent tasks, has a great potential for improvement if implemented as a parallelized algorithm instead. In the following report a description of a parallelized solution of the Shear sort problem will be presented, together with performance experiments where the parallel algorithm is compared to a sequential run, to determine the speedup and effectiveness of the provided algorithm.

2 Algorithms

2.1 Sequential

The sequential algorithm can be described as follows: the first step sorts even numbered rows in ascending order the second one odd in descending order and after this all columns is sorted in ascending order. The process is repeated for $log_2(n)+1$ iterations where n represents the number of rows in the matrix. Over these iterations, the elements gradually move into their correct order following the serpentine pattern.

2.2 Parallelization

For parallelization to be possible the matrix is first divided between the processes row wise by calculating the number of rows divided by the number of processes available. If not possible to evenly divide the rows the rest is assigned one by one to processes in order. The values in the matrix is then distributed among all processes through `MPI_Scatterv`, since there might be a different number of rows per process. Where the number of rows calculated for each row is multiplied by the side of the matrix to ensure the correct number of elements from the original matrix is sent to each process. During this computation the global index of the first row for each process is saved in an array. These values are then used at the beginning of each iteration to compute wether the current row is even or odd. Depending on this each row in the process is sorted in ascending or descending order using the C-library function `Qsort`.

As the matrix is allocated as a continuous block with each value next to each other, row wise, and each process is in possession of a certain number of rows the matrix have to be transposed before the columns can be sorted. This is done in parallel in order to avoid gathering the result on the root each iteration. To achieve this the number of matrix elements each process has to send and receive is collected from every other process and stored in two separate arrays. The amount of elements to be sent and received is based on the processor's assigned rows, as each processor shall receive the same amount of columns as it had rows in the beginning. The transposition is displayed in figure 1 [1].

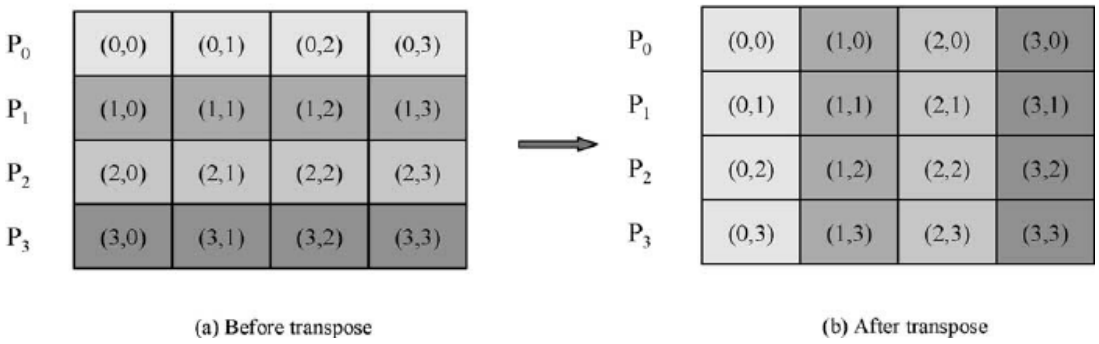

Figure 1: The process of transposing the matrix in parallel [1]

Once the counts are known, each process prepares an array by extracting the appropriate blocks of data from it's local rows. It selects the values it needs to send based on a column offset calculated through equation 1, iterating through each local row and copying the relevant elements into the send buffer. The index for each process within this buffer is stored in a separate array based on the process rank. This organized packing ensures that the data sent to each target process corresponds exactly to the rows it is assigned.

$$\text{columnoffset} = \text{local row} \cdot \text{matrix size} + \text{col\_start} + \text{receiving column} \quad (1)$$

Once each process has prepared it's sending buffer `MPI_alltoallv`, which sends a certain number of values to every process and makes sure each process receives a certain number in return, is used to exchange values. This way each process sends the values it prepared in the send buffer and receives it's new values from all other processes. Since all processes prepares and calculates the index for the new values the same way they can effectively be distributed through `MPI_alltoallv` based on each process rank. The columns, now rows, are then sorted in ascending order before the matrix is transposed back in the same way and the algorithm is repeated.

After $log_2(n)+1$ iterations the result is gathered into the original matrix through `MPI_Gatherv` and the sorted matrix can be printed.

3 Performance experiments

The correctness of the program was tested through several runs on smaller matrices with small and large integer values. The tests where performed with both even and odd number of processes. Correct sorting was assured through a check function.

To investigate the performance of the algorithm AI was used to create several nxn sized matrices with large integers. The time used was recorded through MPI_Wtime taking a time stamp after the reading of the file which was sub-

tracted with the timestamp taken after gathering the Matrix. This time was recorded for 1, 2, 4, 8 and 16 processes and the speedup was calculated through equation 2. The speedup for each matrix was plotted against the number of processes and displayed in figure 2. An Sbatch script was created to simplify the testing process where the four different marices of size 100x100, 500x500, 1000x1000, 5000x5000 and 10 000x10 000 numbers were run with two cores on cluster Snowy for the different number of processes mentioned earlier.

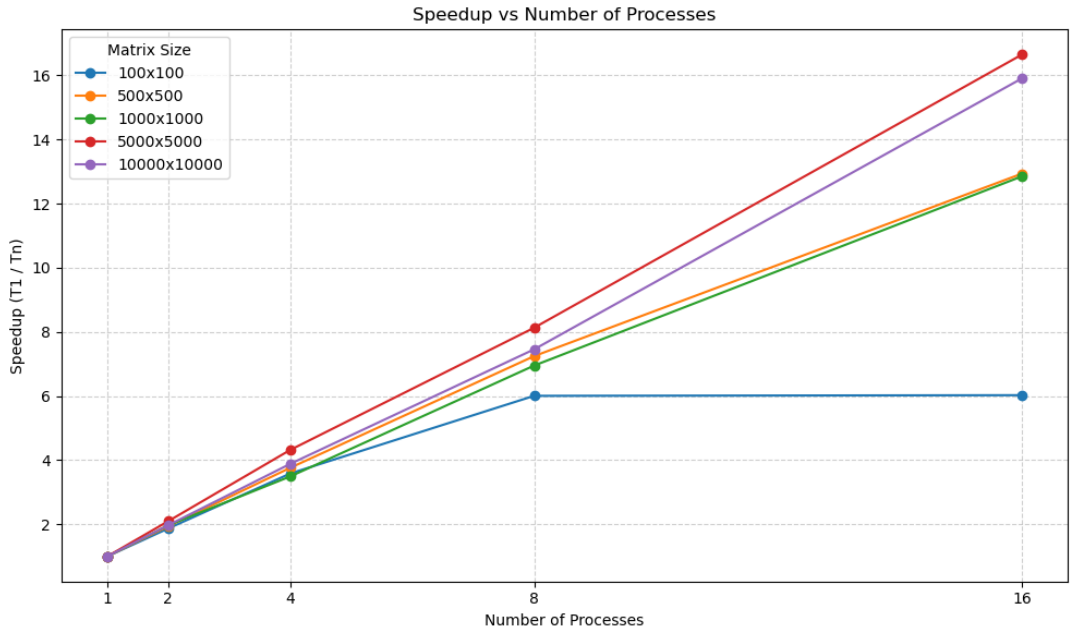

Figure 2: Plot of the speedup vs number of processes. Made from the data in table 1

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{2}$$

The efficiency was investigated through equation 3. The results are plotted in 3.

$$\text{Efficiency} = \frac{T_1}{p \cdot T_p} \tag{3}$$

To investigate the weak scaling each process was assigned an approximately constant number of rows per process. As the problem size scales quadratically the size of each matrix was calculated through the formula 4 where C represented a constant and p the number of processes. To make sure the algorithm was thorughly tested the constants used were 10, 50, 100, 500, 1000 and 1500. The speedup for each load was calculated through equation 2. The results are plotted in 4

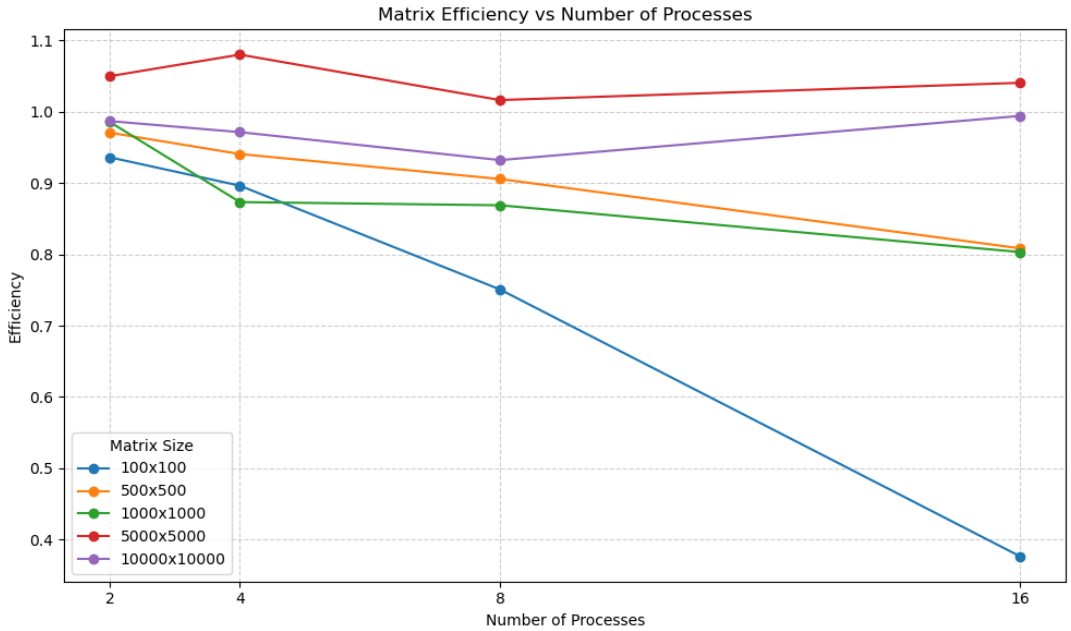

Figure 3: Plot of the efficiency vs number of processes. Made from the data in table 1

$$n(p) = C \cdot \sqrt{p} \tag{4}$$

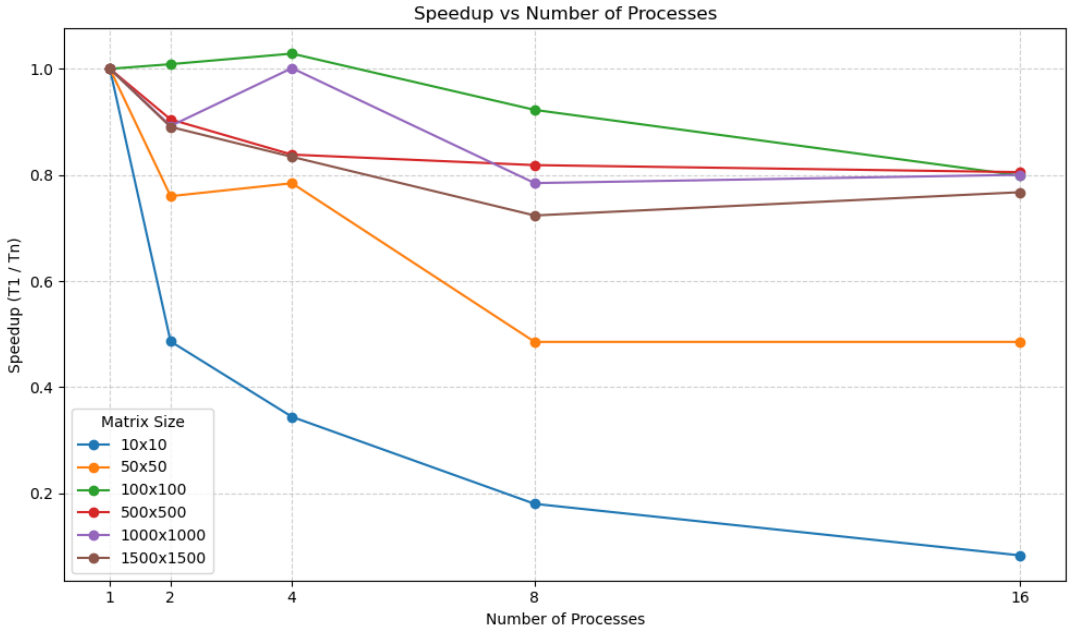

Figure 4: Plot of the speedup vs number of processes for a fixed number of rows per process. Made by the data in table 2

4 Discussion

4.1 Strong scaling

In regards to strong scaling the program is performing well. As can be seen in figure 2 the speedup increases significantly with each process that is added. For most matrices the speedup is close to ideal, that will say it follows the number of processes, which according to Ahmdals law means the parallelization overhead is very small. Since the matrix is evenly distributed between each process at the beginning of the program and each process continues to work with the same number of rows through every iteration, before the matrix is gathered at the end of the program. Each core will continue to work with the same amount of evenly distributed data throughout every run, which ensures proper load balancing. As the processes increase the speedup starts to decrease slowly where it eventually no longer follows the number of processes. Suggesting the communication overhead from especially `MPIalltoallv` slowly overtakes the parallelization benefit.

The one exception to this is the second to largest matrix, where the speedup eventually exceeds the number of processes. A speedup of this magnitude is rare and might indicate a problem where the algorithm no longer behaves as intended. This is difficult to verify for matrices of this size, however, since the algorithm has consistently performed as expected in all other tests, there is no other reason to assume incorrect behavior here. Given the matrix's large size, the substantial speedup could be explained by memory limitations. When using only one process, the entire matrix might no longer fit in the shared L3 cache, potentially causing frequent and slow memory accesses. This could result in significantly slower execution compared to when multiple processes are used to distribute the memory load. The large difference in execution time from the other matrices (when using a single process) supports this explanation.

When investigating the size of the L3 cache through the command *lscpu | grep 'cache'*, in the provided server, the size is stated as 25600K. This means the cache supports roughly 26 million bytes. A matrix of 5000x5000 that contains 25 million large integers of type int64, meaning each element in the matrix is 8 bytes would thereby not fit in the cache. Which means the previous statement could be true and a reasonable explanation to the super scalar speedup.

The largest matrix, however, do not experience this superscalar speedup, at least not for 16 processes which might contradict the previous statement. However the speedup do not decrease at all during the run, instead it increases with each added process, which might indicate this rather means matrices of this size causes memory issues within each process as well and therefore do not experience the same speedup as the second to largest matrix when only used with 16 processes. Another explanation could be that a matrix of this size means sending very large data packages between the different processes and therefore fully saturates the memory bandwidth which leads to a larger communication overhead. It could simply be that a matrix of size 5000x5000 fits perfectly for this algorithm where

it is large enough to benefit from the parallelization but small enough for memory partitioning and cache utilization to work very efficiently.

The one slowdown in the program can be seen for the smallest sized matrix from 8 to 16 processes. Where it is assumed the communication overhead becomes larger than the work available for each core. Since every process will send a few numbers to all other processes twice every iteration the bandwidth might quickly get saturated, as a lot of small packages do not use the bandwidth very well as each package contains not only the data being sent but a lot of additional data as well. This might be the reason to the slowdown as the communication through `MPI_ALLtoallv` is the main overhead present in the program. Both because it sends a lot of packages each run and has to be performed twice each iteration. The other overhead, allocation and reallocation, especially the new local matrix that has to be freed and reallocated twice each iteration when the matrix is transposed is not considered big enough to have any major impact.

4.2 Efficiency

When investigating the efficiency of the algorithm the program continues to perform well. The efficiency remains high during the run for most matrices which is displayed in 3. The exception being the smallest matrix where it decreases significantly as more processes are added. This supports the previous statement claiming the communication overhead overtakes execution time when not working with enough data. For the largest matrices the efficiency do not follow the pattern of a slight decrease with the number of processes, instead it displays a decrease followed by an increase. This might be the result of a more effective use of memory where each process uses the shared cache without evicting any other's data too often or just a major improvement from having few processes work on too large amounts of data.

4.3 Weak scaling

To investigate weak scaling, each process was assigned the same number of rows for every run according to the equation 4. From the results shown in figure 4, it is clear smaller workloads are dominated by a large communication overhead. Where the small loads speedup decreases significantly as more processes are added. Where as the larger loads has a more even performance with a small decrease in speedup followed by a small increase as the processes are increased before it remains constant between 8 and 16 processes.

For an ideal weak scaling the speedup should be roughly the same as the number of processes increase since the workload per process remains the same. This is true with this algorithm for larger workloads and more processes, indicating that this is where it reaches it's full potential. This follows the conclusion drawn from the strong scaling tests where the algorithm performed best for large workloads and more processes as well. This makes sense as the communication overhead

shrinks in comparison to the large amounts of data that is sorted.

The small speedup observed between 2 and 4 processes in some tests could be the cause of several factors, as small-scale parallel runs are sensitive to many variables, such as OS scheduling, thread placement, memory layout and so on. The most reasonable explanation is that MPI manages to handle communication more efficiently as the number of processes increases for these cases, which results in the observed slight speedup. However it is also worth noting that, since the execution times for all tests are very short, a noticeable speedup in the graph corresponds to a very small absolute difference. Which implies that the observed speedup could be caused by a minor factor.

5 Appendix

Table 1: Execution Time (seconds) for Different Matrix Sizes and Process Counts

Matrix Size	**1 Process**	**2 Processes**	**4 Processes**	**8 Processes**	**16 Processes**
100x100	0.012492	0.006674	0.003485	0.002080	0.002073
500x500	0.299077	0.154095	0.079498	0.041287	0.023124
1000x1000	1.233554	0.625935	0.353182	0.177500	0.095997
5000x5000	49.139534	23.407806	11.376486	6.044825	2.952175
10000x10000	220.046917	111.510983	56.644207	29.513026	13.836842

Loads per Process	**1 Process**	**2 Processes**	**4 Processes**	**8 Processes**	**16 Processes**
10	0.000125	0.000257	0.000363	0.000693	0.001501
50	0.001907	0.002509	0.002431	0.003928	0.003928
100	0.009264	0.009184	0.009006	0.010041	0.011596
500	0.224959	0.248659	0.268330	0.274818	0.279339
1000	1.012862	1.135166	1.011557	1.290749	1.265560
1500	2.371744	2.663335	2.843158	3.277362	3.090183

Table 2: Execution Time (seconds) vs Number of Processes for Various Loads per Process (Weak Scaling)

AI Declaration

This report was prepared by the author. Artificial intelligence tools (ChatGPT, developed by OpenAI) were used to support text editing, test data for the algorithm and result figures. The author is responsible for verifying the accuracy and validity of all content.

References

[1] R. Na'mneh, W. Pan, and S.-M. Yoo, "Efficient adaptive algorithms for transposing small and large matrices on symmetric multiprocessors," *Informatica, Lith. Acad. Sci.*, vol. 17, pp. 535–550, Jan. 2006. DOI: 10.15388/Informatica.2006.153.