UPPSALA
UNIVERSITET

Individual Project

Aikaterini Manousidou
Parallel and Distributed Programming
Uppsala University
VT 2021

June 10, 2021

Table of contents

1 Introduction

One of the most important computational problems in the field of computer science is being able to sort a sequence of data efficiently. It is often essential that the sorting implementation results in high performance, which usually means that the implementation should acquire a level of parallelism. There are several well-known parallel sorting algorithms such as Hyper Quick sort or Merge sort, however, this project focuses on the Shear sort algorithm.[1]

1.1 The Shear sort algorithm

Shear sort or snake sort is an algorithm for sorting two dimensional arrays in a snake-like pattern, as seen in figure 1. For a matrix of size $N \times N$ and $d = log(N)$, it takes $d+1$ iterations to sort the rows correctly and d iterations to sort the columns correctly. The algorithm consist of two phases, which are described bellow.[2]

- **Phase 1** Sort the values of the even numbered rows in an ascending order and the values of the odd numbered rows in a descending order. This phase is repeated $d+1$ times.
- **Phase 2** Sort the values of the columns in an ascending order. This phase is repeated d times.

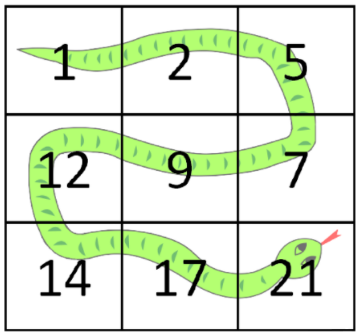


Figure 1: Two-dimensional array sorted in a snake-like pattern.

1.2 The Parallel Shear sort algorithm

Given that there are p processes available, the first step of the parallel Shear sort algorithm is to divide the rows of the matrix into p equal parts and distribute them among the processes. This implies that an assumption that the size of the matrix, N is divisible with the number of processes, p. The indices of the rows send to each process are stored in each process. Thereafter, each process starts by sorting their rows in ascending or descending order depending on the original row indices. The processes then need to exchange values so that each process assembles the columns corresponding to their stored indices and then the values of columns are sorted in ascending order. The sorting of the rows is repeated for $d+1$ iterations and the sorting of the columns is repeated for d iterations, where $d = log(N)$. Finally, the matrix is assembled together once again and is now sorted in a snake-like pattern.

1.3 Performance evaluation

In order to evaluate the performance of the program, two types of experiments were performed. The first experiment involved a strong scalability analysis, meaning that for a constant problem size N the number of processes is increased. For the evaluation of the strong scalability, the speed-up, defined as the ratio between the serial and the parallel run-times of the algorithm, was calculated. The definition of the speed-up can be found in equation 1, where $S(N,p)$ is the speed-up, $T_{serial}(N)$ and $T_{parallel}(N,p)$ are the serial and parallel run-times for a problem size N and p number of processes.[3]

$$S(N,p) = \frac{T_{serial}(N)}{T_{parallel}(N,p)} \tag{1}$$

The second experiment involved a weak scalability analysis, entailing that both the problem size and number of processes are increased in such a manner that the problem size per process is kept constant.

Moreover, the efficiency of the strong and weak scaling experiments was calculated, in order to further be able to analyze the performance of the sorting algorithm. The efficiency is defined in equation 2. The efficiency should increase with an increasing N and decrease with an increasing p, thus remaining constant when simultaneous increasing N and p.[3]

$$E(N,p) = \frac{T(N,1)}{p \cdot T(N,p)} \tag{2}$$

2 Problem description

The purpose of this project is to implement a parallel Shear sort algorithm in C with the help of Message Passing Interface (MPI) and evaluate its performance through strong and weak scaling experiments. Access to a super-computer was available for the performance experiments.

3 Solution approach

The implementation of the algorithm takes one input argument, which specifies the matrix size N. The first part of the program initializes the MPI, reads in the input and checks the assumption that the size of the matrix is divisible with the number of processes. If the assumption does not hold, the program terminates. If the assumption holds, the process of rank 0 allocates memory for the matrix and fills it with uniformly random numbers, with the help of the `drand48()` function.

The size of the matrix is then divided by the number of processes, in order to determine how many rows each process will be responsible for. Memory is then allocates in each process for storing the rows of the matrix and for storing the index corresponding to each row. At this point, the timer is started. The rows of the matrix are then distributed to each process with the help of the `MPI_Scatter()` function.[4]

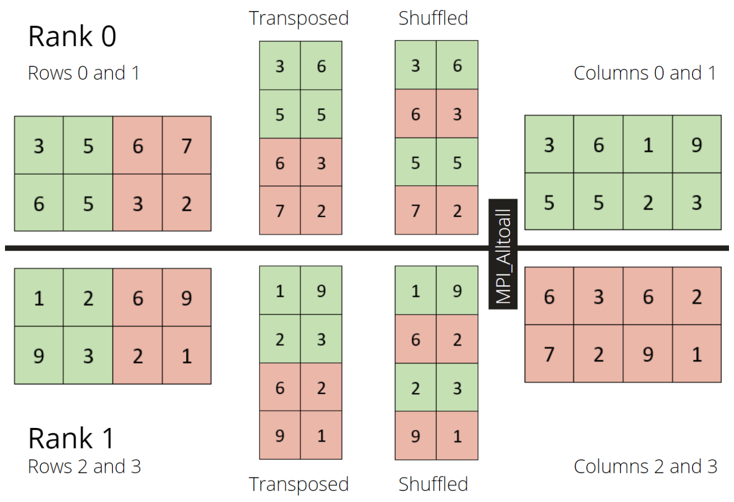


Figure 2: Schematic picture of how the process of the adaptation of the rows into columns is achieved.

The sorting sequence starts with each process sorting their rows with respect to the original index by calling the help-function called `sort()`. This help function takes five inputs which involve the pointer to the local matrix, the size of the matrix, the row index of the row that needs to be sorted, the number of iterations that need to be performed and the type of sorting, ascending or descending order. The next step is to sort the columns of the matrix. For that to happen, each process starts by transposing their local matrix and then rearrange the rows of the transposed matrix in such a way that the function `MPI_Alltoall()` can be utilized, as seen in figure 2. This function is a combination of `MPI_Scatter()` and `MPI_Gather()` and enables each process to scatter parts of its version of a buffer across all processes while at the same time gathering parts of the equivalent buffers from all the other processes. In other words, each process is able to assemble the corresponding columns to the row indices that it stored initially. Afterwards, each process sorts the column elements in an ascending order by calling the `sort()` help-function and repeats the method of transposing and rearranging the rows in order to use the `MPI_Alltoall()` so that each process can, once again, assemble the rows of the matrix. This procedure is repeated for d iterations, where d is defined as `ceil(log2(N))`.[5]

The process of rank 0 gathers the rows from each one of the processes with the help of the `MPI_Gather()` function and the timer is stopped. Furthermore, the longest execution time among the processes is determined with the help of the `MPI_Reduce()` function. The process of rank 0 checks if the matrix has been sorted correctly and prints out a message indicating if the sorting was successful or not. The process of rank 0 also prints the longest execution time. Lastly, the all allocated memory is freed and the MPI is finalized. The program then terminates.

4 Experiments

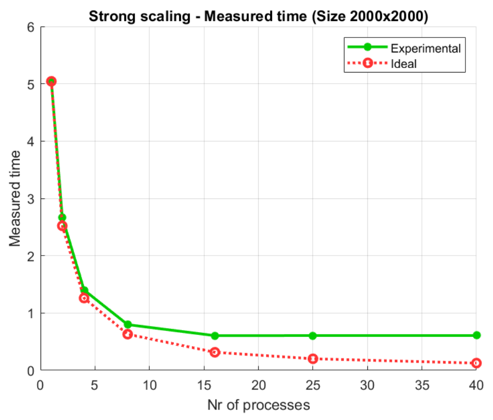
As previously mentioned in section 2 strong and weak scaling experiments were to be carried out in order to determine the performance of the Shear sort implementation. These experiments were performed on the UPPMAX system Snowy, enabling a total of 40 cores to be used in this project.

For the strong scaling analysis a problem size of 2000 was chosen and the number of processes tested were 1, 2, 4, 8, 16, 25 and 40. The measured execution times can be seen in table 1 along with the speed-up which was calculated with the help of equation 1. In figure 3, plots over the measured times and the speed-up against the number of processes can be observed.

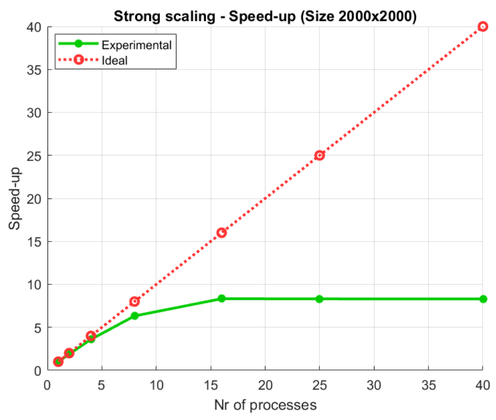
Table 1: Table over the number of processes, the Shear sorting timings acquired during the strong scaling analysis and the calculated speed-up. All values have been rounded to five significant digits.

Processes	Measured time (s)	Calculated Speed-up
1	5.0428	1.0000
2	2.6716	1.8875
4	1.4007	3.6003
8	0.7974	6.3238
16	0.6055	8.3284
25	0.6066	8.3136
40	0.0.6074	8.3027

From figure 3a, it is obvious that the experimental measured times follow the trend of the ideal execution times. However, when studying figure 3b, it can be determined that the experimental speed-up follows the ideal speed-up but seems to reach a maximum of approximately 8.3 with 16 or more cores. This suggests that the implementation acquires some strong scalability and that the optimal number of processes for a problem size of 2000 seems to be 16.



(a) Measured times



(b) Speed-up

Figure 3: Plots over the strong scaling analysis, including the measured execution times and the calculated speed-up.

For the weak scaling analysis a problem size of 2000 per process was chosen. The tested problem sizes were then 2000, 4000, 8000, 16000, 24000 and 32000 for 1, 2, 4, 8, 12 and 16 processes. The measured execution times can be seen in table 2 along with their equivalent total problem size. In figure 4, a plot over the weak scaling measured times against the number of processes can be seen.

Table 2: Table over the number of processes and the Shear sorting timings acquired during the weak scaling analysis. All values have been rounded to five significant digits.

Processes	Problem size	Measured time (s)
1	2000	5.04813000000000
2	4000	12.6159250000000
4	8000	30.0055970000000
8	16000	73.3657930000000
12	24000	137.977380000000
16	32000	721.300811000000

By observing table 2 and figure 4, it is rather clear that the execution time increases while the problem size and the number of processes increases. It was also quite unexpected how much the execution time increased when raising the problem size from 24000 to 32000 and the number of processes from 12 to 16. Consequently, it can be determined that the results of the weak scaling imply that the implementation does not obtain weak scalability and is not recommended for substancial problem sizes.

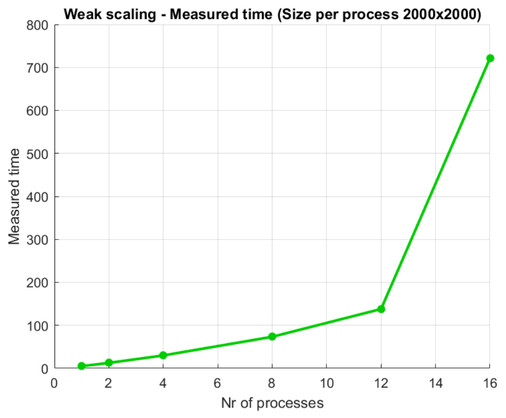
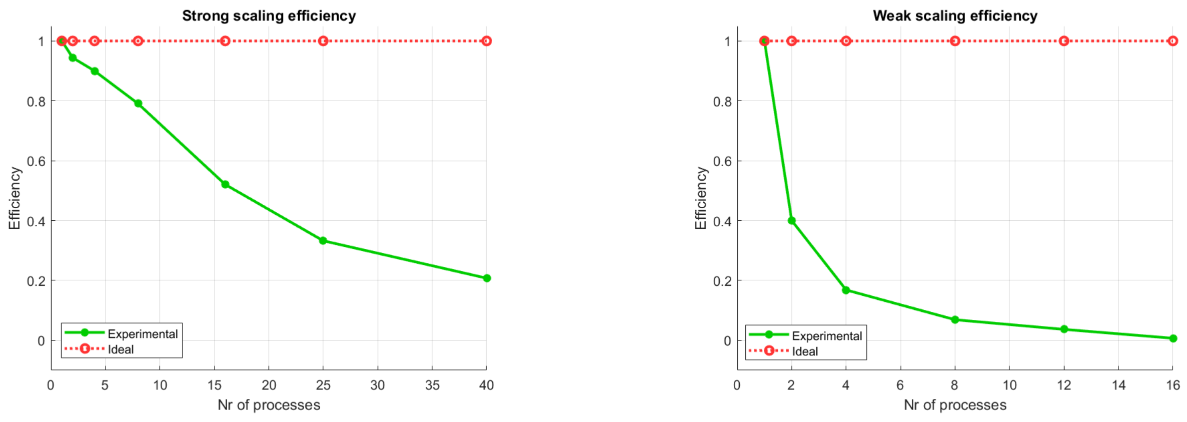


Figure 4: Plot over the weak scaling execution times against the number of processes.

The efficiency of the strong and weak scalability experiments was calculated with the help of equation 2 and plots of the results can be found in figure 5. Figure 5a indicates that the efficiency of the implementation is quite high in the beginning but it seems to decrease linearly and for 16 processes it has reached 50%. This seems to verify the conclusion drawn from figure 3, that for a problem size of 2000 the algorithm seems to demonstrate speed-up close to the ideal until for up to 16 processes. On the other hand, from figure 5b it is quite clear that the implementation does not display any signs of being weakly scalable.



(a) Strong scaling efficiency

(b) Weak scaling efficiency

Figure 5: Plots over the calculated efficiency of the strong scaling and weak scaling experiments.

5 Conclusions

The aim of this project was to implement and study the performance of the Shear sort algorithm. The program utilized MPI which enabled for the algorithm to be parallelized. The performance of the implementation was then analyzed with the help of strong and weak scalability experiments.

From the results of the strong scaling experiments, it was concluded that for a problem size of 2000 the algorithm demonstrated close to ideal speed-up for up to approximately 16 processes. To the contrary, the weak scaling experiment results suggested that the implementation does not display weak scalability and is therefore not recommended for substantial problem sizes.

6 Efforts of improvement

6.1 Code Modifications

By further examining the implementation, it was obvious that the main for-loop for the sorting of the rows and the columns was the part of the code that affected the performance the most. In order to optimize the code, the help-function `sort()` was changed to the `qsort()` function given in the standard library for sorting the even numbered rows and columns. The `qsort()` function takes four arguments, the pointer to the first element of the array to be sorted, the number of elements in the given array, the size of each element and a function that compares two elements.

The code was also altered in order to be able to read in the matrix to be sorted from an input file and to write the sorted matrix in an output file.

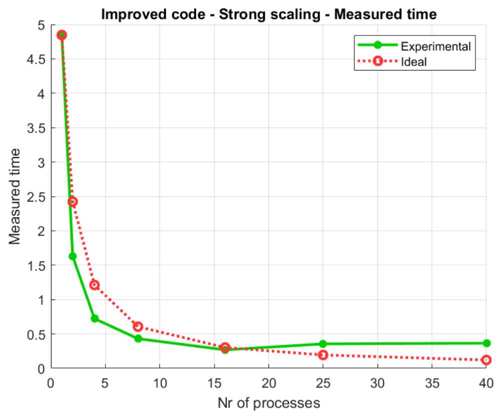
6.2 Results

Once again, for the strong scaling analysis a problem size of 2000 was chosen and the number of processes tested were 1,2, 4, 8, 16, 25 and 40. The measured execution times can be seen in table 3 along with the speed-up which was calculated with the help of equation 1. In figure 6, plots over the measured times and the speed-up against the number of processes can be observed.

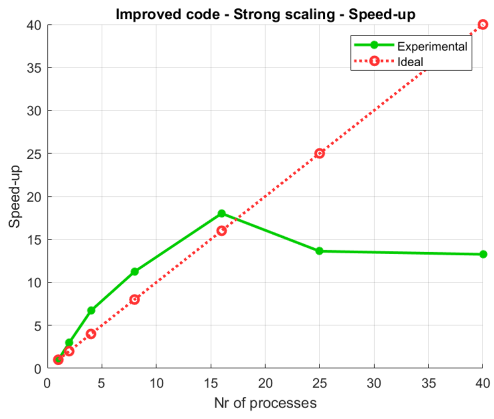
Table 3: Table over the number of processes, the Shear sorting timings acquired during the strong scaling analysis and the calculated speed-up. All values have been rounded to five significant digits.

Processes	Measured time (s)	Calculated Speed-up
1	4.8456	1.0000
2	1.6293	2.9741
4	0.7222	6.7091
8	0.4308	11.2474
16	0.2689	18.0170
25	0.3558	13.6210
40	0.3654	13.2615

By observing table 3 and figure 6, it can be determined that the measured times have slightly decreased and that speed-up has increased tremendously. However, the speed-up seems to flatten out after the use of 16 cores. Consequently, it can be concluded that the results of the strong scaling analysis imply that the improved code acquired strong scalability and the optimal number of processes for a problem size of 2000 seems, once again, to be 16, however with more than double the speed-up.



(a) Measured times



(b) Speed-up

Figure 6: Plots over the strong scaling analysis, including the measured execution times and the calculated speed-up.

For the weak scaling analysis a problem size of 2000 per process was once again chosen. The tested problem sizes were then 2000, 4000, 8000, 16000 and 32000 for 1, 2, 4, 8 and 16 processes. The measured execution times can be seen in table 4 along with their equivalent total problem size. In figure 7, a plot over the weak scaling measured times against the number of processes can be seen.

Table 4: Table over the number of processes and the Shear sorting timings acquired during the weak scaling analysis. All values have been rounded to five significant digits.

Processes	Problem size	Measured time (s)
1	2000	4.8516
2	4000	7.8628
4	8000	15.7123
8	16000	35.9091
16	32000	115.9002

After examining table 4 and figure 7, it is obvious that the execution times for the different problem sizes have decreased in comparison to the ones in table 2. On the other hand, the relation between the number of processes and the problem size appear to have a linear behaviour in comparison to the almost exponential growth of the measured time in figure 4. Thus, it can be determined that the improved code displays more weak scalability than the initial implementation.

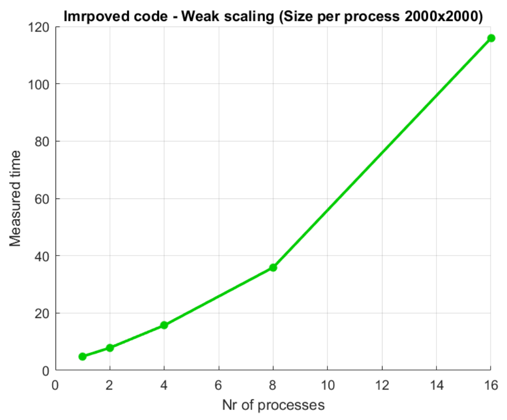
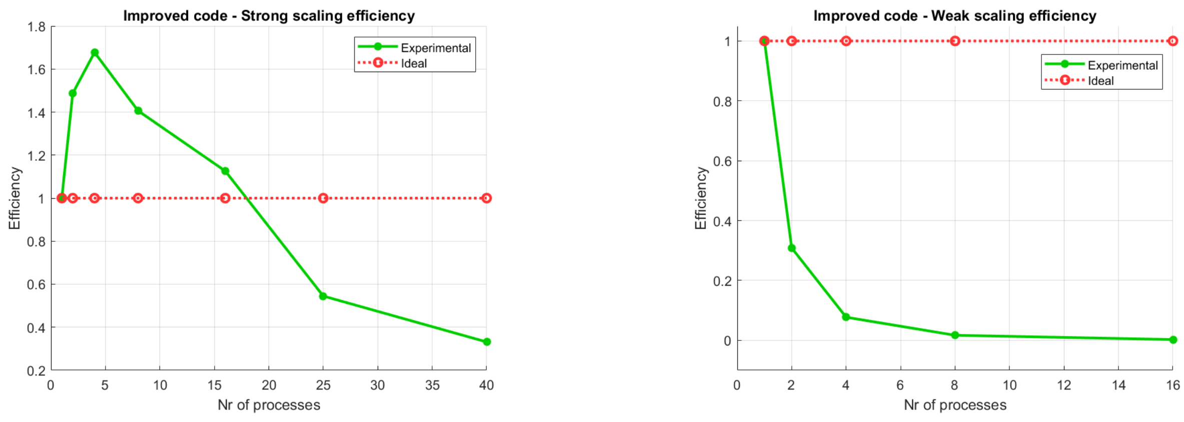


Figure 7: Plot over the weak scaling execution times against the number of processes.

Last, the efficiency of the strong and weak scalability experiments of the improved implementation were calculated with equation 2 and the results can be seen in figure 8. In figure 8a it can be seen that the strong efficiency of the code is above one for 2, 4, 8 and 16 processes. This suggests that the algorithm performs much better than expected. The efficiency then decreases to a slightly under 0.4, which it not bad at all. In figure 8b, however, the weak efficiency seems to retain its previous form, implying that the implementation does not display signs of being weakly scalable.



(a) Strong scaling efficiency

(b) Weak scaling efficiency

Figure 8: Plots over the calculated efficiency of the strong scaling and weak scaling experiments.

At last, it can be concluded that for a problem size of 2000 the improved algorithm displays more than twice the initial speed-up and that the optimal amount of processes seems to be approximately 16. However, when it comes to the weak scaling analysis, the improved implementation does not seem to performed much better than the initial implementation and is, yet again, not recommended for larger problem sizes.

References

[1] Selim G. Akl, *Parallel Sorting Algorithms.* Academic Press Inc., 1985.

[2] Maya Neytcheva, "Parallel and Distributed Programming Lecture 6," 2021.

[3] Maya Neytcheva, "Parallel and Distributed Programming Lecture 1," 2021.

[4] Open MPI Project, "MPI Scatter Documentation."

[5] Open MPI Project, "MPI All to all Documentation."