INDIVIDUAL PROJECT: SHEARSORT

Parallel and Distributed Programming

Rafael Rodriguez Velasco

VT 2024

Contents

1 Introduction

ShearSort is a 2D grid sorting algorithm, particularly designed for parallel computing. Given a matrix of size $n \times n$ (or a sequence of numbers that can be mapped into such a matrix) the algorithm sorts each row of the matrix independently, in alternative directions. Then, it sorts each column of the matrix in the same way. After ceil($\log 2(n)$) + 1 steps the matrix is fully sorted in a snake-like pattern, visualized in Fig. 1 [1].

This report aims to describe my implementation of the ShearSort algorithm, parallelized using MPI in C. Along with a description of the serial and parallel algorithms used, scaling tests are presented to evaluate the performance of the parallel implementation.

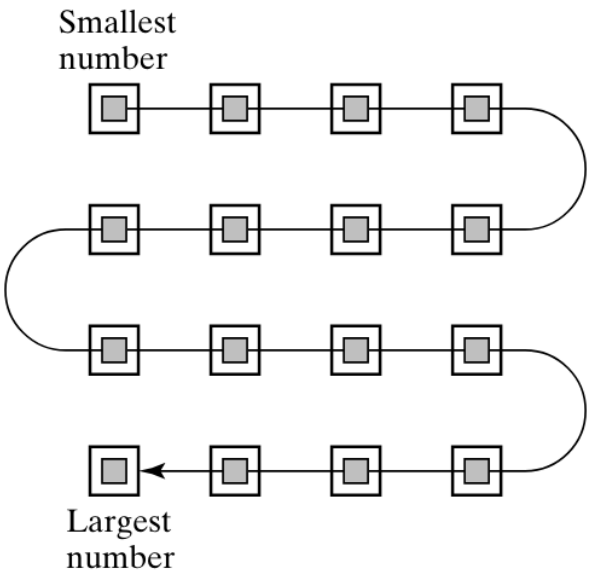

Figure 1: Illustration of the snake-like pattern after ShearSort algorithm (source: [2])

2 Algorithms

Both serial and parallel implementations require an auxiliary sorting function to sort the rows and columns of the matrix. My choice of sorting algorithm is the quicksort algorithm, which is implemented in the `quicksort` function, included in `stdlib.h`. This election is based on its convenient implementation and its efficient time complexity of $O(n \log n)$, ideal for big datasets.

All algorithms described in this section are executed with the same command line arguments, following the format `./snake input_file n output`, where

- `input_file` is the path to the file containing the matrix to be sorted. The program expects a .txt file with a sequence of numbers separated by spaces and new lines, representing the matrix row-wise.
- `n` is the size of the matrix, i.e., the number of rows and/or columns.

- `output` is an integer that determines the output of the program, allowing it to switch between simple debugging or performance testing modes. The possible values are:
  - 0: Only prints the execution time of the algorithm to the console.
  - 1: Reads the specified matrix from the input file, sorts it, prints the output matrix to the console and checks if the matrix is sorted correctly.
  - 2: Ignores the input file and generates a random matrix of size $n \times n$, sorts it, prints the output matrix to the console and checks if the matrix is sorted correctly.

A Python script `matrixgen.py` is provided to generate random matrices and write them to a file, which can be used as input for the program.

2.1 Serial

The serial algorithm is rather simple, yet establishes the main strategy for the parallel algorithms. It handles the main matrix, denoted by A, as a 2D array and sorts each row and column following the steps described in algorithm 1.

Algorithm 1: Serial ShearSort algorithm

```
Calculate the number of steps d = ceil(log 2(n))
for l = 0 to d do
    for each row do
        Sort in ascending order with qsort
    end
    for each row do
        Sort in descending order with qsort
    end
    if l ≤ d then
        Transpose A
        for each row do
            Sort in ascending order with qsort
        end
        Transpose A back
    end
end
```

Note that the algorithm needs to transpose the matrix twice in each column sorting step. This is done since quicksort needs to sort contiguous memory blocks, and the matrix is stored row-wise. The transposition is done with a straightforward function that swaps the elements of the matrix. Figure 2 illustrates these steps.

There are alternative ways to sort the columns avoiding transposing the matrix, such as using an auxiliary array/matrix to store the columns. However, the parallel implementation relies on the transposition, so it is kept in the serial algorithm for consistency.

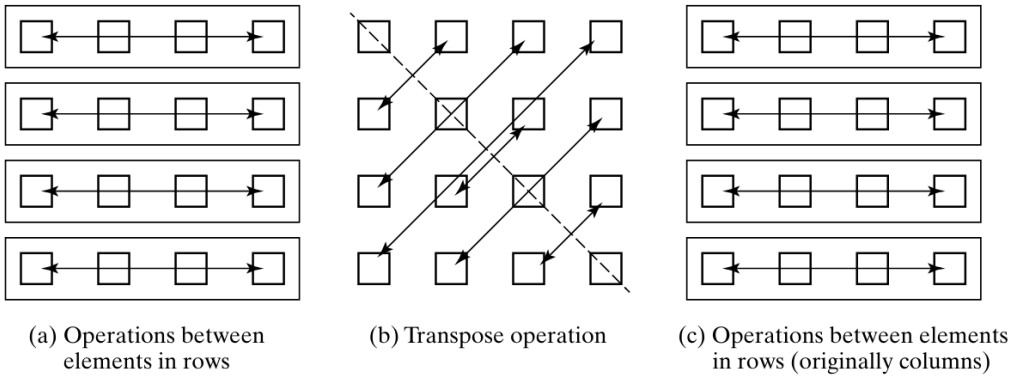

Figure 2: Illustration of the steps of the ShearSort (source: [2])

2.2 Parallel

To carry out the matrix sorting, all algorithms assume that the matrix is squared $n \times n$ and, furthermore, that the number of rows and/or columns is divisible by the number of processes p.

The parallel algorithm is based on the serial algorithm, but it distributes the rows of the matrix among the processes. The main idea is to sort the rows and columns of the matrix in parallel, while also parallelizing the transposition of the matrix. In addition, the algorithm must handle the matrices as 1D arrays, namely flat matrices, to facilitate the distribution of the data among the processes.

There are, however, different approaches to implementing the aforementioned strategy. The following subsections describe the two algorithms that outperformed the others I implemented in the scaling tests, as well as some mentions and comments on the less efficient algorithms.

2.2.1 Ordered algorithm

This algorithm is based on the idea of distributing the matrix into p submatrices, each of size $n/p \times n$. The processes are then assigned to sort the rows of each submatrix, and are transposed locally to sort the columns. The pseudocode for the algorithm is:

Algorithm 2: Parallel ordered ShearSort algorithm

```
1  Scatter the flat matrix A among the processes, which receive a local submatrix of size
     n/p × n
2  Calculate the number of steps d = ceil(log 2(n))
3  for l = 0 to d do
4      if n/p is even then
5          for each row in the local submatrix do
6              Sort even rows in ascending order with qsort
7              Sort odd rows in descending order with qsort
8          end
9      end
10     else
11         if process rank is even then
12             for each row in the local submatrix do
13                 Sort even rows in ascending order with qsort
14                 Sort odd rows in descending order with qsort
15             end
16         end
17         else
18             for each row in the local submatrix do
19                 Sort even rows in descending order with qsort
20                 Sort odd rows in ascending order with qsort
21             end
22         end
23     end
24     if l ≤ d then
25         Transpose the local submatrix
26         Distribute the transposed submatrices among the processes.
27         for each row do
28             Sort in ascending order with qsort
29         end
30         Transpose locally again
31         Distribute the transposed submatrices again
32     end
33 end
34 Gather the sorted submatrices into the global matrix A
```

Now the implementation is more complex, not only because of the scattering and gathering of data, but also since the parity of n/p must be taken into account, and the transposition of the

submatrices is not trivial.

To achieve local transposition, a new array is created to store the transposed submatrix. The transposition is done by iterating over the rows and columns of the submatrix, and storing the elements in the new array in the opposite order. Then, the new $n \times n/p$ array must be divided into chunks of $n/p \times n/p$ elements distributed to every processor so that each processor can have a submatrix of the original size $n/p \times n$ with the transposed elements. This process is represented in Fig. 3 This is achieved by the auxiliary function `TransposedToNormal`, which consists of a series of `MPI_Scatterv` calls to distribute the transposed submatrices among the processes.

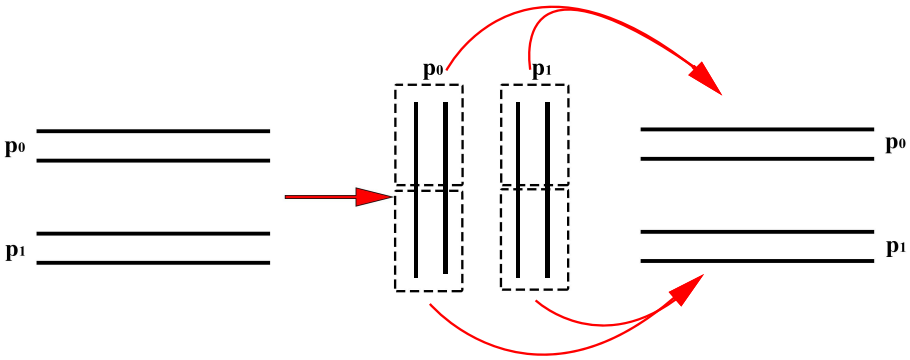

Figure 3: Local transposition process for an example matrix of size 4×4 and two processes. The intermediate matrix is stored in a different array.

```
void TransposedToNormal(int* localTranspose, int* localA, int rows_per_proc,
    int n, int size) {
    int counts[size];
    int displs[size];

    for (int p = 0; p < size; p++) {
        for (int i = 0; i < rows_per_proc; i++) {
            for (int j = 0; j < size; j++) {
                counts[j] = rows_per_proc;
                displs[j] = rows_per_proc*j * rows_per_proc + i*rows_per_proc;
            }

            MPI_Scatterv(localTranspose, counts, displs, MPI_INT, localA + i*n
                + p*rows_per_proc, n * rows_per_proc, MPI_INT, p,
                MPI_COMM_WORLD);
        }
    }
}
```

2.2.2 Splitting algorithm

The splitting algorithm differs from the ordered algorithm in the way the matrix is distributed among the processes. Instead of dividing the matrix into submatrices respecting the row order (hence the name), the matrix is now split into two groups: even processors receive the even rows of the matrix, and odd processors receive the odd rows. All the processors in the same group sort all their rows in the same way and then the transposition is done similarly to the ordered algorithm. This algorithm thus requires p to be even, and does not handle the case when n/p is odd.

The pseudocode for the algorithm is shown in algorithm 3.

Algorithm 3: Parallel splitting ShearSort algorithm

```
1  Assign each processor a color and a pair based on its rank
2  Scatter the flat matrix A among the processes, which receive a local submatrix of size
     n/p × n
3  Calculate the number of steps d = ceil(log 2(n))
4  Use the function NormalToOddEven to split the rows among processes
5  Calculate the number of steps d = ceil(log 2(n))
6  for l = 0 to d do
7      if p is even then
8          for each row in the local submatrix do
9              Sort even rows in ascending order with qsort
10         end
11     end
12     else
13         for each row in the local submatrix do
14             Sort odd rows in descending order with qsort
15         end
16     end
17     Distribute the submatrices to order the rows with OddEvenToNormal
18     if l ≤ d then
19         Transpose the local submatrix
20         Distribute the transposed submatrices among the processes.
21         for each row do
22             Sort in ascending order with qsort
23         end
24         Transpose locally again
25         Distribute the transposed submatrices again, this time splitting by odd and
             even rows
26     end
27 end
28 Gather the sorted submatrices into the global matrix A
```

This algorithm involves more sophistication in the data distribution, and requires the use of

an auxiliary matrix B for the ordering of the rows. However, it improves the locality of the row sorting. The function `NormalToOddEven` works as follows: The already existing rows in the processor are copied to the auxiliary matrix B in the order of the processor's color. Then, n/p send-receives are performed between the processors to exchange the rows in the correct order. The function `OddEvenToNormal` is similar to `TransposedToNormal`, but switching the order of the destination and source arrays.

```
void OddEvenToNormal(int* localA, int* localB, int rows_per_proc, int n, int
↪  size,int pair, int rank) {
    for (int i = 0; i < rows_per_proc / 2; i++) {
        if (rank % 2 == 0) {
            // Copy to even rows of loalB
            memcpy(localB + 2 * i * n, localA + i * n, n * sizeof(int));
        } else {
            // Copy to odd rows of localB
            memcpy(localB + (2 * i + 1) * n, localA + (i + rows_per_proc/2) *
            ↪  n, n * sizeof(int));
        }
    }

    for (int i = 0; i < rows_per_proc / 2; i++) {
        if (rank % 2 == 0) {
            // Send to pair
            MPI_Send(localA + (i + rows_per_proc/2) * n, n, MPI_INT, pair, 0,
            ↪  MPI_COMM_WORLD);
            // Receive from pair
            MPI_Recv(localB + (2 * i + 1) * n, n, MPI_INT, pair, 0,
            ↪  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        } else {
            // Receive from pair
            MPI_Recv(localB + 2 * i * n, n, MPI_INT, pair, 0, MPI_COMM_WORLD,
            ↪  MPI_STATUS_IGNORE);
            // Send to pair
            MPI_Send(localA + i * n, n, MPI_INT, pair, 0, MPI_COMM_WORLD);
        }
    }
}
```

The function `TransposedToOddEven` is analogous to `TransposedToNormal`, but the calculation of the displacements is way less trivial.

```
void TransposedToOddEven(int* localTranspose, int* localA, int rows_per_proc,
    int n, int size, int* counts, int* displs) {
    for (int p = 0; p < size; p++) {
        for (int i = 0; i < rows_per_proc; i++) {
            for (int j = 0; j < size; j++) {
                counts[j] = rows_per_proc;
                if (j % 2 == 0)
                    displs[j] = (j * rows_per_proc + i*2)*rows_per_proc;
                else
                    displs[j] = displs[j - 1] + rows_per_proc;
            }

            MPI_Scatterv(localTranspose, counts, displs, MPI_INT, localA + i*n
                + p*rows_per_proc, n * rows_per_proc, MPI_INT, p,
                MPI_COMM_WORLD);
        }
    }
}
```

2.2.3 Optimizing the transposition algorithm

The transposition methods used in the ordered and splitting algorithms are not optimal, since they require several scatters in a loop to distribute the data. An alternative way of transposing the matrix is to use the MPI function `MPI_Alltoall` to distribute the rows of the matrix among the processes. This function is designed to both scatter and gather data for each process in a single call, thus reducing the communication overhead significatively. This way, there is no need to use the functions `TransposedToNormal` and `TransposedToOddEven`. However, it is not straightforward to use this function in the algorithm, since it is designed to distribute data of the same size as the number of processes.

To overcome this limitation, one may use customized datatypes in MPI to distribute the rows of the matrix among the processes. Custom datatypes allow the user, among other things, to define a new datatype that stores non contiguous data in memory. This way, each submatrix can be subdivided even more into chunks of size $n/p \times n/p$, and then distributed among the processes using `MPI_Alltoall`. This way, each process contains a submatrix of size $n/p \times n$ divided into p chunks, allowing the algorithm to benefit from the `MPI_Alltoall` function. The local transposition needs to be done within each chunk, so it differs slightly from the previous local transposition function.

The details of the data type creation are listed below:

```
MPI_Datatype unaligned_chunk_type, chunk_type;
```

```
2 MPI_Type_vector(rows_per_proc, rows_per_proc, n , MPI_INT,
  ↪ &unaligned_chunk_type);
3 MPI_Type_create_resized(unaligned_chunk_type, 0, rows_per_proc * sizeof(int),
  ↪ &chunk_type);
4 MPI_Type_free(&unaligned_chunk_type);
5 MPI_Type_commit(&chunk_type);
```

`MPI_Type_vector` creates a vector datatype that represents a strided block of data, in this case `rows_per_proc` (p/n) blocks of size `rows_per_proc` and separated by n elements. The `MPI_Type_create_resized` function is used to modify the extent of the datatype, that is, it sets each chunk contiguous in memory. Finally, the datatype is committed with `MPI_Type_commit`.

2.2.4 Less efficient algorithms

Here I will briefly describe other algorithms or changes to the functions that I implemented, but that did not perform as well as the ordered and splitting algorithms.

- **Naive algorithm**: This first version of the algorithm scattered and gathered the matrix in every sorting step of the algorithm. This is very inefficient, since the communication overhead is too high, and the algorithm is not scalable.
- **Not parallelizing transposition**: Since the transposition of the matrix is a costly operation, I tried to check if the algorithm was benefitting from parallelizing this operation. The results showed that the algorithm was indeed faster when the transposition was parallelized, so this was kept in the final version of the algorithm.
- **Non-blocking communication**: Since there are some parts of the algorithm where smaller parts of submatrices are distributed in a loop, I tried to use non-blocking communication to distribute the data, so that each call of the loop wouldn't have to wait for the previous one to finish. However, the results showed that this did not improve the performance of the algorithm, and in fact, made it slower. In general, these algorithms do not benefit from non-blocking communication, since there are no heavy computations that can be done while the communication is being performed.
- **Different ways of splitting the matrix**: For the splitting method, I tried different ways of splitting the matrix among the processes, such as using a series of scatterv calls with some complex displacements. On top of this, I also tried to sort the rows the moment they were received by the processes, but this did not improve the performance of the algorithm. An alternative way of using sends and receives was also tested, where the original matrix was scattered and then the rows were sent to the correct processors, but in a way the columns were not sorted correctly, thus requiring an extra step of transposition and sorting. This was slower than the final version of the algorithm.

- **Dedicating processors to sorting**: One alternative way of implementing ShearSort, is using some processors to work on submatrices and some others to sort the numbers in parallel. Nonetheless, since the greatest problem of my version of ShearSort is the matrix transposition, this did not improve the performance.

3 Experiments and results

The performance of the parallel algorithms was evaluated using the `MPI_Wtime` function to measure the execution time of the algorithm and `MPI_Reduce` to get the maximum execution time across the processes (since it represents the real time taken to complete the execution). These time measurements ignore the matrix reading or the matrix printing and checking, depending on the in-line input. All the experiments were performed in Uppmax, on cluster Snowy on a single node with up to 16 cores.

Two scaling experiments are carried out: Hard and Weak scaling. The hard test measures how much the performance improves for a fixed number of rows/columns n and an increasing number of processors p. The weak test, on the other hand, measures the effect of increasing the number of processors while maintaining the same workload per processor, that is, keeping the quotient n/p constant.

For the hard scaling experiment, we will represent both the speedup and efficiency, defined as follows

$$S_h = \frac{T_s(n)}{T_p(n,p)} \qquad E_h = \frac{S}{p} = \frac{T_s(n)}{p \cdot T_p(n,p)}\,, \tag{1}$$

while for the weak scaling, we define the efficiency as

$$E_w = \frac{T_{p=1}(n)}{T_p(n,p)}\,, \tag{2}$$

where T_s and T_p are the execution times of the serial and parallel codes [3].

Note that, for the weak scaling, the time $T_{p=1}(n)$ cannot be measured for the splitting algorithm, since it requires at least two processors. The value $T_{p=2}(n)$ will be used instead for that case.

Since the figures below don't show great differences between ordered and splitting algorithms, the alltoall transposition is only shown for the ordered algorithm, as it handles odd numbers of rows per processors.

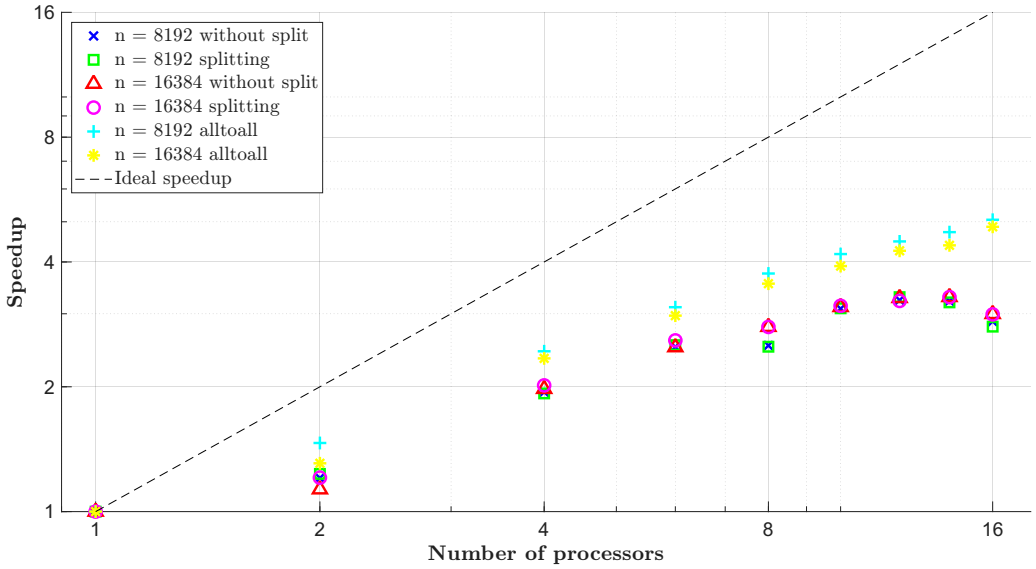

Figure 4: Hard scaling speedup for matrices of size 8192×8192 and 16384×16384 and both ordered and splitting methods

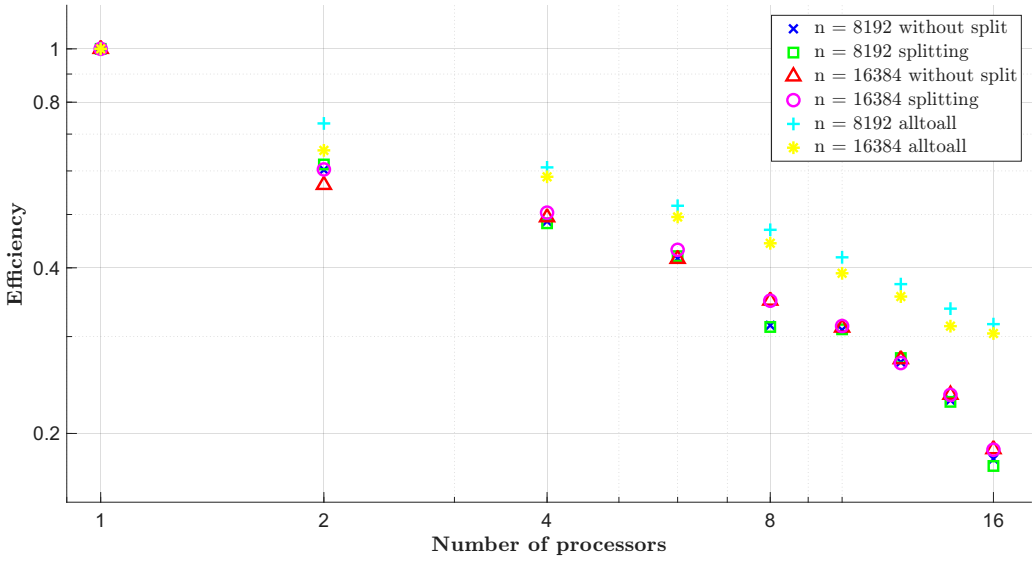

Figure 5: Hard scaling efficiency for matrices of size 8192×8192 and 16384×16384 and both ordered and splitting methods

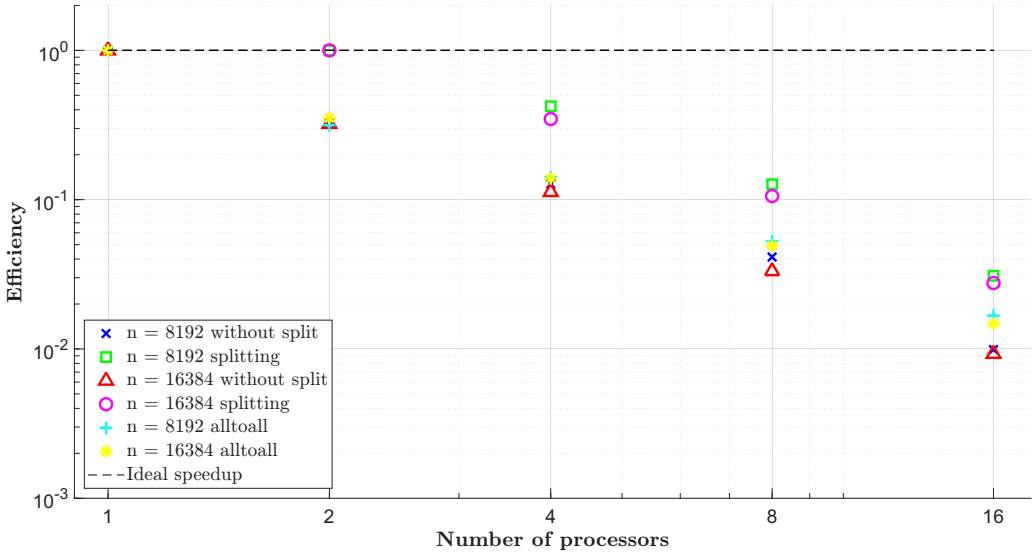

Figure 6: Weak scaling efficiency for matrices of size 8192×8192 and 16384×16384 and both ordered and splitting methods

Number of Processors	**Execution Time (seconds)**	
	n = 8192	**n = 16384**
1	73.335023	332.689703
2	60.845390	293.906082
4	37.739785	168.271384
6	29.379276	133.497028
8	29.153583	119.240967
10	23.771888	106.793505
12	22.667926	101.569868
14	22.780180	101.025274
16	25.504359	111.017326

Table 1: Runtimes for hard scaling test of ordered algorithm

Number of Processors	**Execution Time (seconds)**	
	n = 8192	**n = 16384**
1	73.335023	332.689703
2	59.482343	275.506995
4	38.042014	165.061362
6	29.096231	128.563151
8	29.354413	119.307265
10	23.715078	106.103206
12	22.287731	103.253261
14	22.946194	101.098668
16	26.257196	111.279003

Table 2: Runtimes for hard scaling test of splitting algorithm

Number of Processors	Execution Time (seconds)	
	n = 8192	**n = 16384**
1	75.727839	333.361403
2	51.730589	254.907719
4	31.087949	142.349868
6	24.336194	112.281694
8	20.184779	94.006221
10	18.117914	85.266375
12	16.890382	78.322475
14	16.041758	75.985690
16	14.976526	68.556623

Table 3: Runtimes for hard scaling test of alltoall algorithm

Number of Processors	Execution Time (seconds)	
	n = 8192	**n = 16384**
1	0.253357	1.049682
2	0.766012	3.264267
4	1.990641	9.326189
8	6.134346	31.618342
16	25.585454	113.162578

Table 4: Runtimes for weak scaling test of ordered algorithm

Number of Processors	Execution Time (seconds)	
	n = 8192	**n = 16384**
2	0.801341	3.106938
4	1.897967	8.953949
8	6.310973	29.368787
16	25.917724	112.335712

Table 5: Runtimes for weak scaling test of splitting algorithm

Number of Processors	Execution Time (seconds)	
	n = 8192	**n = 16384**
1	0.253727	0.999423
2	0.802434	2.841281
4	1.792704	7.213300
8	4.812081	20.268827
16	15.176379	67.662916

Table 6: Runtimes for weak scaling test of alltoall algorithm

4 Discussion

So far, both algorithms work and parallelize the serial Shearsort algorithm, speeding its execution time for big input data. They don't exhibit almost any difference in performance, and reach a peek speedup of roughly 3.5 for the hard scaling. Nevertheless, the implementation of the alltoall transposition algorithm improves this maximum speedup above 5. This is a significant improvement, but not a great result. The way the weak efficiency drops almost exponentially in Fig. 6 is a clear indicator that the algorithms are heavily memory-bounded. This should not come as a surprise, since the size of the matrices used is quite large, the memory handling becomes an issue. However, if one uses smaller matrices, then the communication overhead would be even more significant, so the algorithm is bounded both by memory and communication overhead.

In addition to that, one experiment was carried out serially sorting an array of 16384^2 elements, and the execution time was 39.2 seconds. If one compares this to the parallel execution time of the ordered algorithm in its optimal performance, 65.5 seconds, the problem is apparent. This is a clear indicator that the parallelization of the algorithm is not efficient, and the communication overhead is too high. One must notice, though, that while the transposition-inefficient methods reach a maximum speedup at 14 processors after which efficiency drops, alltoall transposition does not show a maximum, suggesting that the code could potentially benefit from using more than 16 processors.

The greatest cause of bottleneck in the algorithm is the transposition of the matrix. This is a very costly operation, and even if implemented in the most efficient way, it still requires a lot of communication between the processes. The alltoall algorithm is the most efficient in this regard, but it is not straightforward to implement, and it still requires a lot of memory to store the transposed submatrices.

One possible solution to this problem is to find a better algorithm for the local transposition of the submatrices. In addition to that, one could try to find a different sorting algorithm that does not require contiguous memory blocks, so that the transposition of the matrix is not necessary, or perhaps implement a parallel sorting algorithm and split the available processors between sorting and dividing the matrix.

5 Peer review

This report was reviewed by Márton Bidlek, who provided valuable feedback on the clarity and organization of the report. The reviewer suggested adding more details to the algorithms and possible ideas to improve its performance, as well as providing more information on the scaling tests. He also pointed out that the report could benefit from a more detailed discussion of the results. The feedback was very helpful and was used to improve the report.

References

[1] Scherson Isaacd, Sen Sandeep, and AD Shamir. Shear sort-a true two-dimensional sorting technique for vlsi networks. In *International Conference on Parallel Processing*, pages 903–908. International Conference on Parallel Processing, 1986.

[2] Philip Wilkinson. *Parallel programming: Techniques and applications using networked workstations and parallel computers, 2/E*. Pearson Education India, 2006.

[3] HPC Wiki. Scaling. https://hpc-wiki.info/hpc/Scaling, 2022.