Parallel ShearSort Algorithm Using MPI

Uppsala University, Department of Information Technology,
1TD070 Parallel and Distributed Programming

June 4, 2025

Carl Löfkvist
Master's Programme in Engineering Physics

UPPSALA
UNIVERSITET

1 Introduction

Sorting structured data is one of the most common challenges in computer science and software development. *ShearSort* is a sorting algorithm designed for two-dimensional matrices. It works by alternately sorting the rows in ascending and descending order, followed by sorting the columns in ascending order. This produces a snakelike ordering pattern, as illustrated below:

$$\begin{array}{cc} \text{Initial matrix} & \text{ShearSorted matrix} \\ \begin{bmatrix} 16 & 2 & 3 & 13 \\ 5 & 11 & 10 & 8 \\ 9 & 7 & 6 & 12 \\ 4 & 14 & 15 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 2 & 3 & 4 \\ 8 & 7 & 6 & 5 \\ 9 & 10 & 11 & 12 \\ 16 & 15 & 14 & 13 \end{bmatrix} \end{array}$$

Efficient and optimized implementations of such algorithms are crucial for the overall performance of many applications. [1]

2 Problem Description

The task is to parallelize the serial ShearSort algorithm (see Algorithm 1) using the open source Message Passing Interface (MPI) implementation OpenMPI in the C programming language. The program is to read a *square* input matrix of size $n \times n$ from a text file, perform a parallel ShearSort using p processes and write the execution time to standard output. The rows of the matrix are distributed amongst the processes (not necessarily evenly) and sorted independently. To sort the columns, the matrix must be transposed, which means processes must exchange elements. Thus, implementing a parallel matrix transposition function is the main challenge of the program.

Algorithm 1 Serial ShearSort for an $n \times n$ matrix

```
Require: $A$ is an $n \times n$ matrix
 1: $phases \leftarrow \lceil \log_2(n) \rceil + 1$
 2: for $i = 1$ to $phases$ do
 3:     if $i$ is odd then
 4:         for $r = 0$ to $n - 1$ do
 5:             if $r$ is even then
 6:                 Sort row $r$ in ascending order
 7:             else
 8:                 Sort row $r$ in descending order
 9:             end if
10:         end for
11:     else
12:         for $c = 0$ to $n - 1$ do
13:             Sort column $c$ in ascending order
14:         end for
15:     end if
16: end for
```

ShearSort is inherently well-suited for parallelization since both row and column sorting operations are completely independent of each other. Each pass of the algorithm is a $O(n^2 \log(n))$ operation, and the matrix is sorted in $O(\log(n))$ iterations. The ideal parallel speedup is therefore $O(n^2 \log^2(n)/p)$. As data sizes grow, serial ShearSort therefore becomes impractically slow and parallelization therefore become a necessity. Although the algorithm can be extended to arbitrary matrix dimensions, this implementation is contained to square matrices. The number of processes used must not exceed the side length n of the matrix. The correctness of the implementation will be compared to a serial implementation of ShearSort

3 Parallel Implementation

This section outlines the strategy used in the parallel implementation of ShearSort. Key design choices include using MPI for process communication and a block distribution with remainder to balance the workload.

Transpositions are used to simplify column sorting. The goal is to maintain correctness while minimizing communication overhead.

As mentioned in Section 2, implementing a parallel transposition function is the main task of this program. Although transposing the matrix before and after column sorting isn't an explicit requirement of the ShearSort algorithm, it resolves critical data locality issues in parallel implementations. In C, arrays are stored in row-major order, making column access inefficient and often impossible in distributed memory programs where processes typically own complete rows rather than having access to entire columns. To resolve this, the global matrix is transposed after row sorting, allowing efficient column-wise access for the column sorting phase. After column sorting, the matrix is transposed back to restore row-major layout. This completes one ShearSort phase, and the algorithm requires $\log(n) + 1$ such phases total.

Algorithm 2 Parallel ShearSort for an $n \times n$ matrix using p processes

```
Require: $A$ is an $n \times n$ matrix distributed across $p$ processes
 1: $phases \leftarrow \lceil \log_2(n) \rceil + 1$
 2: for $i = 1$ to $phases$ do
 3:     if $i$ is odd then
 4:         for all processes $j$ in parallel do
 5:             for each row $r$ assigned to process $j$ do
 6:                 if $r$ is even then
 7:                     Sort row $r$ in ascending order using quicksort
 8:                 else
 9:                     Sort row $r$ in descending order using quicksort
10:                 end if
11:             end for
12:         end for
13:     else
14:         Transpose matrix $A \leftarrow A^T$
15:         for all processes $j$ in parallel do
16:             for each row $r$ assigned to process $j$ (originally columns) do
17:                 Sort row $r$ in ascending order using quicksort
18:             end for
19:         end for
20:         Transpose matrix $A \leftarrow A^T$
21:     end if
22: end for
```

The implementation accepts any number of processes p less than or equal to the side length of the matrix. Although in most practical cases, it is fair to assume that the number of cores available are far fewer than the side length of the typical matrix encountered (i.e., $p \ll n$).

After initializing the global matrix on the root process, rows are distributed in blocks, with cyclic remainder to lower-ranked processes. This means that each process is assigned a base of $\lfloor n/p \rfloor$ rows, and the first $d = n \mod p$ processes are assigned an extra one. This structure is predictable, which simplifies communication and balances workload well.

The implementation of the `shearsort` function follows the structure of Algorithm 2. The C standard library function `qsort` is used to sort rows and columns locally within each process. The function `transpose_square_matrix` performs the transposition of the global matrix. An overview of the function is presented in Algorithm 3. The communication routine `MPI_Alltoallv` is a key design choice in this program. It enables collective communication between processes when data is unevenly distributed, which is the case when $n \mod p \neq 0$

Figure 1 illustrates the ownership of matrix rows and the communication pattern during transposition. In this example, ten rows are distributed amongst three processes, with process 0 having an extra one. Grey squares indicate diagonal elements, which remain unchanged under transposition. The saturated colored squares point to elements that are transposed within the process, meaning that they end up on a row which also is owned by the process after transposing. The remaining colored squares are therefore elements exchanged between processes.

After $\log(n) + 1$ phases, the array is reassembled on the root process. After checking whether the matrix was sorted correctly, the execution time of the slowest process is written to standard output and the program

Algorithm 3 Parallel Transpose of a Square Matrix

```
Require: $A$ is a local block of rows from an $n \times n$ matrix
1: Compute local and global row distribution
2: Initialize send/receive counts and displacements
3: Fill send buffer with column-wise elements destined for other processes
4: Perform MPI_Alltoallv to exchange matrix columns
5: Reassemble received data into transposed layout using temporary buffer
6: Exchange elements to be transposed within process
7: Fill in remaining columns using the received data
```

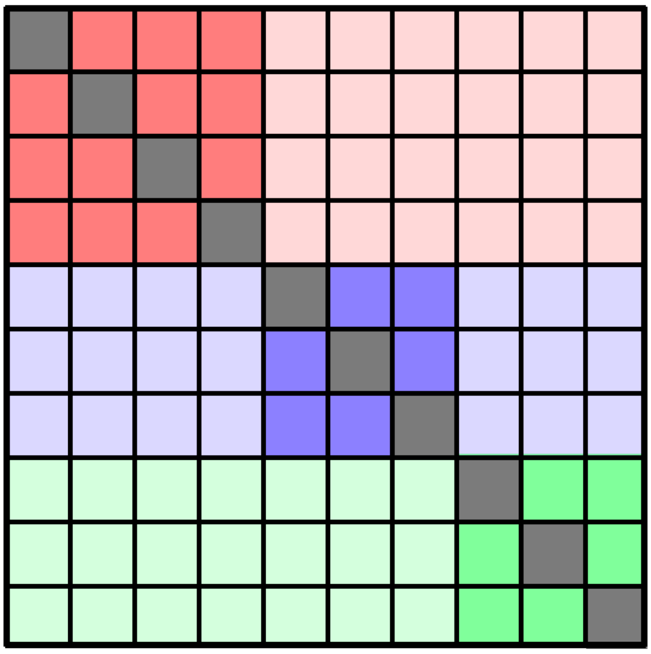

Figure 1: Diagram of row ownership in the square matrix ($p = 3$, $n = 10$)

exits.

4 Experiments and Results

This section presents the performance experiments conducted to evaluate the parallel ShearSort implementation, along with the results. The strong and weak scalability of the program was measured up to 16 processes. Table 1, Figure 2 and Figure 3 present the result of the strong scalability experiments and the resulting speedup for varying input sizes. Table 2 and Figure 4 likewise present the result of the weak scalability experiments and the efficiency per core. The computations were performed on resources provided by UPPMAX under Project SNIC 2025/2-247[2].

Table 1: Performance results of the parallel Shearsort algorithm for different input sizes and number of processes.

n	**Processes**	**Time (s)**	**Speedup**	**Efficiency**
1,000,000	1	1.314	1.000	1.000
	2	0.679	1.936	0.968
	4	0.389	3.377	0.844
	8	0.219	6.009	0.751
	16	0.119	11.063	0.691
4,000,000	1	5.529	1.000	1.000
	2	2.686	2.058	1.029
	4	1.511	3.659	0.915
	8	0.975	5.670	0.709
	16	0.506	10.931	0.683
9,000,000	1	13.634	1.000	1.000
	2	6.607	2.063	1.032
	4	3.683	3.702	0.925
	8	2.401	5.679	0.710
	16	1.276	10.681	0.668
16,000,000	1	25.627	1.000	1.000
	2	12.578	2.037	1.019
	4	6.830	3.752	0.938
	8	4.445	5.766	0.721
	16	2.334	10.979	0.686

Table 2: Weak scalability results showing efficiency with increasing number of processes (1,000,000 elements per process)

Processes	Time (s)	Efficiency	Elements per Process
1	1.187	1.000	1,000,000
4	1.520	0.781	4,000,000
9	2.155	0.551	9,000,000
16	2.325	0.511	16,000,000

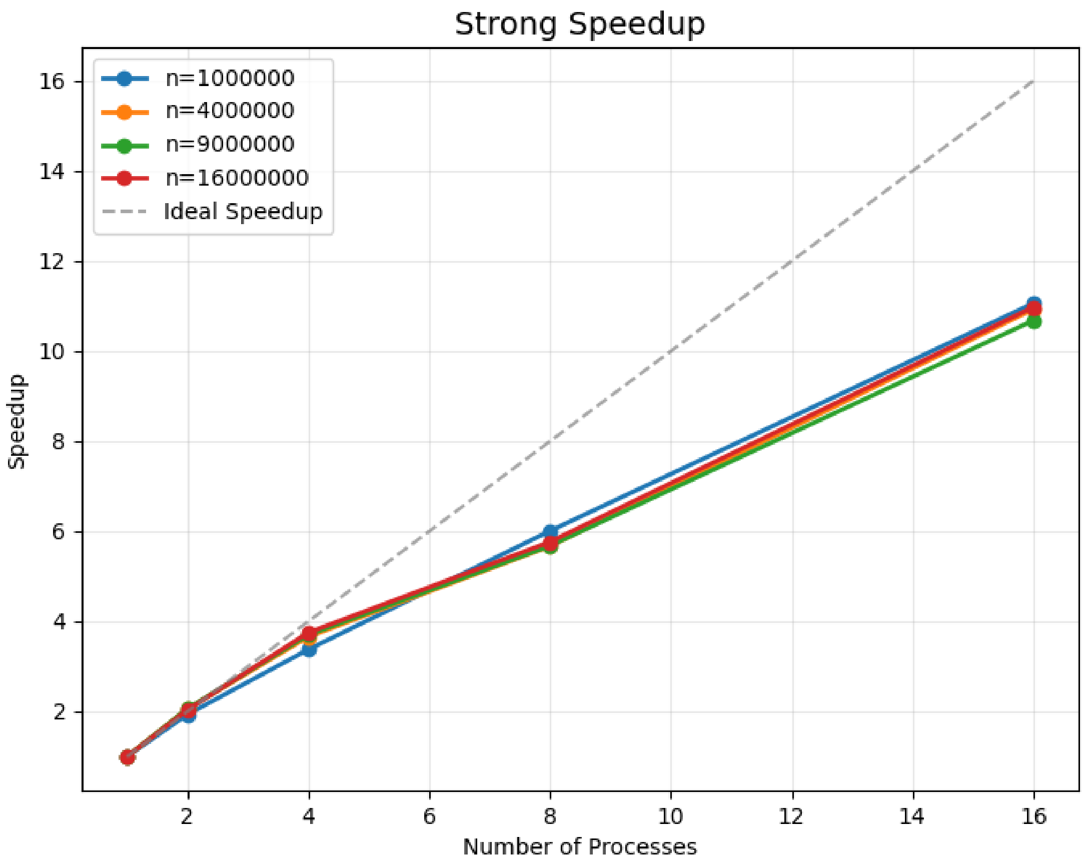

Figure 2: Strong scalability of the parallel Shearsort algorithm, showing speedup as a function of the number of processes for different input sizes.

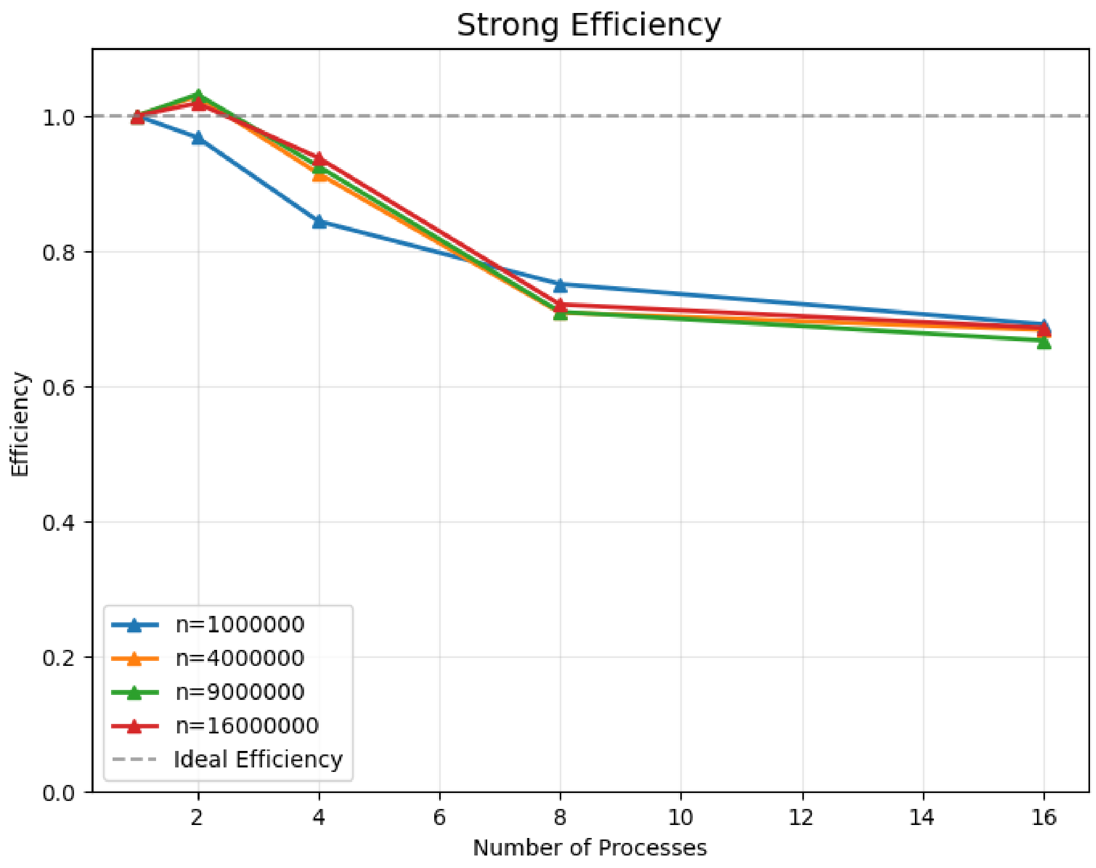

Figure 3: Strong efficiency of the parallel Shearsort algorithm, showing parallel efficiency as a function of the number of processes for different input sizes.

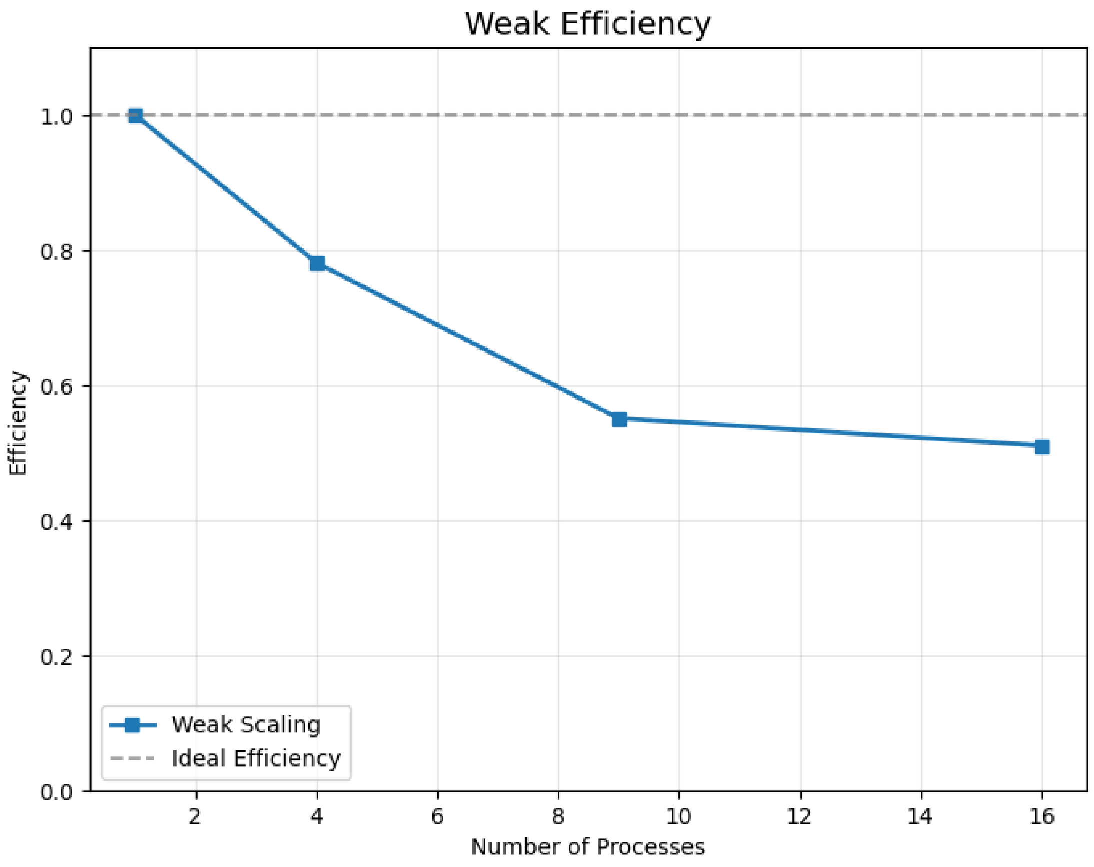

Figure 4: Weak Scalability of the Parallel Shearsort Algorithm showing efficiency with increasing processes and problem size.

5 Discussion

In this section, we discuss the performance and implementation of the parallel ShearSort algorithm. Based on the results presented in Section 4, the implementation demonstrates competitive performance relative to established expectations for similar sorting algorithms. Figure 2 shows a clear relation between increased number of processes and increased speedup, supporting the correctness of the implementation. At $p = 8$ we observe a noticeable deviation from the ideal speedup, which is supported by the dip at the same point in the efficiency per Figure 3. This pattern is mostly observed across all input sizes.

The speedup curve of the input sizes look mostly the same. This is because for all but a few combinations of input size and number of processes, the L1 and L2 caches are too small (at input size 1,000,000 and $p \geq 4$, the memory per process fits in the L2 cache, which likely explains why we observe a slightly higher speedup).

The drop in efficiency can likely be attributed to communication overhead, which of course increase as p increase. As p increase, the *proportion* of elements communicated per process in the transposition function increase. Intra-process communication is of course much cheaper than inter-process communication. The number of elements transposed internally within a process is n^2/p^2, which leads to inter-process communication dominating as higher p. In addition to this, all-to-all communication routines also mean more contention with increased p as processes compete for limited resources and bandwidth.

Under the assumption that each row is independently and uniformly random and that troublesome edge-cases for the `qsort` function are evenly distributed, this implementation had great load balancing. No process should be disproportionally overloaded as rows are distributed evenly. The transposition function introduces several synchronization points that takes a toll on execution time, but in necessary for the correctness of the algorithm.

The weak efficiency exhibited a fast decline to approximately 0.5 at $p = 8$, indicating diminishing returns when increasing the number of processes up to this point. This drop can likely be attributed to increased communication overhead and contention as more processes compete for shared resources, particularly during the all-to-all transposition phases. Interestingly, the efficiency stabilized somewhat at $p = 16$, which may suggest that beyond a certain process count, the overhead growth slows or the workload per process becomes sufficiently small, reducing the relative impact of communication.

In conclusion, the results confirm the success of the methodology. The results support the correctness of the code, and exhibit the behavior one could expect when the program is run on the machine specifics. As the main task was to develop a parallel implementation of the ShearSort algorithm, less focus was spent on serial optimizations like minimizing branching and optimizing loops. Furthermore, since processes send messages of the same length multiple times, one could leverage persistent communication to further optimize the implementation.

References

[1] Sandeep Sen and Isaac D. Scherson. Shear sort: A true two-dimensional sorting techniques for vlsi networks. IEEE, 1986. Available at: https://www.semanticscholar.org/paper/Shear-Sort:-A-True-Two-Dimensional-Sorting-for-VLSI-Sen-Scherson/211b7bdf135a8d2237c261f478ac71bec0a514da.

[2] Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). Rackham cluster at uppmax, uppsala university. `https://www.uu.se/en/centre/uppmax/resources/clusters/rackham`, 2024. Accessed: 2025-05-30.

A Peer Review

Peer reviewing was done on both code and report with Stina Brunzell who did project 2, Conjugate Gradient method

B Sent

Report:

- The introduction is thorough and detailed, which really gets the reader up to speed on the subject without going off course.
- A paragraph could be added on the constraints of the implementation and its input parameters. For example, if n_{base} must be an even number, or if the number of processes needs to be even.
- The section explaining the algorithm implemented may be a bit too verbose, and details like explanation of the `EXP_MODE` constant could be left for the reader to interpret when diving deeper into the code. The same goes for the pseudo-code in this section.
- Overall, the report is of high quality. Many aspects of the algorithm were considered, such as persistent versus non-persistent communication, how the domain was divided between processes, and the details of the machine used. That said, some sections are too verbose and include information that does not add value to the report.

Code:

- Consider refactoring the file structure. Instead of one `cg.c`, place functions into a separate `cg.c` with a corresponding header file `cg.h`, all called from a main script `main.c`. This adds a cleaner interface, better maintainability, and improved modularity.
- Functions lack proper documentation. Consider adding comments explaining the purpose of each function and its input parameters.
- Great use of the MPI functions `MPI_Cart_create` and `MPI_Neighbor_alltoallw`, which are highly applicable to your problem. This avoids the all-too-common ”reinvention of the wheel” that many coders fall into.
- Overall, the code is well structured and easy to read. Everything is designed with optimization in mind, both in the serial and parallel code. It combines clarity and efficiency in a near-perfect manner.

C Received

Report:

- Assuming the Shear sort algorithm was not your invention, it would be a good idea to reference the source you consulted.
- In your report you state that you performed your experiments on one of the university's Linux computers, but the instructions given are: "Your task is to implement and experimentally evaluate a parallel algorithm for one particular application/algorithm using MPI on the distributed memory cluster `rackham` or `snowy`."
- Adding the efficiency metric to the strong scaling results would strengthen your report. Strong efficiency can be easily computed using your timings in Table 2 or 3 and the formula from the lecture *Overhead, speedup, scalability*. Further, discussion about the weak scalability would be good to include.
- Overall, the language quality is high with few to no spelling errors. The disposition follows the instructions. For increased readability, it would be preferable if you placed the Figures in connection to the text you are referring to them in.

Code:

- The README file is nicely designed and contains enough detail for an external to build and run the code.
- The descriptive header file is well-formulated and makes it easy to understand each function's task. Although, in the functions themself it would be clearer how they perform this task if some comments were added, as you did in `is_shearsorted`.
- Did you consider performing the matrix transpose with one-sided communication, similar to the example from the lecture *MPI: One-sided Communication*? I'm not sure this would enhance performance or even if it would be a good idea in your case, but might be worth given thought. Also, use of one-sided communication instead of the scatter/gather methodology enables parallel input/output. If you did try other parallelization methods then the ones in the final implementation it would be a good idea to include them in the report and motivate why your choice is optimal for the target problem
- The return values of the function `compare_asce` might overflow for large integers (happened to me in Assignment 3). A safer solution would be

  ```c
  int compare_asce(const void *v1, const void *v2) {
      int a = *(const int *)v1;
      int b = *(const int *)v2;
      if (a < b) return -1;
      if (a > b) return 1;
      return 0;
  }
  ```

  Same thing for the function `compare_desc`.