Uppsala University

Parallel and Distributed Programming

1TD070

Individual Project:
Shear Sort

Students:
Linjing Shen

Lecturer:
Prof. Roman Iakymchuk

May 22, 2024

Contents

1 Introduction **1**

2 Algorithms **1**
2.1 Sequential Algorithm 1
2.2 Parallel Algorithm 1

3 Experiments and Results **2**
3.1 Strong scalability experiment 3
3.2 Weak scalability experiment 4

4 Discussion **5**

1 Introduction

Shearsort is a sorting algorithm particularly well-suited for sorting on a two-dimensional grid, developed by Sen, Shamir, and Isaac Scherson at the University of California, Irvine [2]. The core idea is to arrange data in a 2D grid and perform alternating row and column sorting steps, gradually moving the data in the entire grid towards the final sorted state. Each row or column sorting operation is referred to as a "shear," and the overall sorting process resembles the action of shearing sheep, hence the name Shearsort. The basic idea behind Shearsort is to perform $\lceil \log(n) + 1 \rceil$ iterations to sort the values in an $n \times n$ matrix with a total of N values. In each iteration, the rows and columns are sorted. Specifically, even-indexed rows are sorted in ascending order, odd-indexed rows are sorted in descending order, and then all columns are sorted in ascending order.
Shearsort is well-suited for parallel processing, where each processing unit can handle sorting a set of rows or columns independently. Sorting of rows can be done independently, while sorting of columns can be efficiently achieved by transposing the matrix. The MPI function `MPI_Alltoall()` is particularly useful for efficient global communication and data redistribution, facilitating data exchange and coordination in parallel computing [1]. Additionally, the course materials from the Parallel Algorithm course at the University of Science and Technology of China mention various other methods for sorting two-dimensional arrays besides shear sort. For example, sorting can be performed using a divide-and-conquer strategy, continually breaking down and merging data for sorting, or employing element-swapping algorithms like odd-even transposition sorting [3].

2 Algorithms

2.1 Sequential Algorithm

In terms of data storage, projecting the obtained two-dimensional array onto a one-dimensional linear array ensures that contiguous array members are stored close to each other in memory, reducing the likelihood of cache misses. However, when shearsort begins to use vertical access in the second step, iterating over each column instead of each row, the code will attempt to access data stored in a non-sequential order. As N grows larger, cache misses become inevitable, leading to poor performance. Therefore, when performing column sorting, the code first transposes the matrix, performs row sorting on the transposed content instead of directly implementing column sorting, and then transposes it back to its normal form after completing this process.
Besides, ShearSort inherently requires local sorting of arrays. I tested the algorithm on an input matrix of size 5,000. Compared to the standard C `qsort` function, I also tried implementing quicksort and mergesort. However, both performed worse than `qsort` to varying degrees. Therefore, the code simply uses the `qsort` function.

2.2 Parallel Algorithm

For the parallel algorithm, another benefit of using a 1D array instead of a 2D array to store matrix data is that the MPI library handles one-dimensional arrays more efficiently,

and it simplifies the process of data distribution and collection. Not only does this storage method adhere more closely to the principle of memory contiguity, improving access efficiency, but it also better aligns with the MPI library's parallel computing model, enhancing program performance and scalability.

In the design of parallel algorithms, considering that the size n of the 2D array may not be evenly divisible by the number of processes, a blocking method is used to allocate matrix data with unequal row counts to different processes. The size of the data block each process receives is determined by the matrix size and the number of processes, using the `sendcounts` and `displs` arrays to describe the size and offset of the data block each process receives. Starting from process 0, redundant rows are allocated to the corresponding processes one by one to avoid allocating all redundant rows to one process. This blocking method helps reduce data communication overhead and makes parallel computation more efficient. Accordingly, `MPI_Scatterv()` and `MPI_Gatherv()` are used instead of the common equal division and aggregation methods `MPI_Scatter()` and `MPI_Gather()`. These two functions are used to distribute data from the root process to other processes and to collect data from each process back to the root process, respectively.

After the matrix is read from the root process and allocated to each process, the iterations of the shear sort algorithm begin. In the first step, each process performs local row-wise ascending or descending sorting. If the current iteration count is less than or equal to $\lceil \log_2(n) \rceil$, the second step begins, which is the matrix transposition algorithm. First, each process converts its data into a format usable by `MPI_Alltoallv()`, then employs `MPI_Alltoallv()` to facilitate data exchange among all processes, and finally reverts the data according to the corresponding rules within each process. This entire process achieves one transposition. Subsequently, each process performs local row-wise ascending sorting, instead of implementing column sorting of the matrix. At the end of each iteration, the matrix is transposed back using the aforementioned transposition method. Furthermore, the algorithm uses `MPI_Wtime()` to measure the program's execution time, calling it at the beginning and end of the program to calculate the runtime. `MPI_Reduce()` is then utilized to perform reduction operations on execution times across all processes, calculating global statistics and obtaining the maximum time consumed.

3 Experiments and Results

In the experimental phase, I mainly aimed to evaluate the scalability of the parallel shearsort algorithm, which includes both strong scalability experiments and weak scalability experiments. Firstly, the strong scalability experiments were conducted by keeping the input size constant and gradually increasing the number of parallel processing processes. Large input files stored in a specified directory were utilized as test data to comprehensively assess the program's performance. The execution time of the program under different numbers of processes was recorded for each experiment, and speedup was calculated to evaluate the performance of the parallel program as the number of processors changed.

Secondly, the experimental plan also included conducting weak scalability experiments to

ensure that the problem size handled by each process remained constant as the number of processes increased. This implies that as the number of processes increases, the total amount of data processed will also increase accordingly, maintaining a proportional relationship between problem size and the number of processes. This experiment required recording the execution time of the program under different problem sizes and numbers of processes and calculating speedup to assess whether the execution time of the parallel program remained stable under different problem sizes and numbers of processes.
To ensure the accuracy and reliability of the experimental results, multiple experiments were conducted for each parameter combination, and average values were obtained.

3.1 Strong scalability experiment

Table 1 and Figure 1 show the results of the strong scaling experiments, which were carried out by keeping the input size constant at 5,000 while incrementally increasing the number of parallel processing processes, using the number of processes ranging from one to sixteen. Each sub-experiment was run five times, and the average of these runs was calculated.
The formula for the acceleration ratio is as follows, where T_s is the serial execution time and T_p is the parallel execution time.

$$\text{Speedup} = \frac{T_s}{T_p}$$

Process Number	Execution Time (s)	Speedup
1	34.1541	1.00
2	19.7221	1.73
3	13.8791	2.46
4	10.7500	3.18
5	9.1708	3.72
6	8.1602	4.19
7	7.4382	4.59
8	6.8754	4.97
9	6.3291	5.40
10	5.7005	5.99
11	5.4988	6.21
12	5.3121	6.43
13	5.1537	6.63
14	5.0282	6.79
15	4.9380	6.92
16	4.8335	7.07

Table 1: Strong scalability experiment

Firstly, from the figure and the table, it can be observed that as the number of processes increases, the execution time decreases gradually, which is as expected. From a single

process to 16 processes, the execution time decreases significantly from 34.1541 seconds to 4.8335 seconds, indicating a notable improvement. This suggests that with the increase in the number of processes, the computational workload is distributed more evenly across multiple processes, thereby accelerating the overall execution of the algorithm.
Secondly, according to the provided data, it can be seen that as the number of processes increases, the speedup gradually increases. From one process to 16 processes, the speedup increases from 1.00 to 7.07. However, inevitably, as the number of processes increases, the rate of speedup growth gradually slows down. When the number of processes is less than 13, the speedup can maintain above half of the ideal speedup, while when the number of processes exceeds 13, the improvement in speedup becomes very slow. This is because with the increase in the number of processors, the communication and synchronization overhead become more significant, limiting further improvement in speedup.
In summary, as the number of processes increases, the execution time decreases, and the speedup gradually increases, which is consistent with the definition of strong scalability. Considering both the degree of improvement and the growth rate of speedup at a certain number of processes, the algorithm demonstrates fair strong scalability.

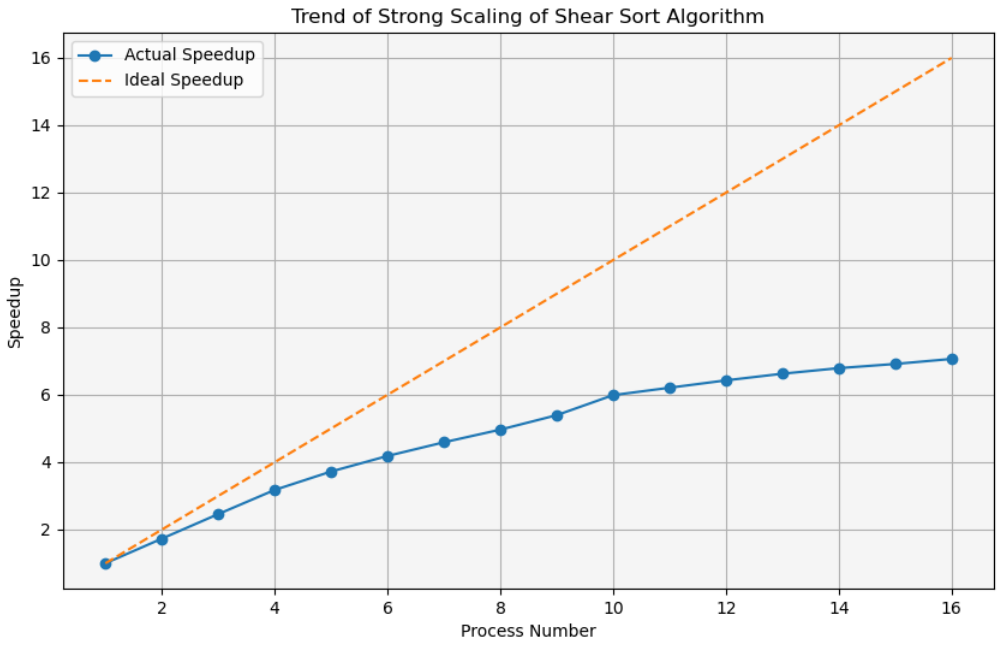

Figure 1: Strong scalability experiment

3.2 Weak scalability experiment

In the weak scalability experiment, the experiment guarantees that each process processes 1,000 rows of one-dimensional array data, and calculates the time-consumption and efficiency when the number of processes ranges from one to eight. The results are

shown in Table 2 and Figure 2. Each configuration is run five times, and the average is calculated.
The efficiency calculation formula is as follows, where p denotes the number of processes, T_s is the serial execution time and T_p is the parallel execution time.

$$\mathrm{E} = \frac{T_s}{p * T_p}$$

Process Number	Input Size	Execution Time (s)	Efficiency
1	1,000	0.9338	1.00
2	2,000	2.5257	0.77
3	3,000	4.4441	0.75
4	4,000	6.3743	0.73
5	5,000	9.1708	0.69
6	6,000	12.0297	0.65
7	7,000	14.9593	0.62
8	8,000	18.3576	0.60

Table 2: Weak scalability experiment

From the figure and table, it can be observed that as the size of the input matrix increases, the execution time gradually increases, while the efficiency decreases. For input sizes ranging from 1,000 to 8,000 and parallelism from 1 process to 8 processes, the execution time increases from 0.9338 seconds to 18.3576 seconds, and the efficiency decreases from 1.00 to 0.60. With the increase in problem size, although more parallel processing units are utilized, the performance cannot be effectively maintained. It is evident that efficiency decreases gradually with the increase in the number of processes. This could be attributed to the increase in communication and synchronization overheads, as larger problem sizes necessitate more communication between processes. Additionally, larger problem sizes also consume more memory space.
Overall, the parallelization of the algorithm on large-scale problems does not effectively utilize additional computational resources, resulting in decreased efficiency. This indicates decent weak scalability of the algorithm. While the algorithm may exhibit some parallel speedup on small-scale problems, the efficiency gradually decreases as the problem size increases, indicating that the parallel performance of the algorithm cannot linearly scale with the problem size.

4 Discussion

Based on the data and analysis from the scalability experiments, it can be concluded that the algorithm cannot achieve the ideal scalability. This is due to the increased frequency of communication and synchronization operations among processes as the problem size or the number of processes increases, resulting in overhead, particularly noticeable when dealing with large-scale problems. Additionally, larger problem sizes require more memory to store data, potentially leading to memory access bottlenecks, which in turn limits

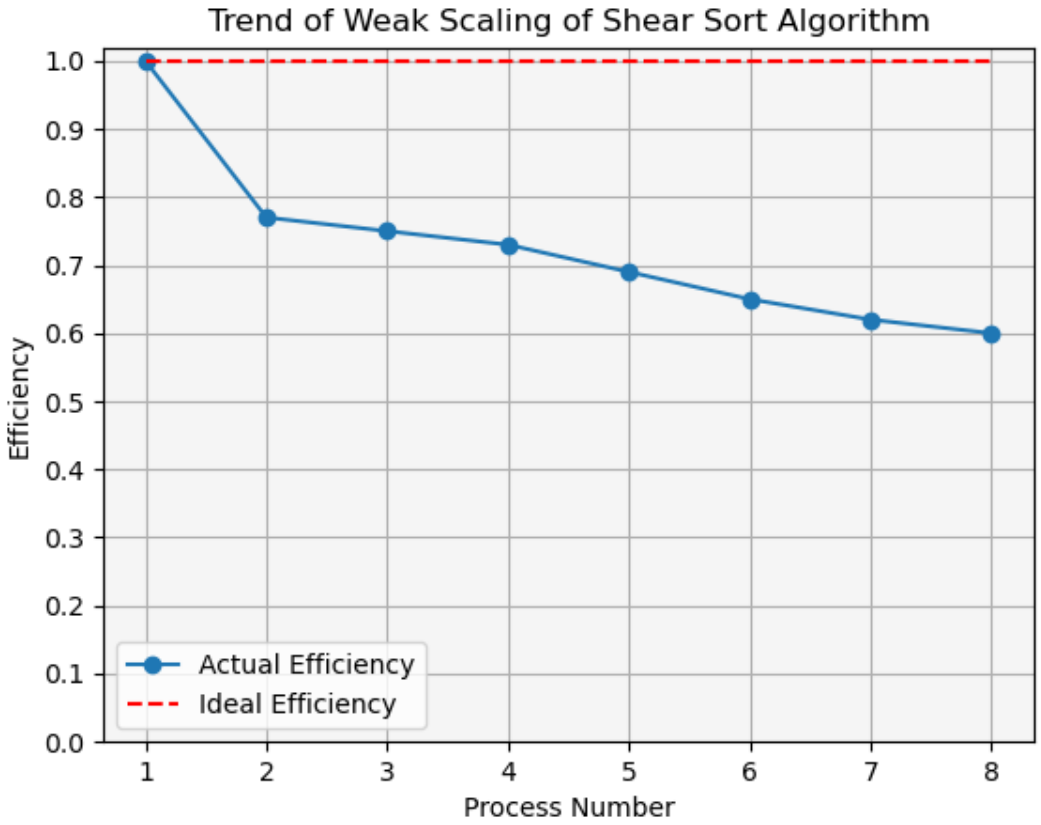

Figure 2: Weak scalability experiment

the improvement in parallel efficiency. Furthermore, there may be load imbalance issues at larger problem sizes, where certain processes may have to handle more workload than others, leading to decreased efficiency.
Apart from this version of the code, I also attempted other implementations of shearsort, such as using `MPI_Scatterv()` and `MPI_Gatherv()` to transpose matrices instead of `MPI_Alltoallv()`. However, test results showed significant improvements in both strong and weak scalability when using `MPI_Alltoallv()` compared to multiple uses of `MPI_Scatterv()` and `MPI_Gatherv()` to transpose matrices. When using `MPI_Scatterv()` and `MPI_Gatherv()`, parallelizing with 16 processes on a 5,000-sized input matrix yielded less than a speedup of 2, possibly due to the increased communication overhead caused by gathering and scattering within iterations, leading to communication imbalance, while `MPI_Alltoallv()` patterns such as full-to-full communication exchange the same amount of data among all processes, avoiding single-point bottlenecks.
Although I compared quicksort and mergesort with `qsort`, there are many other local sorting methods, and some might have better scalability and shorter runtime than `qsort`'s implementation. In my previous tests, for a 5,000-sized input matrix, quicksort performed better in terms of execution time than `qsort` up to 20 processes, but its overall strong and weak scalability was about 5% worse. Mergesort, on the other hand, had longer runtime compared to `qsort`.
To achieve better performance, optimizing memory usage could be considered, such as reusing already allocated memory space to avoid frequent allocations and deallocations, such as within the `temp` variable loop. Additionally, although I accounted for cases where the number of processes does not evenly divide the matrix size, load imbalance issues may still arise. Dynamic adjustment of task allocation to ensure roughly equal loads among processes could be attempted.

References

[1] Norm Matloff. Programming on parallel machines. *University of California, Davis*, 39319, 2011.

[2] Sandeep Sen, Isaac D. Scherson, and Adi Shamir. Shear sort: A true two-dimensional sorting techniques for vlsi networks. In *International Conference on Parallel Processing*, 1986.

[3] Yun Xu. Sorting and selecting in synchronous. Teaching Resources, Feb.–Jun. 2024. Parallel Algorithms (Class 23).