Uppsala University

Parallel and Distributed Programming

1TD070

Assignment 3:

The Parallel Quicksort algorithm

Students:
Houwan Liu
Linjing Shen

Lecturer:
Prof. Roman Iakymchuk

May 15, 2024

Contents

1 Introduction

Quick sort is an efficient divide-and-conquer algorithm that works by breaking down a problem into smaller subproblems and then merging the solutions of these subproblems to obtain the final result. In quick sort, we start by selecting a pivot element from the array to be sorted, and then partition the array into two parts: elements smaller than the pivot are placed on the left, and elements greater than the pivot are placed on the right. This process is known as partitioning.
The partitioning process divides the array into two subarrays, which are then recursively sorted. This recursive process can be executed in parallel because each subarray can be sorted independently without relying on the sorting results of other subarrays. In parallel quick sort, we leverage the divide-and-conquer strategy to break down the larger problem into smaller problems and then solve these smaller problems in parallel.
In parallel quick sort, the sorting process of each subarray can be executed independently, allowing efficient utilization of multi-core processors and distributed computing resources to accelerate the overall sorting process. The core idea of divide-and-conquer and recursion is well-suited for designing and implementing parallel algorithms. Through parallelization, quick sort can efficiently handle large-scale data sorting problems, improving the efficiency and performance of sorting algorithms.

2 Implementation Methodology

Based on the fundamentals of quicksort, we define a parallel strategy for the algorithm. First the array to be sorted is read in process 0, then process 0 divides the array to be sorted equally among p processes. Local sorting is performed within each process, and then p medians are selected as pivots for each process, and each process sends these pivots to process 0. In our algorithm, there are three strategies for pivot selection, the first is for process 0 to randomly select a pivot as the global pivot, the second is to select the median of these pivots as the global pivot, and the third is to select the mean number of all pivots as the global pivot. After the global pivot is selected, the selected pivot is broadcast to all processes, and then all processes are divided into two groups, and the data between the two groups is exchanged to ensure that one group is all less than the global pivot and one group is all greater than the global pivot. Recursive calls are made to divide the groups until there is only one process in each group and the sorting is complete.
After implementing the sorting, we merge the data within all processes based on a tree-based merging approach based on the implementation techniques and optimisations discussed by Kataria [1]. This approach provides a fundamental method for efficiently merging sorted subarrays, which is crucial for the scalability of the algorithm on multiple processors.
In the specific code implementation, the root processor first reads the entire array of data to be sorted from the input file, and then splits the array into p parts called chunks based on the number of processors p. The root processor distributes these parts evenly to each processor using MPI_Scatter.

We then use the `calculate_pivot` function to take care of selecting the local pivot on each processor and collect the median to the root processor. The global pivot is then selected at the root processor based on the `pivot_strategy` and broadcast to all processors via MPI_Bcast.
Each processor performs a fast sorting algorithm on its own chunk based on the global pivot, dividing the array into two parts smaller than the pivot and larger than the pivot, and recursively sorting these two parts. At the same time, we use a tree-structured data merging method, in which the partially sorted array fragments are gradually merged between processors through multiple MPI_Send and MPI_Recv operations after the sorting is completed.

3 Numerical Experiment

Our performance experiments aim to evaluate the scalability of the parallel quicksort program. Scalability involves adjusting resources and capabilities to match application demands by expanding or reducing them.
Firstly, we conduct a strong scalability experiment by maintaining a constant problem size and incrementally increasing the number of parallel processing processes. We utilize large input files stored in the specified directory as test data to comprehensively evaluate the program's performance. Each experiment involves recording the program's execution time with varying numbers of processes and calculating speedup to assess how the parallel program performs as the number of processors changes.
Second, our plan includes performing weak scalability experiments to ensure that the problem size handled by each process remains constant as the number of processes is increased. This means that as the number of processes increases, the total amount of data processed increases accordingly, thus maintaining the proportionality between the problem size and the number of processes. We record the execution time of the program for different problem sizes and number of processes and compute the acceleration to assess whether the execution time of the parallel program remains stable for different problem sizes and number of processes.
To ensure the accuracy and reliability of our experimental results, we conduct multiple experiments for each parameter combination and derive averages for the final results. The execution time and speedup for different parameter combinations are presented using tables and graphs, and we will analyze the differences between actual speedup and ideal speedup.

3.1 Strong scalability experiment

Table1 and Figure1 show the results of our strong scaling experiments, which we conducted for each of the three strategies by keeping the problem size constant at 125,000,000 while gradually increasing the number of parallel processing processes. 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.

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

Pivot Strategy	Process Number	Execution Time	Speedup
1	1	25.6288	1.00
	2	13.3402	1.92
	4	8.6969	2.95
	8	5.8761	4.36
	16	3.4461	7.44
2	1	25.6530	1.00
	2	13.3671	1.92
	4	8.6467	2.97
	8	5.8970	4.35
	16	3.4499	7.44
3	1	25.5921	1.00
	2	13.3988	1.91
	4	8.7048	2.94
	8	5.9038	4.33
	16	3.5280	7.25

Table 1: Strong scalability experiment

From the analysis of the strong scalability data and its corresponding visualization, it is evident that for all pivot strategies, the execution times decrease as the number of processes increases. Specifically, the execution time for a single process is approximately 25.6 seconds across all strategies, which significantly reduces to about 3.5 seconds when 16 processes are utilized.

Initially, the speedup closely matches the increase in the number of processes, demonstrating effective strong scalability. However, as the number of processes continues to rise, the rate of speedup does not maintain this proportionality. Particularly when the process count reaches 16, the speedup plateaus at around 7.

As the number of processes increases, the overhead associated with data communication between processes becomes more significant. In the MPI framework, operations like MPI_Gather, MPI_Bcast, and MPI_Comm_split involve significant communication across different processors. The time spent in these operations grows with the number of processes, which can impede scalability.

Moreover, the function exchange_chunks and the subsequent merge operations are critical for the algorithm's performance. These operations, which involve sending and receiving chunks of data and then merging these chunks, are resource-intensive and scale with the number of processes.

Additionally, the use of MPI_Comm_split for dynamically managing groups adds computational and management overhead. Each split reduces the group size and reassigns process ranks, which is a computationally intensive task when frequently performed in a scenario with many processes.

This section concludes that while the scalability is initially strong, various factors inherent to the MPI operations and the dynamic management of process groups contribute to a plateau in speedup as the process count increases.

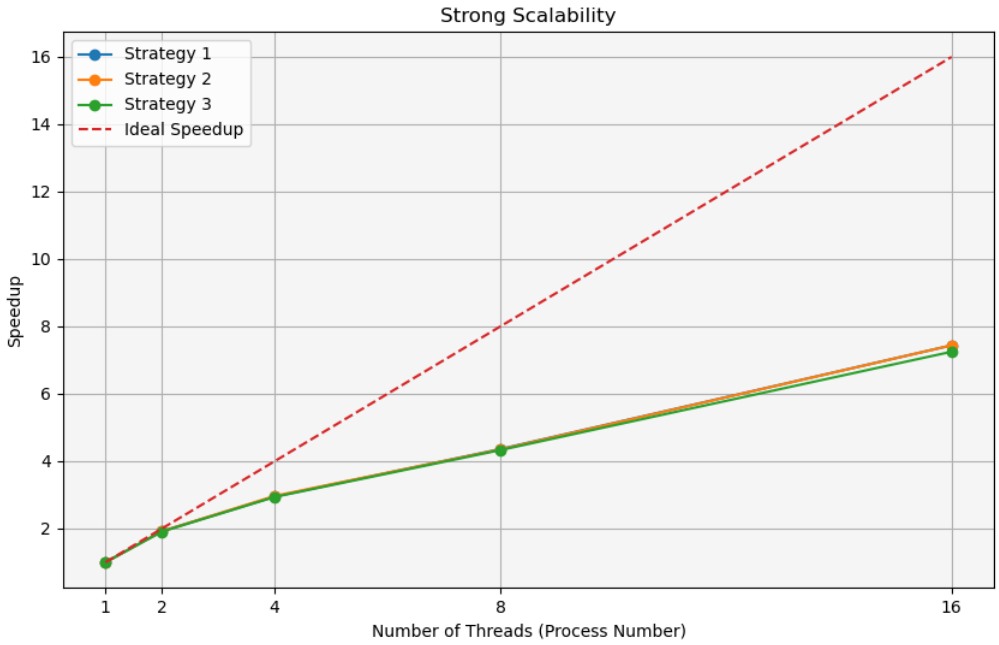

Figure 1: Strong scalability experiment

3.2 Weak scalability experiment

For the weak scalability experiments, we guarantee that each process processes 62,500,000 samples, compute the time-consumption and efficiency when the number of processes is 2, 4, and 8, and compare the differences between the three strategies. The results are presented in Table.2 and Figure.2. Each configuration was run five times, and the averages were computed.
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.

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

From the weak scalability table and its corresponding chart, we observe that all three strategies yield nearly identical results. As the number of processes and the input size increase proportionally, the execution time slightly increases, deviating from the ideal scenario in which execution time would remain constant. This demonstrates a decline in efficiency from 0.96 with 2 processes to 0.53 with 8 processes. This drop suggests that as the workload per processor increases, the system's ability to manage it efficiently decreases, indicating inefficiencies in the system's scalability.

Pivot Strategy	Process Number	Input Size	Execution Time	Efficiency
1	2	125,000,000	13.3402	0.96
	4	250,000,000	18.1126	0.74
	8	500,000,000	26.4861	0.53
2	2	125,000,000	13.3671	0.96
	4	250,000,000	18.1995	0.74
	8	500,000,000	26.2420	0.53
3	2	125,000,000	13.3988	0.96
	4	250,000,000	18.2156	0.73
	8	500,000,000	26.3314	0.53

Table 2: Weak scalability experiment

In summary, the weak scalability demonstrated by the system shows a decrease in efficiency as both the workload and number of processors increase, pointing to scalability limitations in efficiently handling larger parallel workloads. While the system can handle an increased workload by scaling up the number of processes, there is a noticeable decline in efficiency.
The weak scalability performance of this code is likely hindered by significant communication overhead, load imbalance among processes, and the inherent complexity and computational intensity of data merging and pivot calculation steps. The use of collective and point-to-point communication functions such as MPI_Gather, MPI_Bcast, MPI_Irecv, and MPI_Comm_split significantly contributes to the overall communication overhead. Besides, the merging process and the redistribution of data are computationally expensive and can contribute to scalability issues.

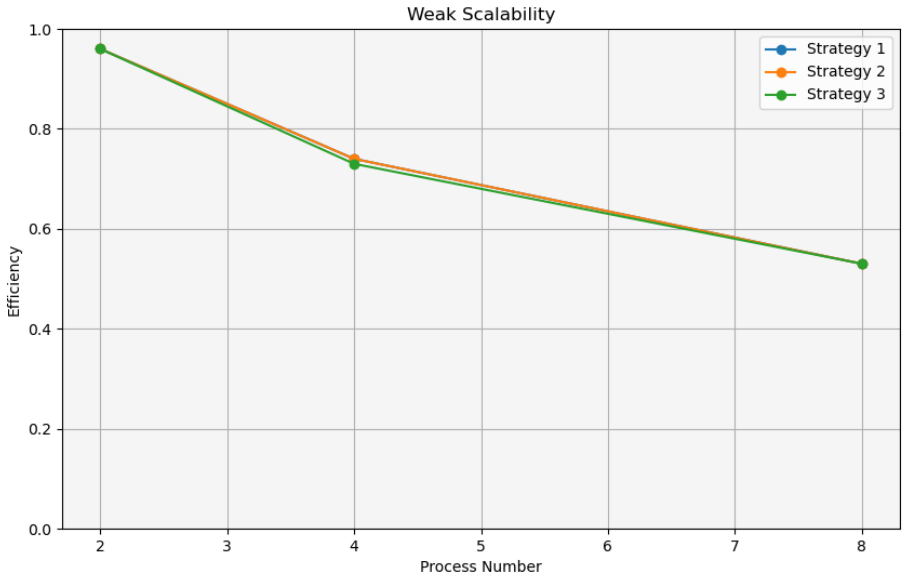

Figure 2: Weak scalability experiment

3.3 Descending order experiment

In addition, we use a fully descending sorted sequence as an input file to compare with a randomly distributed sequence. Table3 and Figure3 show the speedup ratio of descending sorted 125,000,000 numbers under the three strategies.

In the experiment detailed in Table 3, we explored the impact of input data order on the performance of three different pivot strategies by comparing sequences sorted in descending order to those in their original, randomly distributed state. The results clearly demonstrate that data ordered in descending sequence consistently yielded longer execution times and lower speedups across all strategies and process counts compared to the original order. This might be due to the way the pivot is selected and used in the partitioning process, where a descending order could potentially lead to unbalanced partitioning, particularly if the pivot tends to be near one end of the value range.

These findings suggest that optimizations in pivot selection and data partitioning methods could significantly improve the performance of parallel sorting algorithms, especially when handling sorted or nearly sorted data. Such enhancements are crucial for developing more robust algorithms capable of maintaining high efficiency across varying data distributions.

Pivot Strategy	Process Number	Order	Execution Time	Speedup
1	Original	1	25.6288	1.00
		4	8.6969	2.95
		8	5.8761	4.36
	Descending	1	12.3127	1.00
		4	5.5737	2.21
		8	4.2489	2.90
2	Original	1	25.6530	1.00
		4	8.6467	2.97
		8	5.8970	4.35
	Descending	1	12.3021	1.00
		4	6.4308	1.91
		8	4.9604	2.48
3	Original	1	25.5921	1.00
		4	8.7048	2.94
		8	5.9038	4.33
	Descending	1	12.4127	1.00
		4	6.5846	1.89
		8	5.0144	2.48

Table 3: Descending order experiment

Additionally, we have done the same test on sequences of length 93, where the execution times and speedup ratios between the original and reverse order sequences showed smaller variation. This uniformity in performance despite the sequence order suggests that the length of the sequence may be too short to effectively demonstrate the impact of data distribution on the efficiency of the sorting process. This could imply that for very

short sequences, the overhead associated with parallel processing may overshadow any potential gains from optimized data ordering.

Figure 3: Descending order experiment

4 Discussion

In the comprehensive analysis of our parallel sorting algorithms, we evaluated performance under three distinct experimental conditions: strong scalability, weak scalability, and descending order input. Each test provided valuable insights into how our parallel program handles different data distributions and scales with varying numbers of processors. This section of the paper discusses the implications of these results for the efficiency and applicability of parallel sorting techniques in real-world scenarios.

The strong scalability tests demonstrated that as the number of processors increased, the execution time for sorting a fixed-size dataset decreased substantially. This was expected and aligns with the primary advantage of parallel processing—reducing the time complexity of computationally intensive tasks. The algorithm effectively distributes workload among multiple processors, enhancing computational efficiency. However, the scalability was not perfectly linear, suggesting room for optimization in processor utilization and task distribution.

In weak scalability tests, where the size of the dataset increased proportionally with the number of processors, we observed that the system maintained relative efficiency, but with noticeable performance degradation. The efficiency, measured as speedup relative to the increase in workload, tended to decrease. While the algorithm can scale up to

handle larger datasets by increasing the number of processors, it does so with diminishing returns, likely due to overheads such as data communication and management becoming more significant.

Unlike the randomly distributed sequences typically used in scalability tests, sorting sequences pre-arranged in descending order showcased lower performance efficiency across all pivot strategies. The execution times were longer and the speedups less significant when compared to those of sequences in original order. This means our sorting algorithm is sensitive to the initial order of the data, with descending order leading to inefficient pivot selections and unbalanced data partitions.

The analyses from these experiments illustrate that while our parallel sorting algorithms are capable of handling large datasets and scaling up with additional processors, their efficiency is influenced by both the data distribution and the number of processors. The algorithms show potential vulnerabilities when dealing with sorted data, indicating that future work should focus on enhancing pivot selection and data partitioning methods to accommodate various data configurations more robustly.

Furthermore, the results advocate for algorithmic adjustments to minimize overhead, particularly in weak scalability scenarios, and improve the baseline efficiency of the system, especially when processing smaller datasets or data in descending order. The findings from these tests will guide the refinement of our parallel sorting algorithms, ensuring they are not only optimized for theoretical scenarios but are also robust and efficient in diverse, practical applications.

References

[1] Puneet Kataria. Parallel quicksort implementation using mpi and pthreads, 2008.