PDP Assignment 3

Arvid Ehrengren, Viktor Kangasniemi

May 13, 2025

1 Theoretical Background and Implementation

Quicksort is a divide-and-conquer algorithm that recursively partitions data around a pivot and sorts the resulting subarrays. To parallelize the algorithm, the concept is extended to multiple processors to accelerate execution by dividing the workload.

The algorithm begins by evenly distributing the input data among all processors. Each processor sorts its local segment using a sequential algorithm such as qsort. A pivot is then selected, and each processor partitions its data into elements smaller and larger than the pivot. Data is exchanged so that processors in one group contain only smaller elements and those in the other group contain larger ones. After merging the received data, the process recurses independently within each group. This continues until each processor holds a final sorted partition, requiring at most $\log_2(p)$ steps, where p is the number of processors.

Three pivot strategies are implemented: using the median from one processor, the median of all medians, or the mean of all medians. This parallel approach balances the computational load and significantly reduces sorting time for large datasets.

To allow each subgroup of processes to sort their part of the data independently, new MPI groups and communicators are created during each recursive step of the global_sort function. This is done using MPI_Comm_group, MPI_Group_incl, and MPI_Comm_create. The processes are divided into two groups based on whether their data is smaller or larger than the pivot. Each group then gets its own communicator (small_comm and big_comm) and continues sorting recursively within that communicator. This process repeats until each group contains only one process.

2 Numerical Experiments

We evaluated both strong and weak scaling of the parallel quicksort program. Strong scaling was tested on a fixed input size (input250000000.txt and input_backwards125000000.txt) with the 3 different pivot strategies (1. Select the median in one processor in each group of processors. 2. Select the median of all medians in each processor group. 3. Select the mean value of all medians in each processor group.) and varying the number of MPI processes (1,2,4,8,16). Weak scaling was tested by increasing both the input size and the number of processes proportionally, increasing the input size (input125000000.txt,input250000000.txt,
input500000000.txt,input1000000000.txt, input2000000000.txt, and the corresponding backwards files) along with the number of processes used (1,2,4,8,16).

3 Results

3.1 Strong Scaling

Number of Processes	Execution Time (s)	Speedup
1	42.268849	1.00
2	23.562373	1.79
4	12.847158	3.29
8	7.738866	5.46
16	5.768616	7.33

Table 1: Strong Scaling Results for Pivot Strategy 1

Number of Processes	Execution Time (s)	Speedup
1	42.122625	1.00
2	23.793066	1.77
4	14.312863	2.94
8	10.273182	4.10
16	8.631459	4.88

Table 2: Strong Scaling Results for Pivot Strategy 2

Number of Processes	Execution Time (s)	Speedup
1	43.383142	1.00
2	27.842520	1.56
4	15.066095	2.88
8	9.509530	4.56
16	8.574841	5.06

Table 3: Strong Scaling Results for Pivot Strategy 3

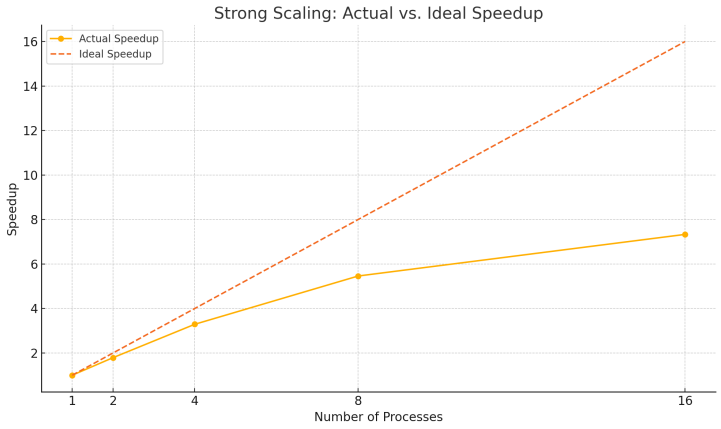

Figure 1: Speedup comparison between 1,2,4,8 and 16 processes.

3.2 Weak Scaling

Number of Processes	Input Size	Execution Time (s)	Speedup
1	125,000,000	20.50	1.00
2	250,000,000	23.40	0.88
4	500,000,000	26.04	0.79
8	1,000,000,000	34.52	0.59
16	2,000,000,000	48.20	0.43

Table 4: Weak Scaling Results for Pivot Strategy 1

Number of Processes	Input Size	Execution Time (s)	Speedup
1	125,000,000	20.33	1.01
2	250,000,000	23.54	0.87
4	500,000,000	28.85	0.71
8	1,000,000,000	42.39	0.48
16	2,000,000,000	69.96	0.29

Table 5: Weak Scaling Results for Pivot Strategy 2

Number of Processes	Input Size	Execution Time (s)	Speedup
1	125,000,000	20.35	1.01
2	250,000,000	23.10	0.89
4	500,000,000	29.25	0.70
8	1,000,000,000	42.45	0.48
16	2,000,000,000	71.78	0.29

Table 6: Weak Scaling Results for Pivot Strategy 3

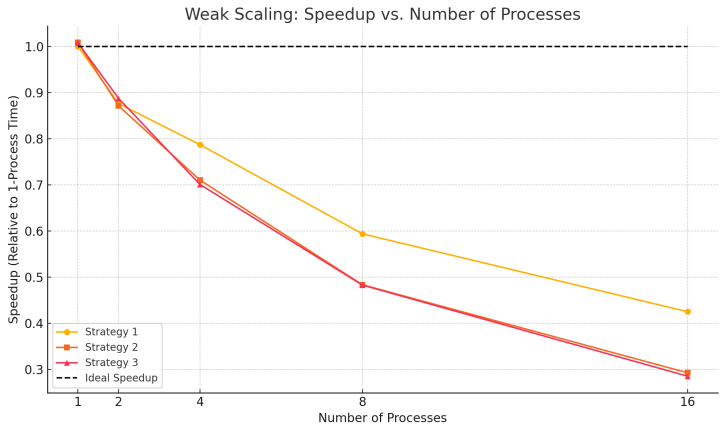

Figure 2: Speedup comparison between 1,2,4,8 and 16 processes.

3.3 Strong Decreasing Order Input

Number of Processes	Execution Time (s)	Speedup
1	90.39	1.00
2	45.47	1.99
4	25.35	3.56
8	15.82	5.71
16	15.62	5.78

Table 7: Strong Scaling Results for Pivot Strategy 1 on Decreasing Order Input

Number of Processes	Execution Time (s)	Speedup
1	88.44	1.00
2	45.36	1.95
4	24.65	3.59
8	15.68	5.64
16	15.86	5.58

Table 8: Strong Scaling Results for Pivot Strategy 2 on Decreasing Order Input

Number of Processes	Execution Time (s)	Speedup
1	88.63	1.00
2	45.41	1.95
4	24.72	3.58
8	24.69	3.59
16	11.19	7.92

Table 9: Strong Scaling Results for Pivot Strategy 3 on Decreasing Order Input

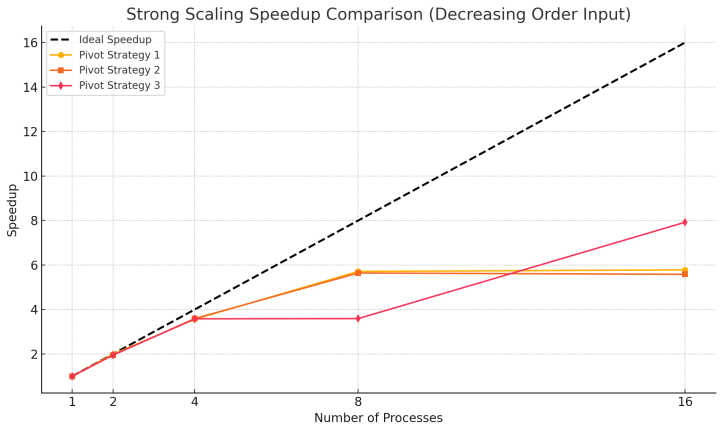

Figure 3: Strong Scaling Speedup comparison between 1,2,4,8 and 16 processes for all Pivot Strategies on Decreasing Order Input.

3.4 Weak Decreasing Order Input

Number of Processes	Execution Time (s)	Speedup
1	9.38	1.00
2	12.31	0.76
4	17.06	0.55
8	28.75	0.33
16	54.73	0.17

Table 10: Weak Scaling Results for Pivot Strategy 1 on Backwards Input

Number of Processes	Execution Time (s)	Speedup
1	9.42	1.00
2	12.17	0.77
4	16.77	0.56
8	28.69	0.33
16	54.20	0.17

Table 11: Weak Scaling Results for Pivot Strategy 2 on Backwards Input

Number of Processes	Execution Time (s)	Speedup
1	9.60	1.00
2	12.32	0.78
4	18.97	0.51
8	28.88	0.33
16	54.18	0.18

Table 12: Weak Scaling Results for Pivot Strategy 3 on Backwards Input

Figure 4: Weak Scaling Speedup comparison between 1,2,4,8 and 16 processes for all Pivot Strategies on Decreasing Order Input.

4 Discussion

The performance of the quick sort parallel program under strong scaling generally follows expectations, with execution time decreasing and speedup increasing as more processes are used see Figure 1. Pivot strategy 1 (selecting the median in one processor) yields the highest speedup (7.33× at 16 processes) due to its minimal communication overhead. Pivot strategy 2 (selecting the median of all local medians) provides more balanced partitioning but suffers from greater communication costs, resulting in a lower speedup (4.88×). Strategy 3 (using the mean of local medians) performs similarly to strategy 2 but may be more affected by outliers, leading to a slightly less optimal partition and speedup (5.06×). The deviation from the ideal linear scaling is due to factors such as communication overhead, and increased per process workload as the processes and workload increases. Overall, the observed results align well with the theoretical expectations, highlighting the trade off between pivot quality and parallel efficiency.

As seen in Figure 2 the weak scaling results reveal that the parallel program does not scale ideally as input size and the number of processes increase. With an ideal weak scaling behavior, execution time should remain roughly constant as the workload per process remains the same. However, all three pivot strategies show increasing execution times with more processes. Strategy 1 performs best overall, but even it experiences a slowdown (from 20.50s at 1 process to 48.20s at 16), resulting in a speedup of only 0.43. Strategies 2 and 3 scale worse, likely due to the higher communication overhead associated with computing medians or means across all processes, and their execution times at 16 processes exceed 69s, corresponding to speedups of just 0.29. These trends are expected to some extent in distributed memory systems where communication costs and synchronization overheads grow with the number of processes. Nonetheless, the results suggest that pivot selection strategies with lower communication complexity (like Strategy 1) are more suitable for weak scaling.

When performing the tests using a decreasingly sorted input set, the execution times across all pivot strategies decreased. This result is likely due to the fact that a decreasing order input represents the worst-case scenario for the standard qsort algorithm used in the local sequential sorting phase. In such cases, qsort performs significantly more comparisons and recursive calls due to unbalanced partitioning, resulting in higher execution time for a single process. However, when distributed across multiple processes in a parallel setting, this initial imbalance can actually accelerate the global sorting process.