Parallel Quicksort Implementation

Hangbiao Meng, Zhiping Hu

May 9, 2025

1 Problem Description

This work focuses on implementing a parallel version of the Quicksort algorithm using MPI. The goal is to leverage data-parallelism to improve sorting efficiency across multiple processes. The algorithm recursively partitions data among process subgroups, exchanging and sorting data based on pivot values until global order is achieved. The overall algorithm is below:

Algorithm 1 Parallel Quicksort Algorithm (Overall)

```
1: Divide the initial data into $P$ (number of processes) roughly equal parts, one part per process. (Handled by distribute_from_root)
2: Each process sorts its local data segment using a sequential sorting algorithm (e.g., qsort).
3: Perform a global sort operation recursively. This is the main parallel phase. (Implemented in global_sort, conceptually described in Algorithm 7)
4: Gather the sorted segments from all processes to the root process. (Handled by gather_on_root)
```

2 Peripheral Functions and Infrastructure

2.1 Pivot selection

1. Select the Median in one Processor in each group of processors.
Choose the median of the local data from the local root rank of each processor group. This is a simple method with minimal communication overhead.

Algorithm 2 MPI Function to Select Pivot from Root Process

```
1: function SELECT_PIVOT_ROOT_MEDIAN($local\_data, local\_size, comm$)
2:     $rank, p \leftarrow$ integers
3:     Call MPI_Comm_rank($comm$, &$rank$)          ▷ Local rank in current communicator
4:     Call MPI_Comm_size($comm$, &$p$)
5:     $pivot \leftarrow 0$
6:     if $rank = 0$ then          ▷ Root of the current subgroup (local rank 0)
7:         $pivot \leftarrow local\_data[local\_size/2]$
8:     end if
9:     Call MPI_Bcast(&$pivot$, 1, $MPI\_INT$, 0, $comm$)          ▷ Broadcast within subgroup
10:    return $pivot$
11: end function
```

2. Select the median of all medians in each processor group.
In this way, each process computes its local median first and then gather all medians at the root. lastly, compute the median of medians as the global pivot. This method is more balanced than Strategy 1. but it Requires MPI_Gather (minor communication overhead).

Algorithm 3 Median of Medians Pivot Selection in MPI

```
1: function SELECT_PIVOT_MEDIAN_MEDIAN($elements, n, communicator$)
2:     Call MPI_Comm_rank($communicator$, &$rank$)
3:     Call MPI_Comm_size($communicator$, &$size$)
4:     $local\_median \leftarrow 0$
5:     if $n > 0$ AND $elements \neq$ NULL then
6:         $local\_median \leftarrow$ get_median($elements, n$)
7:     end if
8:     $all\_medians \leftarrow$ NULL
9:     if $rank =$ ROOT then
10:        $all\_medians \leftarrow$ allocate array of size $size$
11:    end if
12:    Call MPI_Gather(&$local\_median, 1, MPI\_INT, all\_medians, 1, MPI\_INT, ROOT, communicator$)
13:    $pivot\_val \leftarrow 0$
14:    if $rank =$ ROOT then
15:        Sort $all\_medians$ array of size $size$
16:        $pivot\_val \leftarrow$ get_median($all\_medians, size$)
17:        Free $all\_medians$
18:    end if
19:    Call MPI_Bcast(&$pivot\_val, 1, MPI\_INT, ROOT, communicator$)
20:    return get_larger_index($elements, n, pivot\_val$)
21: end function
```

3. Select the mean value of all medians in each processor group.
In this way, each process computes its local median first and then gather all medians at the root processors. Lastly, Compute the arithmetic mean of medians as the pivot. This method can smooth out outliers. However, mean value may not be a good pivot if data is skewed.

Algorithm 4 Mean of Medians Pivot Selection in MPI

```
1: function SELECT_PIVOT_MEAN_MEDIAN($elements, n, communicator$)
2:     // Same steps as select_pivot_median_median until ROOT process handles all_medians
3:     $pivot\_val \leftarrow 0$
4:     if $rank =$ ROOT then
5:         // Difference: calculating mean instead of median here
6:         $sum\_of\_medians \leftarrow 0$
7:         for $i \leftarrow 0$ to $size - 1$ do
8:             $sum\_of\_medians \leftarrow sum\_of\_medians + all\_medians[i]$
9:         end for
10:        $pivot\_val \leftarrow sum\_of\_medians/size$                    ▷ Integer division
11:        Free $all\_medians$
12:    end if
13:    // Remaining steps are the same as select_pivot_median_median
14:    Broadcast pivot value and return index of elements larger than pivot
15: end function
```

2.2 Data distribution and gathering

For data distribution, we use Distribute_From_Root function, This function evenly distributes an array of elements from the root process to all MPI processes. The root computes how many elements each process should receive and their displacements, then uses MPI_Scatter to distribute the counts and MPI_Scatterv to distribute the actual data. The function dynamically allocates memory for the local data buffer on each process and returns the number of elements received.

Algorithm 5 Distribute Elements Evenly in MPI

```
function DISTRIBUTE_FROM_ROOT($all\_elements, n, my\_elements\_ptr$)
    Get $rank$ and $size$ using MPI_Comm_rank/size
    if $rank$ = ROOT then
        Allocate $sendcounts[size]$, $displacements[size]$
        Calculate elements per process: $base \leftarrow n/size$, $remainder \leftarrow n \bmod size$
        Distribute remainder elements one per process until exhausted
        Calculate displacement offsets for each process
    end if
    Distribute counts to all processes via MPI_Scatter
    Allocate local buffer of size $local\_n$
    Distribute actual data via MPI_Scatterv
    Clean up memory (root only)
    return $local\_n$
end function
```

For data gathering, we use gather_on_root function, This function gathers variable-sized arrays of elements from all MPI processes and collects them on the root process. The number of elements each process contributes is first gathered using MPI_Gather, followed by gathering the actual data using MPI_Gatherv. On the root, elements are stored in rank order—i.e., elements from process 0 (root) appear first, followed by those from processes 1, 2, and so on. The root allocates temporary arrays for receive counts and displacements, which are freed after use.

Algorithm 6 Gather Elements on Root Process in MPI

```
function GATHER_ON_ROOT($all\_elements\_buffer\_on\_root, my\_elements, local\_n$)
    Get $rank$ and $size$ using MPI_Comm_rank/size
    if $rank$ = ROOT then
        Allocate $recvcounts[size]$, $displacements[size]$
    end if
    Collect element counts from all processes via MPI_Gather
    if $rank$ = ROOT then
        Calculate displacement offsets for each process
    end if
    Gather all elements to root via MPI_Gatherv
    Clean up memory (root only)
end function
```

3 Core Algorithm: Recursive Global Sort

The core sorting logic resides in the `global_sort` function. Algorithm 7 outlines this procedure:

- **Initial Data Distribution, Local Sort and Pivot Selection:** The root process reads all data and distributes it using MPI_Scatterv (within distribute_from_root). Then, we apply the select_pivot function, called by select_pivot_and_partition. See previous section for out implementations.
- **Data Exchange (Step 3 in Algorithm 7, corresponding to Step 3.3 in Algorithm 1.1):** This is the core communication step within each recursive call of global_sort. After processes are conceptually split into two groups, a pairwise exchange occurs.
    - A process in the "lower" group sends its "large" elements (those greater than the pivot) to its partner in the "upper" group.
    - A process in the "upper" group sends its "small" elements (those less than or equal to the pivot) to its partner in the "lower" group.

This is implemented using MPI_Sendrecv (within the exchange_data helper function in the C code). MPI_Sendrecv is a safe way to perform point-to-point communication where each process both sends and receives data at the same time.

- **Recursive Decomposition:** After data exchange and local merging, the current MPI communicator is split into two new, disjoint communicators based on the color assigned during process grouping (lower half vs. upper half). The global_sort function is then called recursively using these new communicators. This process continues until a communicator contains only a single process, which implements the "for each half until each group consists of one single process" part of Algorithm 1.1.

Algorithm 7 Recursive Parallel Sort (Conceptual)

```
1: function RecursiveParallelSort(Elements, Communicator)
2:     if number of processes in Communicator ≤ 1 then
3:         return            ▷ Base case: data on this process is sorted relative to its group
4:     end if
5:                                                                ▷ 1. Pivot Selection
6:     Select a pivot element $p$ from the distributed Elements using all processes in Communicator.
7:                                              ▷ 2. Data Partitioning and Process Grouping
8:     Divide Communicator into two disjoint sub-communicators: $Comm_{\text{low}}$ and $Comm_{\text{high}}$.
9:     for all process $P_k$ in Communicator do
10:        Partition local data $D_k$ into:
           $D_k^{\leq p}$: elements $\leq p$
           $D_k^{>p}$: elements $> p$
11:    end for
12:                                                               ▷ 3. Data Redistribution
13:    Rearrange data globally such that:
        Processes in $Comm_{\text{low}}$ hold $\bigcup_{P_k} D_k^{\leq p}$
        Processes in $Comm_{\text{high}}$ hold $\bigcup_{P_k} D_k^{>p}$
        (This involves data exchange: $Comm_{low}$ sends $D_k^{>p}$, receives $D_j^{\leq p}$, and vice versa.)
14:    Let $Elements_{\text{low}}$ be the new data on $Comm_{\text{low}}$
15:    Let $Elements_{\text{high}}$ be the new data on $Comm_{\text{high}}$
16:                                                                ▷ 4. Recursive Calls
17:    RecursiveParallelSort($Elements_{\text{low}}$, $Comm_{\text{low}}$)
18:    RecursiveParallelSort($Elements_{\text{high}}$, $Comm_{\text{high}}$)
19: end function
```

4 Experimental Setup and Results

For this assignment, we evaluated the efficiency of our parallel Quicksort algorithm under both strong and weak scalability scenarios. All experiments were conducted on the RACKHAM cluster.

4.1 Strong Scalability Experiments

In the strong scalability tests, we maintained a fixed total problem size while varying the number of MPI processes. We specifically tested datasets with 250,000,000 (250M) and 125,000,000 (125M) elements. For each of these dataset sizes, runs were performed using 1, 2, 4, 8, and 16 MPI processes. A key focus of these strong scaling experiments was to observe performance variations under different conditions:

- **Pivot Selection Strategies:** Three distinct pivot selection strategies (Strategy 1: Median of Root, Strategy 2: Mean of Medians, Strategy 3: Median of Medians) were tested.

- **Data Input Order:** To understand the algorithm's robustness, tests were conducted with **random input data** and **backwards input data**.

The primary metric observed was the execution time for the parallel sort operation.

4.2 Weak Scalability Experiments

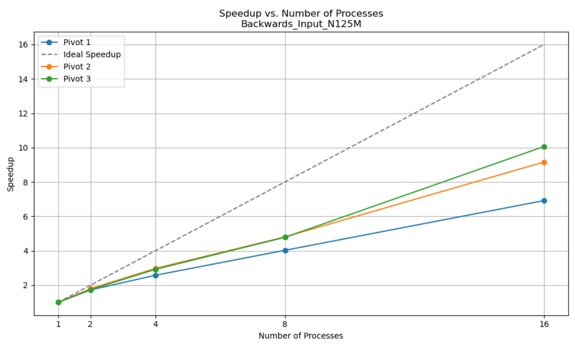
For the weak scalability tests, the workload per MPI process was kept constant, while the total problem size scaled proportionally with the number of processes. Each process was assigned a fixed workload of 125,000,000 (125M) elements. This means the total problem sizes tested were:

- 1 process: 125M elements total
- 2 processes: 250M elements total
- 4 processes: 500M elements total
- 8 processes: 1,000M (1B) elements total
- 16 processes: 2,000M (2B) elements total

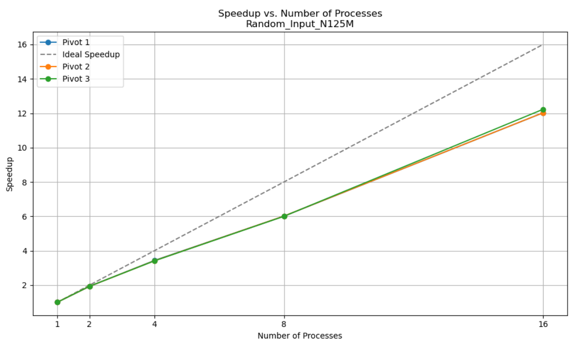
Similar to the strong scaling tests, these weak scaling experiments also evaluated the three different pivot selection strategies and tested both random and backward input data for each configuration.

4.3 Results

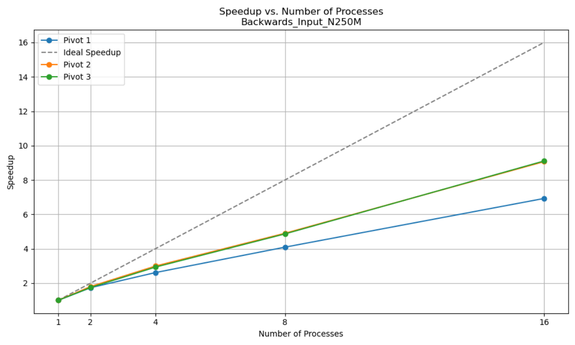
Recall the pivot here: Strategy 1: Median of Root, Strategy 2: Mean of Medians, Strategy 3: Median of Medians



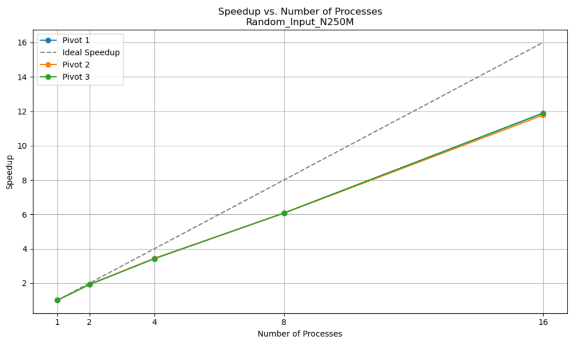
(a)



(b)



(c)



(d)

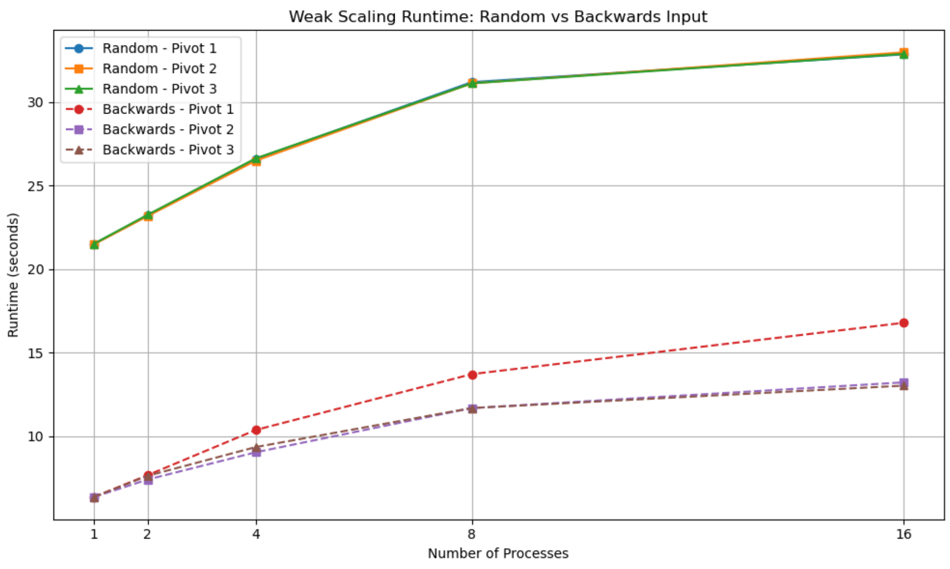
Figure 1: Speedup results for different inputs and sizes

Table 1: Efficiency and execution time for different inputs and pivot strategies

Input Type / Pivot Strategy	**Metric**	**1 proc**	**2 procs**	**4 procs**	**8 procs**	**16 procs**
N=250M						
Random / Pivot 1	Efficiency Time (s)	1.000 44.57	0.960 23.20	0.857 13.00	0.761 7.34	0.745 3.75
Random / Pivot 2	Efficiency Time (s)	1.000 44.65	0.967 23.09	0.861 12.96	0.761 7.36	0.735 3.79
Random / Pivot 3	Efficiency Time (s)	1.000 44.63	0.958 23.26	0.855 13.05	0.761 7.34	0.742 3.76
Backwards / Pivot 1	Efficiency Time (s)	1.000 13.41	0.867 7.74	0.654 5.13	0.511 3.28	0.433 1.94
Backwards / Pivot 2	Efficiency Time (s)	1.000 13.42	0.901 7.45	0.748 4.49	0.613 2.74	0.567 1.48
Backwards / Pivot 3	Efficiency Time (s)	1.000 13.40	0.872 7.68	0.732 4.57	0.610 2.75	0.569 1.47
N=125M						
Random / Pivot 1	Efficiency Time (s)	1.000 21.56	0.964 11.18	0.858 6.28	0.752 3.59	0.752 1.79
Random / Pivot 2	Efficiency Time (s)	1.000 21.51	0.965 11.14	0.855 6.29	0.752 3.58	0.752 1.79
Random / Pivot 3	Efficiency Time (s)	1.000 21.50	0.967 11.12	0.853 6.31	0.752 3.58	0.766 1.76
Backwards / Pivot 1	Efficiency Time (s)	1.000 6.39	0.860 3.72	0.642 2.48	0.504 1.59	0.433 0.92
Backwards / Pivot 2	Efficiency Time (s)	1.000 6.38	0.898 3.55	0.743 2.15	0.602 1.33	0.572 0.70
Backwards / Pivot 3	Efficiency Time (s)	1.000 6.42	0.871 3.69	0.729 2.20	0.597 1.34	0.631 0.64

Table 2: Weak Scaling Performance: Efficiency and Execution Time (seconds)

Number of Processes (P)	Random Input			Backwards Input		
	Strategy 1	Strategy 2	Strategy 3	Strategy 1	Strategy 2	Strategy 3
1	1.0000 (21.50s)	1.0000 (21.47s)	1.0000 (21.51s)	1.0000 (6.37s)	1.0000 (6.37s)	1.0000 (6.37s)
2	0.9273 (23.19s)	0.9265 (23.18s)	0.9249 (23.25s)	0.8294 (7.68s)	0.8589 (7.41s)	0.8340 (7.64s)
4	0.8090 (26.58s)	0.8109 (26.48s)	0.8079 (26.62s)	0.6136 (10.37s)	0.7042 (9.04s)	0.6809 (9.35s)
8	0.6900 (31.18s)	0.6904 (31.10s)	0.6913 (31.11s)	0.4642 (13.71s)	0.5449 (11.68s)	0.5450 (11.69s)
16	0.6548 (32.84s)	0.6515 (32.96s)	0.6544 (32.87s)	0.3793 (16.78s)	0.4816 (13.22s)	0.4889 (13.02s)



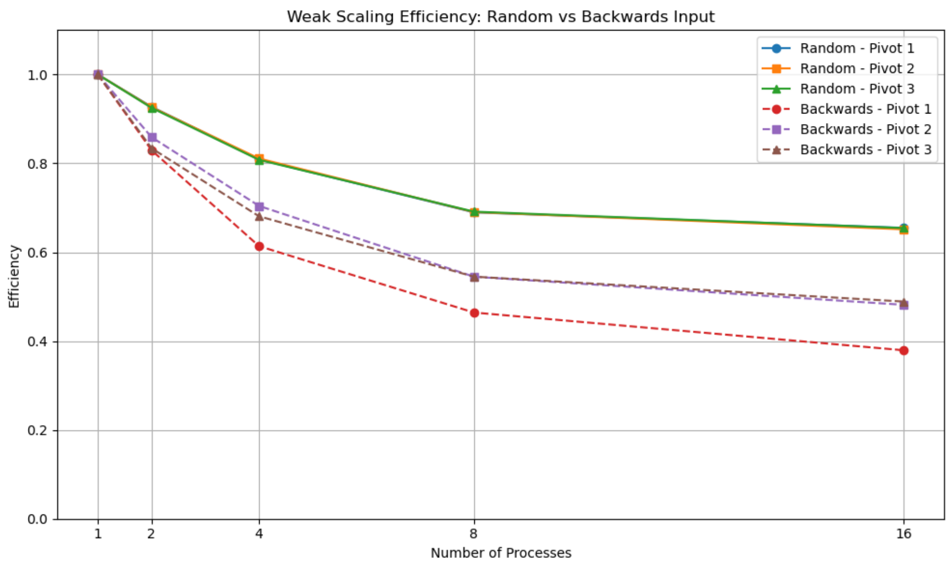


Figure 2: Caption

5 Discussion and Analysis

Reading the results, we noticed some interesting phenomena.

5.1 qsort is faster for backward inputs:

See table 1 and table 2, for problem size $N = 125M$ and $N = 250M$, our quick sort implementation is faster in backwards inputs(eg. 21.50s for random input V.S. 6.37s for backwards input). The main reason is that we used the built-in C library function for in-place quicksort and we set -O3 flag in Makefile.

- **Better built-in qsort:** The qsort function in glibc is not a pure quicksort; instead, it uses a hybrid strategy primarily based on merge sort[1], with heap sort as a fallback.
- **Optimization of memory access patterns:** By swapping data in 64-bit or 32-bit blocks instead of byte by byte, the number of instructions is reduced(see source code for qsort: swap_words_64 and swap_words_32)[1]. When the input is in reverse order, merging can be completed with a single pass, eliminating the need for frequent comparisons. The compiler's -O3 optimization may unroll the merge loop into SIMD[2] instructions (such as AVX, in Rackham, avx2 is enabled).

- **Improved branch prediction:** Since the subarrays are fully sorted before merging, the result of com(b1, b2) becomes highly predictable(eg. elements from b1 are always smaller)[3]. This leads to a near 100% hit rate in the CPU's branch predictor, significantly reducing pipeline stalls.

5.2 Effects of pivot selection

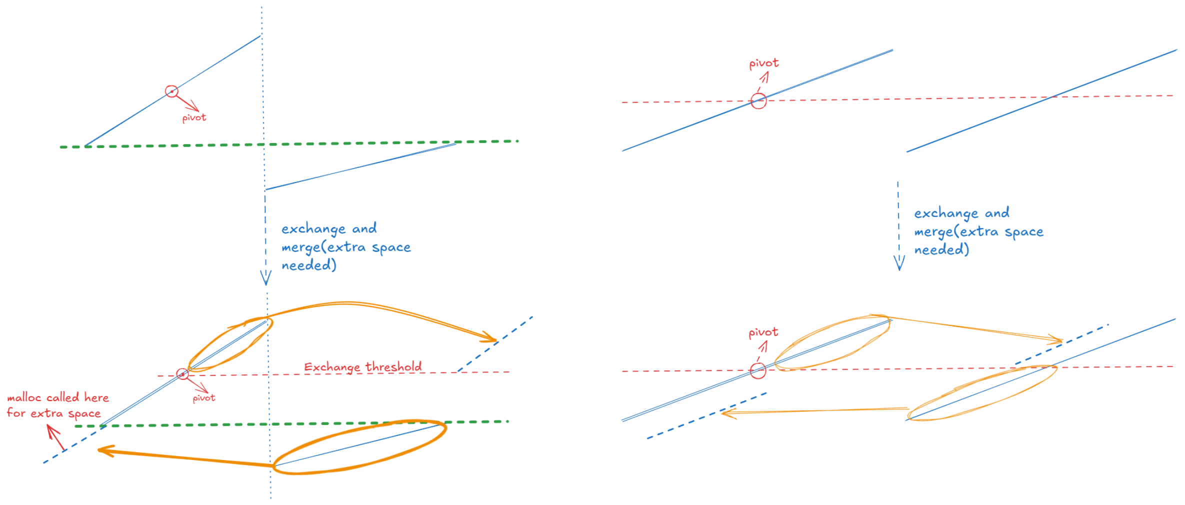
For *random datasets*, all three pivot strategies perform similarly in terms of:

- **Strong scalability** (fixed problem size, increasing processes)
- **Weak scalability** (problem size grows with processes)

Because random distributions inherently balance partitions, making pivot selection less critical. Even a simple pivot (Strategy 1) works well.
For *backwards datasets*, Strategy 2 and Strategy 3 demonstrate nearly identical performance and Strategy 1 (Root median) performs worst, with:

- **Lowest speedup** (due to load imbalance)
- **Poor efficiency** (some processes get overloaded)

Given that Strategy 1 relies on a single process's data, which may select a bad pivot for skewed distributions, causing Uneven splits (some group of processes get most of the data) and increased communication overhead. Strategies 2/3 use global information (median/mean of medians), leading to balanced partitions and better workload distribution. The Figure 3 and 4 shows how the exchange step causes work-load imbalance in our pivot 1 implementation:



pivot
exchange and merge(extra space needed)
Exchange threshold
malloc called here for extra space
pivot
pivot
exchange and merge(extra space needed)
pivot

Figure 3: The left figure shows that under the backwards scenario, the pivot method 1 leads to load imbalance, where the right-side process receives more data. This results in longer merge times. Even if each merge is short, the additional overhead accumulates over multiple recursive steps. In contrast, the right figure shows a more balanced workload between the two processes when data is random distributed.

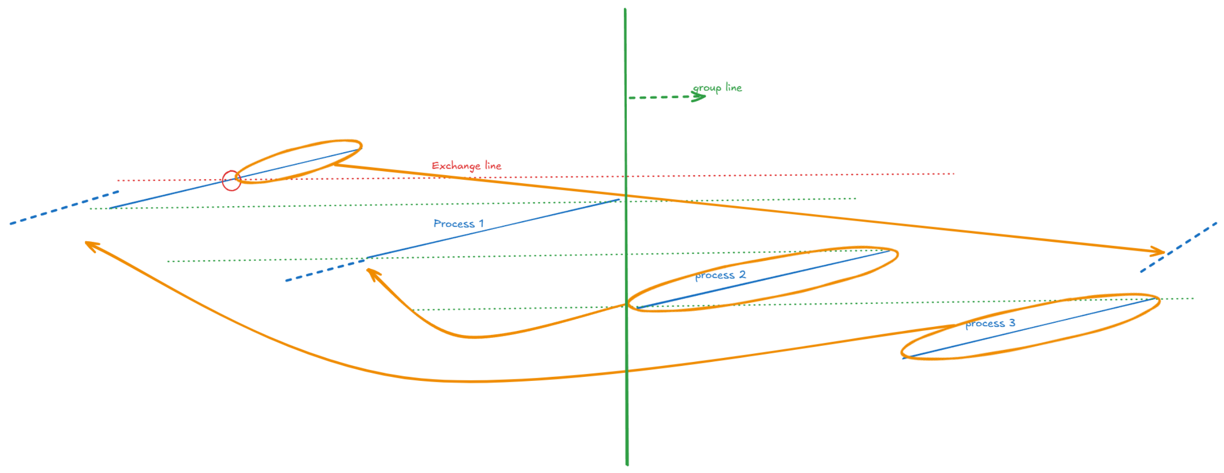


Figure 4: As shown in the figure, the left group of processes received all the data from the right-side processes, while the right group only received half of the data from the root process. This resulted in severe load imbalance in merging and data exchange.

5.3 Decline in overall parallel efficiency:

Apart from the phenomena mentioned above, we also observed that for both weak scaling and strong scaling, the parallel efficiency decreases as the number of processes increases. For example, with random input size $N = 250M$, the efficiency declined from 96%(2 processes) to 74.5%(16 processes) for pivot 1. In our code implementation, there are few bottlenecks:

- **Frequent memory allocation and deallocation:** After each data exchange, merge_data allocates a new array using malloc and releases it with free, leading to memory fragmentation and overhead from frequent system calls.
- **Memcpy:** During merging, we copy the retained data into a temporary array, and the use of memcpy introduces additional experimental overhead, which is expected to incur an $O(n)$ time complexity cost.
- **Communication overhead and synchronization delays:** MPI_Sendrecv is a blocking communication operation, which may cause certain processes to experience additional waiting. Moreover, each recursive call to MPI_Comm_split and MPI_Comm_free introduces internal synchronization overhead within MPI.

6 Conclusion

The observed outperformance of the backwards dataset over the random dataset likely stems from a combination of factors related to using the C library's qsort and compiler optimizations. The glibc qsort is not a pure quicksort, but rather a hybrid, typically merge-sort-based one that benefits greatly from being close to sorted or reversed data. This advantage is amplified by optimized memory access patterns in qsort (such as swapping data in larger chunks) and the -O3 compiler flag, which unrolls the merge loop into efficient SIMD instructions. Crucially, for reversed inputs, the merge process becomes highly predictable (e.g., elements in one subarray are always smaller or larger than elements in another subarray), enabling near-perfect branch prediction in the CPU, greatly reducing pipeline stalls and further improving performance.

Pivot quality significantly impacts performance for non-uniform data, but matters less for perfectly random distributions. Strategy 1 is the simplest with lowest overhead, but should be used only when data randomness is guaranteed. Strategy 2/3 consistently deliver better load balancing and higher parallel efficiency across different dataset distributions.

References

[1] GNU Project. glibc: qsort implementation source code, 2025. Accessed on 2025-05-10.

[2] Wikipedia contributors. Single instruction, multiple data, 2025. Accessed on 2025-05-10.

[3] CMU. Branch prediction. https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module7.4.1-branch-prediction.pdf, 2013. Lecture notes, ECE 740: Computer Architecture, Carnegie Mellon University, Fall 2013. Accessed: 2025-05-12.