Hybrid Distributed Sort with Bitonic Interchanges

Georgios Rousomanis (10703)
Department of Electrical and Computer Engineering
Aristotle University of Thessaloniki
rousoman@ece.auth.gr

July 2025

Introduction

Sorting large datasets efficiently in distributed memory environments is a fundamental problem in parallel computing. This report presents an implementation of a distributed sorting algorithm based on the Bitonic sort paradigm, leveraging MPI for inter-process communication. The total input array of size $N = 2^{p+q}$ is partitioned across 2^p processes, each handling 2^q integers locally. The sorting proceeds through an initial local sort phase followed by iterative bitonic merging stages involving structured data exchanges between processes. Our implementation also incorporates optimizations such as non-blocking communication with buffer splitting and OpenMP-based parallelism within each process to accelerate local sorting. Performance evaluation on the Aristotelis HPC cluster examines the scalability and speedup characteristics across varying numbers of processes and input sizes, providing insights into the algorithm's behavior and bottlenecks.

1 Algorithm Overview

The implemented algorithm is a hybrid distributed sorting method that combines local sorting with parallel Bitonic merging across multiple MPI processes. Each of the 2^p MPI processes receives a block of 2^q integers. The parameter s controls the communication buffer size, dividing each local array into 2^{q-s} chunks for non-blocking communication during the pairwise exchange phase.

1.1 Stages of the Algorithm

The algorithm proceeds in three main stages:

1. **Initial Alternating Sort:** Each process locally sorts its data. Processes with even ranks sort in ascending order, while processes with odd ranks sort in descending order, thus preparing a "bitonic" sequence across the distributed data.

2. **Bitonic Merging with Pairwise Exchanges:** The global merge is performed in $\log_2(2^p) = p$ stages. At each stage, every process exchanges data with a dynamically selected "partner" process, computed using bitwise XOR: $partner = rank \oplus 2^k$, where k depends on the step within the stage. This XOR operation flips the k-th bit of the rank, effectively connecting processes whose ranks differ by a power of two. The process graph formed in this way is a p-dimensional hypercube, where each edge represents communication between processes at Hamming distance 1, 2, 4, etc. As the algorithm progresses, the sorting propagates along these hypercube edges, gradually enforcing global order.

3. **Elbow Sort:** After each merging stage, each process performs an "elbow sort" to fully merge its local array into a sorted sequence, starting from the smallest (or largest) element and expanding outward using a two-pointer merging technique.

The algorithm records timing information for the local sort, communication phases, elbow sort, and total execution time. Synchronization between processes is enforced via `MPI_Barrier` calls to ensure correct timing measurements and data consistency.

1.2 Pseudocode

The following pseudocode summarizes the main steps of the distributed bitonic sort:

Algorithm 1 Distributed Sort with Bitonic Interchanges

Require: Local data array *local_data* of size 2^q
Require: Number of processes $P = 2^p$, buffer size $B = 2^s$, process rank r
Ensure: Globally sorted data distributed across all processes

```
if r mod 2 = 0 then
    SortAscending(local_data)
else
    SortDescending(local_data)
end if
for stage = 1 to log_2(P) do
    chunk ← ⌊r/(P/2^stage)⌋
    ascending ← (chunk mod 2 = 0)
    for step = stage − 1 to 0 do
        partner ← r ⊕ (1 ≪ step)
        if r ≥ partner then
            NonBlockingCommunication(local_data, partner, B)
        else
            AsyncReceiveAndExchange(local_data, partner, B, ascending)
        end if
        Barrier
    end for
    ElbowSort(local_data, ascending)
    Barrier
end for
```

1.3 Communication and Data Exchange

The core of the distributed Bitonic Sort lies in the communication and data exchange between MPI processes during the merging phases. At each step within a stage, a process determines its partner via the XOR operation $partner = rank \oplus 2^k$. This operation reflects a structured traversal of a hypercube topology, where each dimension corresponds to one communication round. The result is a classification network where processes interact with others that differ in exactly one bit position of their binary rank. This structure enables a scalable and regular communication pattern.

Data is exchanged in chunks of size 2^s, resulting in 2^{q-s} buffer splits per process. Non-blocking primitives (`MPI_Isend`, `MPI_Irecv`) are used to overlap communication and computation. The abstract functions NonBlockingCommunication and AsyncReceiveAndExchange encapsulate these steps, enabling higher-ranked processes to initiate sending early, while lower-ranked processes perform receives and apply directional merging logic.

This design ensures bandwidth-efficient data exchange while preserving the required bitonic structure for correct merges at each stage. Global consistency is maintained via barriers that synchronize all processes before moving to the next stage of the algorithm.

2 Validation of the Sorting Algorithm

To ensure correctness of the distributed bitonic sort implementation, a validation function is executed after the sorting completes. The validation performs two key checks:

1. **Local Sorted Order:** Each process verifies that its local data chunk is sorted in non-decreasing order. This is done by asserting that every element is less than or equal to its successor within the local array.

2. **Global Sorted Order at Process Boundaries:** Since the global sorted order spans all processes, the algorithm must confirm the order between adjacent processes. Each process, except the last, sends its last (maximum) element asynchronously to the next higher-ranked process. Each process, except the first, receives the last element from the preceding process and asserts that this received value is less than or equal to its own first (minimum) element.

This two-level validation ensures that the data is correctly sorted both locally within each process and globally across the distributed dataset. Any violation triggers an assertion failure, signaling an error in the sorting or communication phases.

3 Performance Overview

This section provides an overview of the performance characteristics of the distributed bitonic sort algorithm, focusing on how it scales with the number of processes and the size of the input array. We examine both the speedup achieved through parallelization and the scalability of total execution time across different configurations.

3.1 Speedup by Number of Processes

Table 1 illustrates the speedup achieved by increasing the number of processes for various total array sizes, where the size is given by 2^{p+q}. Each row corresponds to a fixed array size, and each column indicates the number of processes used (2^p). The first valid value of each row (always 1.00) represents the baseline execution time. The subsequent values in the row show the speedup achieved by distributing the workload across 2^p processes.

Across all array sizes, a consistent pattern emerges: the first duplication of the number of processes typically results in a near 2× speedup, indicating efficient initial parallelization. However, as the number of processes increases further, the incremental speedup diminishes. This is attributed to communication overheads and synchronization delays introduced during pairwise exchanges and elbow sorting steps of the bitonic sort algorithm.

$p+q$	$p=0$	$p=1$	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$	$p=7$
20	1.00	-	-	-	-	-	-	-
21	1.00	1.64	-	-	-	-	-	-
22	1.00	1.89	3.16	-	-	-	-	-
23	1.00	1.80	3.29	5.59	-	-	-	-
24	1.00	1.98	3.60	6.17	10.28	-	-	-
25	1.00	1.92	3.26	5.90	10.22	16.60	-	-
26	1.00	1.97	3.53	6.28	10.26	17.49	29.26	-
27	1.00	1.81	3.37	5.94	10.44	17.14	29.87	44.78
28	-	1.00	1.84	3.20	5.54	9.84	16.27	24.72
29	-	-	1.00	1.79	3.09	5.47	9.15	13.97
30	-	-	-	1.00	1.76	2.92	5.05	7.95
31	-	-	-	-	1.00	1.72	2.91	4.57
32	-	-	-	-	-	1.00	1.71	2.68
33	-	-	-	-	-	-	1.00	1.58
34	-	-	-	-	-	-	-	1.00

Table 1: Speedup achieved by increasing the number of processes for various total array sizes (2^{p+q}).

3.2 Scalability with Global Array Size

Figure 1 shows how the total execution time scales with the total number of elements 2^{p+q}, while keeping the number of elements per process constant at 2^q. This effectively evaluates strong scaling by holding the local problem size fixed and increasing the total problem size and number of processes proportionally.

We observe that although execution time increases as the global array grows, the rate of increase is significantly sublinear, indicating effective parallel scaling. This is because more processes are recruited to handle the growing workload, reducing the per-process computational burden. However, the increase in total execution time is not negligible, due to factors such as inter-process communication and global coordination required by the bitonic merge steps.

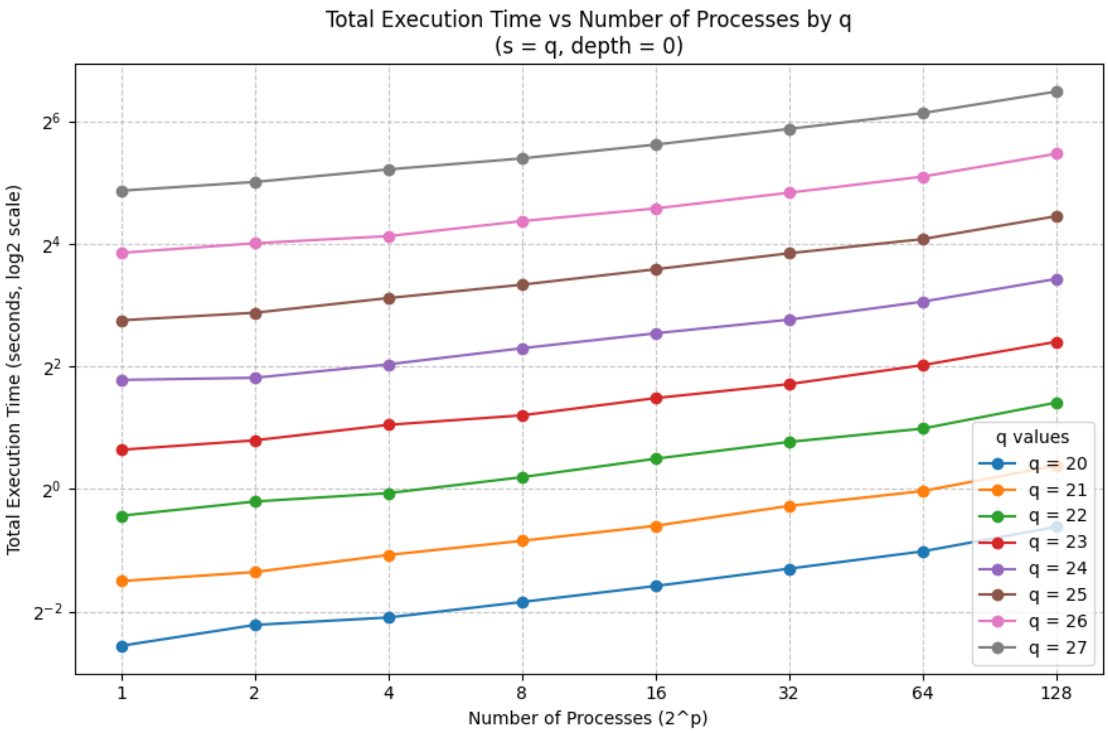

Figure 1: Total execution time vs. number of processes (or global size 2^{p+q}) for fixed per-process size 2^q.

4 Optimization of the Algorithm

4.1 Time Breakdown

Before applying any optimizations, we analyzed how the total execution time is distributed across the different stages of the algorithm. Benchmarks were conducted on the Aristotelis HPC cluster using 8 compute nodes. Figure 2 presents the execution time versus the number of processes 2^p, for a fixed total input size of $2^{p+q} = 2^{27}$ integers.

We observe that for small process counts, the majority of execution time is spent on the *initial sort* phase, which is performed locally on each process. As the number of processes increases, however, the cost of *pairwise communication* grows significantly and eventually becomes the dominant factor, especially when $2^p = 128$. This trend is also reflected in Table 2, which shows the percentage breakdown of time spent in each stage.

As already mentioned, doubling the number of processes from 1 to 2 results in a near-perfect speedup due to effective parallelization of the local sorting phase. However, further doubling leads to diminishing returns. This is because the computational load per process decreases, while communication overhead becomes increasingly significant. Additionally, the complexity of synchronization and data exchange across many processes further hampers scalability.

Processes	Initial Sort (%)	Pairwise Sort (%)	Elbow Sort (%)	Other (%)
1	**100.00**	0.00	0.00	0.00
2	**92.82**	2.60	4.57	0.01
4	**81.74**	6.50	11.76	0.01
8	**70.68**	12.17	17.14	0.01
16	**59.46**	19.91	20.62	0.01
32	**47.80**	30.54	21.63	0.03
64	**39.51**	37.67	22.78	0.04
128	30.06	**49.95**	19.92	0.07

Table 2: Execution Time Breakdown (total size: $2^{p+q} = 2^{27}$)

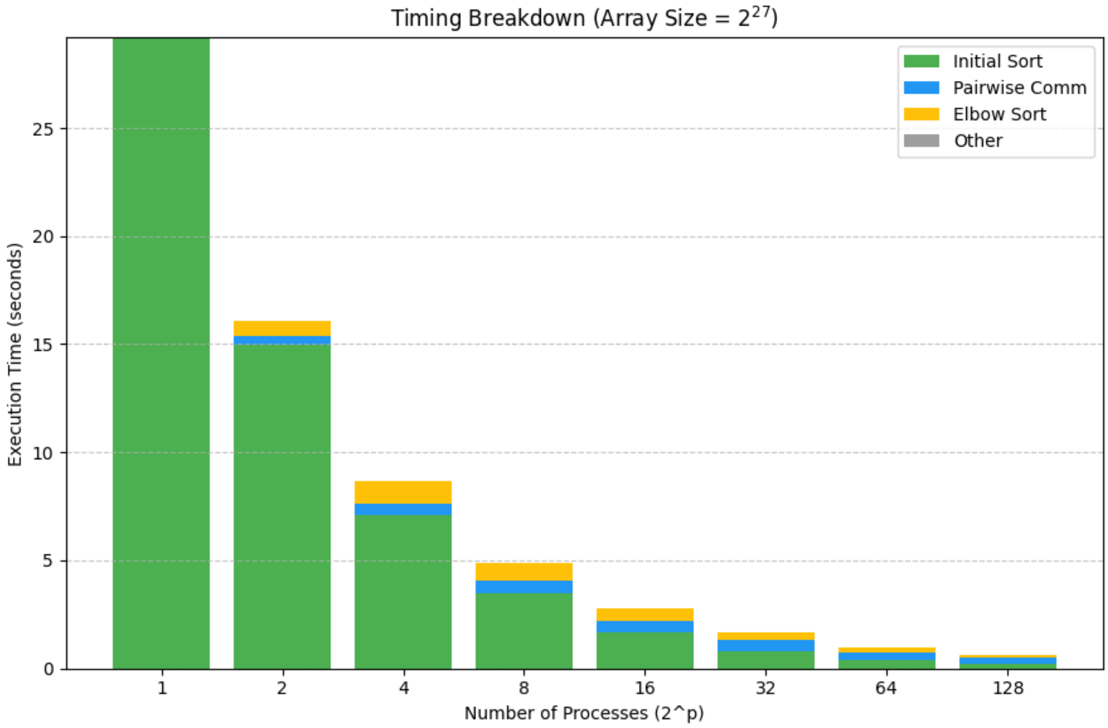

Figure 2: Execution time breakdown per stage for total size 2^{27}

4.2 Optimizing Pairwise Communication

As illustrated in the previous subsection, *pairwise communication* emerges as the primary bottleneck for large process counts. To address this, we experimented with splitting the communication buffer into smaller segments. The goal of this strategy is to overlap communication with computation and enhance performance scalability.

Figure 3 presents the pairwise communication time as a function of the number of processes for various buffer split configurations, where the number of segments is given by $B = 2^{q-s}$. Splitting the communication buffer into two parts consistently improves communication performance across all process counts. However, further splitting into four segments does not provide additional benefit.

An interesting pattern emerges in Figure 3. The pairwise communication time peaks when the number of processes equals the number of compute nodes (8). This configuration forces each process to run on a separate node, resulting in all communication being inter-node and hence more expensive due to reliance on the network interface. As we increase the number of processes beyond 8, multiple processes begin to share the same node. This leads to more intra-node communication, which is significantly faster than inter-node communication, thereby reducing the total communication cost.

Table 3 summarizes the speedup in pairwise communication and total execution time when using two and four buffer splits, relative to the baseline configuration ($s = q$). The maximum speedup observed for communication is about 33%, while for total execution time, it does not exceed 8%. Additionally, speedup diminishes as the number of processes increases. This is because the overhead from communication and synchronization grows, while the benefits of buffer splitting plateau.

In conclusion, while buffer splitting offers some performance improvements in specific configurations, it alone is insufficient to achieve scalability at high process counts. These findings suggest that further optimizations should focus on reducing the local sorting overhead, which becomes increasingly dominant as communication efficiency saturates.

4.3 Optimizing Local Sorting

As previously discussed, local sorting constitutes the primary performance bottleneck, particularly when the number of processes is small. To mitigate this overhead, we implemented a parallel local sorting routine using OpenMP. This optimization aims to exploit multicore parallelism within each process, significantly accelerating the initial sorting phase while keeping the implementation

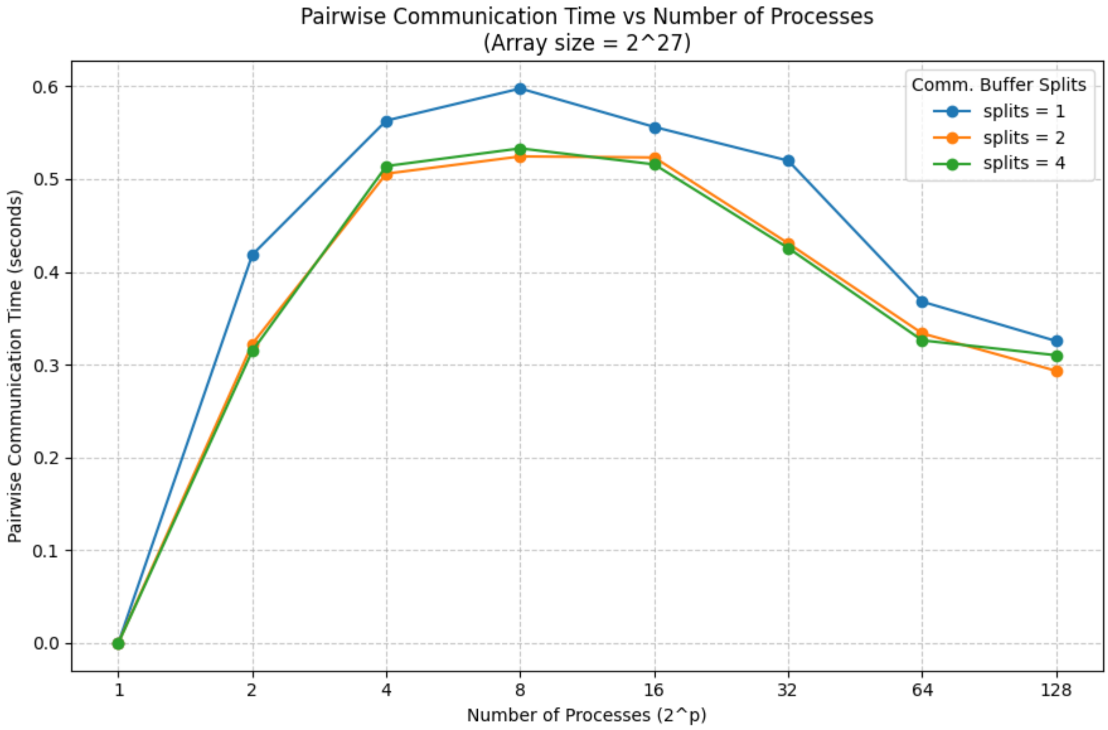

Figure 3: Pairwise communication time vs number of processes for varying communication buffer splits ($B = 2^{q-s}$).

Processes	Comm Speedup		Total Speedup	
	splits: 2	**splits: 4**	**splits: 2**	**splits: 4**
2	1.30	1.33	1.06	1.07
4	1.11	1.10	1.03	1.01
8	1.14	1.12	1.03	1.03
16	1.06	1.08	1.04	1.04
32	1.21	1.22	1.08	1.07
64	1.10	1.13	1.04	1.04
128	1.11	1.05	1.05	1.02

Table 3: Speedup of communication and total execution time for different split counts relative to the baseline ($s = q$)

relatively simple and portable.

The implemented approach is based on a parallel merge sort algorithm. It recursively splits the array into subarrays and sorts them concurrently using OpenMP threads. Once the subarray size is small enough or the recursion depth becomes too large, the algorithm falls back to a serial sort. This hybrid strategy balances parallel performance with overhead control, ensuring efficiency across a wide range of input sizes and system architectures. The implementation also supports both ascending and descending ordering to remain compatible with the alternating sorting stages required by bitonic sort.

Table 4 and Figure 4 present the performance benefits of applying OpenMP-based parallelism to the local sorting phase in our distributed bitonic sort implementation. The benchmarks were conducted using a total problem size of 2^{27} elements, varying the number of processes as 2^p. We evaluated two levels of parallel recursion depth (1 and 2) in the OpenMP merge sort and measured speedup separately for the local sorting time and the overall execution time.

As the number of processes increases, the size of the local data per process decreases, which reduces the effectiveness of thread-level parallelism. Nevertheless, for smaller process counts (larger per-process workload), we observe significant speedups in both local sorting and total execution time. The gains are more pronounced in the initial sorting phase, as the impact of communication overhead becomes more dominant at higher process counts. These observations demonstrate the effectiveness of hybrid parallelism, especially in scenarios where local computation remains a substantial portion of the overall workload.

Processes	**Speedup (Depth 1)**		**Speedup (Depth 2)**	
	Initial Sort	**Total**	**Initial Sort**	**Total**
1	1.83	1.83	1.84	1.84
2	1.92	1.82	1.93	1.83
4	1.92	1.65	1.79	1.55
8	1.88	1.51	1.86	1.51
16	1.82	1.37	1.84	1.38
32	1.80	1.31	1.79	1.30
64	1.77	1.21	1.72	1.20
128	1.66	1.14	1.59	1.12

Table 4: Speedup from OpenMP parallelism in local sorting for total problem size 2^{27} and varying number of processes.

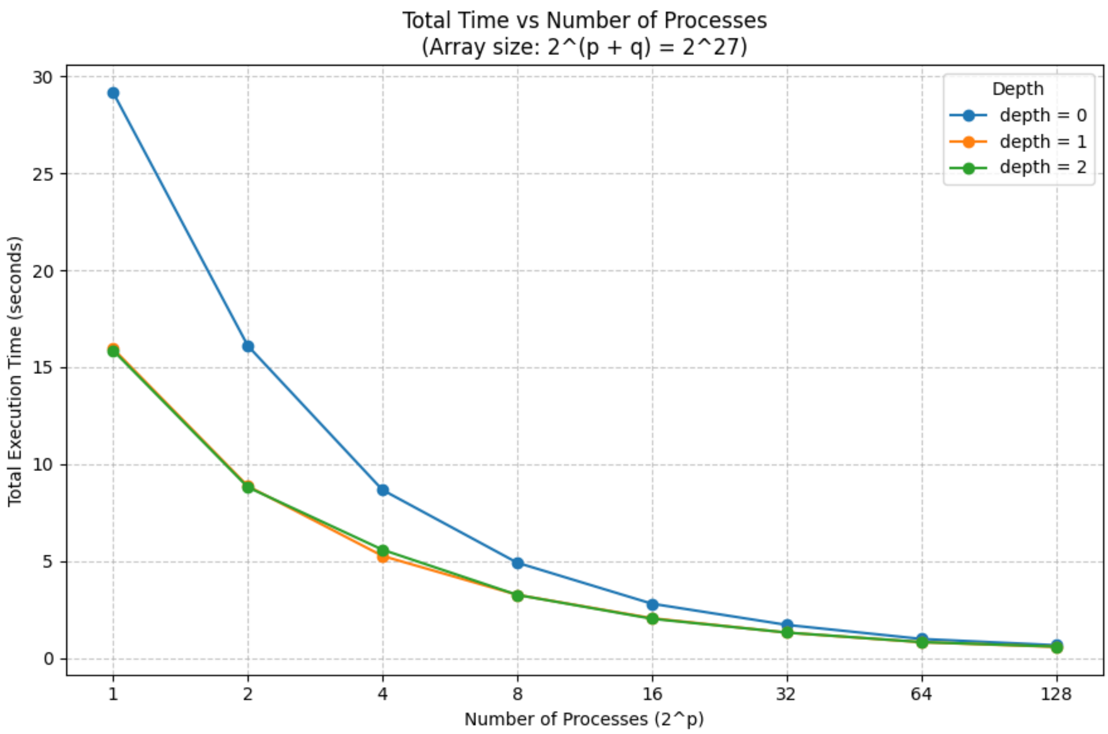

Figure 4: Total execution time versus number of processes (2^p) for different OpenMP recursion depths, for total input size 2^{27}.

Conclusions & Future Work

This work demonstrates that a hybrid distributed Bitonic sort can effectively leverage both inter-node parallelism through MPI and intra-node parallelism via OpenMP to improve sorting performance for large datasets. While initial local sorting dominates execution time for small process counts, communication overhead becomes the primary bottleneck as the number of processes increases. Techniques such as buffer splitting yield modest improvements in communication efficiency, but the scalability remains limited by inter-process data exchanges. Incorporating OpenMP parallelism in local sorting provides significant speedups, particularly when each process has a substantial workload. For future work, exploring GPU acceleration with CUDA for the initial local sorting phase promises further performance gains by exploiting massively parallel architectures within each compute node, potentially alleviating the local sorting bottleneck and improving overall scalability.

Acknowledgments

The experiments presented in this work were conducted using the Aristotelis HPC cluster at Aristotle University of Thessaloniki (AUTH). We gratefully acknowledge the computational resources and support provided by AUTH.