# A Hybrid Parallel-Sequential Approach to Distributed Bitonic Sort using MPI

Fraidakis Ioannis

December 2024

**Abstract**

This research presents a detailed implementation and analysis of the Bitonic Sort algorithm using MPI. We investigate the algorithm's theoretical foundations, computational complexity, and practical performance characteristics, focusing on scalability and communication efficiency. Recognizing the potential for communication overhead in purely parallel implementations, we explore a hybrid sequential-parallel approach. By systematically evaluating the algorithm across varying process counts and data sizes on a high-performance computing cluster, we demonstrate its potential for scalable parallel data organization.

## 1 Introduction

In the era of big data, the ability to efficiently sort massive datasets is paramount across various scientific and industrial applications. As data volumes continue to grow, traditional sequential sorting algorithms become a bottleneck. Distributed sorting algorithms, which leverage the power of parallel computing, offer a solution by distributing the workload across multiple processing units, enabling efficient handling of large-scale data.

This report focuses on the implementation and analysis of the Bitonic Sort algorithm in a distributed memory environment using the Message Passing Interface (MPI). Bitonic Sort, a comparison-based sorting algorithm, is particularly well-suited for parallel architectures due to its regular and structured communication patterns. However, purely parallel implementations can suffer from communication overhead. To address this, we investigate a hybrid approach that combines sequential and parallel sorting techniques.

More specifically, this report details the implementation of a distributed Bitonic Sort designed to sort $N$ elements ($N$ is a power of 2) distributed across a system of $P = 2^p$ processes. Each process initially holds $N/P = 2^q$ elements. The algorithm proceeds in phases. Initially each process performs a local sort on its assigned data using an efficient sequential algorithm like Quicksort or Merge Sort. Subsequently, processes engage in a series of data exchanges and comparisons, coordinated by MPI communication, to achieve a globally sorted sequence. This hybrid strategy aims to minimize the amount of data exchanged during the parallel phases.

## 1.1 Theoretical background

### 1.1.1 Bitonic sequence

A **bitonic sequence** is a sequence of numbers $a_0, a_1, ..., a_{n-1}$ such that:

1. There exists an index $k$ $(0 \leq k \leq n-1)$ such that $a_0 \leq a_1 \leq ... \leq a_k$ and $a_k \geq a_{k+1} \geq ... \geq a_{n-1}$, or
2. The sequence is a cyclic rotation of a sequence satisfying condition 1.

For example:

- Example 1: {2, 9, 14, 16, 20, 23, **32**, 25, 20, 15, 10, 5} (Peak at 32)
- Example 2 (Cyclic Rotation): {27, 31, **25**, 20, 10, 3, 8, 12, 18} (Peak at 25, rotated)

### 1.1.2 Bitonic Sort

**Bitonic Sort** is a comparison-based sorting algorithm that sorts a sequence by transforming it into a bitonic sequence and then merging it into a sorted sequence. It operates in two main phases:

1. **Bitonic Sequence Construction:** Given an arbitrary input sequence of size $(n = 2^k)$ (where $k$ is an integer), this phase constructs a bitonic sequence. It recursively divides the sequence into smaller subsequences of size $2, 4, 8, \ldots, (n/2)$. These subsequences are alternately sorted in ascending and descending order using the bitonic merge process described in step 2. Specifically:
   - For a subsequence of size $2^j$, the elements are grouped into pairs, and the bitonic merge operation is applied to sort these pairs.
   - After each merge operation, the subsequences are doubled in size and alternately sorted (ascending and descending) until the entire sequence forms a bitonic sequence.
2. **Bitonic Merge (or Bitonic Merge Network):** Once a bitonic sequence of size $n$ is obtained, this phase sorts it. The merge operation consists of $(\log_2 n)$ stages. In each stage $i$ (where $0 \leq i < \log_2 n$), for elements at a distance of $2^i$:
   - Compare and swap the elements to split the sequence into two subsequences:
     (a) The first contains the smaller elements.
     (b) The second contains the larger elements.
   - The comparisons are performed such that the resulting subsequences remain bitonic.
3. **Output:** The resulting sequence is sorted in the desired order (ascending or descending).

**Why is Bitonic Sort Useful for Parallel Processing?**
Bitonic Sort is well-suited for parallel processing due to:

- **Parallel Comparisons:** Within each stage of the bitonic merge, all comparisons can be performed independently and concurrently.
- **Regular Communication Pattern:** The algorithm exhibits a structured and predictable communication pattern, which maps efficiently to hypercube networks and other interconnection topologies commonly found in parallel computers. This makes it suitable for both shared and distributed memory systems.
- **Parallel Time Complexity:** While its serial complexity is $O(n \log^2 n)$, its parallel complexity is reduced to $O(\log^2 n)$ due to concurrent execution.[1]

## 1.2 Comparative Algorithm Analysis

While serial time complexity of Bitonic Sort exceeds the time complexity of algorithms such as Merge Sort, its parallel execution fundamentally challenges the traditional $O(n \log n)$ complexity barrier of optimal sequential sorting algorithms. It has the potential to achieve near-linear speedup, although practical constraints typically introduce some overhead.

| Algorithm | Complexity | Communication | Scalability |
|---|---|---|---|
| Quicksort | $O(n \log n)$ | **High** (frequent pivot-based exchanges) | **Moderate** (potential load imbalance) |
| Merge Sort | $O(n \log n)$ | **Moderate** (coordination for merging data) | **Good** (evenly distributed tasks) |
| Bitonic Sort | $O(\log^2 n)$ | **Low** (efficient hypercube pairwise exchanges) | **Excellent** (uniform and predictable) |

Table 1: Different Sorting Algorithms' Characteristics

# 2 Efficiency Strategies

## 2.1 Hybrid Approach: Combining Sequential and Parallel Algorithms

In our approach, we utilize a hybrid sorting strategy that combines the strengths of both parallel and sequential algorithms. Rather than solving entire sorting problem in parallel, we start by sorting the sequence sequentially using an efficient algorithm. Once the sequence reaches a certain point, where parallelism becomes more beneficial, we switch to a parallel sorting algorithm to finish the task. This hybrid approach significantly reduces the communication overhead inherent in parallel processing, as only the later stages of the sorting process—where parallelism offers substantial performance gains—are executed in

[1] While Parallel Bitonic Sort has a theoretical time complexity of $O(\log^2 n)$ with $O(n)$ processes, this requirement is often impractical for large datasets.

parallel. This strategy helps balance the tradeoff between the efficiency of sequential algorithms for smaller tasks and the scalability of parallel algorithms for larger datasets.

## 2.2 Hypercube Communication Topology

### 2.2.1 Topological Structure

In an $p$-dimensional hypercube, there are $P = 2^p$ processes, and each process has $n$ direct connections (neighbors). The partner selection mechanism uses bitwise XOR operations. To communicate along dimension $d$ $(0 \leq d < p)$, the partner's rank is calculated as:

$$partner\_rank = rank \oplus 2^d \tag{1}$$

### 2.2.2 Communication Efficiency in Hypercubes

The hypercube topology is particularly well-suited for Bitonic Sort because it minimizes communication overhead. The communication pattern of Bitonic Sort maps directly onto the hypercube's structure. Each stage $i$ $(0 \leq i < \log_2(n))$ of the Bitonic Merge corresponds to communication along dimension $i$ of the hypercube. This means that elements at a distance of $2^i$ in the sequence communicate with processes whose ranks differ in the $i$-th bit. This logarithmic scaling is crucial for achieving good scalability.

### 2.2.3 Communication Pattern Visualization

The following figures illustrate 2-dimensional and 3-dimensional hypercubes:

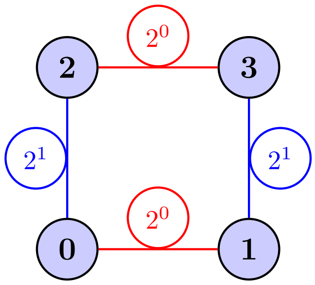

Figure 1: 2-Dimensional Hypercube

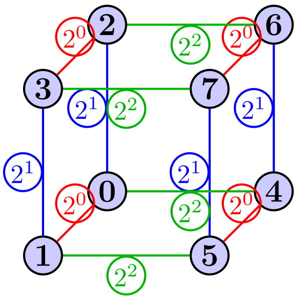

Figure 2: 3-Dimensional Hypercube

## 2.3 Non-Blocking Communication

Non-blocking MPI functions, such as `MPI_Isend` and `MPI_Irecv`, allow the process to continue with other computations (e.g., the `CompareAndSwap` operation) while the communication is in progress, thus overlapping computation and communication and reducing idle time. The `MPI_Wait` function is then used to ensure that the communication has completed before the received data is used.

## 2.4 Load Balancing and Data Distribution

A uniform distribution of elements ensures that each process has a similar amount of work to do, preventing some processes from becoming idle while others are still working. This minimizes idle time and maximizes resource utilization.

# 3 Algorithm Design

The complete source code for the distributed Bitonic Sort implementation,, is available on GitHub at: `https://github.com/fraidakis/PDS_BitonicSort`.

## 3.1 Pseudocode for Bitonic Sort

**Algorithm 1** Distributed Bitonic Sort

```
procedure BitonicSort($data, size, numProcesses$)
    Perform LocalSort($data, size$)                          ▷ Initial local sorting using qsort
    for $dimension \leftarrow 1$ to $\log_2(numProcesses)$ do
        $isAscending \leftarrow$ getSortingDirection($rank, dimension$)
        for $distance \leftarrow 1 \ll (dimension - 1)$ downto 1 do     ▷ $2^{dimension-1}$
            $partner \leftarrow rank \oplus distance$
            $keep\_min \leftarrow (rank < partner) = isAscending$
            NonBlockingExchange($data, partner, keep\_min$)
        end for
        ElbowSort($data, size, isAscending$)
    end for
end procedure
```

**Explanation of `BitonicSort`:**

1. **Local Sort:** Each process initially sorts its local data using `qsort`. This ensures that each process starts with a sorted subsequence.

2. **Outer Loop (Dimensions):** The outer loop iterates through the dimensions of the hypercube (from 1 to $\log_2(numProcesses)$). Each iteration represents a phase of the Bitonic Sort algorithm.

3. **Sorting Direction:** The `isAscending` variable determines the sorting direction (ascending or descending) for the current phase and process.

4. **Inner Loop (Distances):** The inner loop iterates through the distances within the current dimension. These distances determine which processes communicate with each other.

5. **Partner Calculation:** The `partner` process rank is calculated using the bitwise XOR operation ($\oplus$) between the current process's rank and `distance`

6. **Keep Minimum/Maximum:** The `keep_min` variable determines whether the current process should keep the minimum or maximum of the compared elements. This is essential for forming the bitonic sequences.

7. **Non-Blocking Exchange:** The `NonBlockingExchange` function performs the data exchange and compare-swap between current process and partner.

8. **Elbow Sort**: After the communication phase for each dimension, the `ElbowSort` function is called to merge the local bitonic sequence into a sorted sequence.

## 3.2 Communication Strategy

**Algorithm 2** Non-Blocking Chunked Exchange

```
procedure NonBlockingExchange($data, partner, keep\_min$)
    $n_{chunks} \leftarrow$ GetNumChunks($size$)
    $chunk\_size \leftarrow size/n_{chunks}$
    for $chunk \leftarrow 0$ to $n_{chunks} - 1$ do
        Non-blocking receive of $chunk$ from $partner$
        Non-blocking send of $chunk$ to $partner$
        if $chunk > 0$ then    ▷ Process previous chunk while communication in progress
            CompareAndSwap(previous_chunk, $keep\_min$)
        end if
        Wait for send and receive of chunk $chunk$ to complete
    end for
    CompareAndSwap(last_chunk, $keep\_min$)
end procedure
```

**Explanation of `NonBlockingExchange`:**

1. **Chunking:** The data to be exchanged is divided into smaller chunks.
2. **Non-Blocking Communication:** Initiate non-blocking send and receive operations for each chunk. This allows the process to continue executing other code without waiting for the communication to complete.
3. **Overlapping Computation and Communication:** While the current chunk is being sent and received, the `CompareAndSwap` operation is performed on the **previously** received chunk (if there is one). This overlaps computation with communication, reducing idle time.
4. **Waiting for Completion:** Ensure that the send and receive operations for the current chunk have completed before proceeding to the next chunk.
5. **Processing the Last Chunk:** After the loop, the `CompareAndSwap` operation is performed on the last chunk.

## 3.3 Elbow Sort Implementation

We chose `elbowSort` over Bitonic Merge for sorting locally bitonic sequences due to its simplicity and efficiency in a sequential context. Unlike Bitonic Merge, which has a complexity of $O(n \log n)$, `elbowSort` achieves linear complexity $O(n)$ by leveraging the already sorted nature of the two halves of a bitonic sequence and performing a straightforward linear merge, similar to the final phase of Merge Sort. In our distributed implementation, these operations are run sequentially, with each process independently handling its portion of the sequence after the communication phases.

**Algorithm 3** Elbow Sort for Merging Bitonic Sequences

```
procedure ELBOWSORT($data, size, isAscending$)
    $elbow \leftarrow$ FINDELBOW(data, size)                ▷ Find transition point
    if $elbow = -1$ then
        Reverse data if needed
        return                                          ▷ Already sorted
    end if
    Initialize $merged$ array
    $left \leftarrow elbow$
    $right \leftarrow (elbow + 1) \bmod size$
    while not fully merged do          ▷ Merge until all elements are processed
        if first half was ascending then
            Merge $data[left]$ and $data[right]$ in descending order into $merged$
        else
            Merge $data[left]$ and $data[right]$ in ascending order into $merged$
        end if
        Update $left$ and $right$ indices (wrapping around if necessary)
    end while
    Copy $merged$ back to $data$ in the correct order based on $isAscending$
end procedure
```

**Explanation of `ElbowSort`:**

1. **Finding the Elbow:** The `FindElbow` function locates the ”elbow” or transition point in the sequence. This is the index where the sequence changes from increasing to decreasing (or vice versa). It also determines the initial trend of the sequence (ascending or descending) before the elbow.

2. **Handling Already Sorted Cases:** If no elbow is found, the sequence is already sorted. If it is in the reverse order of the desired, simply reverse it.

3. **Merging Process:**

   - The algorithm initializes two indices, $left$ (starting at the elbow) and $right$ (starting at the element after the elbow).
   - It then iteratively merges the two parts of the sequence, comparing the elements at $data[left]$ and $data[right]$ and placing the appropriate element (larger if the first half was ascending, smaller if the first half was descending) into a temporary *merged* array.
   - The indices $left$ and $right$ are updated in each iteration, wrapping around the array using the modulo operator to handle the cyclic nature of the sequence.

4. **Copying Back:** Finally, the contents of the *merged* array are copied back to the original *data* array. The order of copying depends on the desired sorting direction.

**Example:**

Consider the cyclic bitonic sequence: $\{8, 12, 18, 24, 27, 31, 25, 20, 15, 10\}$. The elbow is at index 7 (between 31 and 25). The algorithm would then merge the subsequences $\{31, 25, 20, 15, 10\}$ and $\{27, 24, 18, 12, 8\}$ to produce the final sorted sequence.

# 4 Performance Analysis

## 4.1 Experimental Setup

| Parameter | Range | Description |
|---|---|---|
| Processes ($P = 2^p$) | $2^0$ to $2^7$ | $P$ is the number of processes |
| Local Elements ($2^q$) | $2^{20}$ to $2^{27}$ | Data volume per process ($2^q$ elements per process) |
| Total Elements ($N = 2^{p+q}$) | $2^{20}$ to $2^{34}$ elements | Global dataset ($N$ is the total number of elements) |
| Platform | Aristotelis Cluster | AUTH high-performance computing environment |
| Partition | Rome | Cluster's partition for MPI tasks[2] |

Table 2: Experimental Parameters

## 4.2 Performance Metrics

Key metrics include:

- **Total Execution Time:** $T_{\text{total}}$ is the total time taken for the program to complete execution, encompassing all phases. This serves as a baseline metric for comparing the overall performance of the implementation.

- **Speedup:** $S(P) = \frac{T_1}{T_P}$, where $T_1$ is the execution time on a single process (sequential execution), and $T_P$ is the execution time on $P$ processes. Speedup quantifies the reduction in execution time achieved by using multiple processes. Ideally, as the number of processes increases, $S(P)$ should approach $P$, representing linear scalability.

- **Efficiency:** $E(P) = \frac{S(P)}{P}$. Efficiency measures how effectively the available processes are utilized. An efficiency of 1 (or 100%) indicates perfect utilization, where each additional process contributes fully to reducing execution time.

- **Communication Overhead Ratio:** $T_{\text{comm}}/T_{\text{total}}$, where $T_{\text{comm}}$ is the total communication time, and $T_{\text{total}}$ is the total execution time.

- **Local Sort Ratio:** $T_{\text{local}}/T_{\text{total}}$, where $T_{\text{local}}$ is the time spent sorting each process's local data segment.

- **Elbow Sort Ratio:** $T_{\text{elbow}}/T_{\text{total}}$, where $T_{\text{elbow}}$ is the time taken to merge locally sorted segments.

## 4.3 Experimental Results

### 4.3.1 Speedup and Efficiency with Varying Process

Figure 3 presents the speedup and efficiency achieved by varying the number of processes while maintaining a constant total data size of $2^{30}$. The analysis highlights the scalability of the parallel sorting algorithm as the process count increases.

[2] System Specifications: 17 nodes, each with 128 CPUs and 256 GB of RAM. CPU model: AMD EPYC 7662. Interconnect: 200 Gb/s InfiniBand HDR.

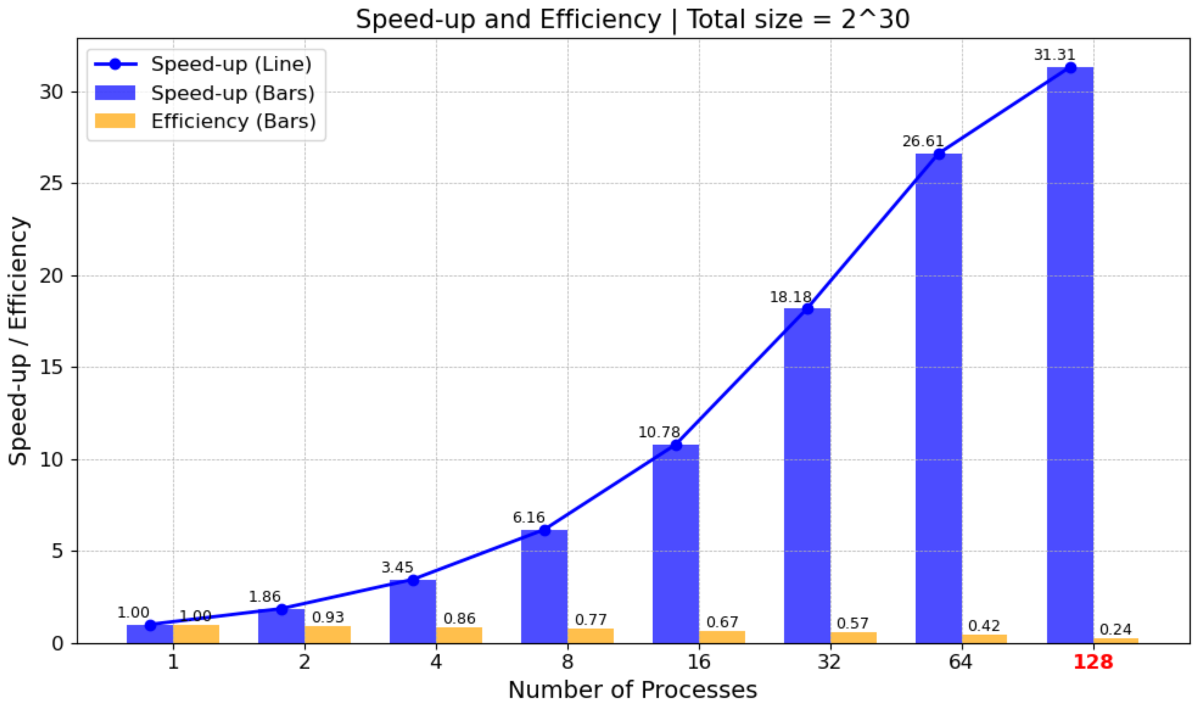

Figure 3: Speedup and efficiency relative to a single process baseline.

**Discussion of Results: Speedup and Efficiency Analysis** The results presented in Figure 3 reveals key insights into the trends of speedup and efficiency.

**Decreasing Efficiency with Increasing Processes.** As the number of processes increases, the efficiency decreases substantially. For example, efficiency drops from 100 % with 1 process to 24 % with 128 processes. This trend can be attributed to two factors:

- **Communication Overhead:** As the number of processes grows, the dataset is divided into smaller chunks, leading to a significant increase in the frequency and volume of inter-process communication. These communication delays, including message passing latency and synchronization overhead, can eventually dominate the overall runtime, especially with a large number of processes.
- **Diminishing Returns:** When the workload is split across many processes, the computation handled by each process becomes smaller. Beyond a certain point, the time saved by parallelizing further is outweighed by the time spent on coordination and synchronization. This leads to diminishing returns in speed-up, where adding more processes results in a disproportionally smaller improvement in performance.

**Speedup Trend.** The results also show that speedup increases with the number of processes but fails to achieve ideal linear scaling. For instance, speedup grows from 1.00 with 1 process to 26.61 with 64 processes, which is far below the ideal speedup of 64.

- **Saturation Point:** Beyond a certain number of processes, the cost of communication dominates, reducing the incremental benefit of adding more processes. This saturation is more pronounced for smaller datasets, where the workload per process is insufficient to justify the additional overhead. See more in Section 4.3.3.

### 4.3.2 Scaling with Dataset Size and Process Count

The performance of the distributed Bitonic Sort algorithm was further evaluated by varying the dataset size for different number of processes.

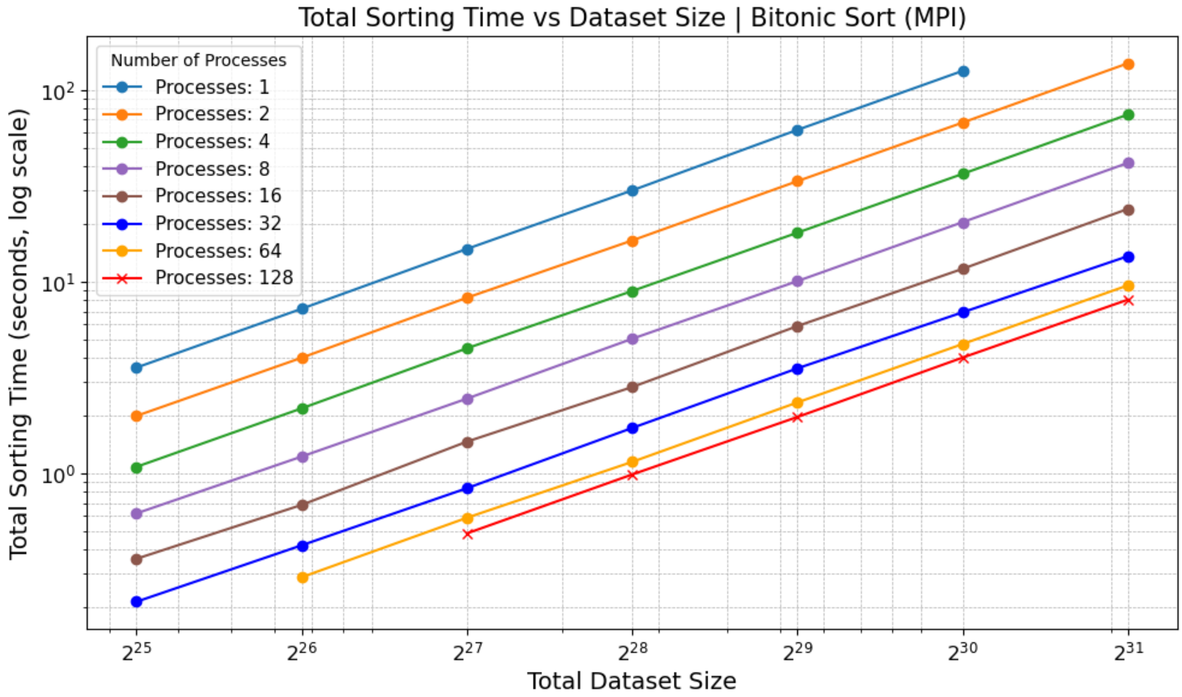

Figure 4: Total Sorting Time vs. Dataset Size for Different Process Counts

**Discussion of Results: Scaling with Dataset Size and Process Count** The figure illustrates the **sorting time** as a function of the **total dataset size**, measured across various numbers of processes. Both axes are plotted on a **logarithmic scale**.

**Trend on Log-Log Scale.** The straight-line relationship between total sorting time and dataset size on the log-log scale indicates a **power-law relationship** of the form $Time = c \cdot N^k$, where $k$ is the slope of the line. This relationship arises because taking the logarithm of both sides of the equation gives:

$$\log(Time) = \log(c) + k \cdot \log(N),$$

which is a linear equation with slope $k$ and intercept $\log(c)$.

- Upon examining the slopes of the lines, which are close to 1, we can conclude that the time **complexity scales as** $\approx \Theta(N)$ for each process count, meaning that doubling the dataset size approximately doubles the sorting time.
- The number of processes affects the constant factor $c$ in the power-law relationship. As the number of processes increases, $c$ generally decreases, indicating a reduction in the sorting time for a given dataset size. However, this decrease in $c$ becomes less pronounced at higher process counts. Thus, while increasing the number of processes reduces the absolute sorting time, the overall scaling behavior with respect to dataset size remains $\Theta(N)$.

**Impact of Process Count.** As discussed in §4.3.1 (Decreasing Efficiency with Increasing Processes), these results further confirm that:

- While smaller process counts (e.g., 2, 4, 8) yield significant reductions in sorting time due to parallelization, the speedup becomes less pronounced beyond 32 processes.

### 4.3.3 Performance Breakdown by Component

The performance of the distributed Bitonic Sort was analyzed by breaking down the total execution time into its constituent components: Local Sort, Elbow Sort, and Communication. Figure 5 visually depicts the time spent in each component for varying numbers of processes and constant dataset size ($2^{30}$). Table 3 provides the corresponding values:

S

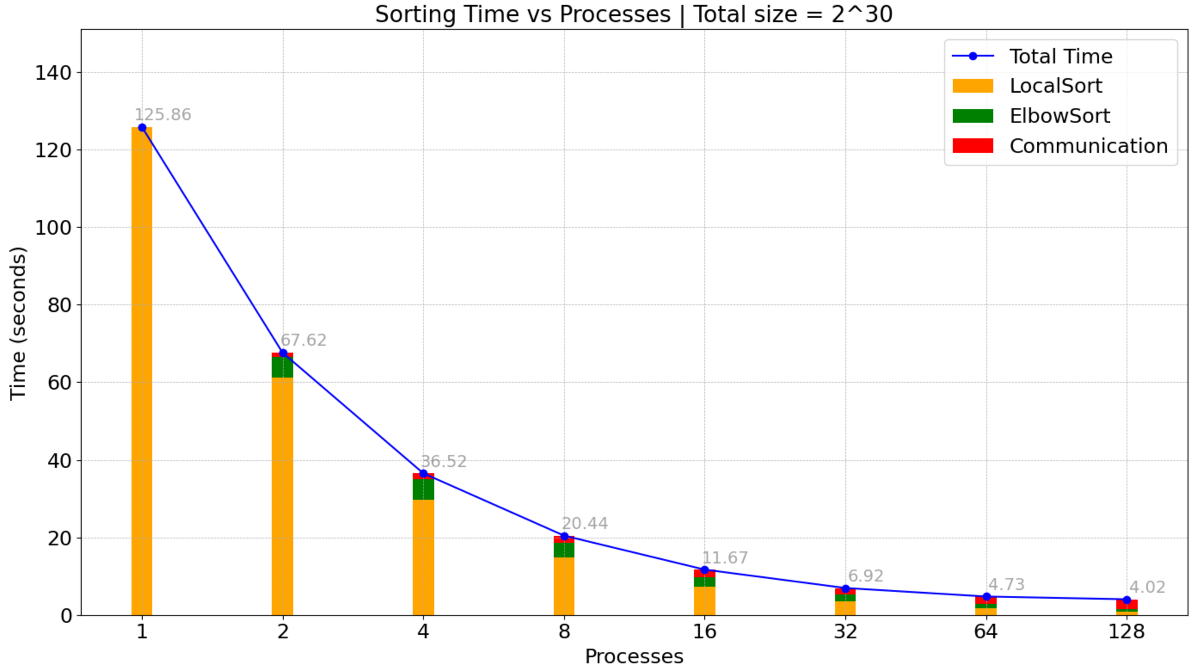

Figure 5: Sorting Time Breakdown by Component ($totalsize = 2^{30}$)

| # Processes | Total Time (s) | Local Sort (s) | Elbow Sort (s) | Communication (s) |
|---|---|---|---|---|
| 1 | 125.86 | 125.9 (**100.0%**) | - | - |
| 2 | 67.62 | 61.3 (**90.6%**) | 5.2 (7.7%) | 1.2 (1.8%) |
| 4 | 36.52 | 29.8 (**81.5%**) | 5.2 (14.2%) | 1.6 (4.3%) |
| 8 | 20.44 | 14.7 (**72.0%**) | 3.9 (18.8%) | 1.9 (9.2%) |
| 16 | 11.67 | 7.2 (**61.9%**) | 2.6 (22.0) | 1.9 (16.2%) |
| 32 | 6.92 | 3.5 (**50.7%**) | 1.7 (23.8%) | 1.8 (25.5) |
| 64 | 4.73 | 1.7 (36.8%) | 1.0 (21.5%) | 2.0 (**41.8%**) |
| 128S | 4.02 | 0.9 (21.6%) | 0.6 (15.3%) | 2.5 (**63.2%**) |

Table 3: Component Performance Analysis ($totalsize = 2^{30}$)

- **Local Sort and Elbow Sort:** As the number of processes increases, the time spent in both Local Sort and Elbow Sort decreases. This is because each process is assigned a smaller partition of the data to sort locally, reducing it's workload.

- **Communication:** While the absolute communication time increases with the number of processes, the **percentage** of total execution time spent on communication grows even more significantly. As shown in Table 3, the communication percentage rises from negligible values (below 10%) with a few processes to dominate the execution time (exceeding 60%) with 128 processes.

- **Total Time and Initial Exponential-like Decrease:** Initially, the total time decreases significantly as the number of processes increases, demonstrating the benefits of parallelization. This initial decrease resembles an exponential decay pattern, highlighting the substantial performance gains achieved during the early stages of parallelism due to effective workload distribution. However, the graph becomes almost horizontal after 32 processes, indicating that no further speedup is achieved by adding more processes, as discussed in previous sections (§4.3.1 and §4.3.2).

#### 4.3.4 Inter-node vs Intra-node Parallelism

The distribution of processes across nodes and cores within each node significantly impacts the performance of distributed algorithms. This benchmark examines the trade-offs between inter-node and intra-node parallelism. Figure 6 shows the total sorting time as a function of the dataset size for three distinct process distributions, each using a total of 128 MPI processes: one configuration with all processes on a single node (intra-node parallelism), and configurations with the processes uniformly distributed across two and four nodes (inter-node parallelism).

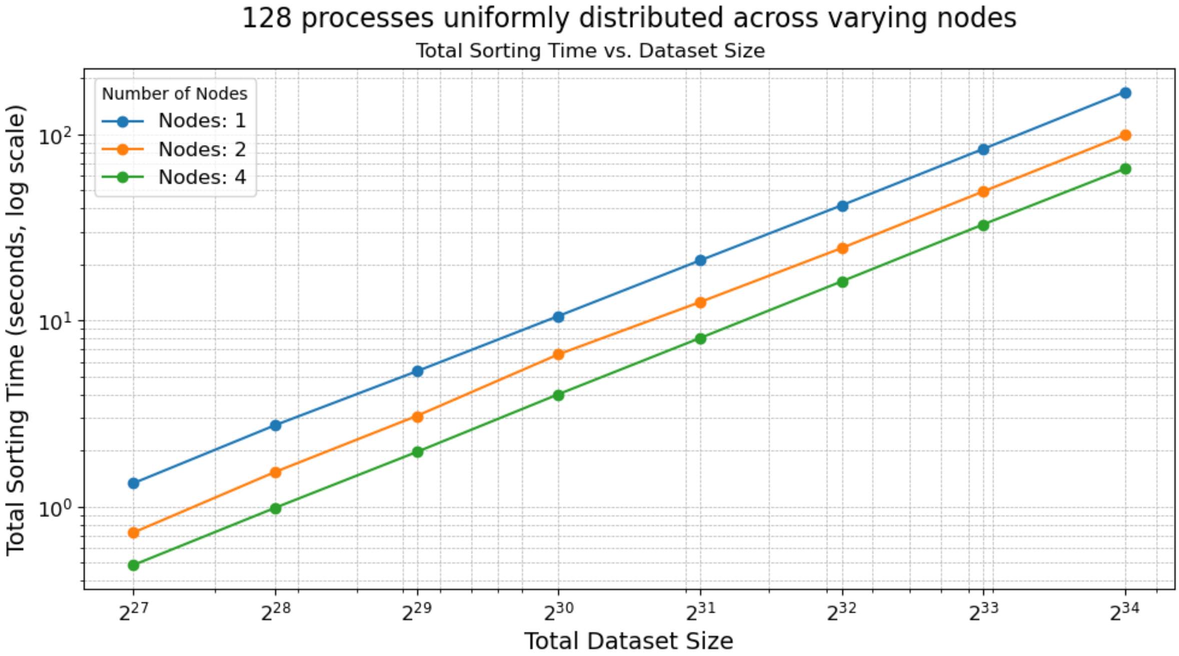

Figure 6: Comparing Intra-node and Inter-node Parallelism (128 MPI Processes)

**Key Observations and Insights.**

- **Impact of Node Distribution:** Distributing processes across multiple nodes (inter-node parallelism) consistently yields better performance compared to concentrating all processes on a single node (intra-node parallelism).

  This suggests that intra-node resource contention, such as memory bandwidth limitations or cache coherence overheads, could bottleneck performance when all processes are confined to a single node. Conversely, spreading processes across nodes reduces these bottlenecks and allows the system to better utilize distributed resources like memory and interconnect bandwidth.

- **Communication Cost Mitigation:** While distributing processes across nodes introduces inter-node communication overhead due to network latency, this cost is effectively mitigated by the architecture's HDR InfiniBand interconnect. With speeds up to 200 Gbps, this interconnect facilitates efficient data exchange, allowing the advantages of distributed processing to outweigh the communication overhead.

- **Architecture-Specific Factors:** The observed trends are specific to the test architecture and may not generalize across all systems. Factors as network topology, interconnect quality, and node-specific memory hierarchies play significant roles in determining the relative performance of inter-node versus intra-node parallelism.

## 5 Acknowledgments

Computational resources were provided by the Aristotle University of Thessaloniki (AUTH) through access to the Aristotelis High Performance Computing (HPC) cluster.

## References

[1] Aythamar Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co.

[2] Lawrence Livermore National Laboratory. *MPI Programming Tutorials.* [Online]. Available: `https://hpc-tutorials.llnl.gov/mpi/`

[3] Aristotelis High Performance Computing Cluster. *Documentation.* [Online]. Available: `https://hpc.it.auth.gr/`