Parallel Computing: MPI Parallel Sorting

Francesco Biscaccia Carrara
Master Degree in Computer Engineering
High Performance and Big Data Computing
University of Padova
Via G. Gradenigo, 6b, 35131 Padova PD, Italy
francesco.biscacciacarrara@studenti.unipd.it

***Abstract*— This paper explores the parallelization of sorting algorithms, focusing on adapting the Merge Sort algorithm for distributed computing environments. It presents a parallel implementation based on the divide-and-conquer paradigm, utilizing a hypercube network topology. The study investigates the scalability and efficiency of this parallel sorting approach across various dataset sizes and numbers of processors. Performance analysis includes measurements of execution time, speedup, and the ratio of computation to communication time. Results demonstrate the effectiveness of the parallelization strategy in significantly reducing sorting time for large datasets while maintaining a high computing-to-communication ratio.**

***Index Terms*— Merge Sort, Parallelization, MPI, Hypercube, Bitonic Sort**

I. INTRODUCTION

In the field of computer science, sorting algorithms are fundamental tools used to organize data. Whether the task is to arrange numbers, words, or any other data types, sorting is a critical operation that enhances the efficiency of subsequent data processing and analysis. The importance of sorting is evident in various applications, from database management to search optimization and even in everyday computational tasks.

The choice of a sorting algorithm is often determined by the size and nature of the dataset, as well as the performance requirements of the system. For example, simple algorithms like Bubble Sort and Insertion Sort are easy to implement and understand, making them suitable for small datasets or educational purposes. On the other hand, other algorithms such as Quick Sort, Merge Sort, and Heap Sort are preferred in professional applications due to their superior performance on large datasets.

As data continues to grow exponentially in today's digital world, the role of sorting algorithms remains as vital as ever, serving as the backbone of effective data organization and retrieval. However, even a robust sequential algorithm like Merge Sort may not be as effective in real-life scenarios: in real-time computation, the timespan is constrained to be very small, making computation time a critical factor.

To address these limitations, parallelization offers a powerful solution by distributing the computational workload across multiple processors. By leveraging parallel computing, the Merge Sort algorithm can be adapted to handle larger datasets more efficiently and within tighter time constraints. This approach not only accelerates the sorting process but also optimizes resource utilization, which is increasingly important in environments where computational power is abundant but time is limited.

In this paper, there will be an exploration of how to distribute the power of the Merge Sort algorithm across a set of processors, focusing on parallelization techniques. The investigation will delve into the scalability of such an implementation, examining how effectively the algorithm can handle increasing amounts of data and processing power. Additionally, the communication cost among processors will be analyzed, as it plays a crucial role in determining the overall efficiency and practicality of parallelized sorting in real-world applications.

II. SORTING ALGORITHM

A. Merge Sort Algorithm

Merge Sort[1] is a divide-and-conquer sorting algorithm that efficiently sorts data by recursively dividing a dataset into smaller subarrays, sorting them, and then merging the sorted subarrays back together. Developed by John von Neumann in 1945, Merge Sort is well-regarded for its stable sorting behavior and predictable time complexity of $O(n \log n)$, where n is the dataset size.

Algorithm 1: Merge Sort

Input: Array A with indices $start$ to end
Output: Sorted array A

Function `Merge`(A, $start$, mid, end)**:**
    $n_1 \leftarrow mid - start + 1$;
    $n_2 \leftarrow end - mid$;
    $A_L[0 \ldots n_1 - 1]$, $A_R[0 \ldots n_2 - 1]$
    for $i \leftarrow 0$ **to** $n_1 - 1$ **do**
        $A_L[i] \leftarrow A_L[start + i]$;
    end
    for $j \leftarrow 0$ **to** $n_2 - 1$ **do**
        $A_R[j] \leftarrow A_R[mid + j + 1]$;
    end
    $i \leftarrow 0$, $j \leftarrow 0$;
    $k \leftarrow start$;
    while $i < n_1$ ***AND*** $j < n_2$ **do**
        if $A_L[i] \leq A_R[j]$ **then**
            $A[k] \leftarrow A_L[i]$;
            $i \leftarrow i + 1$;
        end
        else
            $A[k] \leftarrow A_R[i]$;
            $j \leftarrow j + 1$;
        end
        $k \leftarrow k + 1$;
    end
Function `MergeSort`(A, $start$, end)**:**
    if $start < end$ **then**
        $mid \leftarrow \lfloor \frac{start+end}{2} \rfloor$;
        `MergeSort`(A, $start$, mid);
        `MergeSort`(A, $mid + 1$, end);
        `Merge`(A, $start$, mid, end);
    end

As show in the pseudocode, the algorithm operates by splitting the input array into two halves, repeatedly doing so until each subarray contains a single element. It then merges these subarrays in a way that results in a sorted array. The merging process is the key to Merge Sort's efficiency, as it combines elements in linear time, leading to its overall $O(n \log n)$ complexity.

One of the strengths of Merge Sort is its performance consistency, as it maintains the same complexity in the worst, best, and average cases. However, Merge Sort requires additional space proportional to the size of the input data, which can be a drawback in environments with limited memory.

B. Parallelization of Merge Sort Algorithm

Due to its nature, there are multiple ways to parallelize the Merge Sort algorithm. One straightforward approach is to leverage the hypercube architecture, which provides efficient parallel processing capabilities. This chapter explores a naive solution based on the Bitonic Sorting Paradigm, implemented on a hypercube network.

1) Hypercube Architecture: A hypercube[2] is a d-dimensional structure with 2^d nodes, where each node is connected to d other nodes. The diameter of the hypercube is $\log_2 P$, where P is the number of processors.

2) Bitonic Sorting Paradigm: In the Bitonic Sorting Paradigm[3] on a hypercube, data is exchanged among nodes based on edges connecting nodes that differ by specific bit positions. The process proceeds as follows:

1) Initially, data is exchanged along edges E_0, which connect nodes differing by the first bit.
2) In the subsequent step, data is exchanged along edges E_1 and E_0.
3) This is followed by exchanges over edges E_2, E_1 and E_0, and so on, continuing up to E_{d-1}.
4) Finally, exchanges are performed over all edges $E_{d-1}, E_{d-2}, \ldots, E_0$, incorporating all dimensions.

Each step progressively sorts the data by higher dimensions, leveraging the hypercube's

connectivity to distribute the sorting workload efficiently across multiple processors. This approach ensures that the data becomes sorted in parallel, taking full advantage of the hypercube's architecture.

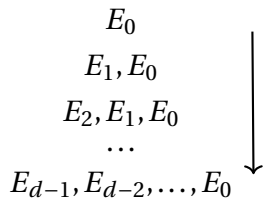

Fig. 1. Bitonic Sorting Paradigm execution schema

3) Bitonic Sorting Paradigm Implementation: Based on the Bitonic Sorting Paradigm, the implementation can be summarized as follows:

1) Divide the dataset into P equal parts, assigning one part to each processor.
2) Use the Merge Sort routine to sort each subset locally on its respective processor.
3) Employ the Bitonic Sorting Paradigm on a virtual hypercube with $\log_2(P)$ dimensions. During each stage:
   - Merge sorted sequences from adjacent processors, where one processor merges sequences from the lower index and the other from the higher index.
   - Continue merging until each processor's sequence reaches the middle point.

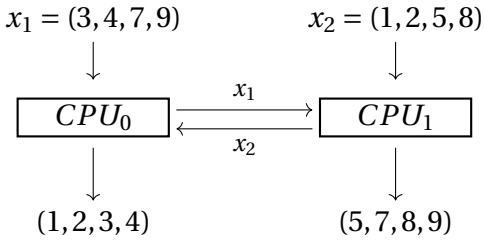

Fig. 2. Bitonic Merging on 2 processors

This approach minimizes inter-processor communication while maximizing parallelism. It is worth noting that the sorting routine used in the initial phase can be tailored to the specific processor type, ensuring compatibility without disrupting the overall computational framework.

Algorithm 2: Bitonic Sort

Input: Array A, Processor Set P
Output: Sorted array A

1: **Function** `BitonicSort(`A, P`)`**:**
   `// Scatter` A `across` P `processors`
2: $A_0, A_1, \ldots, A_{P-1}$;
   `// Execute in parallel on each processor` $p \in \{0, \ldots, P-1\}$
3: `MergeSort(`A_p, 0, $|A_p|-1$`)`;
4: **for** $l \leftarrow 0$ **to** $(\log_2 P) - 1$ **do**
5: **for** $i \leftarrow l$ **to** 0 **step** -1 **do**
6: $pal \leftarrow$ partner(p, i, l);
7: $A_r \leftarrow$ data_from(pal);
   `// Exchange data between` p `and` pal
8: **if** *p merges from low index* **then**
9: `MergeLow(`A_p, A_r`)`;
10: **end**
11: **else**
12: `MergeHigh(`A_p, A_r`)`;
13: **end**
14: **end**
15: **end**
   `// Gather` A_p `from each processor` p
16: $A \leftarrow A_0 \cup A_1 \cup \cdots \cup A_{P-1}$;

III. PROJECT SETUP

The sequential MergeSort algorithm is implemented in C, while the BitonicSort algorithm is executed using MPI. The code and all implementation details are available on GitHub.

For simplicity, the algorithm is tested on datasets whose sizes are powers of two, with the number of processors also being a power of two. This simplification can be mitigated using padding strategies, which become less significant as the dataset size increases. Thus, this approach remains relevant and effective for larger datasets. The tests include 10 measurements of both computation and communication time for each input size, with the datasets generated

randomly for each measurement. Below are the input sizes used:

Instance Size	
Power of 2	Decimal
2^{25}	33554432
2^{26}	67108864
2^{27}	134217728
2^{28}	268435456
2^{29}	536870912
2^{30}	1073741824

All tests were performed on the CAPRI High-Performance Computing (HPC) system, owned by the University of Padova. CAPRI is designed to provide computational power for testing innovative algorithms across various research fields. It is equipped with the following hardware:

- 16 Intel(R) Xeon(R) Gold 6130 @ 2.10GHz CPUs
- 6 TB of RAM
- 2 NVIDIA Tesla P100 16GB GPUs
- 40 TB of disk space

The objective of these tests is to verify that the parallelization approach improves computing time while keeping communication time as a low percentage of the overall execution time.

To evaluate this, the sequential MergeSort was run for each input size 10 times, and the average computation time for each size is referred to as T_{Seq}. Subsequently, the BitonicSort algorithm was executed on 2 to 32 processors, with the number of processors being powers of 2. For each configuration, the average computation time was measured similarly to the sequential tests, and the speedup

$$S(P) = \frac{T_{Seq}}{T(P)}$$

was analyzed, where $T(P)$ is the execution time of the algorithm on P processors. Finally, the Computing-over-Communication Ratio (CCR) was evaluated, defined as

$$CCR(P) = \frac{T_{Calc}(P)}{T_{Tot}(P)}$$

where $T_{Calc}(P)$ is the time devoted to computation, and $T_{Tot}(P)$ is the total execution time of the algorithm.

IV. RESULT AND ANALYSIS

A. Performance Plots

The data collected from the CAPRI infrastructure are available on the GitHub repository. The plot scripts were created using Matplotlib in Python. From the raw data, the following plots can be extracted:

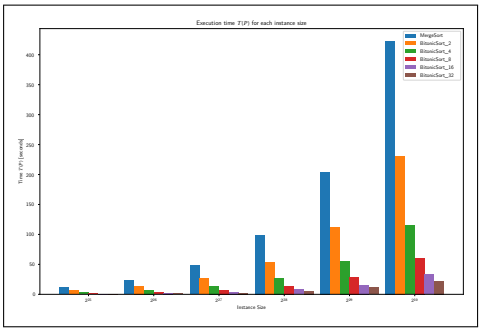

Fig. 3. MergeSort execution time T_{Seq} vs BitonicSort execution time $T(P)$ for different values of P and for each instance size. (lower is better)

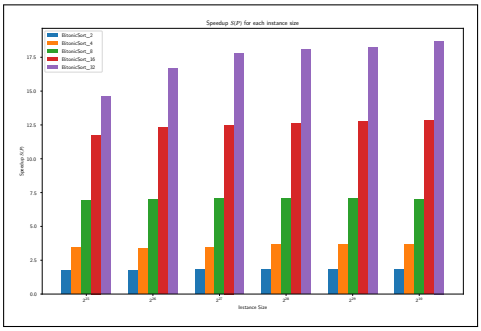

Fig. 4. Speedup $S(P)$ with respect to the MergeSort execution time for different values of P and for each instance size. (higher is better)

B. Data Analysis

First of all, there is a significant reduction in execution time (see Figure 3), showing a linear relationship between the number of processors and the time saved. As observed in Figure 4, the speedup factor $S(P)$ is consistently maintained, even as the input size increases. This consistency indicates that the algorithm effectively divides and manages the workload, likely because the sequential algorithm operates on problems that decrease in size by a factor of 2 up to 32.

Regarding the Computing-over-Communication Ratio $CCR(P)$, it is noteworthy

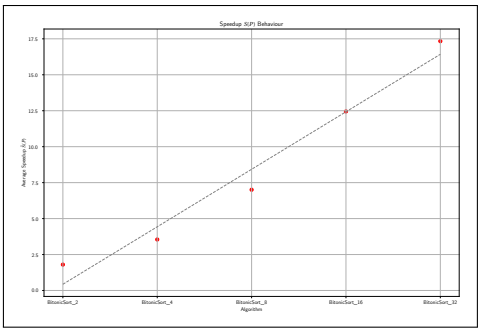

Fig. 5. Estimated speedup $\hat{S}(P)$ based on average speedup $S(P)$ for different values of P.

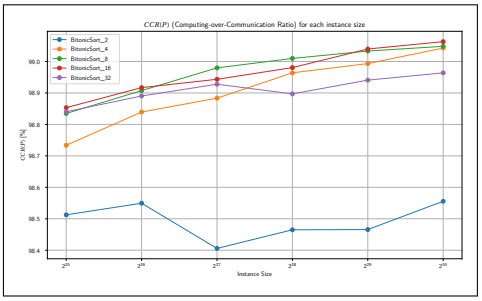

Fig. 6. Computing-over-Communication Ratio $CCR(P)$ for different values of P and for each instance size. (higher is better)

that even with a large number of processors (such as 16 or 32), the ratio remains close to 1, never dropping below 0.98. This demonstrates that the majority of the execution time is spent on computation rather than communication between processors, likely because the data exchanged between processors becomes smaller as the number of processors increases.

As expected, the linear interpolation of the average speedup $S(P)$ for each P (see Figure 5) confirms a linear trend, suggesting that the BitonicSort algorithm scales well.

V. CONCLUSIONS AND FUTURE WORKS

To sum up, the results from the tests performed on the CAPRI infrastructure demonstrate that the BitonicSort implementation is a highly effective solution for handling massive datasets. However, there are potential areas for improvement:

- Executing tests on larger datasets with more processors could further confirm the hypotheses about scalability and execution time reduction.
- Reducing the data exchange between processors: currently, the entire storage of a processor node is sent to another one. This could be optimized with some technical improvements to minimize data transfer.

REFERENCES

[1] John Von Neumann. “A new method for solving a problem in the theory of sorting”. In: *Communications of the ACM* 2.5 (1956), pp. 286–293.

[2] Lars J. Boman et al. “The hypercube: A model for parallel computation”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 443–486.

[3] Kenneth E. Batcher. “Sorting networks and their applications”. In: *IBM Journal of Research and Development* 12.5 (1968), pp. 365–372.