Sorting arrays in parallel with Bitonic Sort using MPI

Eleni Koumparidou and Epameinondas Bakoulas

January 2025

1 Objective

Sorting algorithms usually take $O(nlogn)$ time to sort an array of size n. Bitonic sort is a parallel sorting algorithm that can sort very large arrays faster with the power of MPI and parallel processing. In this report, we will present the implementation of this algorithm. We will also present the results of our experiments and compare the different benchmarks.

2 Algorithm Analysis

2.1 Overview

Our goal is to sort an array of size $p * 2^q$, where $p = 2^k$, $k = [1 : 7]$ is the number of processes and 2^q, $q = [20 : 27]$ the number of elements in each process.

2.2 Parallelism - MPI

The algorithm will follow the standard **Bitonic Sort Network** pattern as shown in Figure 2. Our goal is to create bitonic sequences of size $2, 4, 8, ..., p$ and then, with the correct exchanges between processes, make bigger bitonic sequences. So our first step is to create bitonic sequences using 2 processes each time. In order to do that, we will locally sort the processes in ascending/descending order using `quickSort (qsort)`. This step is quite costly but it is necessary for the algorithm and we will only do it once.

Now we can start the communications between the processes. Each process will keep either the `min` or `max` elements element wise. Once this finishes, every process (block) will be a bitonic sequence, so we can sort them locally in O(n) time by applying the `elbowSort` algorithm explained in detail later on.

We repeat this process but now we initally communicate with the neighbors of distance 2, and then with the neighbors of distance 1. The sorting always takes place right after we communicate with the closest neighbor. In total, this process is repeated $\log_2 p$ times, and each time we double the initial distance. This is clearly visible in Figure 2, where the total steps are equal to $\log_2 8 = 3$.

Finally, we end up with p processes sorted in ascending order, and if we merge them together we get a fully sorted array of size $p * 2^q$.

```
void bitonicSort(int rank, int num_p, int num_q, int **array) {
    // Initial sorting of the processes
    sortProcesses(rank, num_q, *array);
    for (int group_size = 2; group_size <= num_p; group_size *= 2) {
        bool sort_descending = rank & group_size;
        for (int distance = group_size / 2; distance > 0; distance /= 2) {
            int partner_rank = rank ^ distance;
            minmax(rank, partner_rank, num_q, *array, sort_descending);
        }
        elbowMerge(num_p, num_q, array, sort_descending);
    }
}
```

Figure 1: Bitonic Sort Algorithm. The function sortProcesses sorts each process using quick sort, while the minmax function exchanges the min/max elements between two processes. Finally, the elbowMerge function sorts the bitonic sequence locally.

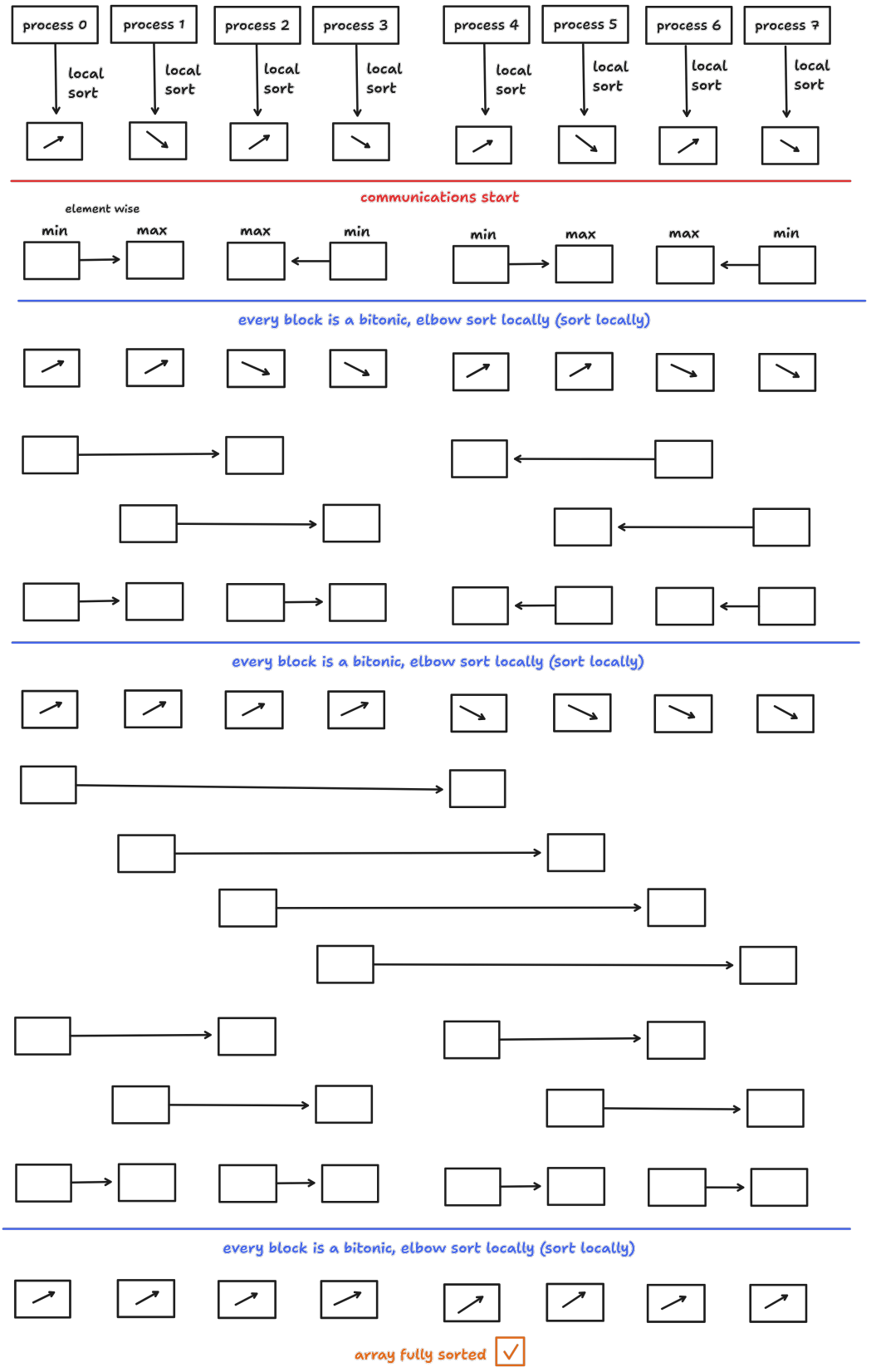


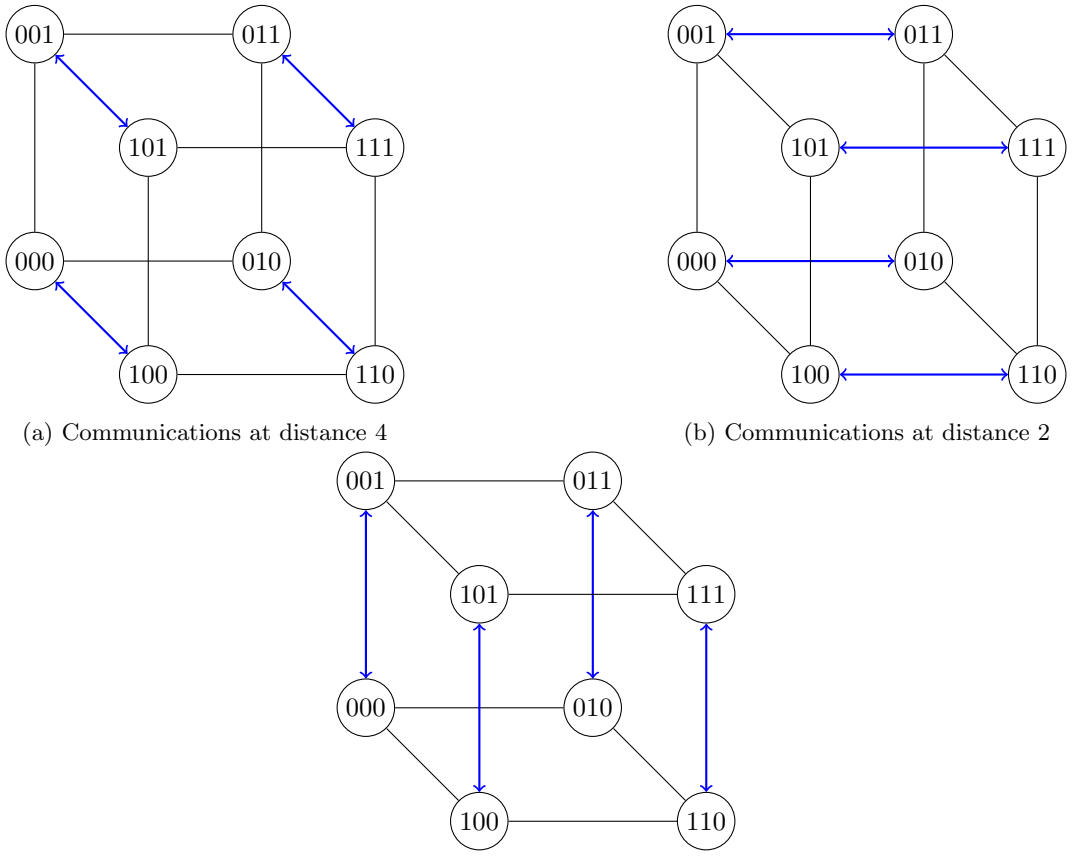
Figure 2: Bitonic Sort Network

2.3 Hypercube model

Finding the right neighbor to exchange data with in each stage of the algorithm can be quite tricky. One possible way is to leverage the power of the hypercube model. First, we give each process a unique $log_2 p$ bit id. In the case of 8 processes, each process will have a 3 bit id, as shown in Figure 3. In order to find the right neighbor, all we need to do is compute

$$pid \oplus distance$$

where $\oplus$ is the XOR (exclusive-OR) operator and *pid* the process id.



(a) Communications at distance 4

(b) Communications at distance 2

(c) Communications at distance 1

Figure 3: (Hyper)cube model with 8 processes.

2.4 Elbow sort

`ElbowSort` sorts efficiently a bitonic sequence in **O(n)** time:

- The minimum element of the bitonic sequence, known as the **elbow**, is found in **linear time** and depending on the order of the sorting, it is placed as the first or last element of the sorted array.
- The array is scanned in a circular way, by placing a `left` and a `right` pointer to the left and right of the elbow. This means that when one pointer reaches the end of the array, it loops around to the other end.

During the scanning process, the smallest of the two elements pointed is placed in the sorted array and the corresponding pointer moves to the next element, until all elements have been placed into the sorted array in **linear time** again. The sorting direction (ascending or descending) and the placement of the elbow (first or last) are determined by the ID of the process combined with the phase of the procedure, as shown in Figure 2.

3 Benchmarks

Different testings took place in **rome partition** of **Aristotel Cluster** to test the performance of our sorting algorithm, inlcluding efficiency and speed. To ensure that valid results were provided, we developed a function `evaluateResult`, which compares only the last element of a process and the first element of the next one to confirm that the global array is sorted.

3.1 Efficiency & Scaling

Figure 4 shows the performance of our code in efficiently handling large datasets. In each execution, the number of elements as well as the number of parallel processes was doubled (2^{27} elements/process).

3.2 Parallel processing

The performance of sorting 2^{27} elements in different processes is presented in Figure 5. The speedup is calculated by comparing the execution time to that of a single process, which is equivalent to using quick sort. In the case of 128 processes, the execution time is almost **20 times reduced**. This confirms that increasing the number of processes accelerates the execution of the algorithm.

3.3 Nodes Performance

In Figure 6, we analyzed the sorting of $q = [20 : 27]$ elements while increasing the number of nodes used for the partition. We observed a significant reduction in execution time, with a decrease of up to 50% for 8 nodes.

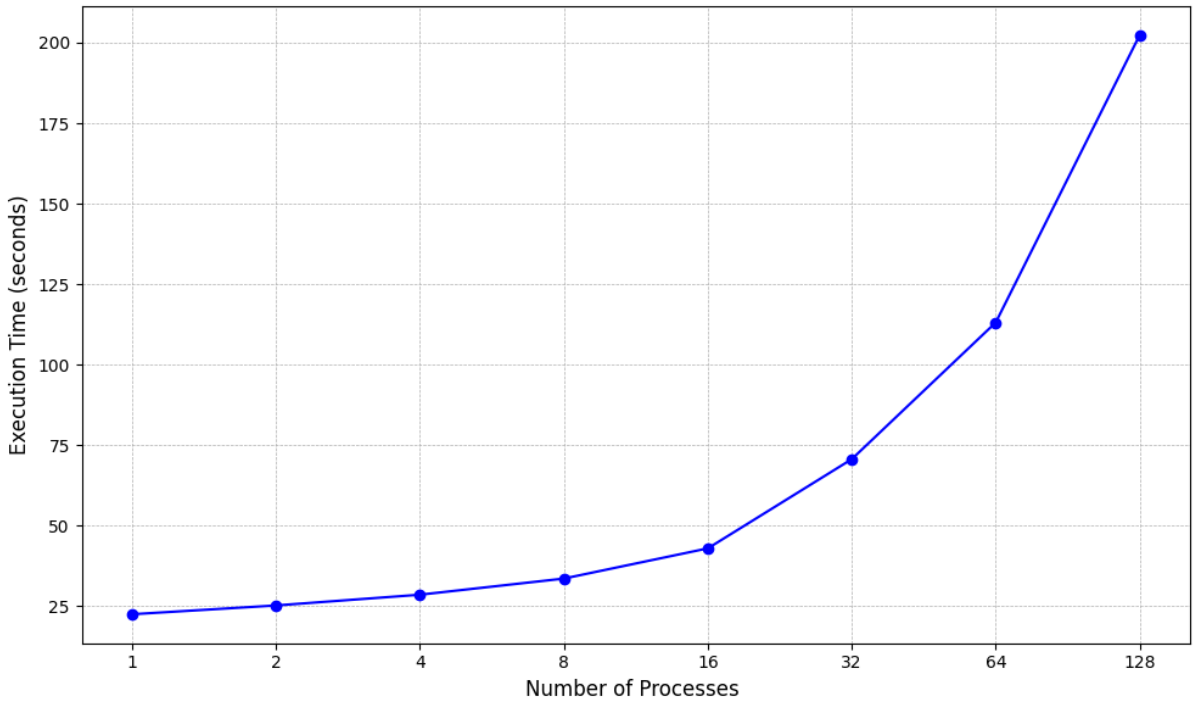


Figure 4: Execution time with different processes

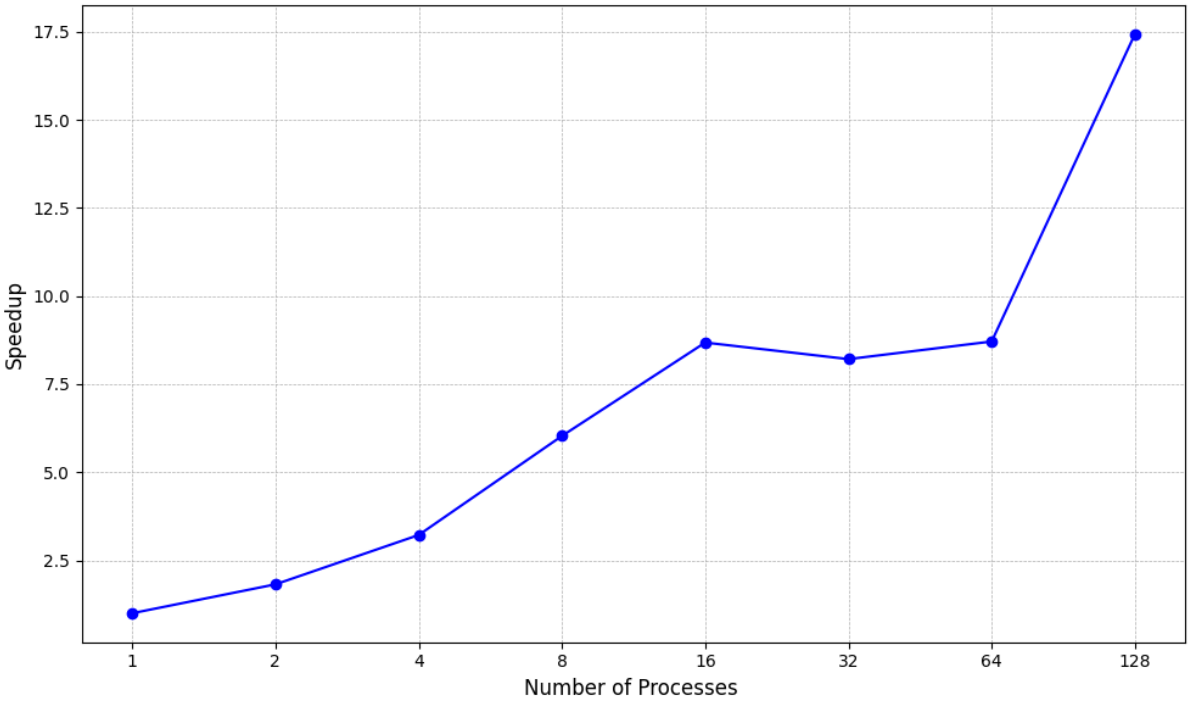


Figure 5: Speedup compared to QuickSort

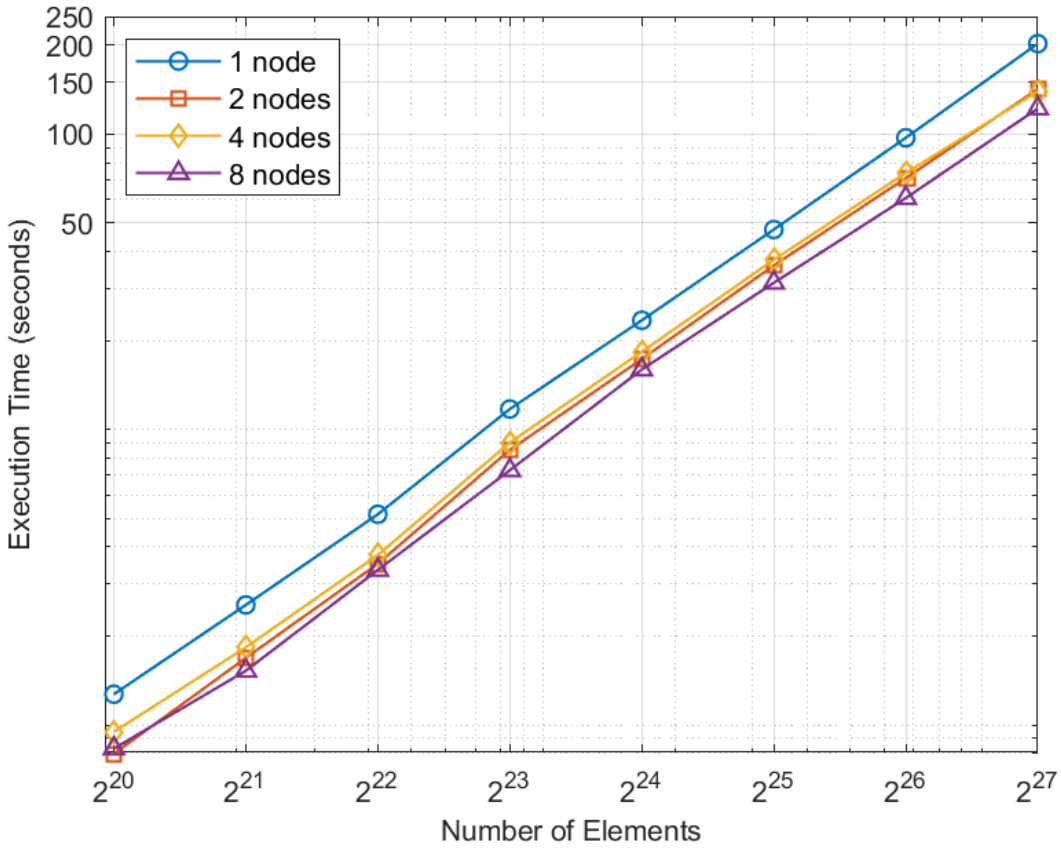


Figure 6: Execution time using different number of nodes