# 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

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

### 2.2 Parallelism - MPI

The algorithm will follow the standard **Bitonic Sort Network** pattern as shown in Figure 1. 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 1, 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 $p * 2^q$.

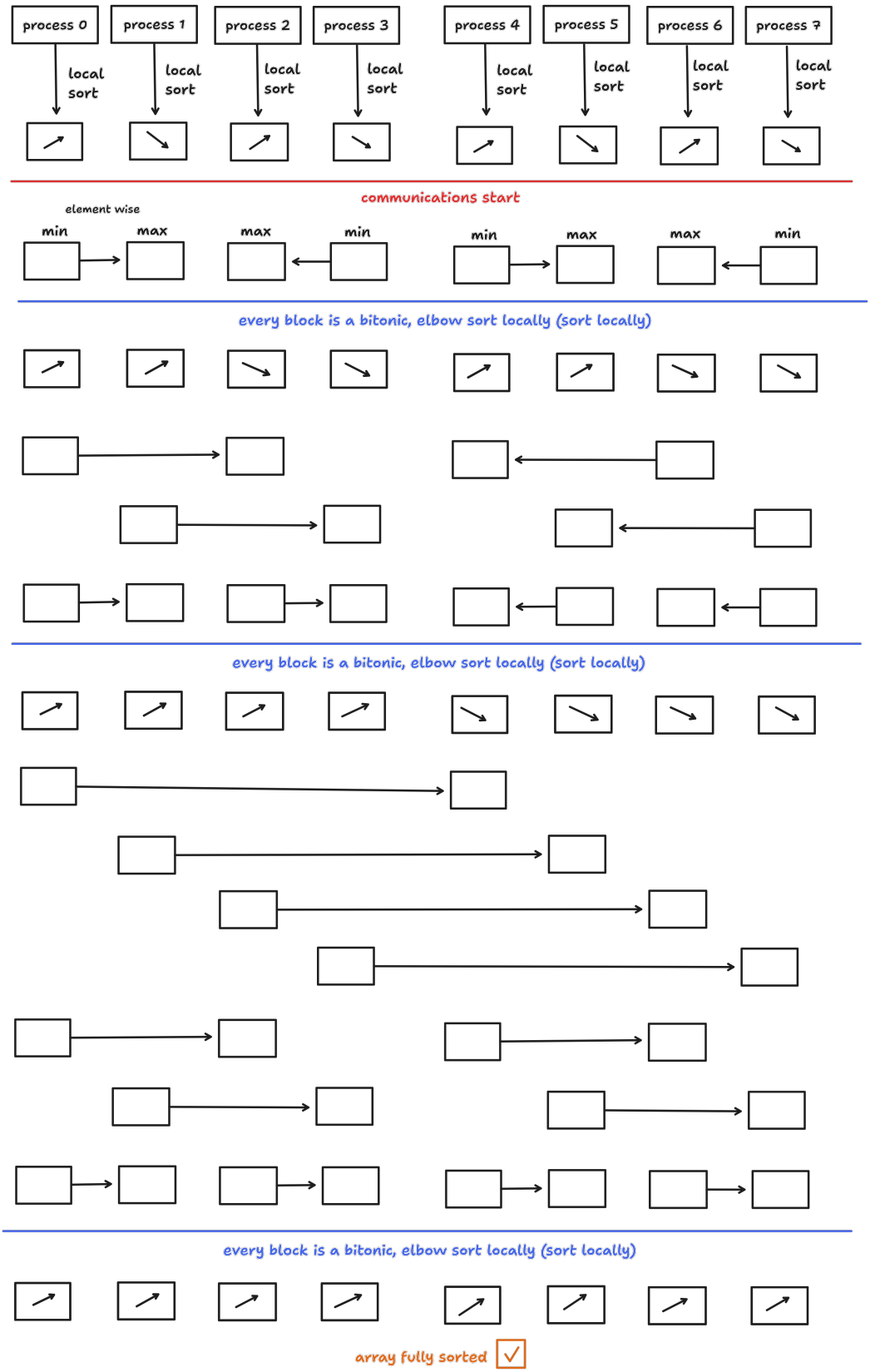


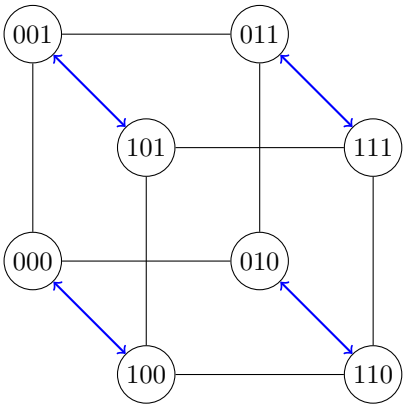

Figure 1: Bitonic Sort Network

### 2.3 Hypercube model

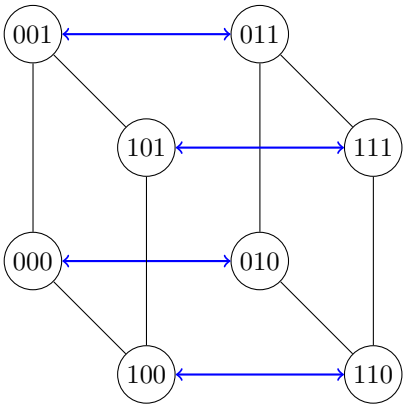
Finding the right neighbor to exchange data with in each stage of the algorithm can be quite tricky. One possible way is to leverage the power of the hypercube model. First, we give each process a unique $log_2 p$ bit id. In the case of 8 processes, each process will have a 3 bit id, as shown in Figure 2. In order to find the right neighbor, all we need to do is compute

$$pid \oplus distance$$

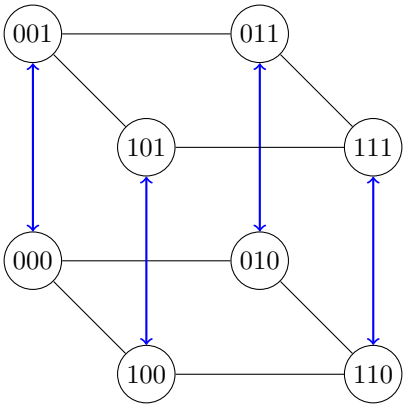
where $\oplus$ is the XOR (exclusive-OR) operator and *pid* the process id.




(a) Communications at distance 4




(b) Communications at distance 2




(c) Communications at distance 1

Figure 2: (Hyper)cube model with 8 processes.

### 2.4 Elbow sort

## 3 Benchmarks