Bitonic Sort with MPI

Ioannis Michalainas, Savvas Tzanetis

January 1, 2025

Contents

Chapter 1

Abstract

This report is part of an assignment for the **Parallel and Distributed Systems** class of the Aristotle University's Electrical and Computer Engineering department, under professor *Nikolaos Pitsianis*.

This project implements *distributed sorting* using the **Bitonic Sort** algorithm and the **Message Passing Interface (MPI)**. The primary objective is to sort a dataset of $N = 2^{(q+p)}$ numbers (where 2^p represents the total *processes* and 2^q the *numbers* assigned per process) utilizing inter-process communication. The implementation employs parallel processing to achieve efficient sorting, making it suitable for large-scale data sets.

Chapter 2

Serial Bitonic

To grasp the concept of distributed **Bitonic Sort**, it is essential to first understand its serial implementation. This foundational knowledge will provide the necessary context for comprehending the distributed version of the algorithm.

2.1 Explanation

Bitonic Sort is a sorting algorithm that operates by sorting and merging bitonic sequences. A bitonic sequence is defined as a sequence of numbers that first monotonically increases and then monotonically decreases (or vise versa), or can be cyclically rotated to exhibit this pattern. Note that sorted sequences are also bitonic. For example, the sequence $6, 4, 3, 1, 2, 5, 8, 7$ is bitonic because it can be rotated to $7, 6, 4, 3, 1, 2, 5, 8$.

The algorithm is has the following characteristics:

- $log(n)$ steps, where n is the total amount of numbers, with $log(step)$ comparison phases in each step.
- In each comparison phase, we swap elements using the **min-max** criteria. The min-max pattern is presented in the image bellow:

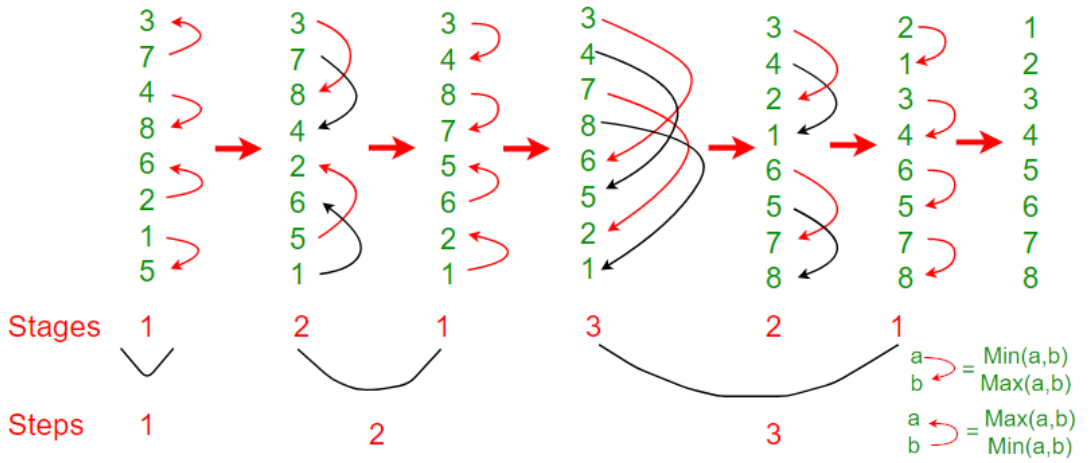


Figure 2.1: Bitonic Sort

2.2 Example

To illustrate the process, consider the following random sequence:

$$S\{3, 7, 4, 8, 6, 2, 1, 5\}$$

In each step each element is compared to its neighbor with distance **stage**.

1. **Step 1, Stage 1 min-max pattern:** *min max max min min max max min*

   $$S\{3, 7, 8, 4, 2, 6, 5, 1\}$$

2. **Step 2, Stage 2 min-max pattern:** *min min max max max max min min*

   $$S\{3, 4, 8, 7, 5, 6, 2, 1\}$$

3. **Step 2, Stage 1 min-max pattern:** *min max min max max min max min*

   $$S\{3, 4, 8, 7, 6, 5, 2, 1\}$$

4. **Step 3, Stage 3 min-max pattern:** *min min min min max max max max*

   $$S\{3, 4, 2, 1, 6, 5, 7, 8\}$$

5. **Step 3, Stage 2 min-max pattern:** *min min max max min min max max*

   $$S\{2, 1, 3, 4, 6, 5, 7, 8\}$$

6. **Step 3, Stage 1 min-max pattern:** *min max min max min max min max*

   $$S\{1, 2, 3, 4, 5, 6, 7, 8\}$$

2.3 Remarks

1. The complexity of this algorithm is $O(nlog^2n)$. While it is higher than other popular sorting algorithms like **Merge Sort** or **Quick Sort**, **Bitonic Sort** is ideal for parallel implementation, because it always compares elements in a predefined sequence and the sequence of comparison does not depend on data.

2. **Bitonic Sort** can only be used if the number of elements to sort is 2^n. The procedure fails if the number of elements is not in the aforementioned quantity precisely.

Chapter 3

Distributed Bitonic

As mentioned above, this algorithm, when implemented in a serial manner, has a time complexity of $\boldsymbol{O(n \log^2(n))}$, which is higher than most sorting algorithms. Despite that, this algorithm is useful, as it is well-suited for parallel implementations, like the one we will be discussing. For a parallel implementation, we can achieve a time complexity of $\boldsymbol{O(\log^2(n))}$, which is significantly faster than our serial implementation. We will be achieving this goal by utilizing the **Message Passing Interface (MPI)** for inter-process communication, allowing us to distribute the load across multiple machines.

3.1 Generating Random Sequences

The sequences we used to test our results were generated using the **rand()** function, which is part of the **C standard library**. More specifically, we selected *integers* ranging from **1** to **999** for simplicity. We use this function to create a random array of N integers, which we then distribute across multiple processes.

```
void randomVec(Vector* vec, int max) {
    srand(time(NULL));
    for (int i=0; i<vec->size; i++) {
        vec->arr[i] = rand()%max;
    }
}
```

Here, **Vector** is a *struct* we implemented with two variables, **arr** for storing the data and **size** for storing the size of the array **arr**.

3.2 MPI Communication

MPI allows us to simulate multiple machines, by creating multiple processes to a single computer running the same code. Each process is given an **ID** called **rank** and is generally working independently from other processes, except for times where synchronization is needed. In our code, we refer to the amount of ranks (processes) as **size**.

```
// run a copy of main for each rank
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Processes should also be able to communicate with each other, in order to exchange data and synchronize. Luckily, the MPI library already provides us with the necessary tools:

```
// Rank 0 Scatters data to all other ranks
MPI_Scatter(data->arr, local->size, MPI_INT, local->arr,
            local->size, MPI_INT, 0, MPI_COMM_WORLD);

// Other ranks accept data but don't send anything themselves
MPI_Scatter(NULL, local->size, MPI_INT, local->arr,
            local->size, MPI_INT, 0, MPI_COMM_WORLD);
```

In the code above, **data** holds the randomly generated array that awaits to be sorted, while **local** represents the sub-arrays of each rank involved in the distributed sort task. After we perform the distributed sort, we synchronize all ranks and then gather our results to showcase them.

```
distributed_sort(local, rank, size);
MPI_Barrier(MPI_COMM_WORLD);

[...]

MPI_Gather(local->arr, local->size, MPI_INT, sorted ? sorted->arr : NULL,
                       local->size, MPI_INT, 0, MPI_COMM_WORLD);

// wait until all ranks reach this point
MPI_Barrier(MPI_COMM_WORLD);
results(sorted, rank, size);
```

3.3 Implementation

In this subsection, we will discuss how the **Bitonic Sort** algorithm was implemented in a distributed manner.

3.3.1 Brief

After each rank performs an initial sort to its local data, we enter a nested for loop. The algorithm has **p** stages, with **stage** steps each. Number of stages **p** is defined as

$$log_2(size)$$

In each step, we calculate the **distance** of the **partner** we need to exchange data and said partner's rank, then we proceed to **exchange** data by following a **min** or **max** pattern. After each stage has concluded, each rank sorts its local data using **elbowmerge**.

```
initialSort(local, rank);
for (int stage=1; stage<=p; stage++) {
    for (int step=stage; step>=1; step--) {
      int distance = 1 << (step-1);
      int partner = rank ^ distance;

      exchange(partner, local, remote);
      minmax(rank, stage, distance, local, remote);
    }
    // Ascending (0) or Descending (1)
    int direction = (rank & (1 << stage)) == 0;
    elbowmerge(local, direction);
  }
```

Later follows a visual outline of the algorithm, featuring an example of **8** processes.

3.3.2 Initial Sort

First and foremost, after the random array data has been distributed evenly across all ranks, each rank will go through an *initial sorting process* that will run exactly once. This sort is performed using the **qsort** function provided by the C standard library.

```
void initialSort(Vector* local, int rank) {
  if (rank & 1) {
    qsort(local->arr, local->size, sizeof(int), compDescending);
  } else {
    qsort(local->arr, local->size, sizeof(int), compAscending);
  }
}
```

Here **(rank & 1)** is a logical operation equivalent to **rank** MOD **2** that essentially allows us to sort the sub-arrays with alternating order. Note that the logical AND (&) operation will be handy later on for creating alternating binary patterns, since the **Bitonic Sort** algorithm is based on **powers of 2**.

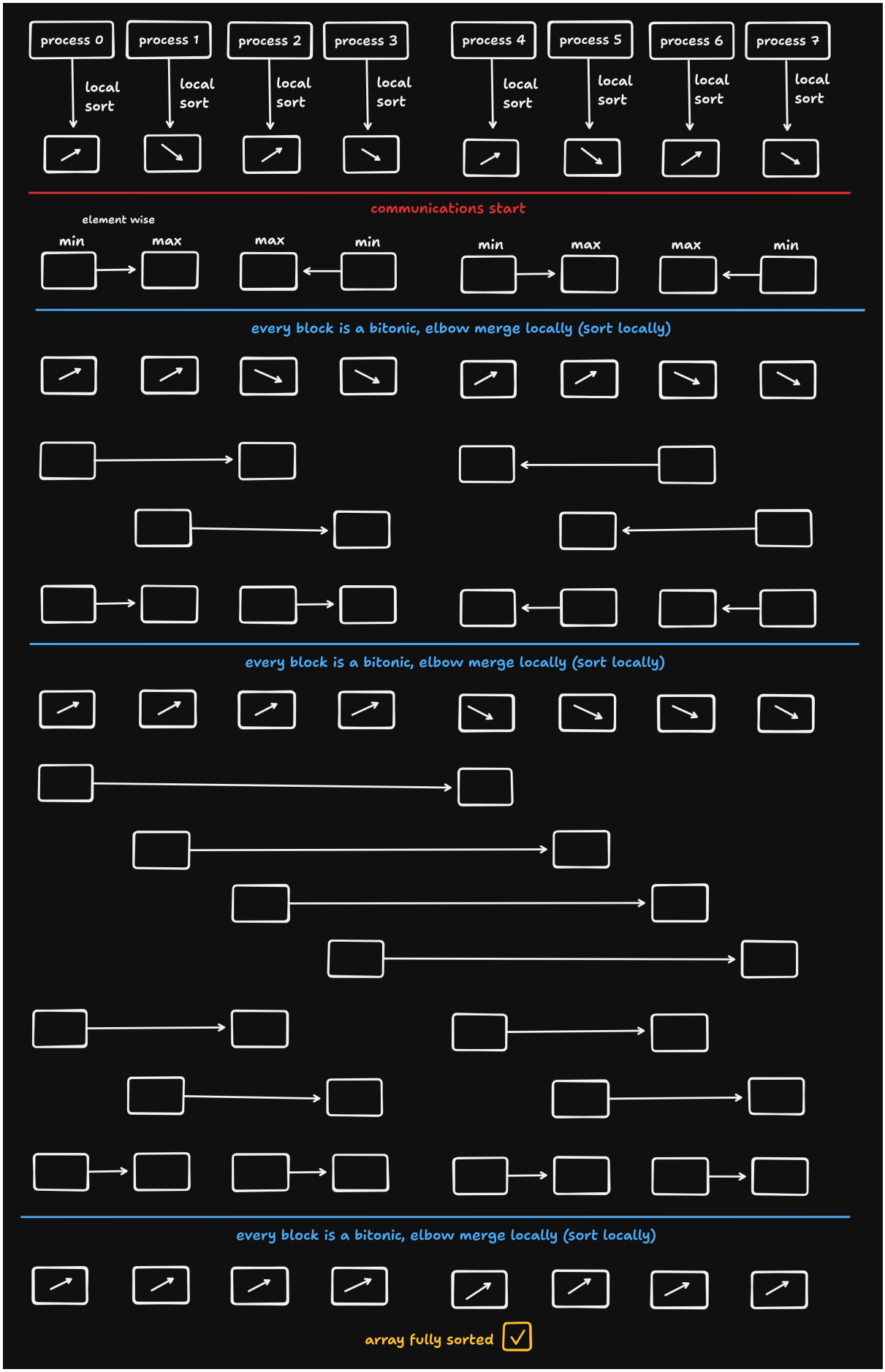


Figure 3.1: Algorithm Outline

Decimal	Binary	AND with 1	AND with 2	AND with 4
0	000	000 *(OFF)*	000 *(OFF)*	000 *(OFF)*
1	001	001 *(ON)*	000 *(OFF)*	000 *(OFF)*
2	010	000 *(OFF)*	010 *(ON)*	000 *(OFF)*
3	011	001 *(ON)*	010 *(ON)*	000 *(OFF)*
4	100	000 *(OFF)*	000 *(OFF)*	100 *(ON)*
5	101	001 *(ON)*	000 *(OFF)*	100 *(ON)*
6	110	000 *(OFF)*	010 *(ON)*	100 *(ON)*
7	111	001 *(ON)*	010 *(ON)*	100 *(ON)*

Above, we can visualize and understand the patterns created using the logical AND (&) operation for **powers of 2**.

3.3.3 Data Exchange

In this part, each process **exchanges** data with its **partner**. We need to determine the **distance** of the partner the process should exchange data with. This is given by $2^{(step-1)}$. We then calculate the partner based on the distance **rank** XOR **step**.

The processes are arranged in a hypercube-like structure, where the rank of each process is treated as a binary number. The **XOR** operation between two binary numbers results in a number where the bits are set to 1 at positions where the two numbers differ. The distance between two nodes in a hypercube is the number of differing bits between their binary representations, known as the **Hamming distance**.

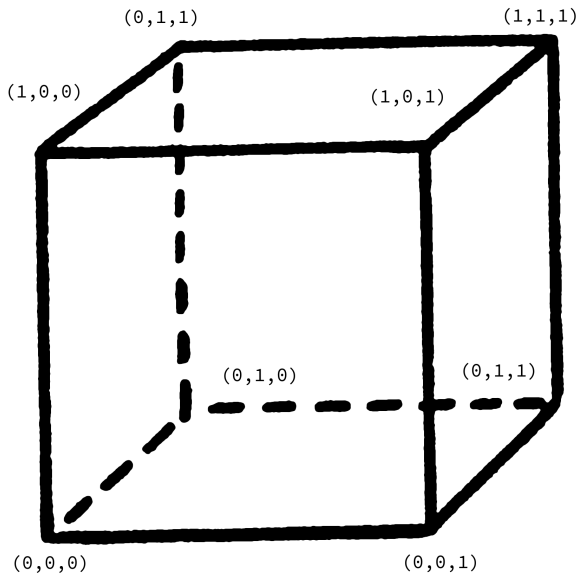


Figure 3.2: 3D Hyper-Cube Representation

The XOR operation helps find a partner that differs from the current process by exactly the number of bits defined by the *distance* at each step. This ensures that in each stage of the algorithm, the process communicates with a partner that is located at a specific Hamming distance, consistent with the hypercube structure. At each stage, the *distance* doubles (based on the step), and the `rank ^ distance` operation calculates a process ID that is at the desired Hamming distance.

Decimal	Binary	XOR with 1	XOR with 2	XOR with 4
0	000	001	010	100
1	001	000	011	101
2	010	011	000	110
3	011	010	001	111
4	100	101	110	000
5	101	100	111	001
6	110	111	100	010
7	111	110	101	011

Every rank then sends their **local** array to their respective partner and receive their partners data in a buffer called **remote**.

```
void exchange(int partner, Vector* local, Vector* remote) {

  // send local and receive remote
  MPI_Sendrecv(local->arr, local->size, MPI_INT, partner, 0,
               remote->arr, local->size, MPI_INT, partner, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

3.3.4 Min-Max Operations

This part of the algorithm involves processing the data each rank receives from its partner. Each rank compares all the local elements to the remote ones element-wise and keeps the **min** or **max** elements, depending on the stage and the distance at the specific point of the algorithm.

A helpful way to visualize the criteria for choosing min or max is to imagine that for each stage *mirrors* are placed before the processes with ranks $2^{stage} + k * w$ where $w = 2 * mirror$ and k any natural number $N[0, 1, 2, ...]$, spanning the whole array of processes. We then determine in which mirrors province does the rank we examine belong and calculate its distance from the mirror, its *reflection*.

Let's examine the example of 8 processes to clarify our approach. Suppose we are on the **first stage** with **distance one**. We place *mirrors* at positions **2** and **6**, like so: $[p0\ p1|p2\ p3, p4\ p5|p6\ p7]$. In this case, $p0, p3, p4, p7$ would have a *reflection* of 1 and $p1, p2, p5, p6$ a *reflection* of 0. The required min max order is: $[minmax|maxmin, minmax|maxmin]$. We notice that there is symmetry around the locations of the mirrors, spanning $w = 2 * mirror$ range. Employing a similar binary pattern creation tactic we used for the initial sort, we determine whether a process should keep the min or the max elements element wise. We concluded to the following formula:

$$reflection \ \& \ 2^{(log_2(distance))}$$

If the above operation results to 0, we get a **max** element-wise comparison, else we get a **min** element-wise comparison.

This approach generalizes for any number of processes that is a **power of 2**.

```
void minmax(int rank, int stage, int distance,
            Vector* local, Vector* remote) {

  int mirror = 1 << stage;
  int w = 2*mirror;
  int pos = rank % w;

  int reflection = pos<mirror ? mirror-pos - 1 : pos-mirror;

  if (reflection & (1 << (int)log2(distance))) {
    min(local, remote);
  } else {
    max(local, remote);
  }
}
```

3.3.5 Elbow Merge

Lastly, **elbowmerge** is a sorting technique specific to bitonic sequences (that being the reason why we do not use it for our initial sort, as the arrays are arbitrary).

We first begin by identifying the **extremum** of the bitonic array, either the **maximum** (elbow) or the **minimum** (pit). To do that, we need to locate a point in the sequence that its adjacent points (in the context of a circular buffer) are either both smaller than said point (elbow) or larger (pit).

If we do not manage to find such point, it means that the sequence is of the form $[a, a, b, b]$ or $[b, b, a, a]$ or $[a, a, a, a]$ where $a > b$. Such sequences may not be considered bitonic, but consist possible inputs for the elbow function. To handle them gracefully, we select the max element and return it as the elbow (by convention).

That way, locating the extremum of a bitonic sequence is performed in $O(n)$ time, even for the corner cases.

```
Extremum elbow(Vector* local) {

  Extremum point;
  point.index = -1;
  point.polarity = 0;

  int  n   = local->size;
  int* arr = local->arr;

  for (int i=0; i<n; i++) {
    int prev = arr[(i-1 + n) % n];
    int curr = arr[i];
    int next = arr[(i+1) % n];

    if (curr>prev && curr>next) {
      point.index = i;
      point.polarity = 1;
      return point; // elbow
    }

    if (curr<prev && curr<next) {
      point.index = i;
      point.polarity = -1;
      return point; // pit
    }
  }

  // invalid bitonic sequences (corner cases)
  int max = 0;
  for (int i=1; i<n; i++) {
    if (arr[i] > arr[max]) {
      max = i;
    }
  }
  // return max as elbow
  point.index = max;
  point.polarity = 1;

  return point;
}
```

After identifying the **extremum**, we add it either on the end or the start of a temporary array, depending on if we have a **minimum** or a **maximum**. For the **elbow** case, we then compare the values of the **left** and **right** indices and the one containing the largest element gets to be placed next in the temporary array and shift its index by one (-1 for left +1 for right) in a circular buffer manner. This processes is repeated until all elements are placed in the tmp array, resulting in a fully sorted array, and follows commensurable logic in the **pit** case. If we want to sort the array in **ascending** order we *directly* copy tmp to local, if we want **descending** order we *reverse* copy our results.

```
void elbowmerge(Vector* local, int direction) {
  Extremum point = elbow(local);
  int  n   = local->size;
  int* tmp = malloc(sizeof(int)*n);

  int left  = (point.index-1 + n) % n;
  int right = (point.index+1) % n;

  if (point.polarity == -1) {
    int i = 0;
    tmp[i++] = local->arr[point.index];

    while (i < n) {
      if (local->arr[left] < local->arr[right]) {
        tmp[i++] = local->arr[left];
        left = (left-1 + n) % n;  // move left index circularly
      } else {
        tmp[i++] = local->arr[right];
        right = (right + 1) % n;  // move right index circularly
      }
    }
  } else if (point.polarity == 1) {
    int i = n-1;
    tmp[i--] = local->arr[point.index];

    while (i >= 0) {
      if (local->arr[left] > local->arr[right]) {
        tmp[i--] = local->arr[left];
        left = (left-1 + n) % n;  // move left index circularly
      } else {
        tmp[i--] = local->arr[right];
        right = (right+1) % n;  // move right index circularly
      }
    }
  }
  if (direction == 1) {
    memcpy(local->arr, tmp, sizeof(int)*n);
  } else {
    for (int j=0; j<n; j++) {
      local->arr[j] = tmp[n-1 - j];
    }
  }
  free(tmp);
}
```

The merging process is also performed in $O(n)$ time, making the total computational cost of the local sort $O(n)$ too.

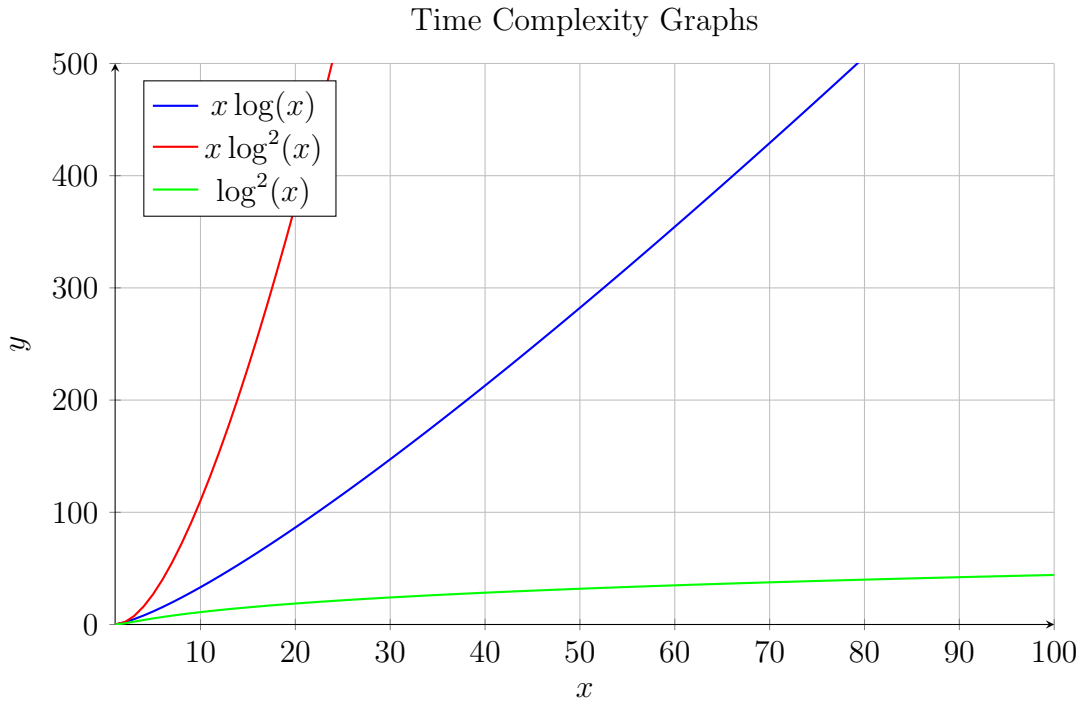
Chapter 4

Results

As previously mentioned, a serial version of this algorithm has a time complexity of $\boldsymbol{O(n \log^2(n))}$, which is not ideal, as there are faster algorithms. However, a parallel or distributed version, significantly lessens this to $\boldsymbol{O(\log^2(n))}$.

Performance Table	**Serial**	**Distributed**	**Quick Sort**
Time Complexity	$\boldsymbol{O(n \log^2(n))}$	$\boldsymbol{O(\log^2(n))}$	$\boldsymbol{O(n \log(n))}$

Table 4.1: Performance Table



This program was executed on the **Aristotle Cluster** provided by the **Aristotle University of Thessaloniki**. We tested our program across a range of parameters: $p = [1 : 7]$ and $q = [20 : 27]$, where the random array (to be sorted) was populated with integers in the range **1–999**. We then verified the correctness of the results using **qsort**.

In the graphic below, we observe the *performance* of the algorithm for different values of p and q. It is important to note that for $p = 1$, the algorithm effectively becomes serial, with the sorting performed entirely using **qsort**. This is because the initial sorting step accounts for the entire array.

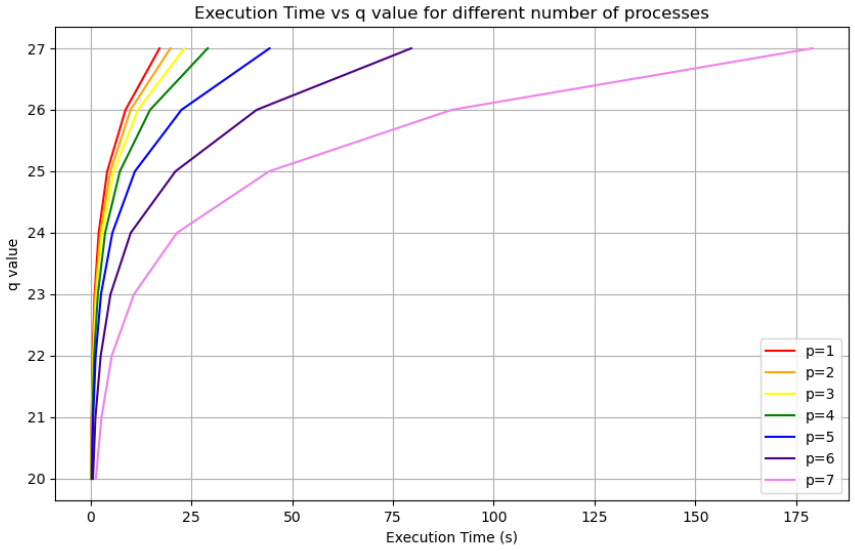


Figure 4.1: Performance

While the above graphic provides an excellent visualization of performance, it lacks a crucial element: it cannot quantify the effectiveness of the distributed approach compared to the serial one, as the number of elements to be sorted depends on both q and p.

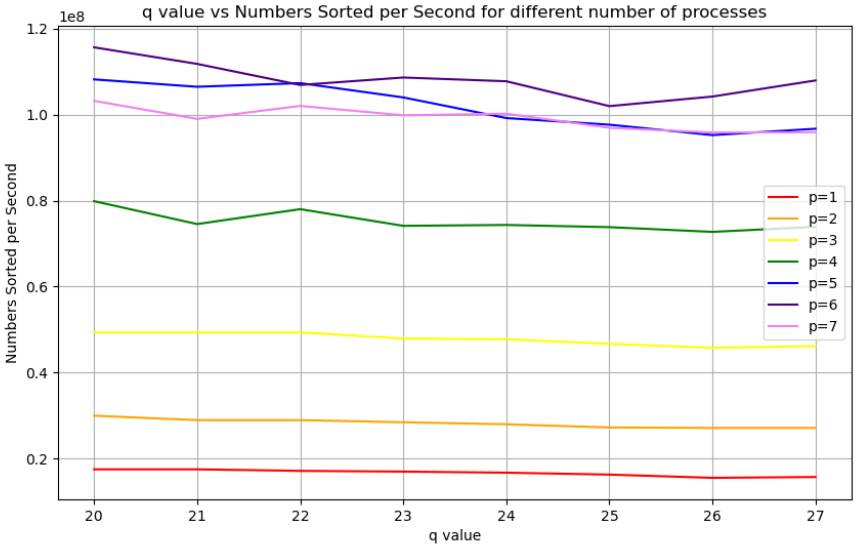


Figure 4.2: Speed

This graphic highlights the number of elements sorted per second of execution. We observe that distributing the load across multiple CPUs achieves nearly a **sixfold** improvement in sorting speed compared to the serial implementation.

Chapter 5

Tools and Sources

In this project, the following tools where used:

1. The **C** programming language.
2. The **OpenMPI** Library for implementing the distributed algorithm.
3. The **Aristotle Cluster** for testing.
4. **Neovim** text editor for development.
5. **GitHub** for version control.
6. **GitHub Copilot** as an AI assistant.
7. **Python** with **matplotlib** for graphics
8. **Gimp** for asset creation.

The following sources were helpful for understanding the problem presented in the assignment:

- `https://en.wikipedia.org/wiki/Bitonic_sorter`
- `https://www.geeksforgeeks.org/bitonic-sort/`
- `https://www.open-mpi.org/doc/v4.0/`
- Lecture notes from the **Parallel and Distributed Systems** course, written by professor **Nikolaos Pitsianis**.
- Credits to *Epameinondas Bakoulas* for keeping lecture notes and creating the distributed-algorithm.png asset.