Distributed Bitonic Sort with MPI

Maria Charisi
Maria Sotiria Kostomanolaki

January 15, 2025

Abstract

This project implements a distributed sorting program using **MPI**, based on the `Bitonic Sort` algorithm. The program sorts $N = 2^{(q+p)}$ integers in ascending order across 2^p processes, each starting with 2^q random integers. It follows the `Bitonic Sort` method, combining local sorting and data exchange between processes to achieve the final result. The program validates the sorting and compares the performance of the parallel implementation for different values of q and p. To evaluate the efficiency of the parallel approach, the program was tested on the Aristotelis cluster for high-scale performance as well as tested against `qSort` and serial `BitonicSort` to verify speedup achievement.

You can find the source code of our implementation on this Github repository.

Contents

Professor: Nikolaos Pitsianis
Department of Electrical and Computer Engineering
Parallel and Distributed Systems

1 Serial Bitonic Sort

Understanding `Distributed Bitonic Sort`, requires a first examination of the concepts of a bitonic sequence, the Bitonic Theorem, and the serial implementation of the algorithm.

1.1 Bitonic Sequence

> A **Bitonic Sequence** is a sequence of integers that first increases monotonically and then decreases monotonically (or vice versa). The reverse of a bitonic sequence or any cyclic rotation of it also forms a bitonic sequence.

To elaborate on the last point with an example, the sequence [6, 4, 3, 1, 2, 5, 8, 7] is bitonic. When the last element, 7, is rotated to the beginning, the resulting sequence [7, 6, 3, 1, 2, 5, 8] satisfies the bitonic property.

1.2 Bitonic Merge

> **Bitonic Theorem:** The element-wise comparison of the first half of a bitonic sequence with its second half, along with the separation into minima and maxima subsequences, also results in bitonic sequences.

Bitonic Merge is an algorithm that takes a bitonic sequence with a length that is a power of two and returns a sorted sequence. `Bitonic Merge` recursively applies the *Bitonic Theorem* and stores first the minima and then the maxima sequence, as demonstrated in the following example:

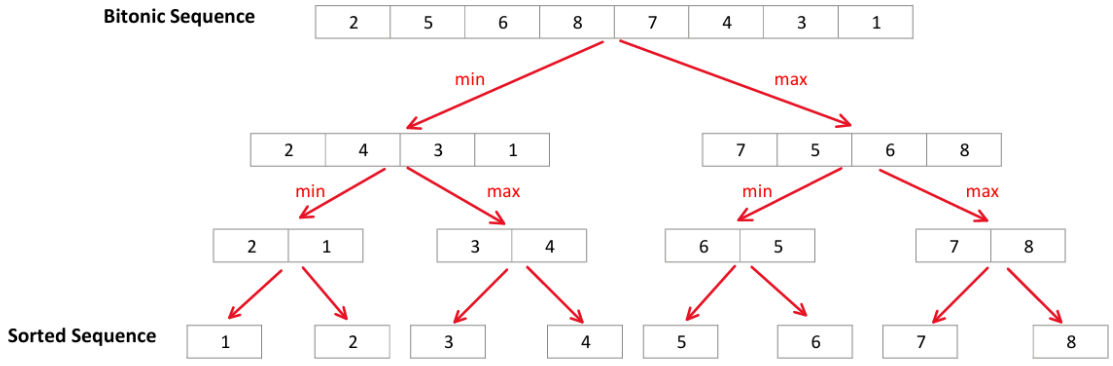

Figure 1: Bitonic Merge

If we assume that the input sequence has length n, then:

- Element-wise comparisons between the two halves of the sequence have a time complexity of $O(n)$.

- `Bitonic Merge` calls itself twice: once for the minima sequence and once for the maxima sequence and it is recursively called $log(n)$ times.

Below is the serial implementation of the `Bitonic Merge` algorithm, where `m` represents the size of the sequence or subsequence being processed in the current recursion and `min.arr[]` and `max.arr[]` are the respective minima and maxima subsequences.

```
    for(int i = 0; i < m; i++){
        if(arr[i] > arr[i+m]){
            min.arr[i] = arr[i+m];
            max.arr[i] = arr[i];
        }
        else {
            min.arr[i] = arr[i];
            max.arr[i] = arr[i+m];
        }
    }

    min = bmerge(min, order);
    max = bmerge(max, order);
```

Therefore, the **time complexity** of `Bitonic Merge` is:

$$T_M(n) = 2T_M(n/2) + O(n) = O(nlog(n))$$

1.3 Bitonic Sort

Bitonic Sort takes an arbitrary sequence of integers as input and transforms it into a bitonic sequence. It achieves this by recursively dividing the sequence, sorting each half in opposite orders. Following that, it calls `Bitonic Merge` to merge the bitonic sequence into the desired order, resulting in a fully sorted sequence.

```
    l = bsort(l, true); //First half ascending order
    r = bsort(r, false); //Second half descending order

    // Create the bitonic sequence
    for(int i = 0; i < m; i++){
        b.arr[i] = l.arr[i];
        b.arr[i + m] = r.arr[i];
    }

    b = bmerge(b, order);
```

Taking into account the algorithm above, the **time complexity** of the `Bitonic Sort` is:

$$T_S(n) = 2T_S(n/2) + T_M(n) = O(n(\log n)^2)$$

2 Distributed Bitonic Sort

Although `Bitonic Sort` is a serial sorting algorithm and is slower than other similar algorithms, such as `Merge Sort` which has a time complexity of $O(n \log n)$ compared to `Bitonic Sort`'s $O(n(\log n)^2)$, it possesses useful properties that are well-suited for parallel and distributed implementations.

The `Distributed Bitonic Sort` takes two integers, p and q, as input. The algorithm sorts a sequence of $N = 2^{(q+p)}$ integers in ascending order across 2^p processes, with each process starting with 2^q random integers. Initially, each process sorts its local elements. Then, using the properties of `Bitonic Sort` and the *Bitonic Theorem*, the algorithm uses inter-process communication via the **Message Passing Interface (MPI)** to combine the sorted local sequences and produce the final globally sorted sequence.

2.1 Step by Step Explanation

Below is a step by step analysis of the behavior of the algorithm. Please note that this analysis is concise; for a complete understanding, the reader should refer to the source code.

For the purpose of our implementation we define:

```
1  typedef struct {
2  int *arr;
3  int size;
4  } Sequence;
```

The program starts with the execution of the command `make run P=<p> Q=<q>`,which specifies the number of processes to be used (2^p) and the respective lengths (2^q) of the `Sequence`s assigned to each process.

- At first, every process sorts its local sequence in ascending or descending order, which is determined using the `ascdesc(rank,stage)` method. For this sorting, we use the `qsort()` function from the C `stdlib` library, with time complexity $O(n \log n)$.
- Next, we enter a loop which is executed exactly p times. At every iteration (`stage`), processes communicate and exchange elements with their neighbor processes. Specifically, in the first iteration, there is one step of communication, in the second one there are two steps and so on.
- These communications are not random. At each `stage` and `step`, each process communicates with a specific `partner` at a defined `distance`, which increases as the number of `step`s progresses. The processes compare their elements element-wise, and depending on the current `stage`, one process keeps the minimum values, while the other keeps the maximum values.
- At the end of each `stage`, each process holds a bitonic sequence, which is sorted using the `elbowSort` function, which has a time complexity of $O(n)$, instead of `qsort`. A detailed description of `elbowSort` can be found in **Section 2.6**.

```
    firstSort(local, ascdesc(rank, 0));
    for (int stage = 1; stage <= log2(size); stage++){
        for (int step = stage; step > 0; step--){
            int distance = 1 << (step - 1);
            int partner_rank = partner(rank, distance);
            Sequence remote = exchange(partner_rank, local);
            minmax(rank, stage, distance, local, remote);
            deleteSeq(remote); }
        elbowSort(local, ascdesc(rank, stage));}
```

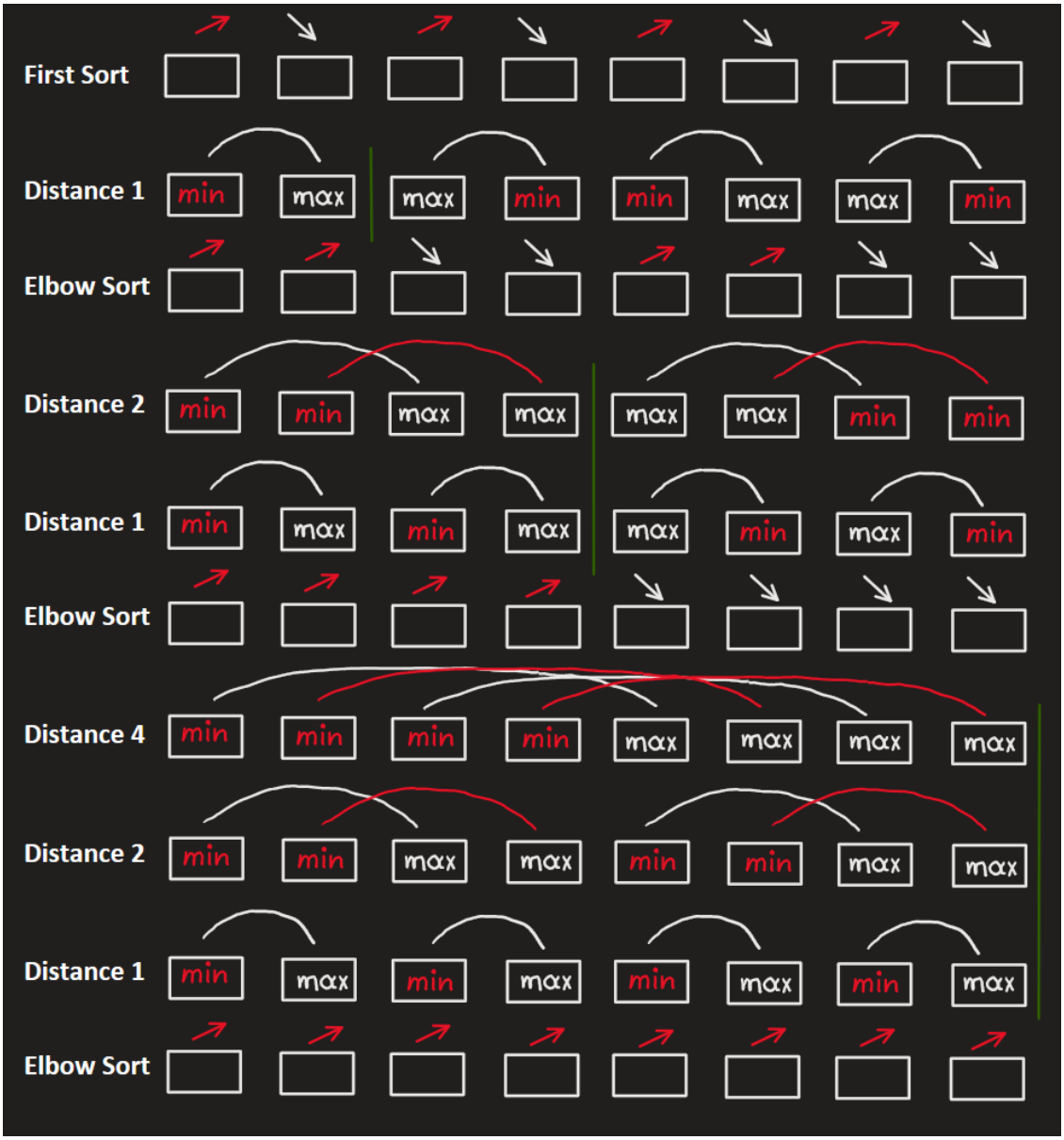

Figure 2: Example of Distributed Bitonic Sort for 8 processes

In the above figure, each rectangle represents a process. Each row is a step of the algorithm, and it is either a communication step or a sorting step.

2.2 Partner

The rank of the `partner` process can be easily calculated using the current communication `distance` and the local process's rank. The `distance` depends on the current `stage` of the algorithm and is always a power of two, ranging between $2^{\text{stage}-1}$ and 2^0. Below is a reference table of the `partner` pairs as they are formed by the following experession:

$$\text{rank} \oplus \text{distance}$$

Decimal	Binary	XOR with distance 1	XOR with distance 2	XOR with distance 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

Table 1: XOR Table

2.3 Exchange Data

Communication between processes is achieved through the `exchange` function. Each process uses `MPI_Sendrecv` to send its local data to its partner and store the received data from the partner in a buffer called `remote`.

```
1  Sequence exchange(int partner, Sequence local)
2  {
3      Sequence remote = createSeq(local.size);
4      MPI_Sendrecv(local.arr, local.size, MPI_INT, partner,
           0, remote.arr, remote.size, MPI_INT, partner, 0,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
5      return remote;
6  }
```

2.4 Maxima or Minima

After receiving data from its remote partner, each process performs an element-wise comparison between its local elements and the remote ones. It then keeps either the **minimums** or the **maximums**. This decision is crucial for the final result and depends on the current `stage` and `distance`.

The algorithm behind this decision-making can be better understood through a mirror analogy. At each `stage`, a mirror is placed in front of the process with rank 2^{stage}. These mirrors, represented by green vertical lines in **Figure 2**, have a width of $2 \times 2^{\text{stage}}$. The processes with ranks in the range $[0, 2^{\text{stage}} - 1]$ are **reflections** of those with ranks in the range $[2^{\text{stage}}, 2 \times 2^{\text{stage}} - 1]$. This pattern repeats until the end of the global sequence.

To identify a process's reflection within this mirror system, we first calculate its position relative to the mirror's width using: `pos = rank % w` ,where $w = 2 \times 2^{\text{stage}}$.

Using this position, the reflection is computed as:

- For processes on the left of the mirror (`pos` $<$ `mirror`), the reflection is `mirror`$-$`pos`-1.
- For processes on the right (`pos` $\geq$ `mirror`), the reflection is `pos` $-$ `mirror`.

With the reflection determined, the decision to keep either the **minimums** or the **maximums** is based on the `distance`. Specifically, the algorithm examines the $\log_2($`distance`$)$-th bit (counting from the right) of the process's rank:

- If the bit is 0, the process retains the **maximums**.
- If the bit is 1, it retains the **minimums**.

```
1  void minmax(int rank, int stage, int distance, Sequence local,
       Sequence remote)
2  {
3      int mirror = 1 << stage;
4      int w = 2 * mirror;
5      int pos = rank % w;
6
7      int reflection = pos < mirror ? mirror - pos - 1 : pos -
           mirror;
8
9      if (reflection & (1 << (int)log2(distance))){
10         // Keep min elements (element-wise)
11     }
12     else{
13         // Keep max elements (element-wise)
14     }
15 }
```

Below is a reference table that illustrates the logic for `stage` 2 of the `Distributed Bitonic Sort` with 8 processes. In this `stage`, there are two communication `steps`. The first `step` involves a `distance` of 2, and the second `step` involves a `distance` of 1. In both `steps`, a single mirror is "placed" in front of rank 4.

During the first communication `step`, the decision to retain either the minimums or maximums is based on the second bit from the right of the process's rank. In the second communication `step`, this decision is made using the first bit from the right of the rank.

Decimal	Rank	Reflection	Distance 1	Distance 2
0	000	011	min	min
1	001	010	max	min
2	010	001	min	max
3	011	000	max	max
4	100	000	max	max
5	101	001	min	max
6	110	010	max	min
7	111	011	min	min

Table 2: Stage 2 for 8 processes

2.5 Ascending or Descending

At the end of every `stage`, each process sorts its elements in ascending or descending order. The function `ascdesc` defines this order in a manner similar to the approach used previously with the reflection. For example, in the first sort (stage 0), we keep the first bit from the right. If this bit is equal to zero, the process sorts its local array in ascending order; otherwise, it sorts it in descending order.

$$\texttt{rank} \;\&\; (1 \ll \texttt{stage})$$

2.6 Elbow Sort

At the end of each stage, all processes hold a bitonic sequence. To increase time efficiency, the `Distributed Bitonic Sort` algorithm takes advantage of this characteristic and uses `elbowSort` for sorting instead of `qsort`.

The `elbowSort` function takes a bitonic sequence as an input and sorts it in the desired order, designated by the `ascdesc` function. It begins by identifying the **"elbow"** of the bitonic sequence, which is defined as the element for which all elements before it are monotonically decreasing (or increasing), and all elements after it are monotonically increasing (or decreasing).

It is important to note that only the **minimum** or **maximum** elements can serve as the elbow of the sequence. If these elements were placed elsewhere, they would disrupt the monotonic increase or decrease, which is a required property for the sequence to be considered bitonic. Furthermore, sequences like $[6, 4, 3, 1, 2, 5, 8, 7]$ are still considered bitonic sequences

because they always have a cyclic rotation with a correct elbow candidate, such as this one $[7, 6, 4, 3, 1, 2, 5, 8]$.

For this implementation, we have chosen to consider the **minimum** element as the elbow and take advantage of the fact that all the elements before and after it are already sorted. We start by placing the elbow in the correct position which according to the sorting order, is either at the start of the sequence, if sorting in ascending order, or at the end of it, if sorting in descending order. After that, we place **two pointers**: one to the **left** and one to the **right** of the elbow.

```
1     int l = (elbowIndex - 1 + s.size) % s.size;
2     int r = (elbowIndex + 1) % s.size;
```

The algorithm then compares the elements at the two pointers. The smaller element of the two will take the next position in the sequence and its corresponding pointer will then move one step to the right (if referring to the right pointer) or one step to the left (if referring to the left pointer). This process continues until the entire sequence is sorted.

When a pointer reaches the end of the sequence, it wraps around to the beginning (and vice versa for the opposite pointer) to ensure that cases like the one we mentioned earlier are sorted correctly.

```
1     // Sorting in ascending order
2     int sortedIndex = 0;
3     sortedSeq.arr[sortedIndex++] = s.arr[elbowIndex];
4
5     while (sortedIndex < s.size)
6     {
7         if (s.arr[l] < s.arr[r])
8         {
9             sortedSeq.arr[sortedIndex++] = s.arr[l];
10            // Wrap around to end if l < 0
11            l = (l - 1 + s.size) % s.size;
12        }
13        else
14        {
15            sortedSeq.arr[sortedIndex++] = s.arr[r];
16            // Wrap around to start if r >= s.size
17            r = (r + 1) % s.size;
18        }
19    }
```

3 Results

For testing, we executed our program on the **Rome** partition of the **Aristotle cluster**, provided by Aristotle University of Thessaloniki. The Rome partition consists of 17 nodes with 128 CPU cores per node. To test our implementation at high scale ranged, we utilized all 128 CPU cores of a single node, and the following results were obtained:

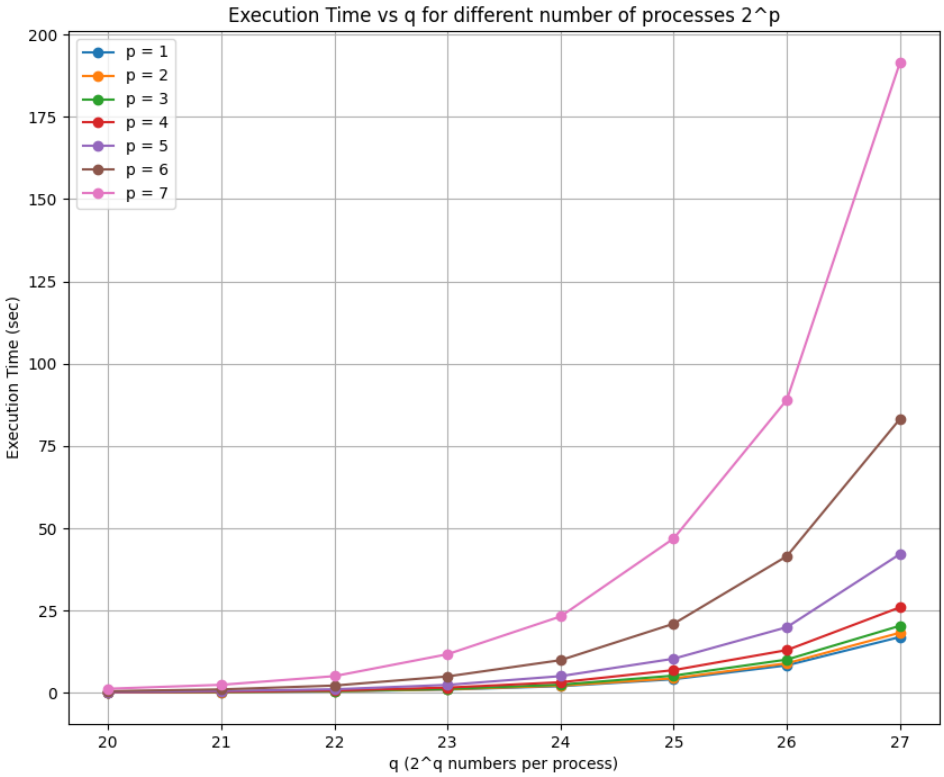

Figure 3: Performance of Distributed Bitonic Sort

To verify the speedup of our algorithm, we have created the following plot comparing the performance of `qsort` and the serial implementation of `Bitonic Sort` against `Distributed Bitonic Sort` across different datasets (varying values of q). The plot clearly demonstrates that `Distributed Bitonic Sort` outperforms both `qsort` and serial `Bitonic Sort`. While the speedup for $p = 1$ is minimal, the speedup becomes significant as the number of processes increases, particularly for datasets where $q = 21$ and $q = 22$.

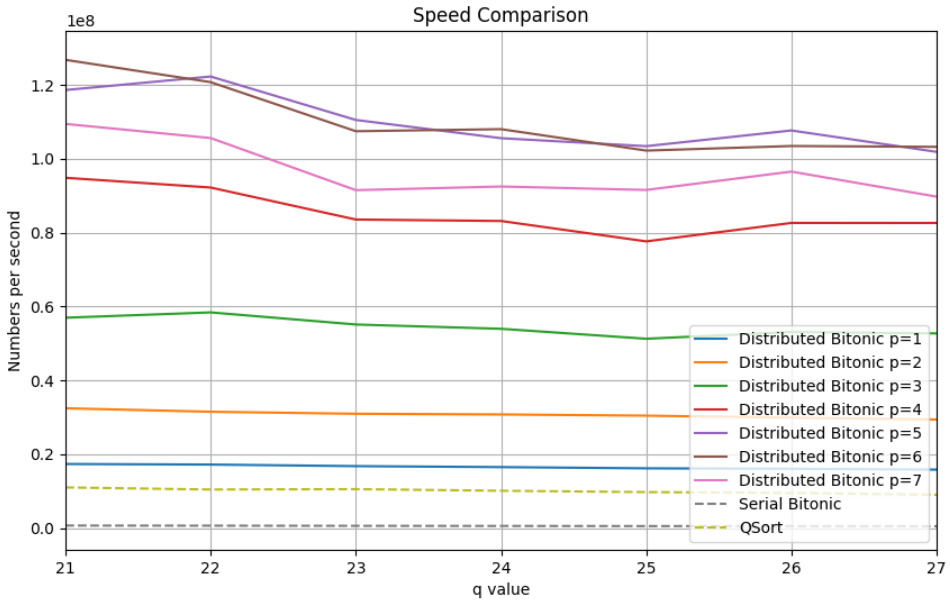

Figure 4: Speed comparison against `qsort` and serial Bitonic Sort

4 Conclusion

In this report, we explored the implementation of the `Distributed Bitonic Sort` algorithm, with a focus on leveraging parallelism through `MPI` to optimize sorting tasks in a distributed environment. The results demonstrated the scalability of our approach, showcasing the potential of `MPI` to significantly accelerate algorithms as the number of processes increases.