Distributed Bitonic Sort with MPI

Maria Charisi
Maria Kostomanolaki

January 5, 2025

Contents

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

1 Serial Bitonic Sort

Understanding *Distributed Bitonic Sort* requires first examining 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.

For example, the sequence [6, 4, 3, 1, 2, 5, 8, 7] is bitonic, because when the last 7 is rotated to the beginning, the resulting sequence is decreasing and then increasing.

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 a function that takes a bitonic sequence as input, where the length is a power of two, and outputs 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:

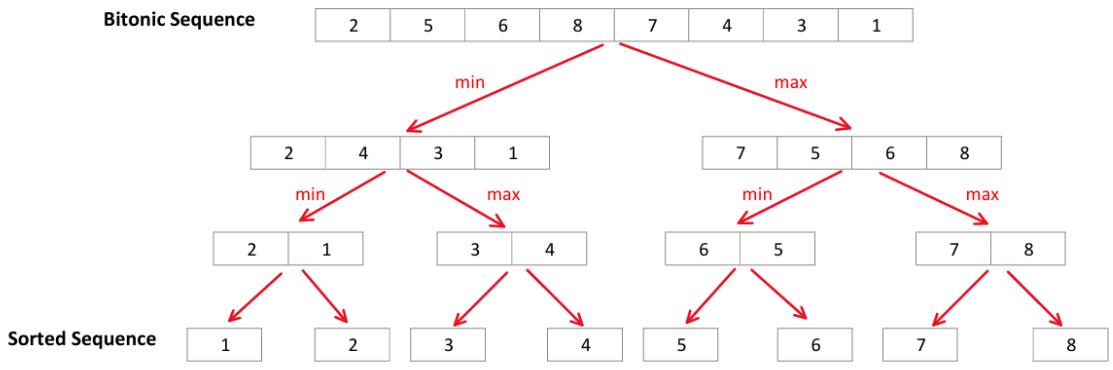

Figure 1: Bitonic Merge

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. Therefore, it is recursively called $log(n)$ times.

```
    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);
```

So, 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 calling itself, sorting the first half in one direction and the second half in the opposite direction. The next step is to call *Bitonic Merge* to sort the resulting bitonic 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

We can understand that there are serial sorting algorithms that have much better performance than the bitonic sort. Specifically, bitonic sort has $O(n(\log n)^2)$ complexity, whereas other algorithms such as merge sort have $O(n \log n)$ complexity. However, bitonic sort is useful because it is well-suited for parallel and distributed implementations.

The *Distributed Bitonic Sort* takes two integers, p and q, as input. p defines the number of processes, specifically 2^p **processes** will start, and each process will handle 2^q **numbers**. In the end, every process will sort its numbers locally. If we then gather all the numbers from all processes in order (first the chunk from the first process, and so on), we will end up with a sorted sequence in ascending order. The final sequence will have a length of $N = 2^{p+q}$. Inner-process communication will be achieved by using **Message Passing Interface (MPI)**.

2.1 Brief Explanation

As an introduction, we will briefly analyze the algorithm step by step. As we already mentioned, we start with 2^p processes, and every process takes a sequence of length 2^q.

- At first, every process will sort its sequence in ascending or descending order (based on the rank of the process). For this sorting, we will use the `qsort()` function from the C `stdlib` library, with complexity $O(n \log n)$.
- Next, we will enter a loop that will execute p times. At every iteration (stage), processes will communicate and exchange elements. Specifically, in the first iteration, there will be one step of communication, in the second, two steps, and so on.
- These communications will not be random. At each step, each process will communicate with a specific partner at a certain distance. They will compare their numbers elementwise, and depending on the current stage, one will keep the minimums, and the other will keep the maximums.
- At the end of each iteration, each process will end up with a bitonic sequence, which will be sorted again using the `elbowsort` function instead of `qsort`. The `elbowsort` function has $O(n)$ complexity and only sorts bitonic sequences.

```
1   firstSort(local, ascdesc(rank, 0));
2
3   for (int stage = 1; stage <= log2(size); stage++){
4       for (int step = stage; step > 0; step--){
5           int distance = 1 << (step - 1);
6           int partner_rank = partner(rank, distance);
7           Sequence remote = exchange(partner_rank, local);
8           minmax(rank, stage, distance, local, remote);
9           deleteSeq(remote);
10      }
11      elbowSort(local, ascdesc(rank, stage));
12  }
```

The functions `partner`, `exchange`, `minmax`, `ascdesc`, and `elbowSort` will be explained further.

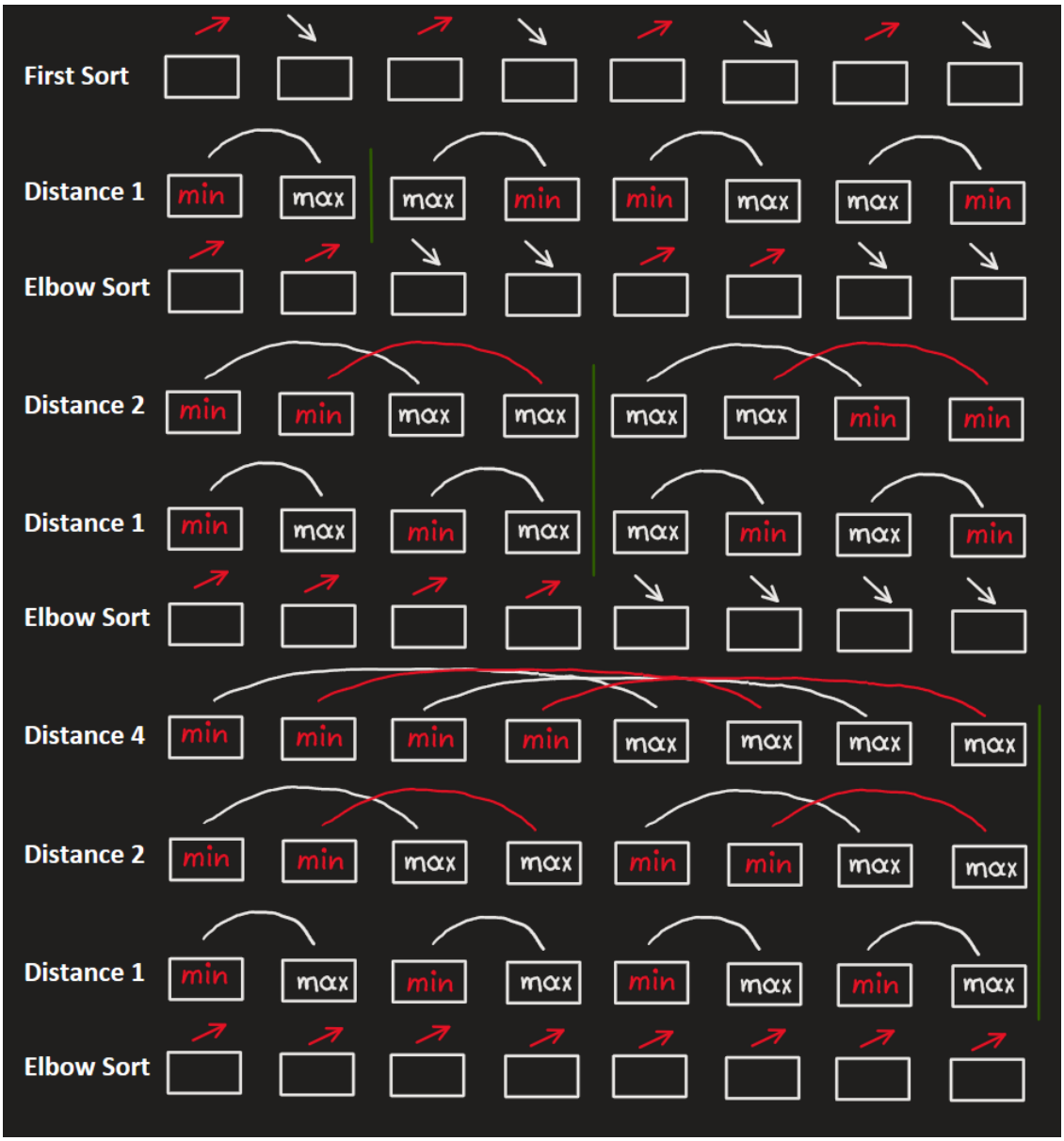

Figure 2: Distributed Bitonic for 8 processes

In the above figure, the 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 if the current communication distance is known. The distance depends on the current stage of the algorithm, is always a power of two, and ranges from $2^{\text{stage}-1}$ to 2^0

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

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

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

Each process compares all its local elements with the remote ones element-wise and keeps either the **minimums** or the **maximums**. This decision is crucial for the final result and depends on the current stage and distance.

To understand the motif behind the algorithm, we can imagine that at each stage, a mirror is placed before the process with rank 2^{stage}. These mirrors are depicted in Figure 2 with green vertical lines. Each mirror has a width equal to $2 \times 2^{\text{stage}}$, and the processes

with ranks in the range $[0, 2^{\text{stage}} - 1]$ are reflections of the processes with ranks in the range $[2^{\text{stage}}, 2 \times 2^{\text{stage}} - 1]$. This pattern repeats until the end of the sequence.

Taking advantage of this motif, we simply calculate the modulo of the rank with the width of the mirror and then find the reflection of the outcome from the first mirror. The ranks that are placed to the left and right of the mirror have a reflection equal to zero, and so on.

Now that we have the reflection, the criterion that determines whether the process will keep the minimums or the maximums is the distance. Specifically, we will retain the $\log_2$(distance)-th bit from the right of the rank. If this bit is equal to zero, the process will keep the maximums; otherwise, it will keep 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 }
```

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

As we have mentioned earlier, at the end of each stage, all processes will have a bitonic sequence. Therefore, instead of using `qsort` for sorting the sequence, we will use `elbowSort`.

The function `elbowSort` takes a bitonic sequence as input and sorts it in the desired order. It begins by identifying the **"elbow"** of the bitonic sequence. We define the elbow of a bitonic sequence as the element such that all elements before it are monotonically decreasing (or increasing), and all elements after it are monotonically increasing (or decreasing).

It is clear that only the elements with the **minimum** or **maximum** value can serve as the elbow of the sequence. If these elements were placed elsewhere, they would disrupt the monotonic increase or decrease. Furthermore, sequences like $[6, 4, 3, 1, 2, 5, 8, 7]$ are not problematic because they always have a cyclic rotation with a correct elbow candidate, like this one $[7, 6, 4, 3, 1, 2, 5, 8]$.

In our case, we choose the minimum element as the elbow and take advantage of the fact that all the elements before and after it are already sorted. We place the elbow in the correct position (at the beginning if we are sorting in ascending order, or at the end if sorting in descending order). Then, 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;
```

From this point, we compare the two pointers. The smaller element will take the next position in the sequence. The pointer corresponding to the smaller element 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 will wrap around to the beginning (and vice versa for the opposite pointer). This way, we ensure that cases like the one we mentioned earlier are sorted correctly.

```
    // Sorting in ascending order
    int sortedIndex = 0;
    sortedSeq.arr[sortedIndex++] = s.arr[elbowIndex];

    while (sortedIndex < s.size)
    {
        if (s.arr[l] < s.arr[r])
        {
            sortedSeq.arr[sortedIndex++] = s.arr[l];
            // Wrap around to end if l < 0
            l = (l - 1 + s.size) % s.size;
        }
        else
        {
            sortedSeq.arr[sortedIndex++] = s.arr[r];
            // Wrap around to start if r >= s.size
            r = (r + 1) % s.size;
        }
    }
```

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 program, we utilized all 128 CPU cores of a single node, and the following results were obtained:

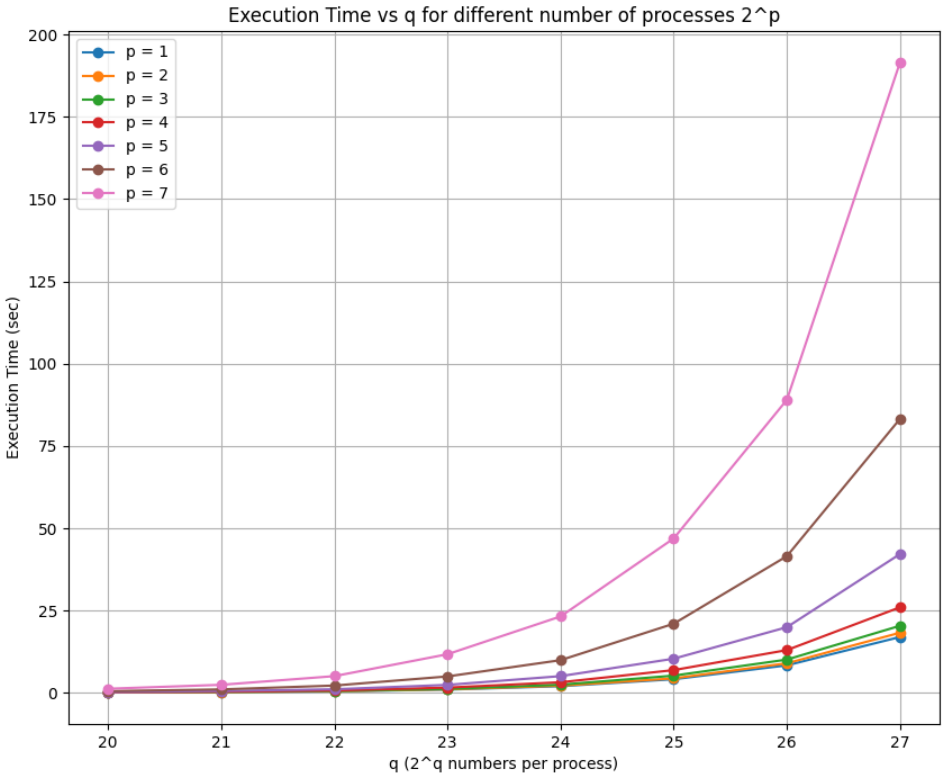

Figure 3: Performance