Hybrid Distributed Sort with Bitonic Interchanges

January 20, 2025

Author: Aristeidis Daskalopoulos
Github Repo: https://github.com/arisdask/BitonicSortMPI
A Parallel & Distributed Systems' Project

1 Intro

Problem Introduction

The primary objective of this project is the implementation of a distributed bitonic sort algorithm. Specifically, given the number of processes, denoted as p, that can execute concurrently on the Aristotelis HPC system, each process is assigned a random list of numbers of size q. *In more domain-specific applications, these datasets may not be random but instead extracted from a large predefined list.* The task is to sort all $p \times q$ elements by implementing inter-process communication using the Hamming distance. In more detail, the data exchanges follow the edges of a hypercube, whose dimensionality corresponds to the binary logarithm of the number of processes.
This work focuses on parallel systems with distributed memory and the message-passing model, MPI. The core algorithm under consideration is the 'hybrid' bitonic sort, which dictates the communication steps among processes running on different computational nodes. Each process manages a portion of the data, as well as the corresponding positions (addresses) for sorting, and exchanges data with other processes. These exchanges indicates that each process holds - at most - two lists of data, the *local and the received.* At the end of the sorting process, the data should be fully sorted and distributed across all processes. For instance, process 0 should hold the first segment of the sorted data, process 1 the next segment, and so forth, ensuring a contiguous distribution of the sorted results across the processes.

> Note
>
> To better understand the concept and implementation, we can "visualize" the system of processes as a matrix. In this abstraction, the ***rows*** of the matrix represent the different ***processes***, while the ***columns*** within each row correspond to the ***numbers held*** by each process in the distributed memory system. This matrix, denoted as A, is a $p \times q$ matrix. Such a representation provides a clearer perspective on how inter-process communications are performed and how the algorithm operates.

Repo Layout

The repository includes the following:

- The MPI Bitonic Sort implementation written in C, with a main function that validates the final sorting before returning 0. For the source code, refer to the `inc` and `src` folders.
- Scripts to build and run various test cases (for different numbers of nodes, tasks per node and elements per process). Refer to the `tests` folder to view all the scripts and test cases.
- Log files and results from running the code on the Aristotelis HPC, which document the time taken and whether the sorting was successful. A brief `report.md` file contains a table summarizing all the test runs and highlights key outcomes derived from these results. *For detailed test results, check this markdown file.*

- A Julia implementation of the Bitonic Sort algorithm, providing a better theoretical understanding of the algorithm in a Julia Pluto notebook demonstrating the algorithm. For more details about the algorithm's functionality, refer to the notebook in the `julia` folder.
- A detailed `README.md` file with instructions on how to set up and run the project correctly. For more information about the repository, refer to this README file.

2 Theoretical Algorithm Analysis

In this section, we introduce the bitonic sort algorithm that has been implemented. We explain its theoretical foundation, detailing how it functions step-by-step. Following this, we calculate its complexity, identify areas for potential optimization, and compare its performance with the classical serial approach to sorting.

2.1 First Algorithmic Approach

Algorithm 1 Initial Implementation of Hybrid Bitonic Sort

Require: ***local_row***: the array of integers that the process holds, ***rows***: total number of processes, ***cols***: size of local array, ***rank***: current process rank

Ensure: *Full sorted array, distributed across all processes*

stages $\leftarrow \lfloor \log_2(\text{rows}) \rfloor$
received_row $\leftarrow$ AllocateBuffer(cols) ▷ Pre-allocate communication buffer
InitialAlternatingSort(local_row, cols, rank)
Synchronize() ▷ MPI Barrier
for stage $\leftarrow$ 1 to stages **do**
    num_chunks $\leftarrow 2^{\text{stages}-\text{stage}}$
    chunk_size $\leftarrow$ rows/num_chunks
    chunk $\leftarrow \lfloor$rank/chunk_size$\rfloor$
    is_ascending $\leftarrow$ (chunk mod 2 = 0)
    for step $\leftarrow$ stage $-$ 1 downto 0 **do**
        partner $\leftarrow$ rank $\oplus\ 2^{\text{step}}$ ▷ XOR for Hamming distance
        if rank $\geq$ partner **then**
            Send(local_row, partner) ▷ Send current local row to partner
            Receive(local_row, partner) ▷ Receive new local row after 'PairwiseSort'
        else if rank $<$ partner **and** partner $<$ rows **then**
            Receive(received_row, partner) ▷ Receive local row of partner
            PairwiseSort(local_row, received_row, cols, is_ascending)
            Send(received_row, partner) ▷ Send new local row after 'PairwiseSort'
        end if
        Synchronize() ▷ MPI Barrier
    end for
    ElbowSort(local_row, cols, is_ascending)
    Synchronize() ▷ MPI Barrier
end for
FreeBuffer(received_row)

2.2 Explaining the Algorithm

Here, we provide a detailed explanation of how the above algorithm works. We describe the key stages of the algorithm, the role of each process, and how data exchanges occur to achieve the final sorted outcome.

1. **Initialization:**
   - Compute the total number of stages as `stages` $= \log_2($`rows`$)$. The stages of the algorithm execute sequentially, and they are defined based on the total number of processes (or rows, as described in the matrix analogy introduced earlier). Each stage coordinates the sorting steps across processes, ensuring that data is progressively aligned and sorted according to the bitonic sort logic.
   - Allocate a buffer for communication between processes. This buffer is used to store the list of numbers received from the process it is communicating with.
2. **Initial Alternating Sort:**
   - Perform an initial alternating sort on the local row using the function `initial_alternating_sort`.
   - Synchronize all processes using an `MPI_Barrier`: We want to ensure that all processes performed their local sort (ascending or descending based on the process id/rank) before starting the communications needed at each step.
3. **Iterative Bitonic Stages:**
   - For each stage from 1 to `stages`:
     (a) Compute the number of chunks and chunk size:

$$\texttt{num\_chunks} = 2^{\texttt{stages}-\texttt{stage}}, \quad \texttt{chunk\_size} = \frac{\texttt{rows}}{\texttt{num\_chunks}}$$

     In all stages, except for the final one, not all processes need to communicate with each other. For instance, in stage 2, the initial communication occurs between `pid 0` and `pid 2`, as well as `pid 1` and `pid 3`. Subsequently, communication happens between `pid 0` and `pid 1`, and between `pid 2` and `pid 3`. In this stage, `pid 0` to `pid 3` operate independently from the other processes, ***forming their own chunk***. At each step, these chunks define which processes will communicate exclusively with each other.

     (b) To determine the chunk to which the current process belongs and whether the sort should be in ascending order, we have:

$$\texttt{chunk} = \lfloor \texttt{rank}/\texttt{chunk\_size} \rfloor, \quad \texttt{is\_ascending} = (\texttt{chunk} \bmod 2 = 0)$$

     After the internal communication between the correct partners within each chunk, the chunk should be fully sorted in ascending order if `is_ascending` is `true`; otherwise, it will be sorted in descending order. Achieving this requires executing the appropriate internal communication between the designated partners and applying the correct functions accordingly.

     (c) Perform internal iterative steps for each bitonic merge stage:
       i. Compute the communication partner of the local process based on Hamming distance:

$$\texttt{partner} = \texttt{rank} \oplus 2^{\texttt{step}}$$

We begin with `step` = `stage` − 1 and decrement it until it reaches zero. This approach ensures that communication starts with the most "distant" processes and progressively moves to the nearest ones. Within each chunk, this part must also execute sequentially: first handling the more distant communications and then reducing the distance to handle closer ones. This process forms a loop that consists of $\log_2($`rows`$)$ steps.

ii. We have two cases:
   - If `rank` ≥ `partner`:
     A. Send local row to partner process
     B. Receive sorted row back from partner process
   - If `rank` < `partner` and `partner` < `rows`:
     A. Receive row from partner process
     B. Perform pairwise sort, of local row and received row, based on is_ascending flag
     C. Send sorted received row back to partner

iii. Synchronize all processes using `MPI_Barrier`

(d) Perform the elbow sort on the local row using the function `elbow_sort`.

(e) Synchronize all processes using `MPI_Barrier`.

4. **Cleanup:**
   - Free the allocated communication buffer.

2.3 Time Complexity

From the algorithm's explanation, we derived that certain iterations must be performed sequentially. The first such iteration is the number of stages required to complete the sorting. This part has a total time complexity of $O(\log p)$, where p is the total number of processes (or rows in our matrix analogy). For each stage, we also perform $O(\log p)$ steps: initially communicating with the most distant partners, followed by communication with the nearest ones. Therefore, the total time complexity is:

$$O(\log^2 p) \times (\text{time complexity of the communication} + \text{pairwise sort time}) + O(\log p) \times \text{local elbow sort time}.$$

The elbow sort itself takes $O(2 \cdot n)$ time: one n for finding the elbow and another n for sorting the two parts (n is the number elements/numbers the local list has, in our matrix analog is represented by the number of columns of the matrix). The pairwise sort maintains the smallest values from each pair within the local process if `is_ascending` == `true`. Specifically, given two lists `local_list` and `received_list`, the i-th pair is the element $[$`local_list`$[i],$ `received_list`$[i]]$. Thus, the time complexity for this process is also $O(n)$.

Of course, we cannot complete the time complexity analysis without addressing the initial local sorting algorithm. This algorithm has a time complexity of $O(n \cdot \log n)$, and as we can observe, it is the second most time-inefficient term, following the term related to the communication between processes.

2.4 Algorithm's Optimization

Focusing on the two terms that, in the previous analysis, seem to significantly slow down the implemented algorithm, we propose the following solutions:

For the initial `qsort`:

- If each process supports only one thread, we cannot improve the time complexity beyond $O(n \log n)$.
- However, if we have access to multiple threads (let's say t, where $t \leq n$ and $t \mid n$ - t divides n), we can implement a multithreaded bitonic sort. The practical implementation is nearly identical to the distributed version, with the main difference being the need to clean up and implement a wrapper function.
- Because this aspect is not a primary objective, and the main focus of this project is to 'enhance' the communication time between processes, the provided/tested timing results are obtained by simply applying the `qsort` algorithm to the local data.

For the communication time *we implemented* the following optimizations to our algorithm:

- There are two types of communication: *send* and *receive*. When receiving the data to perform the pairwise sort, we do not need to send the entire list in a single piece. Instead, we can send it in multiple smaller pieces (let's say s). After receiving the first piece, we can begin running the first part of the pairwise sort.
- In the initial algorithm, the process sends its local values to its partner. The partner performs a pairwise sort operation (retaining either the minimum or maximum values as explained) and then returns the updated list to the original process. However, we can *eliminate this second transmission*, following the pairwise sort, by introducing an additional send operation during the initial exchange of local values. In this approach, the process and its partner exchange their initial local values *simultaneously*, allowing both to independently perform the corresponding pairwise sort operation. Each retains the appropriate values based on their respective process ID.

2.5 Comparison with the Serial Approach

The total sorting problem, given the segmentation we already follow, has a size of $p \times n$, where p is the number of processes and n is the number of elements per process. A classical serial approach would involve implementing a merge sort or a quicksort, resulting in a total time complexity of $O(n \cdot p \cdot \log(n \cdot p))$. However, this does not account for the fact that the entire problem, given the restrictions on p and n, could result in a data size that reaches 68.8GB! This means that a serial implementation would require writing data to disk or necessitate extremely costly RAM, exceeding 70GB. From this, we can conclude that *the hybrid distributed sort* we proposed and implemented makes the whole procedure ***practically feasible***.

Now that we have established the theoretical foundation, we proceed to examine the validation tests and corresponding results.

3 Validation Tests & Results

In this section, we will describe the methodology used for validation and the calculation of results. To maintain clarity and keep this section free of results and diagrams, these have been included in the `report.md` file located in the `/log` folder of our repository. Additionally, this markdown file contains diagrams that visualize the performance of our algorithm executed on the "Aristotelis HPC".

3.1 Validation Tests Explained

The validation process is executed immediately after the completion of all bitonic interchanges. It is divided into two parts:

- **Part 1: Local Sorting Validation** Each process verifies that the data it holds is locally sorted. This step ensures that all bitonic sequences were sorted locally correctly using the *elbow sort* function.
- **Part 2: Global Sorting Validation** After confirming the correctness of local sorting for each process, the global sorting correctness is validated. This is achieved as follows: each process, from rank 0 to (total processes -1), sends its last value to the next process in sequence (e.g., process 0 sends to process 1, process 1 sends to process 2, and so on). The receiving process compares the received value with its first local element. If the received value is greater than the first local element, *the sorting is deemed incorrect.* Once all processes complete this check, they send a boolean value (`true` for correct sorting, `false` for incorrect sorting) to rank 0. If one or more processes report `false`, rank 0 prints a message indicating that the sorting was incorrect.

3.2 Explanation of the log files

The `/log` directory contains the results of executing all the `.sh` test scripts located in the `/tests` folder. For each test run, two files are generated: a `.out` file and a `.err` file.

- `.out` **Files:** These files include the total sorting time in milliseconds, as well as an indicator confirming whether the sorting was successfully completed.
- `.err` **Files:** These files provide detailed information about any issues that occurred during execution, particularly if the sorting was unsuccessful. Common issues might include the job being terminated due to time limits or other runtime errors.

Each file follows the naming format `A_B_C.*`, where `A` represents 2^A numbers per process, `B` represents 2^B processes and `C` is the job's ID.

3.3 Nodes and Tasks per nodes

As noted in `report.md`, when we aim to use, for example, 32 processes and allocate them across 4 nodes with 8 tasks per node, the execution time is generally less efficient compared to using fewer nodes with more tasks per node. This behavior arises because inter-node communication is significantly more costly than intra-node communication among tasks within the same node.

Inter-node communication involves data transfer across the network, which introduces additional latency and bandwidth limitations compared to intra-node communication, which leverages faster data exchange mechanisms.
Conversely, keeping more tasks within the same node is far more efficient due to higher throughput. This efficiency highlights the importance of optimizing the distribution of tasks across nodes to minimize communication overhead and improve overall performance.