Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
Parallel and Distributed Computer Systems

Nikolaos Kylintireas
December 2024 / January 2025

source code: https://github.com/NicKylis/MpiSort

MPI Array Sorting Solution with Bitonic Sort

1 Introduction

The Bitonic Sort algorithm is a sorting algorithm that operates efficiently for parallel and distributed systems, leveraging the Bitonic sequence properties. In this project, a parallel implementation of Bitonic Sort using the Message Passing Interface (MPI) has been developed to sort an array of integers distributed across processes. The MPI framework facilitates the exchange of data between processes to ensure efficient sorting.

Each process is assigned a chunk of the overall array, and the processes communicate with each other to exchange data and merge their results progressively. The implementation steps are as follows:

- Data Distribution:

  The array of size N is divided into smaller chunks of size N/p (p = number of processes), with each process receiving an equal portion of the data. Each process is responsible for sorting its local array segment using the built-in qsort() function from the standard C library (stdlib).

- Communication of the partners:

  The core idea of the parallel Bitonic Sort is that, at each step, each process exchanges data with its neighboring processes. The direction of the merge (ascending or descending) is determined on the basis of the rank of the processes involved. After exchanging data, each process sorts the merged data locally using Bitonic Merge (or elbow sort).

- Validation:

  After sorting, the root process gathers the sorted data from all processes. The final sorted array is validated by comparing it with the result obtained by sorting the entire array using the qsort() function.

2 Algorithm design

The parallel Bitonic Sort implementation using MPI is structured into several key components, each of which contributes to efficient and accurate distributed sorting. The following pseudocode summarizes the function of the code.

Algorithm 1 Parallel Bitonic Sort using MPI

```
1: Input: Array of integers $A$ of size $N = 2^{q+p}$
2: Output: Sorted array $A$ in ascending order
3: Initialize MPI:
4:    Rank $r$, Total processes $p$ are initialized using MPI functions
5: Local Sorting:
6: for each process $r$ from 0 to $p-1$ do
7:     Sort $A_r$ locally using qsort (ascending or descending based on rank)
8: end for
9: Parallel Bitonic Sort:
10: for step = 1 to $p$ do
11:     $dist = 2^{(step-1)}$
12:     for each process $r$ from 0 to $p-1$ do
13:         if process $r$ and partner $r'$ need to exchange data then
14:             Exchange data between $A_r$ and partner $A_{r'}$ using MPI_Sendrecv
15:             Merge data using Bitonic Merge:
16:             for each pair of elements in $A_r$ and $A_{r'}$ do
17:                 if (merge condition is satisfied) then
18:                     Swap elements between $A_r$ and $A_{r'}$
19:                 end if
20:             end for
21:         end if
22:     end for
23: end for
24: Final Gathering and Validation:
25: if process == root process then
26:     Gather all sorted subarrays $A_r$ from all processes using $MPI\_Gather$
27:     Sort the gathered array using qsort for validation
28:     Compare the MPI sorted array with the reference sorted array to check correctness
29: end if
30: Finalize MPI
```

2.1 MPI Bitonic Sort function

This function is the main function of the program, used to sort the Array of N elements. It takes as parameters two buffers, local data and local size, an integer (rank) to keep track of the process' ID and the total number of processes initialized by MPI, p.

The main task of this function is to orchestrate the exchanges between the partners. Each partner is determined based on the algorithm (1) that calculates the distance among them during each stage (step) of the merging process.

$$distance = 2^{step-1} \tag{1}$$

As the algorithm progresses, the distance increases, leading to more distant communication between processes to merge increasingly larger subsequences.

3 Execution Time

3.1 Time Complexity

Considering both computation and communication, the parallel time complexity of the algorithm can be represented by the following recurrence relation:

$$T(p) = O\left(\frac{N}{p}\log\frac{N}{p}\right) + O(\log p) \tag{2}$$

Where:

- $O\left(\frac{N}{p}\log\frac{N}{p}\right)$ accounts for the local sorting performed by each process.
- $O(\log p)$ accounts for the communication and merging steps.

This explains why, for large arrays, parallel sorting can be faster than serial sorting, assuming $\frac{N}{p}! \approx N$ and that $logp << N$.

3.2 Performance Analysis and Results

The algorithm's performance was tested on a local 8-core device. For this test, N was set to 2^{20} and p to 8. The following results shown in Figure 1 proved that parallelizing the Bitonic Sort algorithm improves performance as more processes are used, but the improvement decreases as p increases. Beyond a certain threshold, the overhead of communication and synchronization among processes likely outweighs the benefits of parallelism.

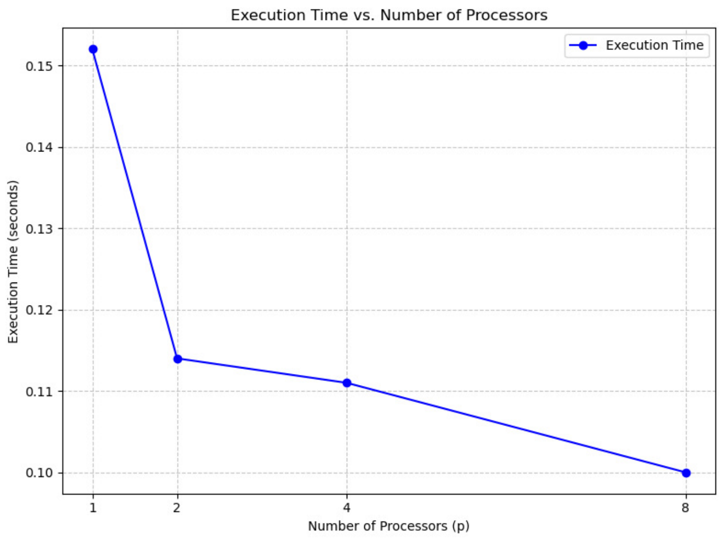

Figure 1: The decrease of execution time using 1-8 processes

4 Tools used

The following list contains all the tools used for this project:

- C programming language/compilers/standard libraries
- Visual Studio Code
- Windows Subsystem for Linux (WSL)
- OpenMPI library
- Jupyter notebook
- ChatGPT for debugging
- Latex with Overleaf
- source control with git/github