

# Report of Distributed Sorting using MPI

Edoardo Coli

March 2024

# 1 Introduction

The objective of this report is to practice into the implementation and performance analysis of a distributed merge sort algorithm using Message Passing Interface (MPI). Specifically, I will focus on the limitations associated with the Merge Sort algorithm, considering factors such as the varying sizes of the files and the benefits that can be gained from exploiting cache and the “near” memory. Moreover, I will perform scalability tests to analyze the algorithm's efficiency, assessing its ability to scale effectively under different workloads and resource capacities.

In the following sections of this report, I will outline the setup of the experimental cluster, my development choices to gain the most from the divide and conquer paradigm and some graph to discuss efficiency and performance.

## 2 Reference Setup

The name chosen for this cluster is *Steffe*. In its mass storage I have created different files of random bits, using bash command '`dd`', with different sizes to perform sorting. Writing numbers directly in binary rather than using ASCII (American Standard Code for Information Interchange) characters can be more efficient in terms of both storage space and processing speed. Besides, ASCII introduces additional overhead because it is designed to represent a variety of characters, beyond just numbers, by using an encoding standard that needs 7 or 8 bits to represent each.

However, it's important to notice that using binary representation makes the data less human-readable compared to ASCII. For this reason, we can use a bash command '`od -t d8 -A n binaryfile.bin`', that allows us to inspect a binary file knowing that its purpose is to encode 64-bit numbers.

### 2.1 Physical machine

Before start this is a detailed overview of the hardware specification of the cluster, Rock4+ nodes and the TP-Link switch and the Western Digital Red mass storage, highlighting their key features and capabilities:

1. Rock4+ Single-Board Computers:

- CPU: Rockchip PK3399 Hexa-core processor (Dual-core Cortex-A72 and Quad-core Cortex A53)
  - GPU: Mali-T840MP4
  - RAM: 4GB LPDDR4 dual channel
  - Connectivity: Gigabit Ethernet (POE)
  - Operating System: Linux-based distribution
2. TP-Link Switch (Model: TL-SG1428PE)
    - Ports: 26x 10/100/1000 Mbps RJ45
    - PoE Ports: 24x PoE (802.3at/af)
    - Power Budget: Up to 350W
  3. Western Digital Red (Model: WDC WD40EFAX-68JH4N1)
    - Capacity: 4TB
    - Form factor: 3.5 inches
    - Sector size: 512 bytes logical, 4096 bytes physical

The network topology of the cluster has 21 nodes, where one node is designed as the master (referred to as ‘steffe0’) and the remaining nodes are connected as a star topology. The mass storage of the cluster is interfaced with the host computer through a USB connection, it is achievable to all other nodes given the fact that there is a shared filesystem. Every single node is equipped with a microSD on which is in the operating system. The remaining storage in each node is not important for our analysis so we can forget about it.

## 2.2 Cluster handler

Orchestrating resource allocation and task scheduling can be very difficult to manage with. For this reason, we decided to use the open-source workload manager called Slurm\*. It provides a framework for starting, executing, and monitoring works (generally a parallel job) on the set of allocated nodes.

To run our jobs, we used a script which contains multiple commands, to achieve a certain task with the right parameters such as: Partition, Number of nodes, Time of work, ...

---

\* <https://slurm.schedmd.com/overview.html>

```
#!/bin/bash
#SBATCH --job-name=$(JOBNAME)
#SBATCH --partition=$(PARTITION)
#SBATCH --time=$(TIME)
#SBATCH --mem=$(MEM)
#SBATCH --nodelist=$(NODELIST)
#SBATCH --cpus-per-task=$(CPUS_PER_TASK)
#SBATCH --ntasks-per-node=$(NTASKS_PER_NODE)
#SBATCH --output $(OUTPUT)
#SBATCH --error $(ERROR)
mpirun $(NAME) $(ARGS)
```

Commands are executed by command line interpreter that for Slurm is called **sbatch**. Sbatch exit immediately after the script is successfully transferred to the Slurm controller and assigned a Slurm job ID; the system is not necessarily granted resources immediately so our job may be in a pending state until its required resources become available.

Useful commands that can be utilized in the command line to have an overview of the system queue status are:

- **sinfo**
- **squeue**
- **scancel**

### 3 Sorting Algorithm

Merge sort operates on a simple yet powerful principle: the *divide et impera* strategy. Merge sort excels in breaking down the complexity of sorting a large dataset, the algorithm initiates by dividing the unsorted array into two equal halves, recursively applying this process until each subset consists of a single element. This division ensures that we have the smallest, sorted components ready for merging. These sorted subsets are systematically combined to produce larger, ordered arrays, when we go up the recursive call tree. One of the features of Merge sort is its time complexity of  $O(n \log n)$ , where 'n' represents the number of elements to be sorted, but on the other side it's not an in-place algorithm.

In this context, the distributed Merge sort algorithm leverages MPI, a widely adopted communication protocol for parallel computing, to distribute the sorting workload across multiple nodes.

### 3.1 Merge limitation

Usually, we are used to approach the design and analysis of algorithms assuming a very simple model of computation, known as *Von Neumann model*. This model consists of a CPU and a memory of infinite size, with constant access time for each of its memory cells. Nowadays a modern computer consists of one or more CPUs and a very complex hierarchy of memory levels. Each of these memory levels has its own cost, latency and bandwidth; is common that the closer a memory level is to the CPU the smallest and faster it is, and we must take into account those characteristics.

Returning to our algorithm time complexity of Merge sort can be modeled by a recurrence relation<sup>[1]</sup>

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

Let us now assume that the size of numbers to be sorted  $n$  is bigger than the memory  $M$  of the node, so that must be stored on disk and I/Os become the most important computational resource to be analyzed and minimized. We notice that  $O(z/B)$ , where  $B$  is the disk page size, is the cost of merging two ordered sequences of  $z$  items. This holds if  $M \geq 3B$ , because the Merge procedure needs  $2B$  for read the elements and another  $B$  to write the merged one. As a result, the recurrent relation for the I/O complexity of Merge sort can be rewritten as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\frac{n}{B}\right) = O\left(\frac{n}{B} \log n\right) \text{ I/Os}$$

So, when a subsequence of length  $z$  fits in internal memory then it is entirely cached using  $O(z/B)$  I/Os, and for this the subsequent sorting steps do not incur in any I/Os miss. The overall saving leads to reformulate the complexity as  $\Theta\left(\frac{n}{B} \log \frac{n}{M}\right)$  I/Os.

### 3.2 K-way merge

In order to generate ordered sequences of numbers, denoted as Runs, we engaged all nodes to partition the original file into equal slices, stored after sorting in temporary files. This step is imperative because the entire file may exceed the available memory capacity. Following the completion of the partitioning and sorting stages, the next task is to consolidate these ordered Runs into a single comprehensive file, mirroring the initial one but in a sorted order. Employing a 2-by-2 merging strategy results in the formation of a binary tree, until at the end we will have the sorted file. The conventional method becomes less optimal if, at some point, the size of the merged file definitively surpasses  $M$ .

An alternative approach for file merging involves buffering the file to be merged and gradually writing it down. It's essential to keep in mind that within a 2-level memory hierarchy, there exists a more efficient method for merging these files: by utilizing a heap to facilitate a  $k$ -way merge. This merging process takes  $O(\log_2 k)$  time per item, and gain  $O(z/B)$  I/Os to merge  $k$  runs of total length  $z$ . As a result, the  $k$ -way merging scheme recalls a  $k$ -way tree with  $O(n/M)$  leaves. Elements in the heap are pairs containing a value and its corresponding file descriptor. When a value is popped out, a new value from the same file descriptor is inserted until all files are read entirely.

### 3.3 Future works

As previously mentioned, we used all available nodes to partition the original file into equal slices. However, this becomes impractical if the file size exceeds the combined memory capacity of all nodes. To avoid this situation in the code is set a condition that divides the total file in slices, each limited to the maximum computational capacity of the node. The nodes continue processing these slices until the entire file is processed.

It would be optimal if we could enhance the size of the initial (sorted) runs generated by each node boosting the available RAM. Analyzing the I/O complexity reveals that a larger " $M$ " results in fewer merge passes over the data. To achieve improved performance without physically upgrading the memory of the nodes we can use a trick, based on the elegant idea, called Snow-Plow and attributed to Donald Knuth. Implementing this will allow us to increase virtually the memory by a factor of two on average.

## 4 Implementation

The algorithm begins by dividing the input dataset into slices, with each slice assigned to a different MPI process. The MPI processes read their designated portions of the dataset from the input file using MPI file I/O functions. The use of MPI ensures efficient communication and data distribution across nodes. Once the data is read, each MPI process performs an in-memory sorting of its slice using the standard C++ `std::sort` algorithm. This step ensures that each process independently handles its data in parallel, taking advantage of available resources. To optimize read operations, the implementation supports both buffered and non-buffered modes. The buffer size is configurable, allowing for performance comparison between the two modes. The flexibility to toggle between buffered and non-buffered modes provides insights into the impact of I/O optimization on overall sorting performance.

After sorting, the algorithm writes the sorted runs to temporary files. Each MPI process has its corresponding temporary file, which contains the locally sorted data. This step prepares the data for the subsequent k-way merge process. The final phase involves a k-way merge to combine the locally sorted runs into a single sorted output file. This merging process is executed on a single node in a sequential manner, as MPI is not necessary for merging on a single node.

The coordination and synchronization of the parallel sorting algorithm rely on various MPI functions. Here are key MPI functions used in the implementation:

- `MPI_Init(&argc,&argv)` : Initializes the MPI environment
- `MPI_Comm_size(MPI_COMM_WORLD,&mpiSize)`:  
Retrieves the number of processes in the communicator
- `MPI_Comm_rank(MPI_COMM_WORLD,&mpiRank)`:  
Retrieves the rank of the calling process in the communicator
- `MPI_Bcast(&id,1,MPI_INT,0,MPI_COMM_WORLD)`:  
Broadcasts a message from the root process to all other processes in the communicator
- `MPI_File_open(MPI_COMM_WORLD,argv[1],  
MPI_MODE_RDONLY,MPI_INFO_NULL,&file)`: Opens an MPI file

- `MPI_File_read_at(file,currentOffset*sizeof(int64_t),bufferRead,elementsToRead,MPI_INT64_T,&status):`  
Reads data from a file at the specified offset
- `MPI_File_write_at(tmpFile,offset,bufferWrit,count,MPI_INT64_T,&status):` Writes data to a file at the specified offset
- `MPI_File_write_at(tmpFile,offset,bufferWrit,count,MPI_INT64_T,&status):`  
Writes data to a file at the specified offset
- `MPI_Barrier(MPI_COMM_WORLD):` Blocks the caller until all processes in the communicator have called it
- `MPI_Finalize():` Cleans up the MPI environment

## 4.1 Workflow

To execute the program, or through the use of the Makefile to automate the entire workflow, we can follow this pipeline to understand the necessary files and commands to orchestrate everything.

The initial phase involves the compilation of the source code using MPI compiler '`mpicxx`'. Following successful compilation, we need to run the program using '`mpirun -n <numProcesses> sortingAlg.exe <inputFile>`' or like in the Makefile can be created a specialized launcher script, aptly named *launcher.sh*. This script is designed to be executed within the Slurm job scheduler environment through the '`sbatch`', it encapsulates all the pertinent parameters essential for configuring and executing the Slurm job effortlessly.

The Makefile orchestrates the creation of a specialized launcher script, with fixed parameters, but it relies on the input of two parameters: NODES to specify externally the list of nodes in our cluster that we want to use in the computation; ARGS to specify the file that we want to parse.

`'make run NODES="2-7,9-10" ARGS=/mnt/raid/testlists/8Gib.bin'`



## 5 Scalability test

In our parallel sorting algorithm, scalability is evaluated by varying both the size of the input dataset and the number of MPI processes. The algorithm's performance is assessed across different combinations of these parameters to evaluate its ability to efficiently handle increasingly larger datasets and exploit additional computational resources.

Scalability in parallel computing is often classified into: Strong scalability measures the algorithm's performance as the problem size remains constant, and the number of processing elements (MPI processes) increases. In the context of our sorting algorithm, we examine how well the execution time scales as we distribute a fixed-size dataset across an increasing number of MPI processes. The goal is to determine the algorithm's ability to efficiently utilize additional resources without degrading performance. Weak scalability, on the other hand, evaluates the algorithm's performance as both the problem size and the number of processing elements scale proportionally. In our case, this involves increasing both the dataset size and the number of MPI processes simultaneously. The objective is to test the algorithm's capability to handle larger datasets and a growing number of processing elements while maintaining performance.

In parallel computing, accurately measuring the execution time of algorithms is important for performance evaluation. `MPI_Wtime` is a valuable tool for measuring the execution time of parallel applications. This function returns a timestamp with high precision, allowing for accurate timing measurements in parallel environments. In our context, `MPI_Wtime` is employed to capture the start and end times of various operations, enabling precise time measurements during the execution of the parallel sorting algorithm.

### 5.1 Execution

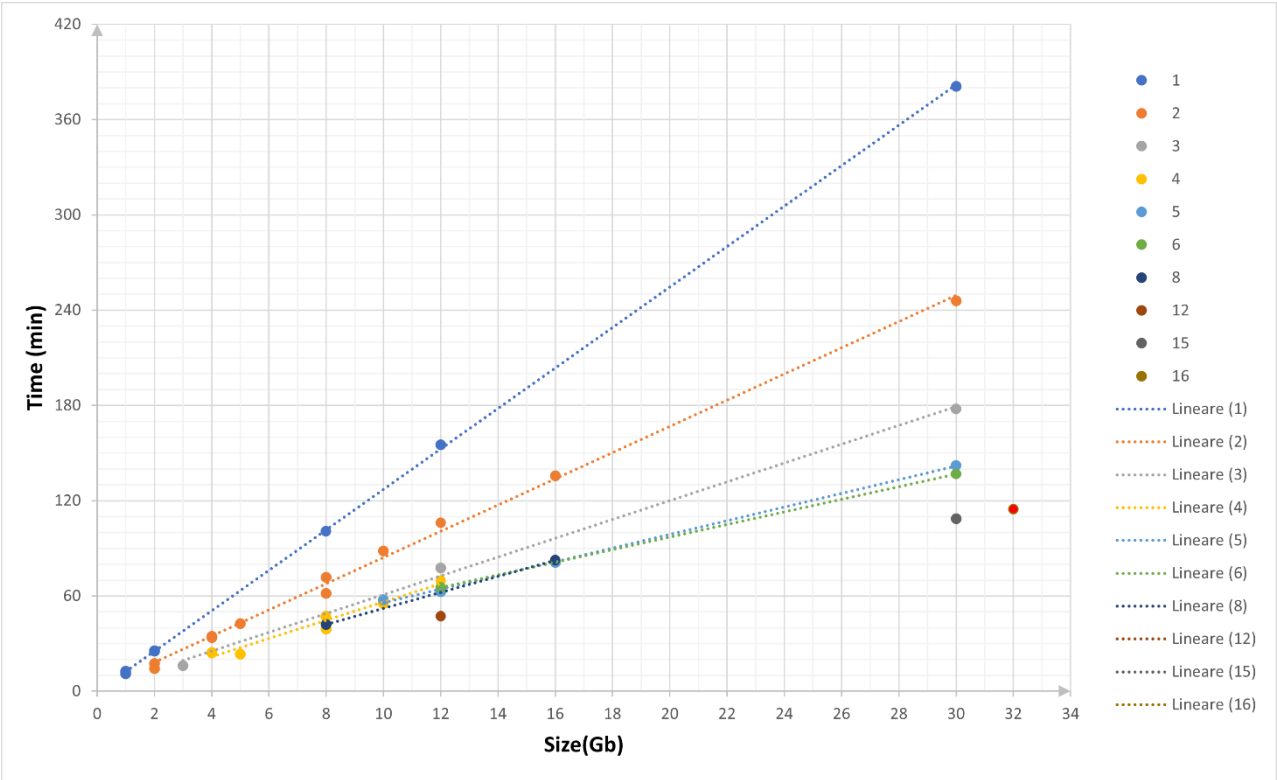
To streamline the process of conducting scalability tests with varying workloads, we have developed a script named *doTest.sh*. This script serves as a versatile tool for launching different types of workloads and automating the execution of our parallel sorting algorithm under various configurations.

All the data retrieved during the execution of the parallel sorting algorithm is systematically organized and stored in a dedicated directory called **dataset**. The directory is used as a repository for various test results and is structured in two subfolders, inside which, in turn, there are folders related to each “test-code” of doTest.sh.

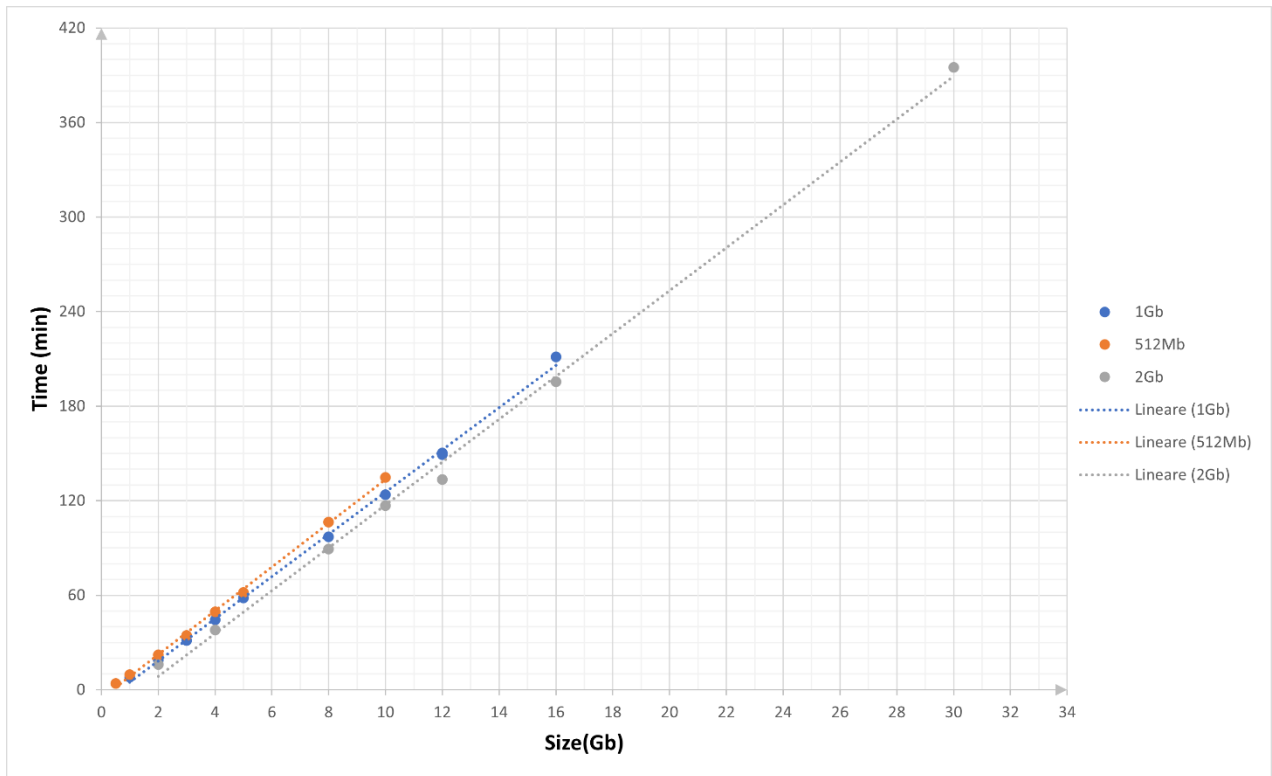
All the test files created contains numerous timing measurements, which have not been reported entirely in the following graphs for simplicity. To have a clear vision of the output I want to remember that the nodes were imagined with a maximum limit of 1Gb (hard-coded) of memory for the sorting space and the goal was to test performance on sorting files larger than this size, taking into account the overhead generated by reading and writing to disk.

## 5.2 Data analysis

This first graph focuses on the parallel execution phases of reading, sorting, and writing slices of the initial dataset. This graph showcases the performance (in time) of the parallel sorting algorithm in distributing the workload across multiple MPI processes, given a fixed dataset size. The number of nodes used are associated to colors as shown in the table on the right.



This second graph focuses on the sequential phase, specifically the merging process executed by Node 0 on all the temporary files generated from other nodes. This phase follows the parallel sorting of slices and involves the consolidation of sorted runs into a unified final file. Given the fact that it is a sequential computation, the graph shows the performance of a node to k-merge on an increasing number of files all of the same size; these performances can be concatenated with the previous ones to obtain an estimate of the total execution time.



## Bibliography

- [1] Ferragina, P. (2023). *Pearls of Algorithm Engineering*.  
<https://doi.org/10.1017/9781009128933>