LABORATORY 2

Parallel Merge Sort using MPI

Juan Sebastián Reyes
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

May 20, 2021

Contents

1 The Code to Parallelize

1.1 Algorithm

The Merge Sort algorithm proposed to be parallelized is as follows:

```
void merge (int i, int j, int mid, List a, List aux) {
    int pointer_left = i;
    int pointer_right = mid + 1;
    int k;

    for (k = i; k <= j; k++) {
        if (pointer_left == mid + 1) {
            aux[k] = a[pointer_right];
            pointer_right++;
        }
        else if (pointer_right == j + 1) {
            aux[k] = a[pointer_left];
            pointer_left++;
        }
        else if (a[pointer_left] < a[pointer_right]) {
            aux[k] = a[pointer_left];
            pointer_left++;
        }
        else {
            aux[k] = a[pointer_right];
            pointer_right++;
        }
    }

    for (k = i; k <= j; k++) {
        a[k] = aux[k];
    }
}


void merge_sort(int i, int j, List a, List aux) {
    if (j <= i) {
        return;
    }
    int mid = (i + j) / 2;

    merge_sort(i, mid, a, aux);
    merge_sort(mid + 1, j, a, aux);
    merge(i, j, mid, a, aux);
}
```

Input:

- ***i:*** is an integer representing the index where the part of the array to sort begins.
- ***j:*** is an integer representing the index where the part of the array to sort ends.
- ***a:*** is an array of integers of at most 10^9 elements, which will be sorted by the algorithm.
- ***aux:*** is an array of integers with the same number of elements as a.

The number of elements in the array a is the worst case supported by the machine on which it was run.

Output:

- ***a:*** is the sorted array.

Description:

The algorithm uses the notion of divide and conquer by following the steps described below:

1. Divide by finding the number `mid` of the intermediate position between `i` and `j`: add `i` and `j`, divide by 2 and round down.
2. Conquer by recursively ordering the subarrays in each of the two subproblems created by the division step. That is, recursively sort the subarray a[i . . . mid] and recursively sort the subarray a[mid+1 . . . j].
3. Combine the two sorted subarrays into the single sorted array a[i . . . j].

The function `merge_sort()`, which receives two integers `i` and `j` and two arrays `a` and `aux`, computes the midpoint (`mid`) between `i` and `j`, then makes a recursive call to order the two portions of the array, the first from `i` to `mid`, and the second part from `mid` to `j`. Finally it calls the function `merge()` which takes the two aforementioned parts and merges them to obtain an array.

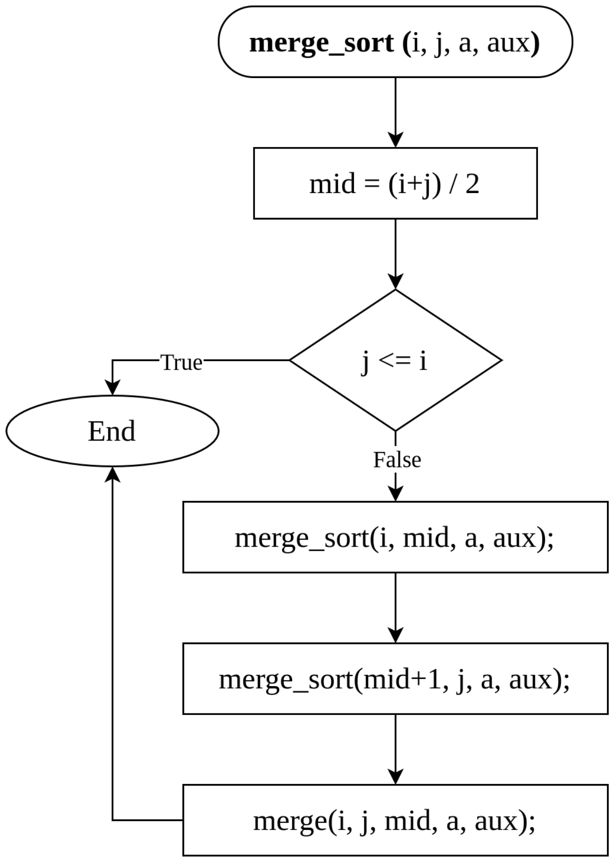
The function `merge()`, which receives three integers `i`, `j`, and `mid`, and two arrays `a` and `aux`, fills the portion of the array `aux` ranging from `i` to `j`, storing each element of the array `a` according to the following options:

- If left pointer has reached the limit, the element at the position of the right pointer is stored and the right pointer is incremented.
- If the right pointer has reached the limit, the element at the position of the left pointer is stored and the left pointer is incremented.
- If the left pointer points to the smallest element, the element at the left pointer position is stored and the left pointer is incremented.
- If the right pointer points to a smaller element, the element at the position of the right pointer is stored and the right pointer is incremented.

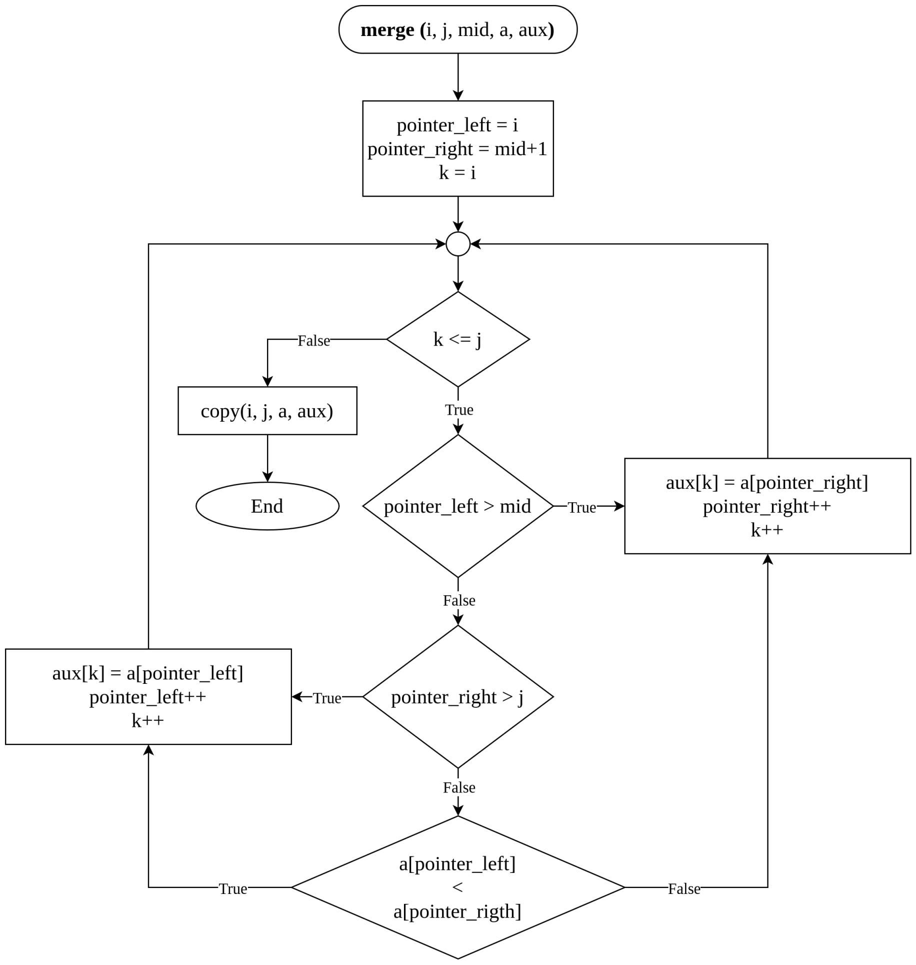
Finally, the elements of the range [i . . . j] are copied from the array `aux` to the array `a`.

Flowchart:

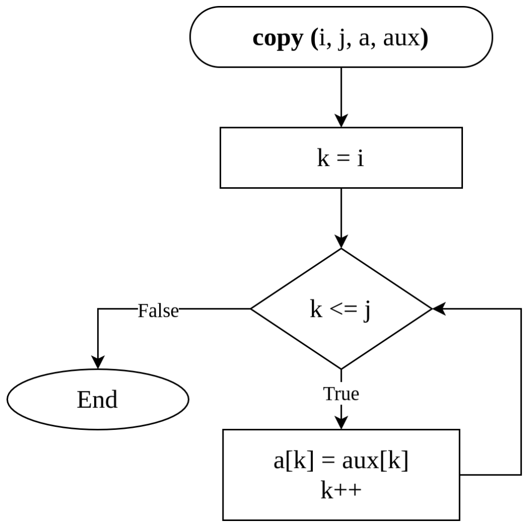
The operation of the algorithm can be better understood with the following flowcharts. The first one is the `merge_sort()` algorithm:


merge_sort (i, j, a, aux)
mid = (i+j) / 2
j <= i
True
End
False
merge_sort(i, mid, a, aux);
merge_sort(mid+1, j, a, aux);
merge(i, j, mid, a, aux);

The following is the `merge()` algorithm:



Which uses the following `copy()` algorithm to copy the elements of the array `aux` to the array `a` in the interval [i . . . j]:


copy (i, j, a, aux**)**
k = i
k <= j
False
True
End
a[k] = aux[k]
k++

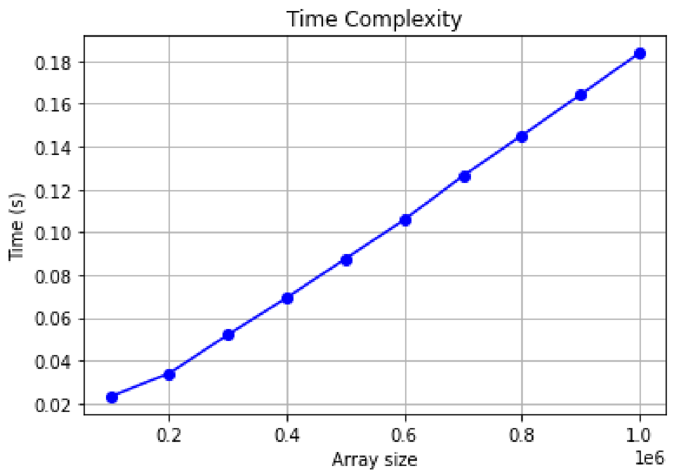
1.2 Complexity

The algorithm has a worst-case complexity of $O(n \log n)$. This is because the `merge` function has a complexity of $O(n)$ by mixing the elements of the sorted arrays. The `merge` function makes a recursive call by splitting the problem into 2 parts and then calling the `merge` function, which makes the complexity of this $O(n \log n)$.

The following table shows the different sizes of the integer array and the corresponding execution time:

Array size	**Time (sec)**
100000	0.023228
200000	0.034091
300000	0.052114
400000	0.069401
500000	0.087621
600000	0.105844
700000	0.126354
800000	0.145179
900000	0.164322
1000000	0.183726

Which can be better seen in the following graphic:



1.3 Time Analysis

The experimental environment where the tests were taken are as follows:

Configuration	**Value**
System Version	Ubuntu 18.04.5
Compiler	GCC 7.5.0
CPU	Intel Core i5-7200U 2.50GHz x4
GPU	Intel HD Graphics 620
Memory Capacity	11,5 GiB
OpenMP Version	OpenMP 4.5

Configuration (CPU)	**Value**
Model Name	Intel(R) Core(TM) i5-7200U @ 2.50GHz
Architecture	x86_64
CPU(s)	4
Thread(s) per Core	2
Core(s) per Socket	2
Socket(s)	1
CPU Max MHz	3100,0000
CPU Min MHz	400,0000
BogoMIPS	5399.81
Virtualization	VT-x
L1d Cache	32K
L1i Cache	32K
L2 Cache	256K
L3 Cache	3072K

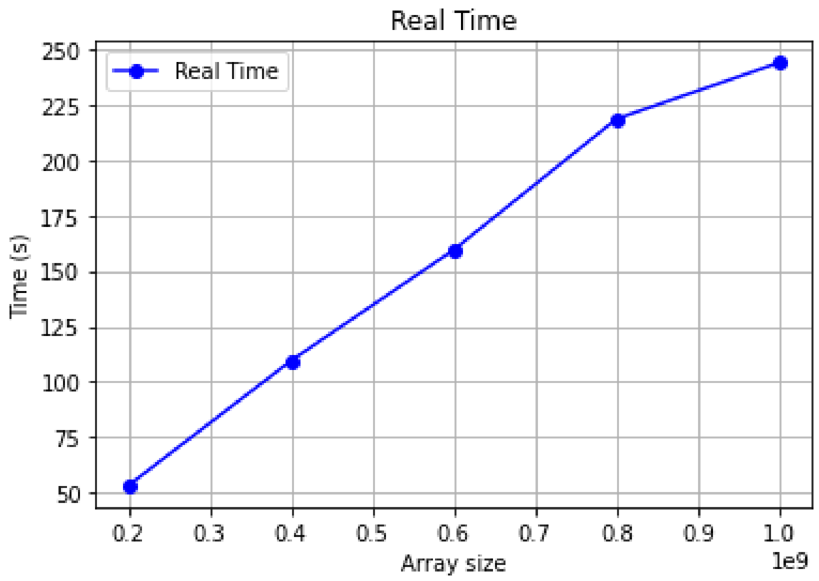
Configuration (RAM)	**Value (RAM 1)**	**Value (RAM 2)**
Total Width	64 bits	64 bits
Data Width	64 bits	64 bits
Size	4GB	8GB
Type	DDR4	DDR4
Type Detail	Sync. Unbuffered	Sync. Unbuffered
Speed	2133 MT/s	2133 MT/s
Manufacturer	Samsung	Kingston
Configured Clock Speed	2133 MT/s	2133 MT/s

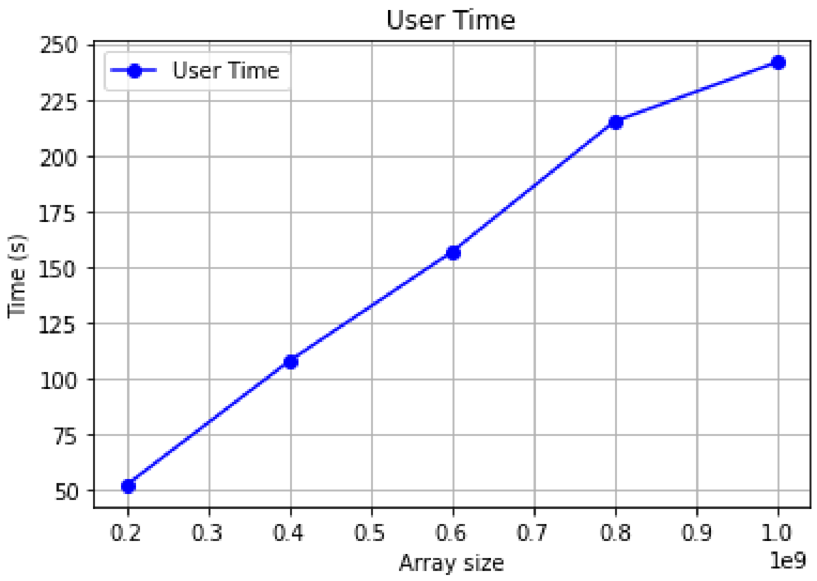
Configuration (Disk)	**Value**
Model Name	ST2000LM007
Capacity	2 TB
Width	32 bits
Speed	66MHz
Buffer size	128 MB
Average Seek Time	13 ms
Data Transfer Rate	600 MBps
Internal Data Rate	100 MBps

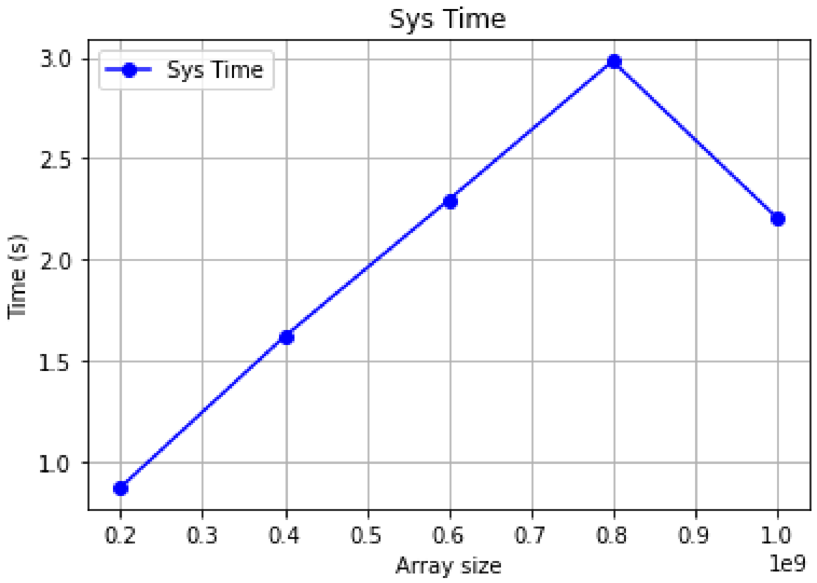
When the worst-case execution time ($n = 10^9$, STACKSIZE = 8G) was measured with the `time` command, the following results were obtained:

n	**Real** $\bar{e}$	**Real** σ_e	**User** $\bar{e}$	**User** σ_e	**Sys** $\bar{e}$	**Sys** σ_e
2×10^8	53.240	2.131	52.270	1.900	0.873	0.098
4×10^8	109.512	3.219	107.761	3.197	1.620	0.115
6×10^8	159.489	9.648	156.661	8.816	2.295	0.775
8×10^8	218.474	12.665	215.141	11.742	2.983	0.713
10×10^8	243.893	0.184	241.725	0.099	2.209	0.134

The average times can best be seen in the following graphics:



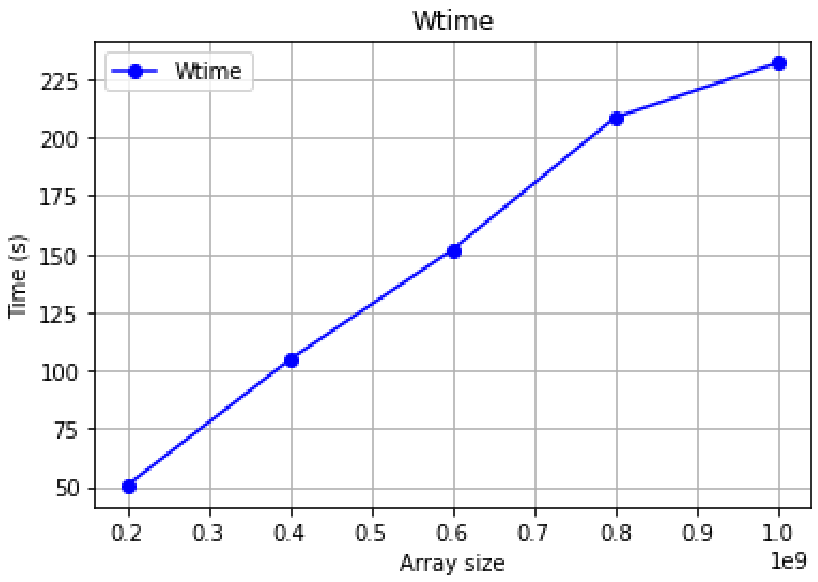

User Time
User Time
Time (s)
Array size
1e9
250
225
200
175
150
125
100
75
50
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0


Sys Time
Sys Time
Time (s)
Array size
1e9
3.0
2.5
2.0
1.5
1.0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

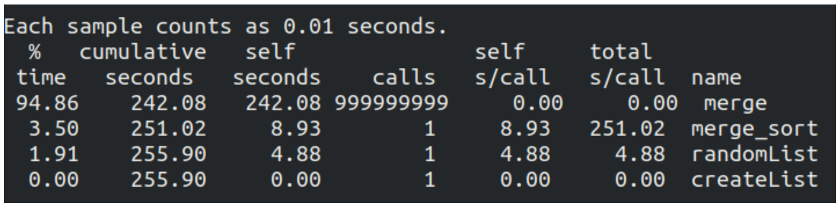
Using the `wtime` tool to measure the execution time of the `Merge_Sort` function at the same input sizes, the following results are obtained:

n	**Wtime** $\bar{e}$	**Wtime** σ_e
2×10^8	50.62590	2.00381
4×10^8	104.65054	3.23631
6×10^8	151.89929	9.44581
8×10^8	208.60174	12.89196
10×10^8	232.09716	0.19621

The average times can best be seen in the following graphics:



Using the `gprof` tool, the following results were obtained:



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 94.86    242.08   242.08 999999999    0.00     0.00  merge
  3.50    251.02     8.93         1     8.93   251.02  merge_sort
  1.91    255.90     4.88         1     4.88     4.88  randomList
  0.00    255.90     0.00         1     0.00     0.00  createList
```

2 How to Parallelize the Algorithm

2.1 Proposal for Parallelization

The main idea about the proposed parallelization of the algorithm with MPI is as follows:

1. The manager process (0) and worker processes (> 0) perform the following steps:

   (a) The array is divide into the number of process that exists i n the called.

   (b) Each slice is given to any process.

   (c) The processor realice the mersort function call, after that mersort function ends, that portion return to a temporal array.

2. It is mixed the portions depend of the number of processors that is called when the algoritms start to run. This process is realized by root process (0).

3. Finally, the result obtainer in the temporal array is copy on the original array.

2.2 Data Independence

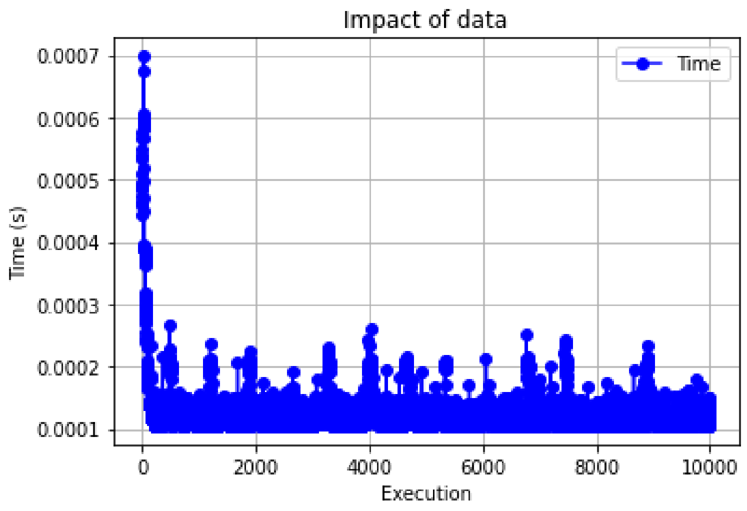
Since the assignment of tasks to worker processes is done by partitioning the array `a` into two parts, each of the parallelizable processes have completely independent data, which are transferred via messages, the only dependency there is the one that a parent process has when waiting for the results of the worker processes, for that the MPI tools will be used to ensure that the algorithm does not continue unless the data have already arrived.

In this sense, the partitioning to be done is on the data, since the assignment of tasks to processes is about assigning the sorting of a portion of the array to the different worker processes.

3 Verification and Analysis

3.1 Impact of data

To verify the impact of the data content on the performance of the algorithm, the creation of the array `a` (with 1.000 elements) was executed 10.000 times, and the execution time of the sorting with each of these arrays was taken. As a result, the mean time was $\bar{t} = 0.00012$ seconds and the standard deviation was $\sigma_t = 3.77 \times 10^5$ seconds. In the following graph you can see the time it took for the algorithm to run in each iteration.



As can be seen from the graph above and based on the standard deviation, the execution time does not vary much depending on the array data.

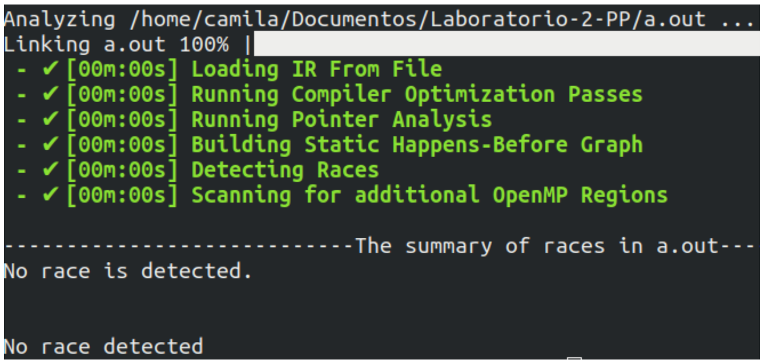
Theoretically, the complexity of the algorithm depends only on the size of the array, since, in the case of the function `merge_sort()` an equal partitioning of the array is performed without taking into account the values of the elements found there. Similarly, the complexity of the `merge()` function is directly proportional to the size of the array, since a cycle of n iterations is performed to place the values of `a` in `aux`, where the value of the *k-th* element only influences the forward pointer (right or left), without changing the number of times to iterate.

Likewise the cycle used to copy the elements of the array `aux` to the array `a` is linear depending on n.

From the above it can be concluded that the performance of the algorithm does not depend on the array data but on the size of the array.

3.2 Data races

After performing the data race analysis with the Coderrect tool, the following results were obtained:



```
Analyzing /home/camila/Documentos/Laboratorio-2-PP/a.out ...
Linking a.out 100% |
 - ✔ [00m:00s] Loading IR From File
 - ✔ [00m:00s] Running Compiler Optimization Passes
 - ✔ [00m:00s] Running Pointer Analysis
 - ✔ [00m:00s] Building Static Happens-Before Graph
 - ✔ [00m:00s] Detecting Races
 - ✔ [00m:00s] Scanning for additional OpenMP Regions

---------------------------The summary of races in a.out---
No race is detected.


No race detected
```

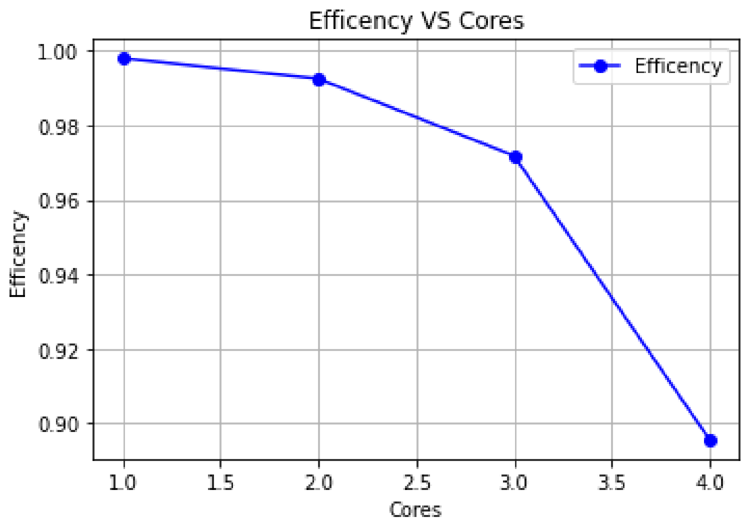
3.3 Strong and weak scaling

3.3.1 Strong scaling

For the strong scaling analysis we took the size of the worst-case problem, and calculated the efficiency for each number of cores with the given n and obtained the following results:

Cores	Efficency (Average)	Efficency (Standard Deviation)
1	0.99804	0.01924
2	0.99255	0.03114
3	0.97186	0.04604
4	0.89558	0.26482

Which can be seen better in the following graph:

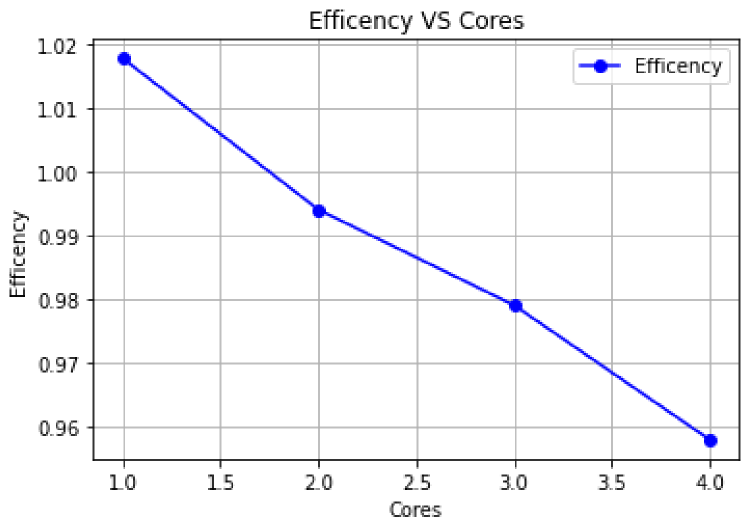


3.3.2 Weak scaling

For the weak scaling analysis we took the problem size for each core $n = 25.000.000$ and calculated the efficiency for each number of cores with the given n and obtained the following results:

Cores	Efficency (Average)	Efficency (Standard Deviation)
1	1.01773	0.01924
2	0.99405	0.03114
3	0.97911	0.04604
4	0.95806	0.26482

Which can be seen better in the following graph:


Efficency VS Cores
Efficency
Cores
Efficency
1.02
1.01
1.00
0.99
0.98
0.97
0.96
1.0
1.5
2.0
2.5
3.0
3.5
4.0

4 How it was paralyzed

The first thing we did was to discuss about the behavior of the mergesort and as we saw that the merge function (which is in itself the one that performs the whole mixing process so that the array is ordered) presented a lot of dependency, we preferred to use the divide and conquer strategy. For them we made use mainly of 2 key functions in MPI which are:

```
MPI_Scatterv(a, counts, displacements, MPI_INT, ap, counts[rank],
    MPI_INT, 0, MPI_COMM_WORLD);

MPI_Gatherv(ap, counts[rank], MPI_INT, a1, counts, displacements,
    MPI_INT, 0, MPI_COMM_WORLD);
```

By means of the above functions, the data is subdivided into disparate parts by means of scatter and each of these parts is sorted by means of the mergesort algorithm. After being sorted, we proceed to regroup its parts in a temporary array that will allow us to make all the necessary changes. When all these parts are already grouped, we proceed to mix them in order to obtain the ordered array, always trying to take 2 partitions and leaving them ordered. This strategy works, since previously the code when they were partitioned ordered the partitions in an ascending way and when mixing them by means of the merge, the resulting arrangement maintains this form. Finally when all the partitions are mixed, we proceed to modify the values in the original array.

4.1 Scatterv

This function is responsible for distributing the data of a process among all of them. In this case Scatterv provides the possibility that each process receives a different number of elements. This allows us to partition the initial array without introducing junk values in them.

4.2 Gatherv

This function is responsible for receiving data from all processes in the root. In this case Gatherv offers the possibility that each process sends a different number of elements and it stores them in an orderly manner in the reception buffer.

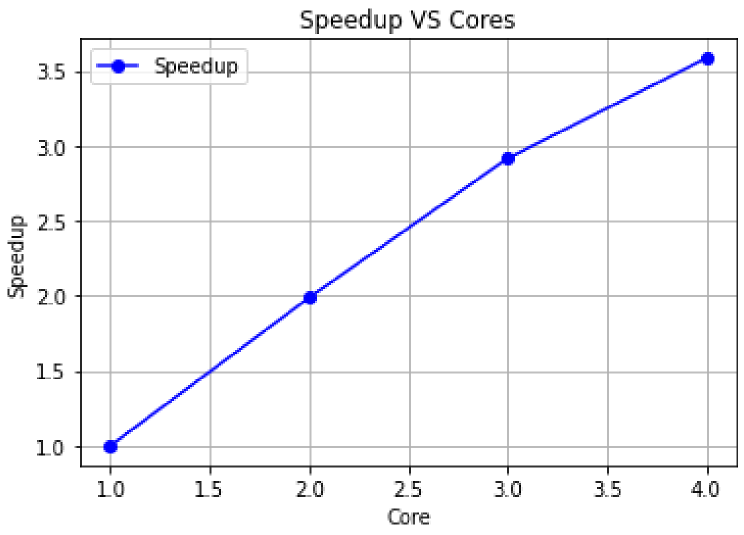
5 Results

5.1 Speedup

To calculate the speedup, it was taken 5 times (with the worst-case) and the average and standard deviation were obtained as shown in the following table:

Cores	**Speedup (Average)**	**Speedup (Standard Deviation)**
1	0.99804	0.00207
2	1.98510	0.00647
3	2.91559	0.01042
4	3.58233	0.07634

Which can be seen better in the following graph:

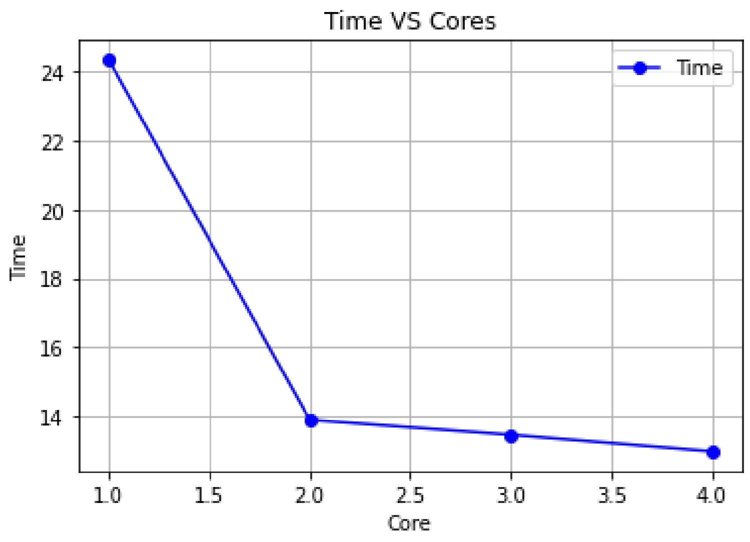


5.2 Execution Time

To calculate the execution time, it was taken 5 times (with the worst-case) and the average and standard deviation were obtained as shown in the following table:

Cores	**Time (Average)**	**Time (Standard Deviation)**
1	24.35950	0.91189
2	13.88640	0.23217
3	13.45082	0.66937
4	12.97074	0.30401

Which can be seen better in the following graphic:



The execution time was also measured depending on the data size and the number of cores used. The following two tables show the average and deviation, respectively, taking into account these factors.

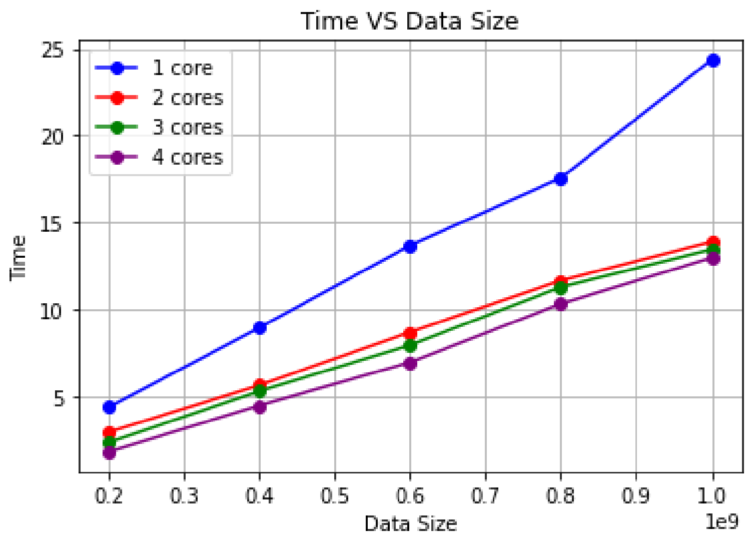
Data Size	**1 core**	**2 cores**	**3 cores**	**4 cores**
2×10^8	4.37178	2.94158	2.34884	1.81132
4×10^8	8.95334	5.64209	5.29859	4.47458
6×10^8	13.68205	8.69558	7.94750	6.95650
8×10^8	17.56162	11.66401	11.28434	10.32447
10×10^8	24.35950	13.88640	13.45082	12.97074

Table 1: Time (Average)

Data Size	**1 core**	**2 cores**	**3 cores**	**4 cores**
2×10^8	0.00756	0.03913	0.04580	0.44945
4×10^8	0.05077	0.10240	0.06414	0.26378
6×10^8	0.46483	0.23507	0.51314	0.29536
8×10^8	0.08187	0.21761	0.49652	0.23399
10×10^8	0.91189	0.23217	0.66937	0.30401

Table 2: Time (Standard Deviation)

Which can be seen better in the following graphic:

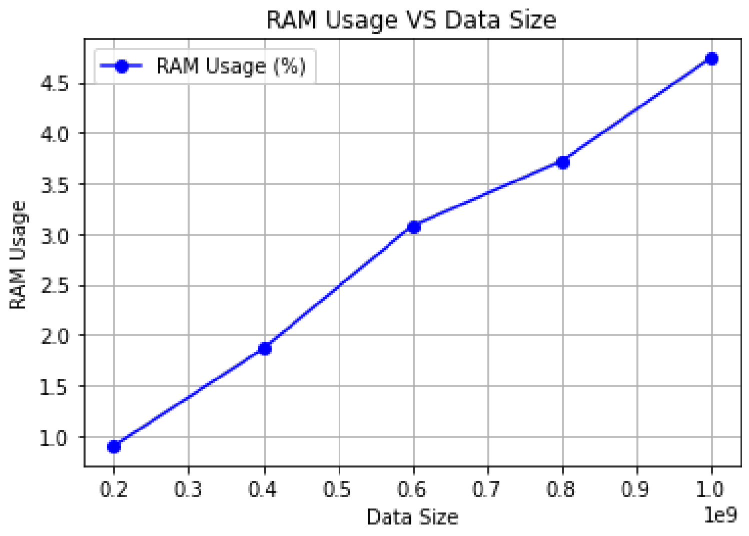


5.3 RAM usage

The percentage of RAM used by the program during execution was measured for each of the input sizes for one core. The results can be seen in the following table:

Data Size	**RAM Usage (Average)**	**RAM Usage (Average)**
1×10^6	0.90000	0.15811
2×10^6	1.86000	0.23022
3×10^6	3.08000	0.13038
4×10^6	3.72000	0.14832
5×10^6	4.74000	0.18166

Which can be seen better in the following graphic:



6 Problems with parallelization

In general we had no problems parallelizing the algorithm. It was a challenge for us as it required a bit more analysis and effort, however we managed to improve the performance of the algorithm and apply the concepts seen in class, and implementing some that we researched on our own.

7 Merge Sort Applications

As the Merge Sort is a sorting algorithm, then all the uses of any other can also be exploited with the Merge Sort. Sorting algorithms are used in different things, some of them are:

- Ordering phone numbers for a phone directory.
- Order scheduled flights at an airport.
- Search algorithms use ordered data structures.
- Greedy algorithms use ordered data structures.
- . . .

In addition to these general applications, Merge Sort is also useful:

- Can be used for the Inversion Counting Problem of a list.
- Can be used in External Sorting.

8 References

- Course material, available on BlackBoard
- Algorithm's author: `Hackrio`. 2018. Taken from this GitHub gist: `merge-sort.c`
- MPI Library
- Stdio Library
- Stdlib Library
- Sorting Algorithms Applications
- Merge Sort Applications
- MPI Guide