LABORATORY 2

Design

Juan Sebastián Reyes
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

May 14, 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.

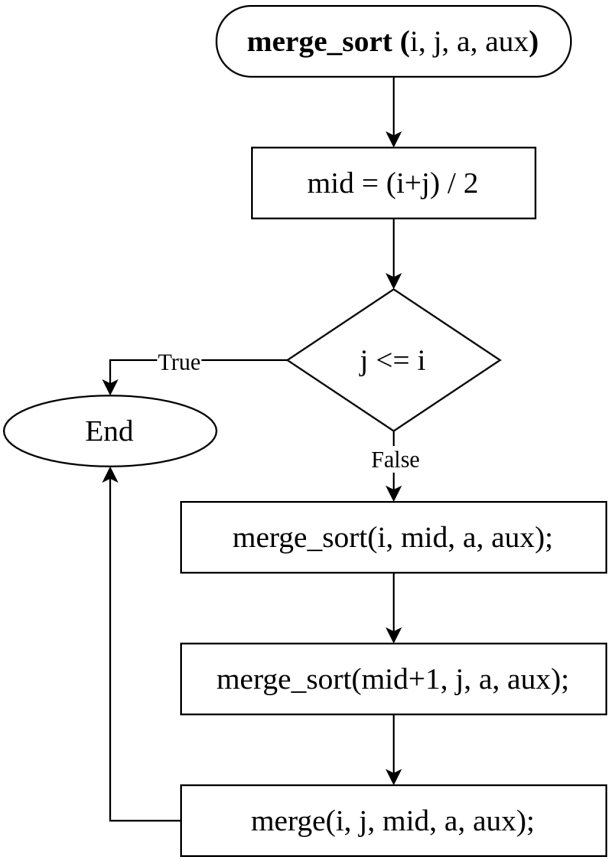
The function `merge()`, which receives three integers `i`, `j`, and `mid`, and two arrays `a` and `aux`, fills the portion of the array `aux` ranging from `i` to `j`, storing each element of the array `a` according to the following options:

- 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.

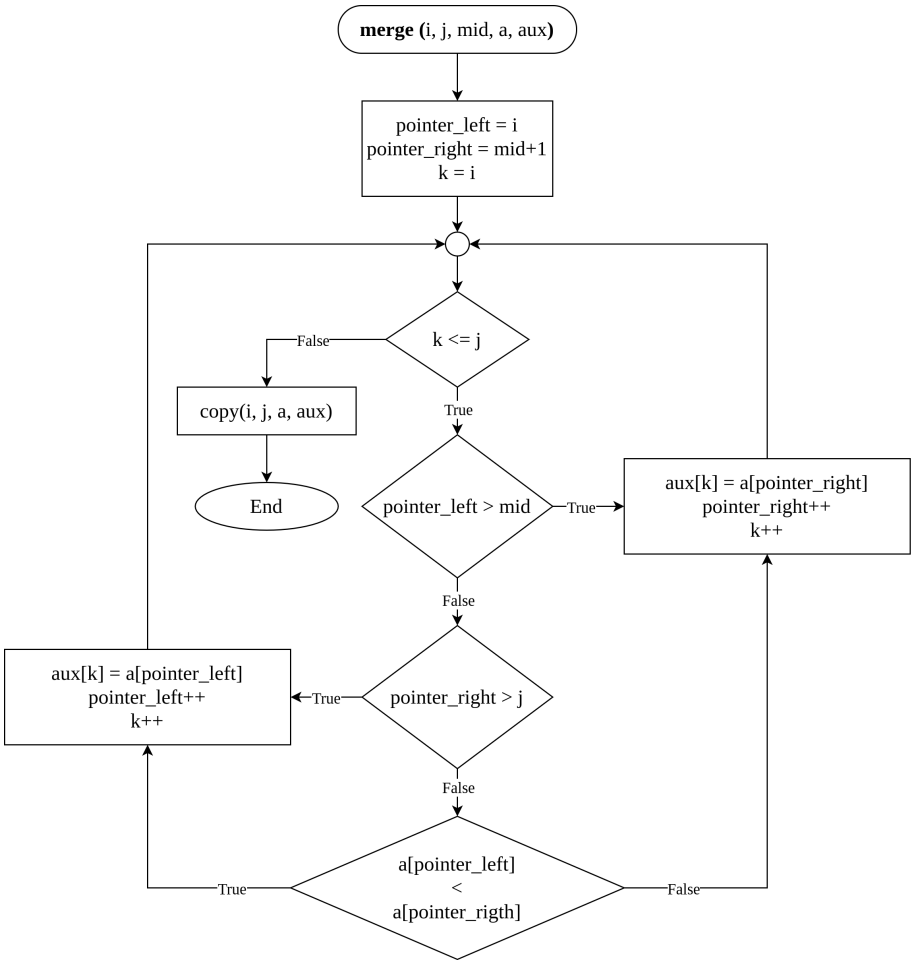
Finally, the elements of the range [i ... j] are copied from the array `aux` to the array `a`.

Flowchart:

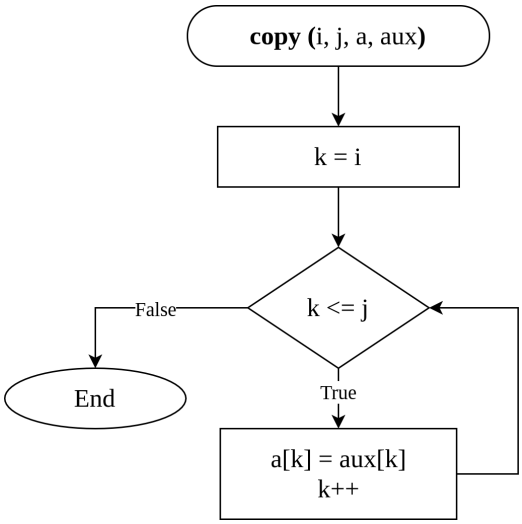
The operation of the algorithm can be better understood with the following flowcharts. The first one is the `merge_sort()` algorithm:



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
End
True
a[k] = aux[k]
k++

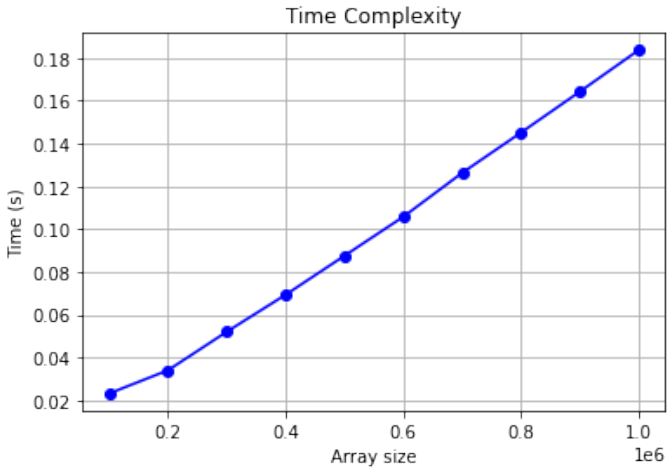
1.2 Complexity

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

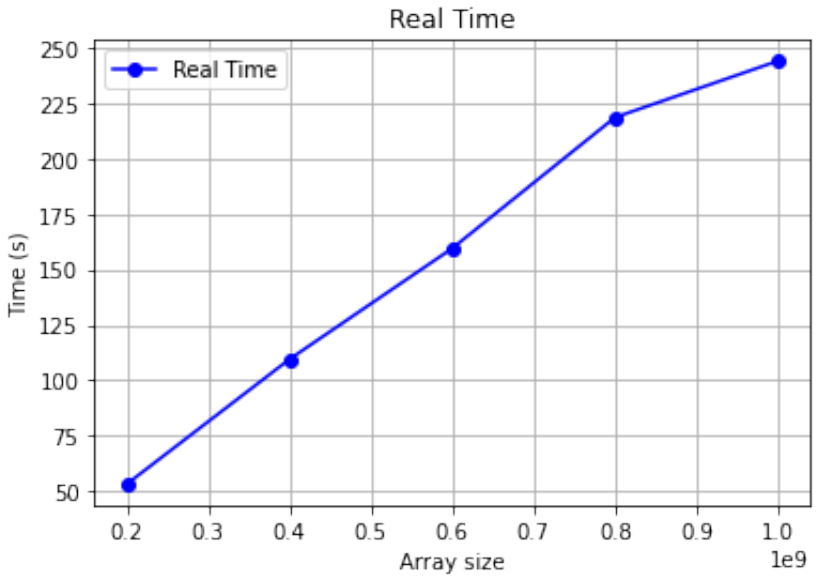
Configuration (RAM)	Value (RAM 1)	Value (RAM 2)
Total Width	64 bits	64 bits
Data Width	64 bits	64 bits
Size	4GB	8GB
Type	DDR4	DDR4
Type Detail	Sync. Unbuffered	Sync. Unbuffered
Speed	2133 MT/s	2133 MT/s
Manufacturer	Samsung	Kingston
Configured Clock Speed	2133 MT/s	2133 MT/s

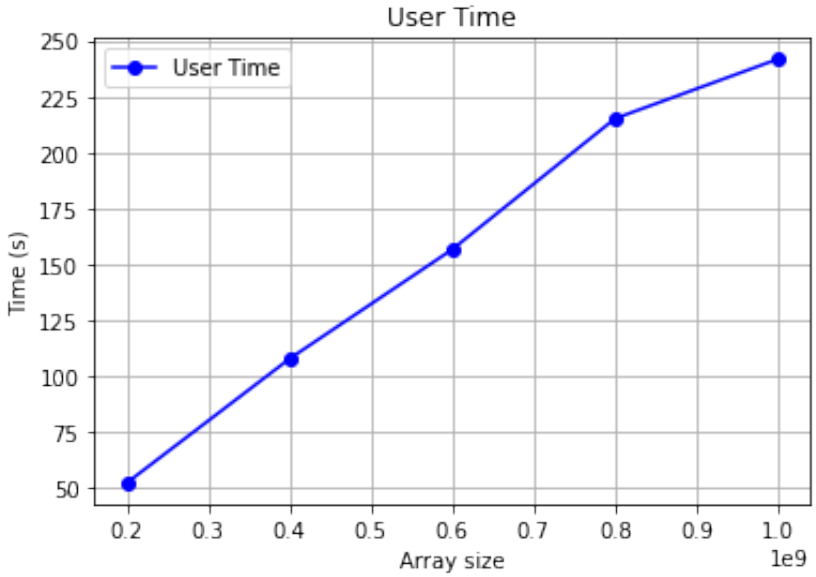
Configuration (Disk)	Value
Model Name	ST2000LM007
Capacity	2 TB
Width	32 bits
Speed	66MHz
Buffer size	128 MB
Average Seek Time	13 ms
Data Transfer Rate	600 MBps
Internal Data Rate	100 MBps

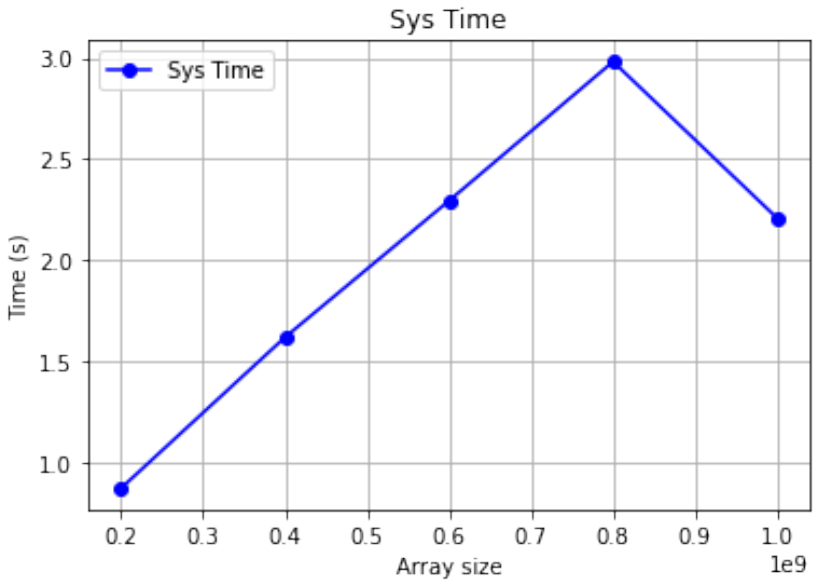
When the worst-case execution time ($n = 10^9$, STACKSIZE = 8G) was measured with the `time` command, the following results were obtained:

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:



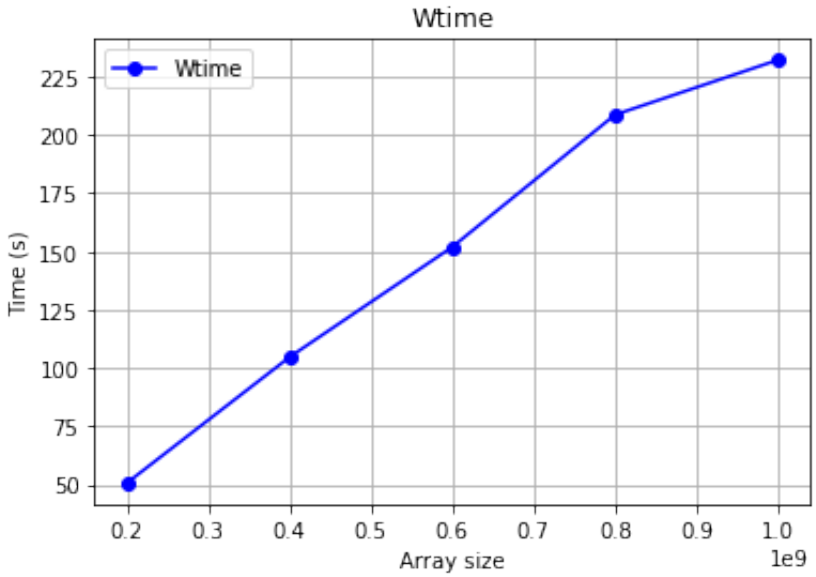

User Time
User Time
250
225
200
175
150
125
100
75
50
Time (s)
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1e9
Array size


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

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) Splits the data into two parts: a left[i . . . mid] array and a right[mid+1. . . j] array.
   (b) Sends the left half of the array to the left worker process.
   (c) Sends the right half of the array to the right worker process.
   (d) Receives the sorted data from the left worker process.
   (e) Receives the sorted data from the right worker process.
   (f) Mixes the sorted left and right halves.
2. If it is an administrator process, the algorithm ends.
3. If it is an worker process, the sorted data is sent to the administrador node.
4. Finally, the manager node (0) will have the array completely sorted.

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 References

- Course material, available on BlackBoard
- MPI Library
- Stdio Library
- Stdlib Library