LABORATORY 1

Parallel Bucket Sort with OpenMP

Nicolás Delgado
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

April 8, 2021

Contents

1 Time Measure

1.1 Hardware Features

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

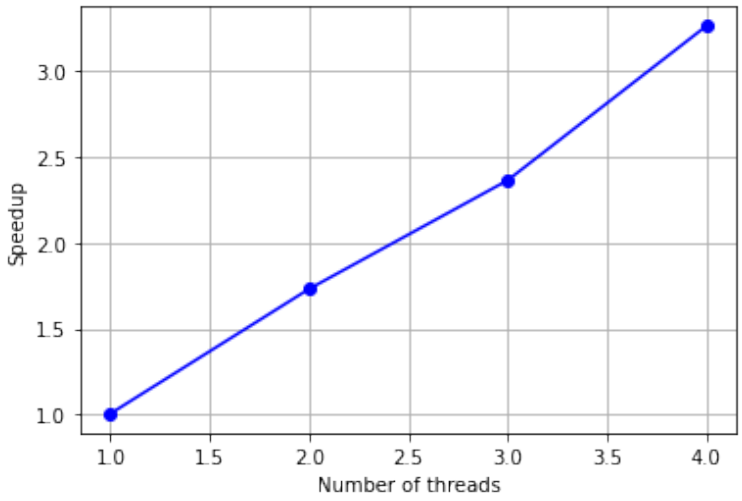
Configuration (RAM)	Value (RAM 1)	Value (RAM 2)
Total Width	64 bits	64 bits
Data Width	64 bits	64 bits
Size	4GB	8GB
Type	DDR4	DDR4
Type Detail	Sync. Unbuffered	Sync. Unbuffered
Speed	2133 MT/s	2133 MT/s
Manufacturer	Samsung	Kingston
Configured Clock Speed	2133 MT/s	2133 MT/s

1.2 Speedup

To calculate the speedup, it was taken 5 times and the average and standard deviation were obtained as shown in the following table:

Threads	Speedup (Average)	Speedup (Standard Deviation)
1	1.00553	0.00759
2	1.73208	0.09149
3	2.36541	0.19348
4	3.26446	0.18907

Which can be seen better in the following graph:

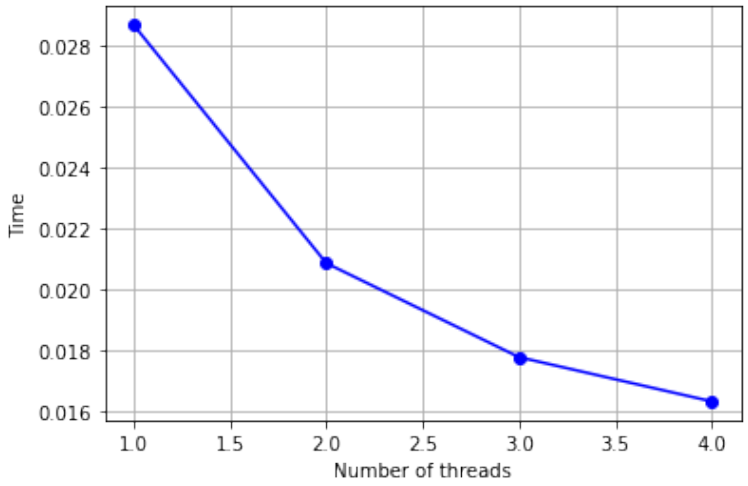


1.3 Execution Time

To calculate the execution time, it was taken 5 times and the average and standard deviation were obtained as shown in the following table:

Threads	Time (Average)	Time (Standard Deviation)
1	0.02870	0.00584
2	0.02088	0.00314
3	0.01779	0.00613
4	0.01634	0.00322

Which can be seen better in the following graph:



2 How It Works

As proposed in the design document, `#pragma omp parallel for` was used to parallelize the cycles of the algorithm. In Part 1 the two non-nested cycles were parallelized, in Part 2 the only cycle in it was parallelized, and in Part 3 only the outer cycle was parallelized, since the inner cycle had the behavior of a `while`.

The parallel algorithm is as follows:

```
void Bucket_Sort(int array [], int n, int max, int min) {

    int i, j = 0;
    int cpos[max+1], cneg[-(min-1)];

    // Begin Part 1
    #pragma omp parallel for
    for (i = 0; i <= max; i++)
        cpos[i] = 0;

    #pragma omp parallel for
    for (i = 0; i <= -(min-1); i++)
        cneg[i] = 0;
    // End Part 1

    // Begin Part 2
    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        if (array[i] >= 0)
            (cpos[array[i]]) ++;
        else
            (cneg[-array[i]]) ++;
    }
    // End Part 2

    // Begin Part 3
    #pragma omp parallel for
    for (i = -min; i > 0; i--)
        for (; cneg[i] > 0; (cneg[i])--){
            #pragma omp critical
            array[j++] = -i;
        }

    #pragma omp parallel for
    for (i = 0; i <= max; i++)
        for (; cpos[i] > 0; (cpos[i])--){
            #pragma omp critical
            array[j++] = i;
        }
    // End Part 3

}
```

Part 3 of the algorithm presents elements that were not mentioned in the design document (`#pragma omp critical`). The decision as to why they were placed will be explained in the next section.

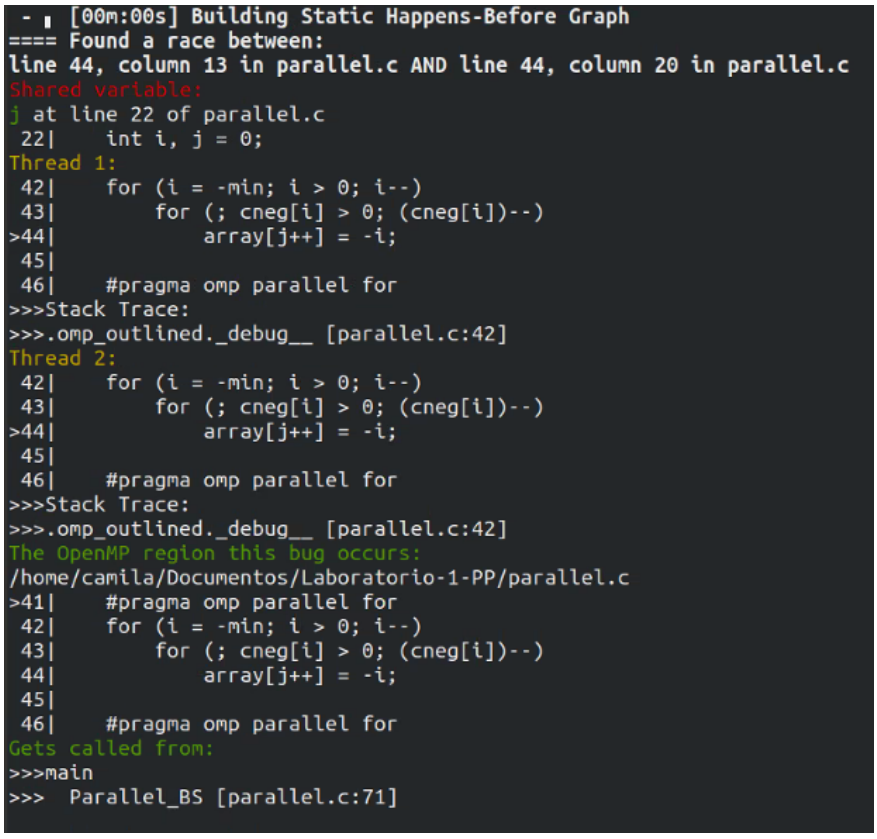
The algorithm separates the data set (*array*) into different buckets, sorts each bucket separately and then, as each bucket is sorted, joins them together. Thus, the whole data set is completely sorted. In essence, this is what the code shown above does.

Being a little more rigorous with the explanation, the code above separates the data set into two arrays: `cneg` and `cpos`. Which are of size `min` and `max`, respectively. Subsequently, zeros are placed in each of the positions of these arrays. Each of these positions will represent how many elements go into each bucket. Since a `parallel for` was used, then the number of positions that each thread will have to fill with zeros will be N/k, where N can be $\{max$, $min\}$ and k is the number of threads to be used, i.e., the number of zeros to be placed by each thread should be equal. Something similar happens with the parallelization of Part 2, only there the data is distributed in `cneg` and `cpos`.

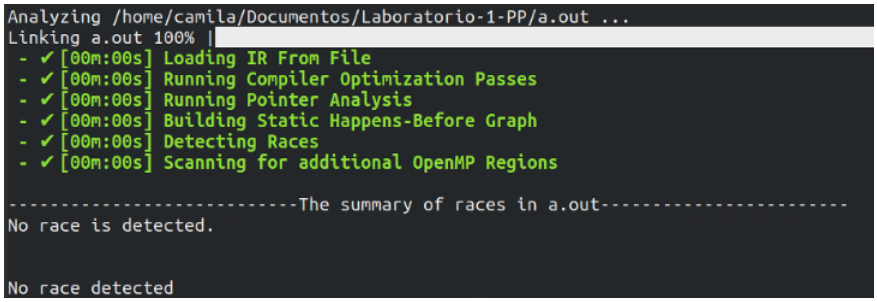
Then, we iterate over each of them and sort them. This process is repeated in both `cneg` and `cpos`. After this, take `cneg` and iterate over it, from the smallest element to zero, placing the elements from the smallest to zero in order in the original array. Then this process is repeated but with `cpos`, only now it will be from the largest to the smallest. Thus leaving the original array completely sorted in ascending order. Care must be taken here, since only the outer for is being parallelized (this is Part 3), i.e., each thread is being told to do N/k iterations of the outer for, where N can be $\{max$, $min\}$ and k is the number of threads to be used. Then each thread must execute the inner for separately, without being parallelized.

3 Problems With Parallelization

From the design document we realized that the internal for statement Part 3 pretended to be a while statement, so it could give us problems if we parallelized it. We were able to avoid this scenario by having a good parallelization design. However, what happens inside the internal for causes a race condition to be generated, since parallelizing the external `for` causes each thread to execute the external `for` separately, which generates a possible trace in which two (or more) threads try to increment that shared variable. This can be seen in the following image, where Coderrect was used to verify this type of problem.



This problem was solved satisfactorily thanks to the use of the instruction `#pragma omp critical`, since this instruction guarantees that the portion of code it encapsulates will be executed by one thread at a time, i.e., if within that portion of code there were variables that were previously shared and generated a race condition, then now they will not be, due to what the instruction guarantees.



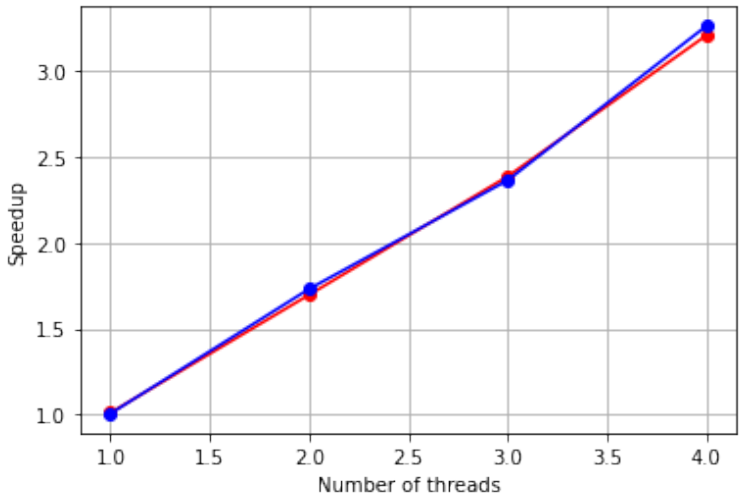
In addition, due to the good structure of the code we could quickly see that there was almost no data dependency, so there were no further complications when parallelizing the code.

4 Speedup Analysis With Critical Statement

In the Problems with Parallelization Section, it was shown that with the initial proposal, race conditions were present in Part 3 of the algorithm, which led us to use the `#pragma omp critical` statement, which allows data to be accessed by only one thread at a time. We found that doing this (practically) eliminates the parallelization that was proposed for Part 3, since each thread must wait for the data to become free before it can be used.

This hypothesis led us to compare the speedups having the parallelization in Part 3 of the algorithm (with the for parallel and the critical) and without parallelizing Part 3, these were the results, comparing the number of threads vs. speedup (in red the average speedup of the algorithm with serial Part 3, in blue the parallel with critical):

Threads	Speedup (Serial)	Speedup (With Critical)
1	1.00553	1.01039
2	1.73208	1.69779
3	2.36541	2.38793
4	3.26446	3.20957



The graph shows that in any of the ways that we propose, there is not much difference in the speedup of the algorithms, they are practically the same. So, with this graph we can support our hypothesis, which was that it was just as well to parallelize Part 3 (the external for and placing a critical in the body of the internal for) or not to do so.

We thought to compare also the `atomic` statement, but unfortunately what is done inside the internal for (an assignment) is not supported by `atomic`. We tried to modify the algorithm a little, incrementing the variable *j* inside an atomic and making the assignment outside the `atomic` statement and it was possible, but the correctness of the algorithm was violated, i.e., it was no longer sorted.

5 Bucket Sort Applications

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

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

6 References

- Course material, available on BlackBoard
- Algorithm's author: `Hemanth Kumar`. 2019. Taken from this Git repo: `Bucketsort-in-C`
- OpenMP Library
- Stdio Library
- Stdlib Library
- Sorting Algorithms Applications