LABORATORY 1

Design

Nicolás Delgado
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

March 29, 2021

Contents

1 The Code to Parallelize

1.1 Algorithm

The Bucket Sort algorithm proposed to be parallelized 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)];

    for (i = 0; i <= max; i++)
        cpos[i] = 0;
    for (i = 0; i <= -(min-1); i++)
        cneg[i] = 0;

    for (i = 0; i < n; i++) {
        if (array[i] >= 0)
            (cpos[array[i]]) ++;
        else
            (cneg[-array[i]]) ++;
    }

    for (i = -min; i > 0; i--)
        for (; cneg[i] > 0; (cneg[i])--)
            array[j++] = -i;

    for (i = 0; i <= max; i++)
        for (; cpos[i] > 0; (cpos[i])--)
            array[j++] = i;
}
```

Input:

- ***array:*** is the array of integers to be sorted.
- ***n:*** is the number of elements in the array.
- ***max:*** is the maximum element (supremum) in the array.
- ***min:*** is the minimum element (infimum) in the array.

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.

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.

1.2 Complexity

This algorithm has a worst-case complexity of $O(n^2)$. Which can be concluded once the algorithm shown above is analyzed.

The part of the code where the supremum or infimum is found has a complexity of $O(n)$, but that is not part of the essence of the Bucket Sort, so that part of the code is not included.

The following is a theoretical analysis of the complexity of the algorithm shown above. The algorithm will be divided into three sections, which will be further analyzed at run-time.

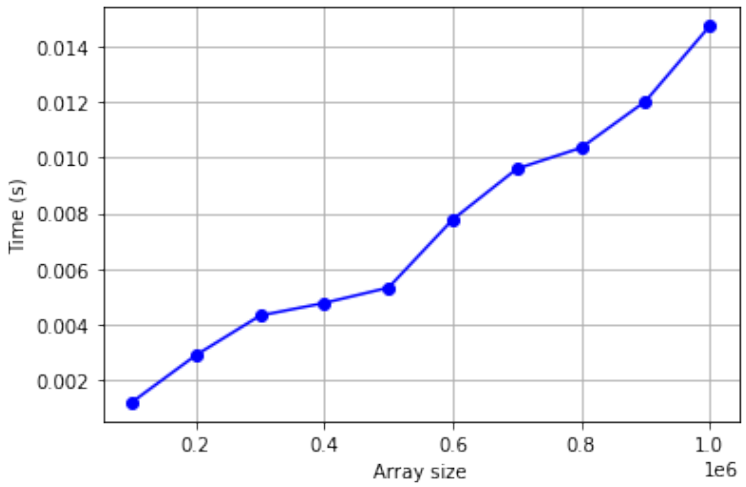
- **Part 1:** between lines 6 and 9. This portion of code is in charge of filling the `cpos` and `cneg` arrays with zeros. Its time complexity is in $O(\max\{max, -min\})$.
- **Part 2:** between lines 11 and 16. This portion of code is in charge of segmenting the original array implicity. It indicates how many elements of the original array will go in each bucket inside `cpos` or `cneg`. Its time complexity is in $O(n)$.

- **Part 3:** between lines 18 and 25. This portion of code is in charge of ordering the original array using the `cpos` and `cneg` arrays, more specifically, the buckets within each array. Its time complexity is in O(n^2) (worst-case).

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

Array size	**Time (sec)**
100000	0.001189
200000	0.002888
300000	0.004315
400000	0.004769
500000	0.005323
600000	0.007776
700000	0.009603
800000	0.010358
900000	0.012032
1000000	0.014742

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 items	Item value
System version	Ubuntu 18.04.5
Compiler	GCC 7.5.0
Server model	Unknown
CPU	Intel Core i5-7200U 2.50GHz x4
GPU	Intel HD Graphics 620
Memory capacity	11,5 GiB
Memory version	Unknown
OpenMP version	OpenMP 4.5

When the worst-case execution time ($n = 10^6$, STACKSIZE = 8G) was measured with the `time` command, the following results were obtained:

Type	Average (sec)	Standard deviation (sec)
user	0.031	0.012
sys	0.01	0.004

If better performance is desired, without problems with the stack size, then obviously the size of the RAM memory must be increased. The good thing is that the arrays in C/C++ have a size of $2^{1024} - 1$, which in a laboratory is difficult to achieve.

Using the `wtime` tool to measure the execution time of the `Bucket_Sort` function on the worst-case, we obtained that the average was 0.016s, with a standard deviation of 0.002s.

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

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.41      0.01     0.01        1    10.02    10.02  Bucket_Sort
 33.41      0.02     0.01        1    10.02    10.02  Build_Array
 33.41      0.03     0.01                             main
```

The execution time of the three parts of the algorithm are:

Part	**Average (sec)**	**Standard deviation (sec)**
1	0.007	0.003
2	0.006	0.002
3	0.005	0.002

2 How to Parallelize the Algorithm

2.1 Proposal for Parallelization

The following are the proposals for parallelization within the chosen code:

- We propose to parallelize `Part 1` mentioned in the **Complexity** section, which has two (*non-nested*) cycles. This proposal is made because this section is the most delayed according to its (*temporal*) metrics.
- It was also considered to parallelize `Part 2`, mentioned in the **Complexity** section. This is because it is the second worst in performance of the (*temporal*) metrics shown in the previous table.
- In addition, we have thought about the possibility of parallelizing Part 3. This with the use of `#pragma omp parallel for` only for the outer cycle, since the behavior of the inner cycle is that of a `while`.

2.2 Data Independence

- In the proposal that was made for `Part 1`, no data dependency is present in either of the two for cycles (not nested), so they can be easily parallelized using `#pragma omp parallel for`, assigning to each thread a portion of the array (N/k). Where N is the size of the array and k is the number of threads to be used.
- In the proposal made in `Part 2`, care must be taken. By not having dependency in the iterators of the cycle, one could think of using `#pragma omp parallel for`, assigning to each thread a portion of the array (N/k). Where N is the size of the array and k is the number of threads to be used. Be careful because in the `if/else` inside this part it can happen that two (or more) threads try to increment the same data at the same time. It must be known that there will be shared memory between the threads.

- For the proposal made in `Part 3`, something similar happens with the previous point. Care must be taken with the inner for, since by only parallelizing the outer for, each thread will do the inner cycle independently. However, the variable j is incremented, so there may be an error in the **correctness of the algorithm** if it is not set that this variable can only be modified by one thread at a time. It is probable (there are no impossible situations) that in the development of the laboratory we will realize that this proposal could not be executed due to synchronization problems. If its possible, it will be parallelized using `#pragma omp parallel for`, assigning to each thread a portion of the array (N/k). Where N is the size of the array and k is the number of threads to be used.

3 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