LABORATORY 3

Design

Juan José Betancourt
Nicolás Delgado
Camila Paladines

Parallel Programming

Professor: Roger Alfonso Gómez Nieto

May 31, 2021

Contents

https://www.overleaf.com/download/project/60b13b01f641dc155b70671b/build/179bb31fe16-72c60daea9937710/output/output.pdf?compileGroup=standardclsiserverid=clsi-pre-emp-n1-b-g0crpopupDownload=true

1 The Code to Parallelize

1.1 Algorithm

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

```python
def Bucket_Sort(array, n, max, min):
    j = 0
    cpos = []
    cneg = []

    # Begin Part 1
    for i in range(max+1):
        cpos.append(0)

    for i in range(-(min-1)+1):
        cneg.append(0)
    # End Part 1

    # Begin Part 2
    for i in range(n):
        if (array[i] >= 0):
            cpos[array[i]] += 1
        else:
            cneg[-array[i]] += 1
    # End Part 2

    # Begin Part 3
    for i in range(-min, 0, -1):
        while cneg[i]:
            array[j] = -i
            j += 1
            cneg[i] -= 1

    for i in range(max+1):
        while cpos[i]:
            array[j] = i
            j += 1
            cpos[i] -= 1
    # End Part 3

    return array
```

Input:

- ***array:*** is the array of integers that will be sorted. The maximum dimension of the array is 10^8, since for larger values it exceeds the RAM usage.
- ***n:*** is an integer representing the number of elements in the array.
- ***max:*** is an integer representing the maximum element of the array.
- ***min:*** is an integer representing the minimum element of the array.

Output:

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

Description:

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.

Flowchart:

The operation of the algorithm can be better understood with the following flowchart:

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 (indicated in the code shown above), which will be further analyzed at run-time.

- **Part 1:** 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:** 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:** 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).

1.3 Time Analysis

The code was executed in Google Colaboratory. The experimental environment where the tests were taken are as follows:

Configuration	**Value**
System Version	Ubuntu 18.04.5 LTS
CPU	Intel(R) Xeon(R) CPU @ 2.20GHz x2
GPU	NVIDIA Tesla T4
Memory Capacity	13 GB
PyCuda Version	2021.1

Configuration (CPU)	**Value**
Model Name	Intel(R) Xeon(R) CPU @ 2.20GHz
Architecture	x86_64
CPU(s)	2
Thread(s) per Core	2
CPU MHz	2199,998
BogoMIPS	4399,99
Cache size	56320K
clflush size	64
cache_alignment	64
address sizes	46 bits physical
address sizes	48 bits virtual

Configuration (RAM)	**Value**
Total Width	64 bits
Data Width	64 bits
Size	13GB
KernelStack	4644 kB
PageTables	8788 kB
Hugepagesize	2048 kB
VmallocTotal	34359738367 kB

Configuration (GPU)	**Value**
Model Name	Tesla T4
Manufacturer	NVIDIA
Capacity	16 GB
Type	GDDR6
Turing Tensor Cores	320
NVIDIA CUDA Cores	2560
Single Precision Perf.	8.1 TFlops
Mixed Precision	65 FP16 TFlops
INT8 Precision	130 TOPS
INT4 Precision	260 TOPS
Bandwidth	320+ GB/s
Power	70W
Interconnect GEN3	x16 PCIe

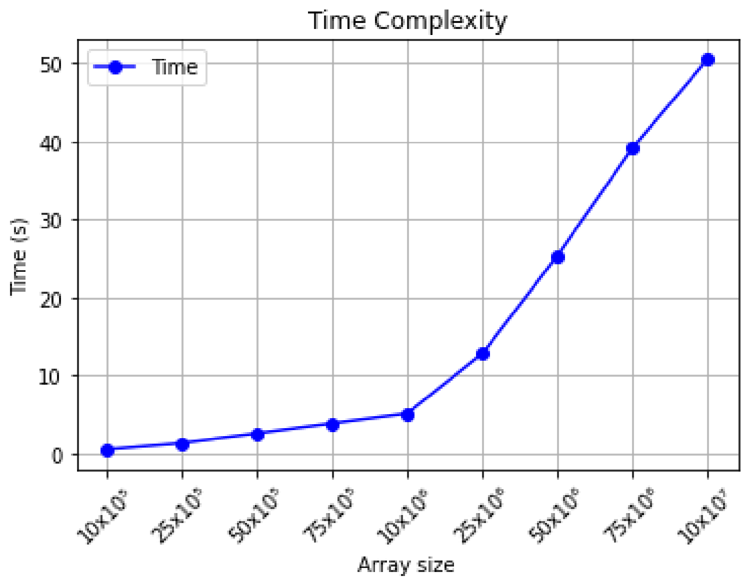
The following results were obtained when the analysis was performed with the cProfile tool.

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
100000002    7.309    0.000    7.309    0.000 {method 'append' of 'list' objects}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.compile}
        2    0.000    0.000   62.771   31.386 {built-in method builtins.exec}
        2    0.000    0.000    0.000    0.000 /usr/local/lib/python3.7/dist-packages/IPython/co
        2    0.000    0.000    0.000    0.000 /usr/local/lib/python3.7/dist-packages/IPython/co
        2    0.000    0.000    0.000    0.000 /usr/local/lib/python3.7/dist-packages/IPython/co
        2    0.000    0.000   62.771   31.386 /usr/local/lib/python3.7/dist-packages/IPython/co
        2    0.000    0.000    0.000    0.000 /usr/lib/python3.7/codeop.py:140(__call__)
        2    0.000    0.000    0.000    0.000 /usr/local/lib/python3.7/dist-packages/IPython/ut
        1    1.706    1.706    1.706    1.706 {built-in method builtins.max}
        1    1.780    1.780    1.780    1.780 {built-in method builtins.min}
        1   51.693   51.693   59.002   59.002 <ipython-input-18-6b07310851ab>:10(Bucket_Sort)
        1    0.283    0.283   62.771   62.771 <ipython-input-18-6b07310851ab>:47(<module>)
        1    0.000    0.000    0.000    0.000 <ipython-input-18-6b07310851ab>:48(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

The following result was obtained when performing the analysis with the Python tool `time()`.

Array size	**Time (Average)**	**Time (Standard Deviation)**
1000000	0.5080671787	0.0075660801
2500000	1.3199320793	0.0297093656
5000000	2.5738748074	0.0494647611
7500000	3.8517268181	0.0431936657
10000000	5.1187510014	0.0683367129
25000000	12.7931775570	0.0521723820
50000000	25.3757541656	0.2787641850
75000000	39.1753437042	2.8974439063
100000000	50.5354355812	0.3507678206

Which can be better seen in the following graphic:



2 How to Parallelize the Algorithm

2.1 Proposal for Parallelization

2.2 Data Independence CORREGIR

- 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 `Hemanth Kumar`. 2019. Taken from this Git repo: `Bucketsort-in-C`
- CUDA Library
- Stdio Library
- Stdlib Library
- cProfile in Google Colaboratory