Parallel Computing Bucket-Sort Algorithm

Ana Carneiro
Informatics Department
University of Minho
Braga, Portugal
pg46983@uminho.pt

Ana Peixoto
Informatics Department
University of Minho
Braga, Portugal
pg46988@uminho.pt

Luís Pinto
Informatics Department
University of Minho
Braga, Portugal
pg47428@uminho.pt

***Abstract*—Given an unsorted array of N positive integers, the bucket sorting algorithm works by distributing the elements into a number of ordered buckets each containing zero or more elements. Each bucket is then sorted using the quick-sort algorithm. At the end, elements are gathered from each bucket in order, so the input array is sorted. The bucket sorting algorithm can execute and finish sorting very quickly in sequential processing. However, this execution time can even be improved by parallelizing it. In this paper we will analyse the sequential version and convert it into a parallel version using OpenMP API. Furthermore, the performance in terms of execution time will be measured considering different metrics.**

***Index Terms*—OpenMP, BucketSort, algorithm, performance, parallel, sequential, threads, dependencies, complexity, critical zone, optimization, metrics, SpeedUp**

I. INTRODUCTION

The main goal of this project was to evaluate the benefits of parallel programming in shared memory, using C and OpenMP, by developing a sequential and a parallel version of the Bucket-Sort algorithm, which sorts an array by distributing the elements into a number of buckets.

We started off by implementing the sequential version, which served as a starting point for the parallel version. During this process, we dealt with data dependencies, critical zones, race conditions, amongst other limitations inherent to parallel programming.

Furthermore, we performed tests using several metrics, such as data input size, number of buckets, threads and cores. Lastly, the obtained results were analysed and compared with the expected results.

II. SEQUENTIAL BUCKET SORT

A. Implementation

The sequential version of the Bucket-Sort algorithm consists of four main steps. For the first step, we start by initializing a bucket vector with a predefined size (number of buckets).

Secondly, we allocate each element of the input data to a bucket, considering the range of values for each bucket. This step was implemented by iterating the array of buckets and verifying which elements of the input array should be inserted into the current bucket. Although this implementation requires more computational power, due to iterating the input array N times, with N being the number of buckets, it is more effective in terms of exploiting parallelization, because we are able to distribute the iterations load between threads while avoiding data dependencies and race conditions.

In the third step, we needed to sort each bucket using a sorting algorithm, like quick sort, insertion sort or recursively using the same bucket. In our case, we decided to use quick sort because it is regarded as the best sorting algorithm and has a significant advantage in terms of efficiency, because it is able to deal well with a huge list of items.

Finally, we concatenate all sorted buckets in an orderly manner, resulting in the input array being sorted. Bellow we defined the pseudo-code for the sequential version of bucket sort.

```
for i in N_BUCKETS
    buckets[i] = makeSpace()
for i in N_BUCKETS
    for j in N_ARRAY
        bucket = allocateBucket(input[j])
        if bucket==i
            insert(buckets[i],input[j])
    qsort(vector[i])
for j in N_BUCKETS
    output = concate(output + i, buckets[j])
    i += elements of buckets[j]
```

Now, let us analyse the time complexity which describes the amount of computer time it takes to run the bucket sort algorithm in the best, average and worst cases. We will also indicate the space complexity of the bucket sort, that is the amount of memory space required to solve an instance of the algorithm.

Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. In Bucket sort, best case occurs when the elements are uniformly distributed in the buckets. The complexity will be better if the elements are already sorted in the buckets. If we use the quick sort to sort the bucket elements, the overall complexity will be linear, i.e., θ(n + $k \log_2 k$), where θ(n) is for making the buckets, and $\theta(k \log_2 k)$ is for sorting the bucket elements using algorithms with linear time complexity at best case. The best-case time complexity of bucket sort is θ(n + $k \log_2 k$).

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. Bucket sort runs in the linear time, even when the elements are uniformly distributed. The average case time complexity of bucket sort is $\theta(n + k \log_2 k)$.

Worst Case Complexity - In bucket sort, the worst case occurs when the elements are of the close range in the array, because of that, they have to be placed in the same bucket. So, some buckets have more elements than others. The complexity will get worse when the elements are in reverse order. The worst-case time complexity of bucket sort is $\theta(n^2)$.

Case	Time Complexity
Best Case	$\theta(n + k \log_2 k)$
Average Case	$\theta(n + k \log_2 k)$
Worst Case	$\theta(n^2)$
Case	**Space Complexity**
BucketSort	$\theta(n * k \log_2 k)$
Stable	Yes

TABLE I: Time and space complexity

B. *Analysis of Possible Optimizations*

There are some optimizations that can be applied to this algorithm, in order to decrease execution time, such as loop unrolling and vectorization. This techniques are applied by the gcc compiler, when using specific flags to achieve the desired purposed.

For this version we analysed the impact of possible optimizations, namely using gcc compiler flags like -O0, -O1, -O2 and -O3. Combining these flags with the "-funroll-loops" flag we can apply loop unroll methods to the code. Each -O flag serves a certain purpose: -O0 is faster but does not use any optimizations; -O1 enables the core optimizations in the compiler; -O2 performs more loop unroll than the previous level and produces vectorial instructions; -O3 enables optimizations that require significant compile-time analysis and resources, and the amount of loop unrolling that is performed is increased.

In the table bellow is the execution time associated with each optimization, considering an input size of 50000 elements. This tests were performed using the "day" partition of the cluster, and the "compute-134-115" node. As expected, the execution time decreases as the level of optimization increases, obtaining the best time for the flag -O3.

Since the developed code contains 4 main loops, we decided to test the performance of the sequential version with the "-funroll-loops" flag along with the -O2 optimization to apply loop unroll techniques, which reflects on lower time to execute the program.

Otimization	Execution Time (μs)	Input Size
-O0	7493	50000
-O1	4848	50000
-O2	4420	50000
-O3	4254	50000
-02 -funroll-loops	4409	50000

TABLE II: Optimizations in sequential version

III. PARALLEL BUCKET SORT

A. *Implementation with OpenMP*

To create the parallel version of the Bucket-Sort algorithm, we added OpenMP directives to the previously mentioned sequential version. We analysed the sequential implementation to verify which blocks of code could be parallelized, in order to obtain better performance in terms of execution time and thus distributing efficiently the computational load for the various parallel threads.

The main focus, to effectively parallelize the bucket sort algorithm, was to use the OpenMP directives in the loop cycles, because it is where the largest computational load occurs.This algorithm has 4 main steps each one being described as a loop, however the second step is the one with the largest amount of computational load so it is beneficial to parallelize this section.

As referred in the section I, the implemented algorithm starts off by allocating memory for each bucket by using a for loop. This is the first target for parallelization, since no data dependencies or critical zones are found. For this loop we created N threads that corresponds to the number of buckets so that each thread would perform each iteration.

```
#pragma omp parallel num_threads(threads)
#pragma omp for
for i in N_BUCKETS //STEP 1
    buckets[i] = makeSpace()
```

The second step of the algorithm was also parallelized since it is where most of the computational power is needed. This section of code was carefully inspected to decide if both the outer and nested loops could be parallelized. The parallelization of the nested loop would cause race conditions and data dependencies, when inserting the element in the right bucket. This insertion creates a critical zone in the code, which can only be executed sequentially. For this reason, the nested loop can not be executed in parallel.

We concluded that only the outer loop could support parallel execution, and therefore the quick sort algorithm will also be executed in parallel, to sort each individual bucket (step 3). For the outer loop parallization, we considered the number of threads equal to the number of buckets to ensure a good load distribution.

```
#pragma omp parallel num_threads(threads)
#pragma omp for
for i in N_BUCKETS
    for j in N_ARRAY  //STEP 2
        bucket = allocateBucket(input[j])
        if bucket==i
            insert(buckets[i],input[j])
    qsort(buckets[i]) //STEP 3
```

Finally, after analysing the last step of the algorithm (step 4), we concluded that it could not be executed in parallel, because there are data dependencies between iterations. The shared variable "i" would disturb the correct execution of the

algorithm because it introduces a RAW dependency between threads. This loop builds the output array in correct order using the variable "i" to find the right place in the array to put the current bucket. Since there is a RAW dependency in this variable, the loop would not be able to perform this operation correctly in parallel. In fact, parallelizing this section using multiples threads would result in poor execution time and incorrectly sorting of the output array.

```
for j in N_BUCKETS //STEP 4
    output = concate(output+i, buckets[j])
    i += elements of buckets[j]
```

To parallelize the steps mentioned above, as seen in the snippets of code, we used the OpenMP "pragma omp" directive along with the "parallel num_threads" parallel construct, to create the desired amount of threads. We also used the "for" work-sharing construct to distribute the computational load of the section to the threads.

When comparing this version to the sequential, we can suppose before-hand that it will perform much better in terms of performance, and we expect to have a considerable gain in execution time. This will allow faster results even when dealing with a huge amount of data in the input array.

IV. PERFORMANCE TESTS

In this section we performed several tests in the parallel version, in order to analyse the scalability of the implementation. To accomplish this goal we used the performance API, PAPI, that measures the execution time of the bucket sort algorithm, number of clock cycles, number of instructions, misses in the cache L1 and in the cache L2. These tests were performed using different sizes of the input array, number of buckets, number of threads, cores and memory's hierarchy. For each test we created a table showing the values of the execution time in μs for both the parallel and sequential version and the value of "speedUp" which measures the relative performance of the two versions by using the expression $speedup_{A->ref} = \frac{ref}{A}$. Note that, by default, the gcc flag used in the tests was the -O2.

A. Size of input array

To analyse how the algorithm behaves for different input sizes, we ran both the parallel and sequential version for several elements of the array in the "day" partition of the cluster and in the "compute-134-115" node. We also used the number of threads equal to the number of buckets that is 10 buckets. Bellow it is shown the table with the execution time of the sequential and parallel versions for the several inputs and the respective SpeedUp.

	Execution Time		
Size of input array	**Sequential (μs)**	**Parallel (μs)**	**SpeedUp (μs)**
1 000	135	87	1.552
10 000	1 221	235	5.196
100 000	9 004	1 602	5.620
1 000 000	95 195	16 984	5.605
10 000 000	812 507	199 524	4.072

TABLE III: Performance varying the input array size

B. Number of Buckets

To analyse how the algorithm behaves for different number of buckets, we can run the parallel and sequential versions for 1 000 000 elements of the input array in the "day" partition of the cluster and in the "compute-134-115" node. For the parallel version we considered 10 threads. Bellow it is shown the table with the execution time of the parallel version for different number of buckets.

	Execution Time		
Number of Buckets	**Sequential (μs)**	**Parallel (μs)**	**SpeedUp (μs)**
5	71892	24361	2.95
10	95331	20444	4.66
20	136319	22193	6.14
50	273674	36561	7.485
100	389157	68522	5.678

TABLE IV: Performance varying the number of buckets

C. Number of threads

To analyse how the algorithm behaves for different number of threads, we ran the parallel and sequential version for 1 000 000 element of the input array in the "day" partition of the cluster and in the "compute-134-11" node with the number of buckets being 10. Bellow it is shown the table with the execution time of the sequential and parallel version for the number of threads.

Number of Threads	**Execution Time (μs)**	**SpeedUp (μs)**
Sequential	95 225	1
2	62 998	1.513
4	38 943	2.448
8	28 099	3.39
16	18 990	5.200
24	21 098	4.518
32	24 352	3.914
48	26 363	3.616

TABLE V: Performance varying the number of threads

D. Number of Buckets/Threads

To analyse how the algorithm behaves for the same amount of buckets and threads, we ran the parallel and sequential versions for 1 000 000 elements of the input array in the "day" partition of the cluster and in the "compute-134-115" node. Bellow it is shown the table with the execution time of the sequential/parallel version for number of bucket/threads and the speed up.

	Execution Time		
Number of Buckets /Threads	**Sequential (μs)**	**Parallel (μs)**	**SpeedUp (μs)**
10	95284	20443	4.66
20	136255	21308	6.39
30	171107	23317	7.34
40	229191	21995	10.42
50	276844	22314	12.41
60	323737	27278	11.868
70	361982	27146	13.33
80	414261	28017	14.786

TABLE VI: Performance varying the number of buckets and threads

E. *Different Cluster Nodes*

To analyse how the algorithm behaves in different cluster nodes, we ran the parallel version for 1 000 000 element with 10 threads and buckets. Bellow it is shown the table with the execution time of the parallel version in several cluster nodes.

The main differences in the chosen nodes were the np value which indicates the numbers of CPUs available in each machine. For the nodes "compute-113-2", "compute-134-1" and "compute-134-115" the number of CPUs were 16, 32, 48, respectively and in each node we are able to run up to 8, 16, 24 threads for each CPU.

Cluster Nodes	Execution Time (μs)
compute-113-2	24062
compute-134-1	19301
compute-134-115	15191

TABLE VII: Performance varying cluster nodes

F. *Memory's Hierarchy*

To analyse how the algorithm behaves when the input array fits in different cache levels, we ran the parallel version for 1 000 000 element of the input array with 10 threads and buckets in the "day" partition of the cluster and in the "compute-134-115" node. Bellow it is shown the table with the execution time of the parallel version and the cache misses for the levels one and two.

In the node "compute-134-115", the size of the cache for L1 is 64KB, 32 KB for data and 32 KB for instructions. To create an array that fits into this level of cache we need to build an array with 15 000 integers. For the size of the cache for L2 level the node allows 256 KB of storage letting an array of 52 000 integer to fit into to this level. Finally, the cache of level L3 stores 8192 KB of data, so an array with 2 050 000 integers manages to fit into this storage. All the arrays with the number of elements above 2 125 800 integers does not fit into any level in cache so it is store in to the RAM.

	Execution Time	Cache Misses	
File Size	**Parallel (μs)**	**L1_DCM**	**L2_DCM**
15 KB (L1)	380	4 258	1 862
52 KB (L2)	851	14 508	5 243
2.05 MB (L3)	35 762	689 323	298 150
2.125 MB (RAM)	36 143	711 922	299 262

TABLE VIII: Performance varying file size

V. RESULTS ANALYSIS

After performing multiple tests considering the most relevant metrics, we can analyse the obtained results, by comparing them to the expected results and studying the scalability of the implemented solution.

A. *Size of input array*

For the analysis of this section, due to the enormous gap between input file sizes, we used two illustrations merely to facilitate the viewers understanding. In the first figure, we can see a comparison between the execution time of the sequential and parallel versions, amongst files of a thousand, ten thousand and hundred thousand random integers.

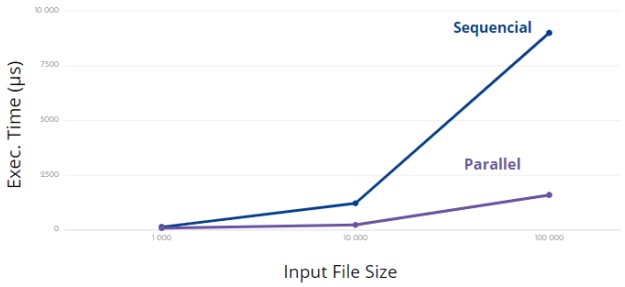

Fig. 1: Input File from 1K to 100K

Now we will see the full scope of the obtained results, that is, having into account input files from one thousand to ten million random integers.

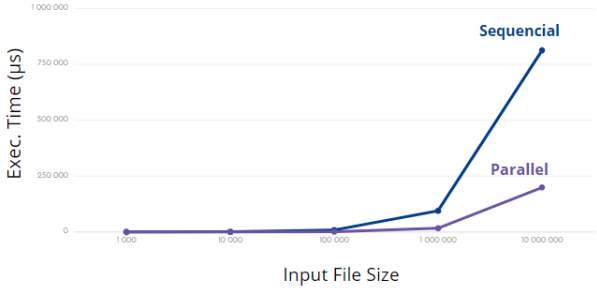

Fig. 2: Input File from 1K to 10M

The results are completely plausible with what was expected, as the input file increased, the impact of the parallel version was more notorious, showing its strength in processing large values of data in significantly less time when compared to the sequential version.

To truly grasp the scope of the gain with the parallel version we can use the SpeedUp values, presented in the previous chapter. For incrementally bigger input files, the SpeedUp for the parallel version reaches it's peak value at around 100 000 elements in the input file, being x5.6 times better than the sequential counterpart, as is shown in the figure below. The decrease factor in large input files, such as 10 000 000 elements, has to do with accessing memory and cache misses, which will be explained in greater detail in section F.

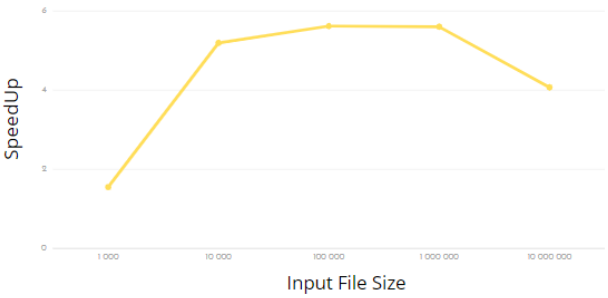

Fig. 3: SpeedUp

B. *Number of Buckets*

In the following graph, we can see how the execution time varies in the sequential and parallel versions for different numbers of buckets, and how the program scales in each version.

For the sequential version, we can see that execution times are a lot bigger than the parallel version. The relation between execution time and number of buckets for the sequential version is linear: when increasing the number of buckets, the execution time increases accordingly. On the other hand, in the parallel version, the relationship between execution time and number of buckets is a lot smoother and starts to grow when the number of buckets is bigger than 50.

We can conclude that the parallel version of the algorithm is much more scalable since it does not increase execution time significantly when increasing the number of buckets. The sequential version is not scalable because the execution time increases outstandingly as the number of buckets increases. For these reasons, we can conclude that the parallel version behaves a lot better for bigger quantities of buckets.

Nonetheless, the number of buckets must be chosen carefully because even though it does not depend on the input size, it depends on the range of values allowed for the input data, and can have a huge impact on performance. It is not recommended that the number of buckets is bigger or equal to the range of values allowed in the input array, since it will degrade performance.

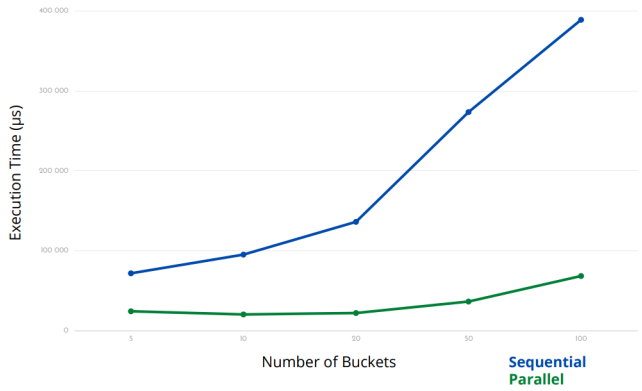

Fig. 4: Number Of Buckets

C. *Number of Threads*

The figure 5 shows a graph that represents the speedup between the sequential version (number of threads equal to 1) and the parallel version for different threads.

As we can see from the graph, when the number of threads increases the speedup also increases. This happens because when more threads are running, more parallelism is introduced to the algorithm making the program faster and more scalable. However, when the number of threads reaches 16 the speedup decreases and consequentially making the algorithm slower. The reason for this decline has to do with the number of cores being used when running the algorithm. The node "compute-134-11" of the cluster has 24 cores for each CPU, allowing for up to 24 threads to run in each CPU. When the number of threads is less than 24 then each thread is allocated to a core allowing to distribute the resources uniformly. We can see that when the number of threads is 16 there is a balance between the resources distribution and the paralellism, making the algorithm faster than with other number of threads. Nevertheless, when the number of threads surpasses 16 then all the resources available will be allocated to each thread making the algorithm slower and the speedup smaller due to the increase processing in each core and consequentially lack of resources.

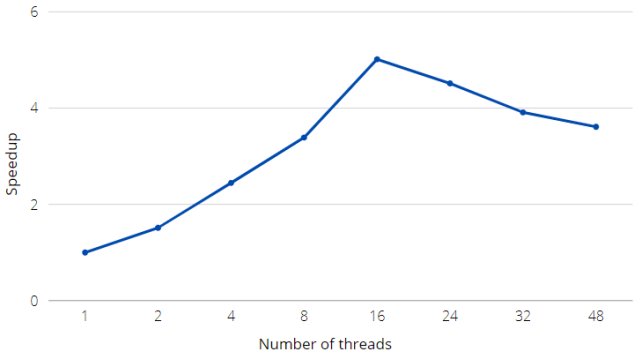

Fig. 5: Number Of Threads

D. *Number of Buckets/Threads*

In the following graph, we can see how the execution time varies in the sequential and parallel versions for different numbers of buckets/threads, and how the program scales in each version. We performed the tests considering the same number of buckets and threads.

In figure 6 we can see how the execution time varies as the number of buckets/threads increases for the sequential and parallel versions. For the sequential version, as the number of buckets/threads increases, the execution time also increases linearly, and therefore this version is not scalable. As in the parallel version there is almost a constant relationship between the number of buckets/threads and the execution time, which will allow more scalability. Nonetheless, considering an equal number of buckets and threads is not the best decision since execution times do not decrease as the metric increases.

This lack of scalability in the sequential version is due to the fact that the code is executed by a single thread and thus increasing the number of buckets will degrade performance. In the parallel version, the fact that the number of buckets equals the number of threads allows each thread to compute the load of each bucket and therefore maintain the performance of the implementation.

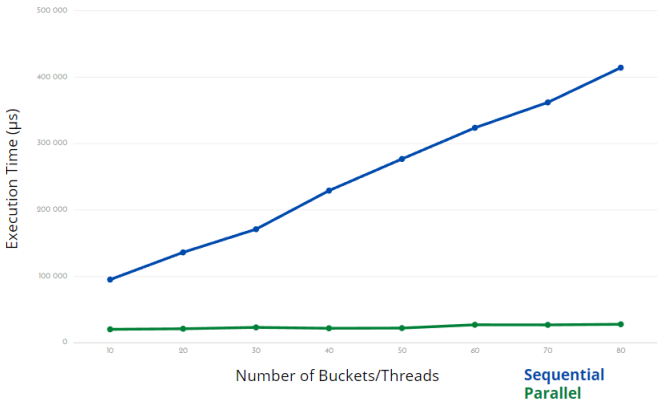

Fig. 6: Execution time for each nr. of threads/buckets

E. *Different Cluster Nodes*

As we can see in the figure below, the execution time, for an input file of a million random integers between the range of 1 and 1000, tends to lower as the nodes progressively gain more computational power (CPUs, cores and threads). The obtained results go accordingly to the expected trend, that is, a machine with more threads per CPU will have a lower execution time.

Fig. 7: Execution time for each node

F. *Memory's Hierarchy*

The 8th figure shows a graph that represents the execution time for the parallel version with several different inputs, each created to fit into different levels in the memory's hierarchy. For the figure 9 we show the caches misses in level L1 and L2 for the several inputs.

We can conclude by the graph below that the smaller the input array, the smaller is the execution time. If an array is small enough to fit into cache L1 or L2, then the process will take less time to get elements from the array because all of its elements are stored in the closest cache to CPU. When an array is too large to fit all of it's elements in the cache L1 and L2 than the processes are going to take more time to get values from the array making the execution less efficient.

In the graph bellow we can analysed the number of cache misses in level L1 and L2 which happens when a process makes a request to retrieve data from a cache, but that specific

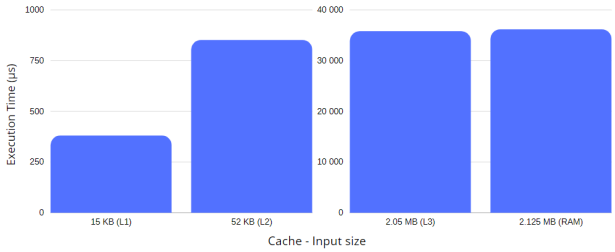

Fig. 8: Execution Time for input size

data is not currently in cache memory. For arrays that fit in cache L1 (15 KB) the number of misses for L1 is a lot smaller because all of its elements are stored in that cache. However for the 15KB array there are still a few misses in cache L2, that may happen when the cache L2 stores some variables needed to performed the algorithm. We can also see an increased of the number of misses in L1 when the input has size 50KB because the array does not fit in its entire in cache L1, having some elements stored in L2. Finally, when the input does not fit into the any levels L1 or L2 than the number of misses largely due to the probability of an element being in those levels hugely decrease.

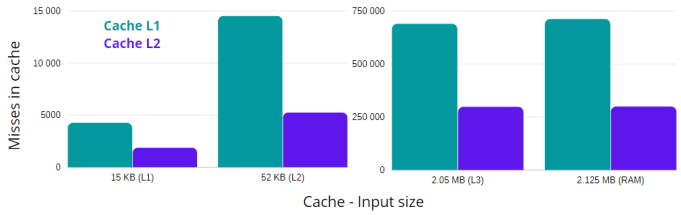

Fig. 9: Misses in cache L1 e L2

VI. CONCLUSION

Given the completion of the project, we present a critical, thoughtful and comprehensible view of the work developed.

On one hand, we consider that the parallelization developed for the bucket sort algorithm allows to better exploit its benefits and potential. We also consider that the tests and metrics applied do the parallel and sequential versions are diversified enough to correctly evaluate the algorithm presented.

On the other hand, the project could benefit from some upgrades like improving the parallelization of the algorithm by adding a schedule construct to the OpenMP directive. This construct would distribute the load of the loop cycles between the threads and potentially making the algorithm more efficient.

Furthermore, this report shows a detailed explanation of the bucket sort algorithm, parallelization implemented and the tests developed, allowing the reader to fully understand the reasons behind each choice made. To sum up, we consider that the work developed is complete, the problems that arose were overcome and the goals of the project were achieved.