Bucket Sort Algorithm Optimization

Universidade do Minho, Computação Paralela

Pedro Veloso PG47578
Carlos Preto PG47089

Abstract — **In this report we will explain the processes that led to the implementation and optimization of the Bucket Sort, using OpenMP, and also show and analyze the different tests that were conducted, in order to study the scalability of the algorithm.**

I. Introduction

In the context of Parallel Computing, were asked to develop a parallel version of the bucket-sort algorithm, using the programming language C and also using OpenMP. This algorithm can be implemented in 4 different phases, where the first one consists in initializing a bucket vector, the second one consists in allocating each number in a bucket, the third one consists in sorting each bucket and finally, the fourth phase consists in putting each number in the final ordered vector.

Before developing the parallel version, it was necessary to develop a sequential version of this algorithm, and only after this one was finished, we could start to use OpenMP to explore the different parallel parts of the code

Throughout the report, we are going to indicate and explain each decision we made to develop the most efficient version of the bucket-sort algorithm we managed to implement.

II. Sequential Version

A. *Sequential Version Implementation*

As it was mentioned before, the first stage of the development of the parallel version of the bucket-sort algorithm was developing its sequential version.

Firstly, we decided to create a structure named Bucket, composed by three arguments. The first argument represents the index where the next element in the specific bucket will be inserted in, the second argument represents the number of elements inside the bucket in a given moment, and finally, the third argument represents que list of elements inside the bucket.

Once the structure was defined, we could start to think on the bucket sort algorithm.

Firstly, we start by storing the values, which we were given, in an array. At the same time, we count the total number of elements in that given array and store its highest value. This highest value will be helpful to decide in which bucket a number should be stored in.

After all the values are stored and we got all the information we need, it's time to start separating each value into its respective bucket.

In order to know in which, bucket each value will go, we have to take in consideration its value, the number of buckets and the maximum value in the array. That being said, we multiply the number of buckets by the value of an element and divide the results by the maximum value in the array. Then, using the floor function, we get the index of the bucket in which we will insert the element.

However, if the element in question is the highest value, then the index of the bucket in which it will be inserted into is decremented, in order to maintain the number of buckets defined in the beginning.

Once the division is finished, we use a sorting algorithm to sort each bucket, and once each bucket is sorted, we put each ordered value in its correct position.

B. *Sequential Version Optimizations*

After the development of the first sequential version of the bucket sort algorithm, it was time to analyse the areas where we could improve it.

The first choice we made was to store each element of a given bucket inside an array, instead of using linked-lists, because when we use linked-lists, we are working with memory pointers which allows for the values to not be stored in continuous memory positions. This implies that some of the values stored in cache do not represent the elements in the bucket, which doesn't allow us to take advantage of the faster cache accesses.

That being said, we observed a reduction of L1 and L2 cache misses when changing from linked-lists to arrays.

The second choice implied choosing an efficient sorting algorithm to sort each bucket. After brief research, we decided to try two different algorithms, the Merge Sort and the Quick Sort (recursive version). Both these algorithms have an average complexity of $\Theta(n*\log(n))$, where n represents the number of elements in the array. However, after a few tests, we noticed that Merge Sort had less cache misses and when working with big arrays it showed better results. For example, while testing the sorting of an array with 10 million integers, five times, we got the results shown in the following table.

Table 1: Quick Sort vs Merge Sort

	Algorithms	
	Quick Sort	***Merge Sort***
PAPI_TOT_CYC	38926968142	4059040405
PAPI_TOT_INS	115405672945	7163346956
PAPI_L1_DCM	320325280	7816266
PAPI_L2_DCM	1446106	761918
CPI	0.34	0.57
AVERAGE TIME	16.15s	1.52s

As we can see in the table, for a big array, the L1 and L2 cache misses are way lower, and so is the number of instructions.

After these decisions, we generated a detailed file with the perf report tool, in order to see the overhead of each function in our code. After a careful analysis, we noticed that one of the costliest operations was associated with the *realloc* of the buckets, where we were allocating space in the same time as we were iterating through the array. This process had an overhead of around 8%, and in order to remove this overhead, we tried to, instead of always using *realloc* to increment the bucket size by one, when another element would be inserted in that bucket, to reallocate more 100 integer space in memory. This decision didn't reduce the overhead that much, so we decided to firstly count how many elements each bucket would have, and only after knowing the total number of elements of each bucket, allocate all the memory necessary for each bucket in one go, using malloc.

Another important decision involved using the flag -O3 when compiling the code, which reduces the total number of instructions of the program by almost half, and also decreases the execution time.

After all the decisions explained above, we noticed that the most time consuming task was related to the functionality of the Merge Sort algorithm. In order to reduce this time consumption task, it would be necessary to parallelize the Bucket Sort algorithm.

III. Parallel Version

A. *Parallel Verision Optimizations*

As it was said before, the next step to optimizing the algorithm resorted to the parallelization of some of its functions, using OpenMP.

Analysing the perf report of the sequential version of the Bucket Sort algorithm, we noticed that around 64% of the algorithm's execution time was consumed, by the merge function, which is called in in the Merge Sort recursively.

The first thing we tried to was to parallelize this function, by creating two different tasks, where the first tasks would copy the first received array into a temporary array, and the second tasks would copy the second received array and copy it into another temporary array. After some tests, we realized that this strategy, instead of decreasing the execution time, it actually raised it. This can be explained by the fact that the time to create and manage threads when we have a small array is higher than when we have a big array, but because the majority of time we have small arrays, the performance gets degraded. In order to treat this, we decided that if the size of the array that we are going to copy into another temporary array has more than 3 million integers, we parallelize the filling of the temporary array, otherwise, we let the filling of the arrays be sequential.

This improvement can be seen when we have a lot of numbers per bucket. For example, if we sort 10 million with one single bucket, the time differences can be seen in the following table.

Table 2: Array Filling Optimization

	Without Optimization	***With Optimization***
AVERAGE TIME	2.49s	2.26s

The rest of the Merge Sort algorithm had some dependencies, so it was very difficult to parallelize those parts in an efficient way, due to the critical operation.

After the sorting algorithm been optimized, it was time to improve the performance in the Bucket Sort, by sorting each bucket simultaneously. In order to do that, we distribute the threads dynamically for the number of buckets. This is beneficial when we have a lot of buckets to sort, which implies using the Merge Sort for a reduced number of integers. As we can see on the table, when we want to sort 10 million integers and use 1000 buckets, the average time are as follows

Table 3: Bucket Sorting Optimization

	Without Optimization	***With Optimization***
AVERAGE TIME	1.55s	0.63s

Once this optimization was concluded, we analysed the perf report one more time and realized that now, 66% of the execution time of the algorithm is spent in memory allocation (mallocs), which is necessary for the correct behaviour of the algorithm and can't be parallelized.

That being said, it was time to test the scalability of our developed algorithm.

IV. Scalability Study

Once the parallel version was fully developed, it was time to study the scalability. In order to do that, we ran a series of tests, where we used different data dimensions, to exploit the different levels of hierarchy memory, such as L3, L2 and RAM.

It's important to refer that the tests for the sequential vs parallel version and the tests for the number of threads were executed in the node 134-115, which has 48 processor units of Intel Xeon Processor E5-2695 v2, with 12 core each and 24 threads each. It also has 62 GB of RAM in total, 32kbytes L1 cache and 256kbytes L2 per core.

However, we also test the impact of executing the program on another machine, mention in the next sections.

A. *Sequential vs Parallel*

As it was expected, for all the different input sizes, when using only one bucket, we get parallel execution times values similar to the sequential version, given that the main optimization in the parallel version was to divide the different threads by the different buckets, and when there's only one bucket, there won't be more than one thread executing. However, there's a minimal difference because of the parallelization of the copy of the values in the array to a temporary array, which makes it a tiny bit faster.

With smaller size arrays, it's possible to store the majority of its values in the different cache levels. This makes memory accesses faster, which leads to a faster performance. With parallel algorithms we have more L2 cache misses (around 48% higher), which implies that there's little difference between the parallel and the sequential version. So, for small arrays, it may be beneficial to use the sequential version instead of the parallel version.

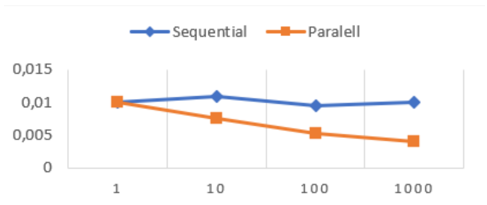

Figure 1: Sequential vs Parallel - 50 Thousand Integers

When working with a bigger input number, we noticed that the difference between the L1 and L2 cache misses in both version is lower, because the cache misses in the sequential version increases and is almost identical to the misses in the parallel version. This way, with the increase of the number of buckets, we have smaller execution times in the parallel version.

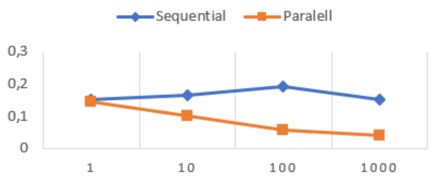

Figure 2: Sequential vs Parallel - 10 Million Integers

Finally, when working with an even bigger array (100 MB), the L3 cache is not sufficient to store all the array, so we see bigger gains in the parallel version, especially when using a higher number of buckets.

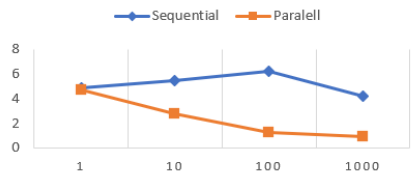

Figure 3: Sequential vs Parallel - 30 Million Integers

B. *Thread Numbers*

In order to know how different number of threads affects our program, we decided to execute the algorithm with 2, 8, 16, 32, 64 and 128 threads. For each thread number, we used 1, 10, 100 and 1000 number of buckets of, and also an input size of 50 thousand (195,32 KB), 10 million (3.8147 KB) and 30 million (114.44 MB), to test the different memory hierarchical levels.

a) 2 Threads

Firstly, we tested the sorting time of an array with 50 thousand integers. As we can see in the Figure, the execution times vary between 0,009s and 0,015s. These values are similar to the ones obtained in the sequential version, so we conclude that it's not very advantageous to have 2 threads and a large number of buckets.

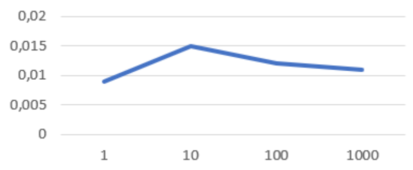

Figure 4: 2 Threads – 50 Thousand Integers

After that, we tested the sorting time of an array with 10 million integers. As we can see on the Figure, because of the optimizations, we can have similar times with few and more buckets.

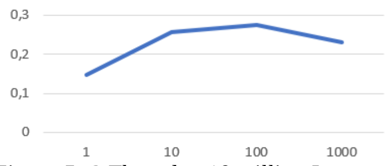

Figure 5: 2 Threads - 10 million Integers

Finally, we tested the sorting time of an array with 30 million integers, and the behaviour was the same as it was in the 10 million integers. The only difference is that the times are a lot higher, ranging between 5 and 8 seconds.

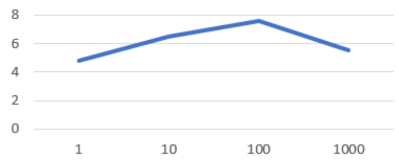

Figure 6: 2 Threads - 30 million Integers

b) 8 Threads

With 50 thousand integers, we noticed that the number of buckets doesn't impact that much the execution time, with values ranging from 0,01s to 0,005s.

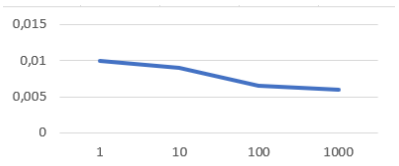

Figure 7: 8 Threads - 50 Thousand Integers

The same happens for 10 million integers, where 1 or 100 buckets have the same impact on the execution time of the algorithm. Only when using more than 100 buckets we see a execution time 0,05s lower than the rest.

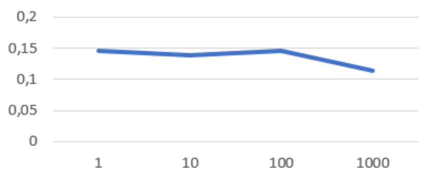

Figure 8: 8 Threads - 10 Million Integers

When using 30 million integers, we noticed that the execution times are pretty similar, rounding the 5s, with the exception of the scenario when we use 100 buckets. This can be considered and outlier, as it does not represent the usual behaviour of the algorithm.

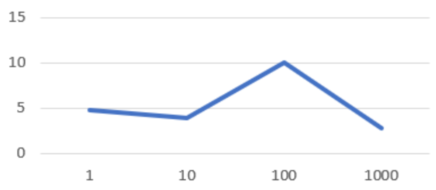

Figure 9: 8 Threads - 30 Million Integers

c) 16 Threads

As we can see in the Figure, when using 50 thousand integers, after reaching 100 buckets, we don't need to increase the number of buckets, because after that the time is constant, edging the 0,005s.

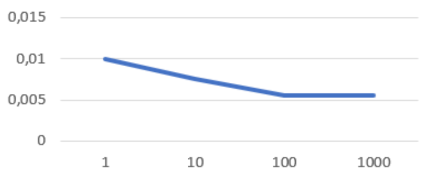

Figure 10: 16 Threads - 50 Thousand Integers

When using 10 million integers, we start to notice the advantage of having more buckets. As we can see in the figure, the time decreases from 0,15s to nearly 0,05s, we change from using 1 bucket to 1000 buckets.

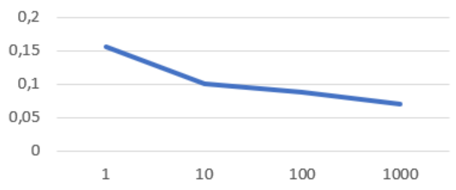

Figure 11: 16 Threads - 10 Million Integers

The same happens for 30 million integers, where the behavior is exactly the same as with 10 million integers, with the difference being the execution times being way higher, what is expected given the massive size of the array.

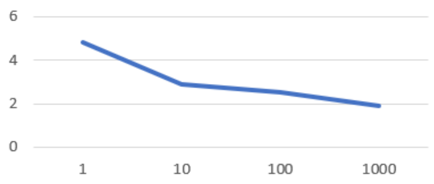

Figure 12: 16 Threads - 30 Million Integers

d) 32 Threads

With 32 threads, we start to see small advantages on using a large number of buckets in a small size array, with times continuously decreasing with the increase of the number of buckets.

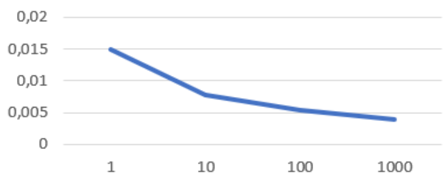

Figure 13: 32 Threads - 50 Thousand Integers

For arrays with 10 million integers, after we reach 100 buckets, the time execution difference starts to decrease massively, being almost constant after the 100 buckets.

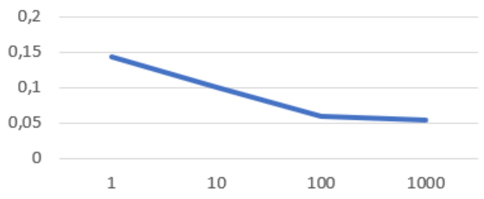

Figure 14: 32 Threads - 10 Million Integers

Finally, when working with arrays with 30 million integers, we can see that with the continuously increase of the number of buckets, the execution time decreases, decreasing from 5s to nearly 1s.

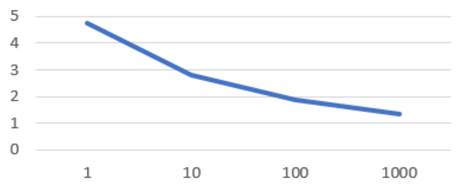

Figure 15: 32 Threads - 30 Million Integers

e) 64 Threads

Once again, for small arrays, nor the number of threads, nor the number of buckets seems to have a big impact performance wise, with the time varying between 0,014s and 0,005s, which is an unnoticeable difference.

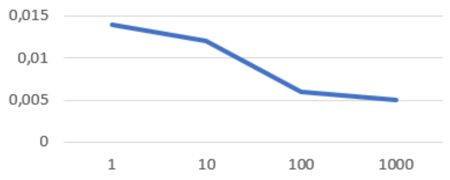

Figure 16: 64 Threads - 50 Thousand Integers

For big arrays however, the number of buckets and threads starts to have more impact, with times varying between 0,2s and 0,05s.

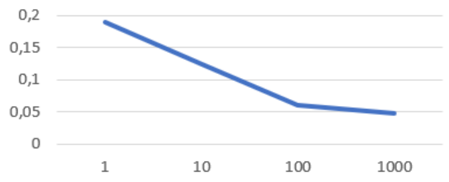

Figure 17: 64 Threads - 10 Million Integers

Finally, when working with enormous arrays, we notice the impact of the 64 threads, with the execution times having a maximum of 5s and reaching a minimum of 1s.

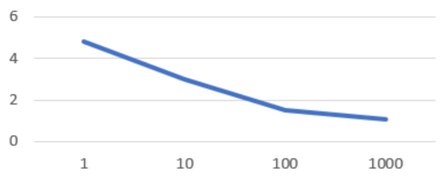

Figure 18: 64 Threads - 30 Million Integers

f) 128 Threads

For 50 thousand integers, nor the number of threads nor the number of buckets has a big impact on the results, because as, we can see, the difference from 1 bucket to 1000 buckets is around 0,01, which is unnoticeable.

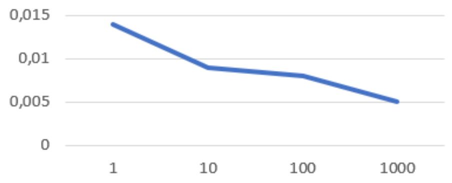

Figure 19: 128 Threads - 50 Thousand Integers

For 10 million integers, the 128 threads make the time vary between 0,05s and 0,15s.

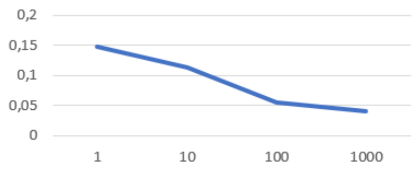

Figure 20: 128 Threads - 10 Million Integers

Finally, for 30 million integers, we see a big improvement in terms of execution times, managing to sort the array in a minimum time of 0,95s which is a great result for the sorting of a array of this size,

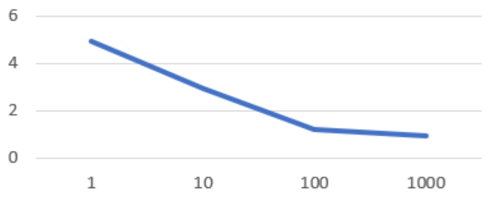

Figure 21: 128 Threads - 30 Million Integers

C. Different Machines

The final conducted tests involved analysing the impact of using a different machine for our tests. As we said before, the original machine where the prior results were conducted had a total of 48 cores and is represented by the blue line in the graphs. Then, we re-runed the tests on a machine with 8 cores, which is represented by the orange line in the graphs.

In addition to the number of buckets we had tested, we decided to add a new test for 10 thousand buckets, because we thought that we would see a lot more differences.

Firstly, when leading with relatively small arrays, the difference is minimal, with both machines presenting similar times of around 0,08s.

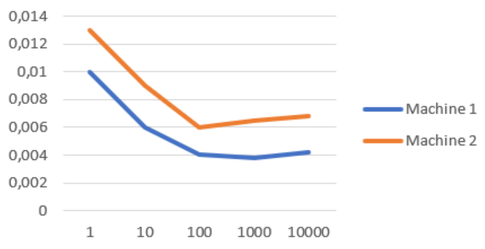

Figure 22: 50 Thousand Integers

For 10 million integers, we start to see that the major difference is in sorting the array using only one bucket. As we can see in the Figure, the execution time of the second machine is 0,15s slower than the execution time of the machine 1.

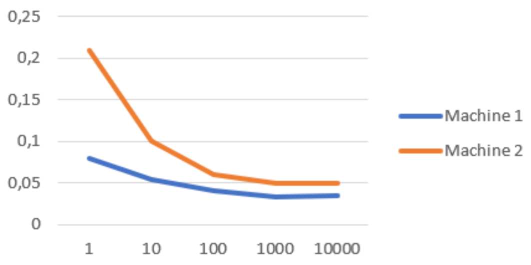

Figure 23: 10 Million Integers

Finally, for 30 million integers, we can see that, because of the better characteristics, the times in the first machine more than 1s lower, in each scenario.

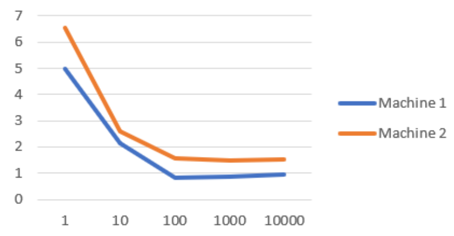

Figure 24: 30 Million Integers

V. Results Explanation

Observing the results shown in the previous chapter, we can make some assumptions on the impact of the number of threads and type of machine where the code is executed.

When talking about the number of threads, we realized that the usage of a small number of threads doesn't allow to take advantages of the parallel optimizations we made to the algorithm. This is very noticeable when trying to sort arrays with a large number of integers. However, an increase of the number of threads doesn't always translate an increase in the performance, because the machines where the code is executed have limited resources. For example, in the Machine 2, we can observe that the performance is practically the same when using 64 and 128 threads.

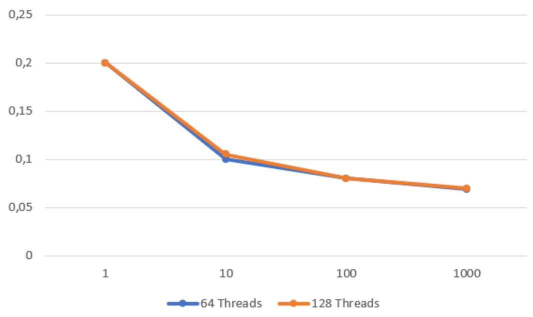

Figure 25: Thread Times Comparation

Another important aspect are the different machines used. For example, in the conducted tests, we obtained better results with the Machine 1, which had as specifications already mention, in chapter IV, than Machine 2, which have 16 processor units of Intel Xeon processor e5520. That implies, that Machine 1 has 3.75x more cache than Machine 2.

This difference has an impact in the observed results, especially when only using one bucket to sort the array, because it has a lot less misses in the cache.

VI. Conlusion

In this project, we were asked to develop an algorithm that could sort a large number of integers. In order to obtain that, we needed to improve the Bucket Sort algorithm, using the programming language C and also using OpenMP.

However, the Bucket Sort can use another algorithm to sort each of its buckets, so we studied different sorting algorithms, like Quick Sort and Merge Sort, and after a few tests, we decided to use the Merge Sort.

Then, we tried to improve the sequential version even more, by using the PAPI tool, to see info like cache misses, number of instructions and execution times. Some of the improvements implied changing the structures being used, reduce the number of *reallocs*, and using the -O3 flag to compile de code.

After that, we started to parallelize all the algorithm, where the best optimization was to divide each bucket to different threads.

Finally, we conducted a lot of tests to see the scalability of the developed algorithm and tried to understand the values and explain them.

References

[1] https://ark.intel.com/content/www/us/en/ark/products/40200/intel-xeon-processor-e5520-8m-cache-2-26-ghz-5-86-gts-intel-qpi.html

[2] https://ark.intel.com/content/www/us/en/ark/products/75281/intel-xeon-processor-e52695-v2-30m-cache-2-40-ghz.html.