Computação Paralela

Carlos Carvalho
UC de Computação Paralela - Dept. de Informática
Universidade do Minho
Braga, Portugal
pg47092@alunos.uminho.pt

Luís Vieira
UC de Computação Paralela - Dept. de Informática
Universidade do Minho
Braga, Portugal
pg47430@alunos.uminho.pt

***Abstract*—This document is about the project proposed by the teachers of the Parallel Computing Curricular Unit in the 2021/22 school year.**

***Index Terms*—OpenMP, C, bucket-sort, PAPI**

I. Introduction

This work aims to evaluate the learning of parallel programming in shared memory using the C language and OpenMP. For this we were proposed to develop a parallel version of the bucket-sort algorithm. This algorithm can be implemented following a scheme of four steps, these being:

- Start an array of buckets
- Putting each number in a bucket
- Sort each bucket
- Put the numbers in the final vector

It was also suggested that the project be divided into four phases, which will be more easily evaluated following these parameters:

1º Development of the sequential version of the algorithm, in C, for sorting a vector of integers, including impact analysis of possible optimizations to the sequential. version.
2º Development of the parallel version of the algorithm using OpenMP, including the study of the various trade-offs in exploiting parallelism in this algorithm.
3º Study the implemantatio scalability on one of the nodes of the SeARCH cluster (with a minimum of 16 cores).
4º Explanation of the obtained results.

II. Resources Used

Before we proceed to the development of the content concerning the implemented algorithm, we think it is important to mention the resources that were used throughout the project.

In this project, making use of the University of Minho Informatics Department *cluster*, we had access to different partitions with different nodes where we could execute our algorithm. Although in the statement it was mentioned that we should use at least 16 cores (compute-145-1 or compute-165-1), when we tried to execute the algorithm with PAPI instructions it was not successful. In order to maintain consistency we used the same node for all measurements, *compute-134-115*, which has only 12 cores, present in the partition *"day"*.

III. Bucket Sort Algorithm

A. Algorithm Explanation

Bucket Sort is an algorithm that divides the elements of an unordered array into several groups, called *buckets*. Each *bucket* is then sorted using some sorting algorithm or by recursively applying the Bucket Sort algorithm. Finally, the already sorted buckets are combined to form the final sorted array.

IV. Sequential Version

A. Test Scenarios

In order to best test our algorithm we made use of two main test scenarios, one with an array of size 500000 (five hundred thousand) and another with an array of size 1000000 (one million). In addition to testing the algorithm with different size arrays, we also changed the *bucket* size in each implementation, varying these sizes between 10, 100, 1000 and 10000.

That said, the table below shows the results obtained for the various mentioned values. Note that our focus was given to arrays of this size in order to better understand what significant differences the results may present depending on the different bucket sizes. We consider it important to mention that the test scenarios were not performed with smaller arrays since when testing the execution was almost instantaneous.

	SIZE	BUCKET RANGE	TIME
0 - 500000	500000	10	0.232s
	500000	100	0.255s
	500000	1000	1.370s
	500000	10000	20.646s
0 - 1000000	1000000	10	0.486s
	1000000	100	0.529s
	1000000	1000	2.593s
	1000000	10000	44.606s

In the table above we can see the significant time differences generated by changing the *bucket* sizes. This growth may also be due to the algorithm used for sorting within each *bucket*.

Another aspect that we decided to investigate for the sequential version was the memory accesses that it made, making use of **PAPI**, in order to investigate cache misses, so that we could later compare with the parallel version and verify the differences in these values, as well as the number of cycles and instructions calculated, allowing us to compare the CPIs

of the different executions. The results obtained from these measurements will be presented later.

B. Possible Optimizations

As we can see in the previous section, there are large temporal disparities when we increase the size of the *buckets*. This could be due to the algorithm we use for sorting within each *bucket*. So a possible performance improvement, in order to correct these significant time differences, would be to change the algorithm (*Insertion Sort* to a more efficient one, such as, for example, *Quicksort*, among others.

Another problem that we identified in this algorithm, and that can be seen in the image below, is the fact that the array provided may not contain numbers in a certain range, thus leading to unnecessary memory usage since space is allocated for empty *buckets*. This could be corrected by checking, after all elements are distributed, whether or not there are empty *buckets* and thus removing them, in order to free up the occupied memory.

```
Size: 5
NBUCKETS: 5
Initial array: 3 4 15 35 48
-------------
Bucket[0]: 4 3
Bucket[1]: 15
Bucket[2]:
Bucket[3]: 35
Bucket[4]: 48
-------------
Bucktets after sorting
Bucket[0]: 3 4
Bucket[1]: 15
Bucket[2]:
Bucket[3]: 35
Bucket[4]: 48
-------------
Sorted array: 3 4 15 35 48
```

Fig. 1. Empty Buckets after Insertion

V. Parallel Version

A. Objective

Having implemented, tested and evaluated the sequential version of the algorithm we now proceed to its parallelization in order to improve its performance using the OpenMP library. This library allows us to better explore the hardware we have at our disposal and allows us to use threads to execute our algorithm.

Relying on several different instructions, our main goal in using this library and this parallelization technique is to improve our algorithm, ensuring that it remains functional in its entirety meaning that in the end the array remains ordered.

In the following sections we will present the decisions we made with the aforementioned goal always in mind, explaining the reasons for them and possible failures and possibilities for implementing other solutions.

B. OpenMP Implementation

The first steps to implement parallelization is to evaluate the previously developed sequential version and find out what can be parallelized. So, we start by looking for instructions in the main algorithm that can be parallelized. As almost all of them make use of *for* cycles, we assume the possibility of using the *#pragma omp parallel for* instruction in all scenarios in an attempt to increase the performance of our algorithm.

The truth is, the algorithm's performance increased. However, its functionality was not maintained, that is, the final array did not come sorted as intended. So we decided to take another approach to observing our possibilities and started implementing the same instruction one cycle at a time.

In the first cycle, responsible for initializing the buckets that will later be filled, the presence of this instruction effectively improved the algorithm's performance and maintained its full functionality. We were able to conclude that this cycle is parallelizable and soon started looking for the use of some more instructions that could further affect the performance, but we quickly realized that it was unnecessary given the atomicity of the instruction performed in this cycle.

Moving on to the next cycle, responsible for filling the *buckets* with the elements of the array to be ordered, in the respective index, according to the interval stipulated for each *bucket*, we followed the previously stipulated strategy. We added the *#pragma omp parallel for* instruction and tested the results of this change. It was here that we began to realize the problem we had encountered when we tried to parallelize all the cycles. The algorithm's performance was improved but its functionality was compromised, since the final array was not properly sorted. We tried to use other instructions in order to counteract this behavior, namely the use of *schedule*. With the use of this instruction and its parameters, *dynamic*, *static* or *guided*, the goal was to divide the cycle and execution of its instructions by the various threads as an attempt to counteract possible read/write errors that might be occurring. However the end result was the same. In order to maintain the full functionality of the algorithm we decided not to parallelize this cycle.

Continuing with the defined strategy, in the next cycle, responsible for sorting each of the buckets, by adding the parallelization instruction we again found that the algorithm's performance was improved and its functionality was not altered. With this established, we looked for new situations where we could make use of parallelization. Since only one instruction is executed within the cycle, we investigated the possibility of parallelizing the content of each bucket's sorting algorithm, the *InsertionSort*. However, we did not find any solution that met our goal. Thus, for the third cycle under study, we concluded that also only the simple instruction would be used.

Finally in the last cycle, responsible for re-introducing the ordered values from the buckets into the array, we encountered the same problem that appeared in the second cycle. Having searched for other solutions that could help and not having

any success, we decided not to parallelize this cycle either, so as not to affect the functionality of the developed algorithm.

Now that the parallelized version of the algorithm has been defined, we can proceed to evaluate it by running several test scenarios and then comparing them to the results obtained from the sequential version.

C. Test Scenarios

Again, in order to best test our algorithm we made use of two main test scenarios, one with an array of size 500000 (five hundred thousand) and another with an array of size 1000000 (one million). Besides testing the algorithm with arrays of different sizes, we also changed the *bucket* size in each implementation, varying these sizes between 10, 100, 1000 and 10000.

That said, the table below shows the results obtained for the various values mentioned. Note that the focus was once again, as for the sequential version, the use of arrays of this sizes in order to better understand what significant differences the results may present depending on the different bucket sizes. We consider it important to mention that these values are the same used for the sequential version in order to make the tests as reliable as possible.

	SIZE	BUCKET RANGE	TIME
0 - 500000	500000	10	1.045s
	500000	100	0.442s
	500000	1000	0.250s
	500000	10000	1.203s
0 - 1000000	1000000	10	0.883s
	1000000	100	0.369s
	1000000	1000	0.459s
	1000000	10000	2.156s

In the table above we can observe the significant time differences generated by changing *buckets* sizes, for a version using parallel OMP. We can also observe that there are no very significant differences in execution times, which, contrary to what was observed in the sequential version, where the times had an exponential growth, is undoubtedly a very significant improvement.

Besides this evaluation of execution times, and as we did in the sequential version, we investigated the various memory accesses that the parallelized version performs as well as the number of instructions and cycles executed, in order to compare it later on with the sequential version.

VI. Results Obtained

In the following figures we can see a comparison of the sequential and parallelized versions. We present below three bar graphs in order to investigate cache misses and variations of the CPIs of the different runs. These values were collected for the various test arrays of 500000 and 1000000 entries and also varying the various bucket sizes between 10, 100, 1000, 10000.

For the following tests we did not specify the number of threads to use, so the algorithm used the number it had at its disposal, in this case 48 threads.

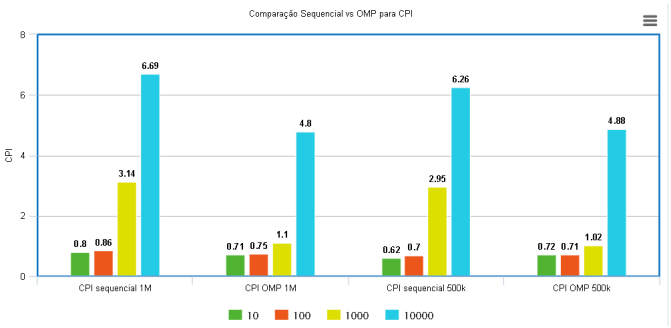

Fig. 2. Double Bar Graph comparing CPI

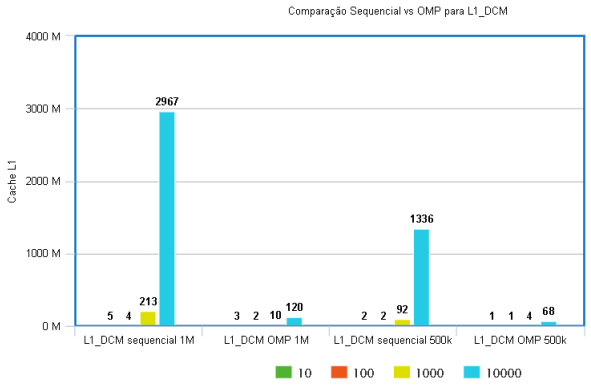

Fig. 3. Double Bar Graph comparing L1_DCM

It is notable that the main differences in any of the metrics come mainly from the different intervals for the buckets we stipulated, the most notable being when we use an interval of 10,000 (ten thousand).

Starting by analyzing the graph comparing the CPIs between the two versions and the two test arrays, we can conclude that for the same version, the values between the different arrays do not vary in a very noticeable way. However, going from the sequential version to the parallel version we can confirm that there is a significant decrease in the number of cycles per instruction, which already shows the impact that parallelization had on our algorithm.

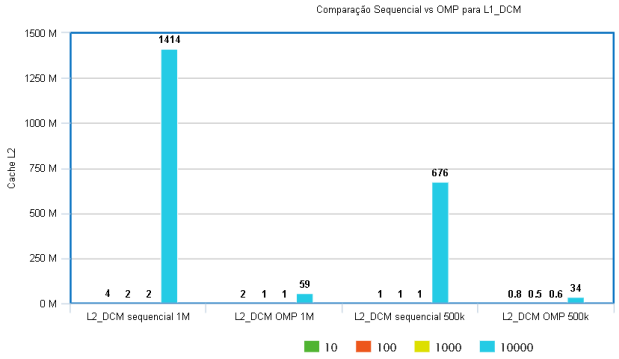

Fig. 4. Double Bar Graph comparing L2_DCM

Conversely, when we look at the graphs containing the number of misses at the L1 and L2 level of the cache we see a greater disparity between the test scenarios of each version, being undeniable for the case where the range of each *bucket* is 10 000 (ten thousand). This factor comes from the number of instructions needed to be performed for the different test scenarios given the larger number of elements to be sorted. However, similarly to what was seen for CPI, there is a significant improvement from the sequential version to the parallelized version when comparing the number of misses at these levels.

Having made the comparison between the different versions of the algorithm, we decided to find out what difference the number of threads running the parallelization would make. To do this we varied the number of threads executing the algorithm between 1, 2, 4, 8, 16 and 32. From these measurements we were able to draw some conclusions that dictated the remaining measurements.

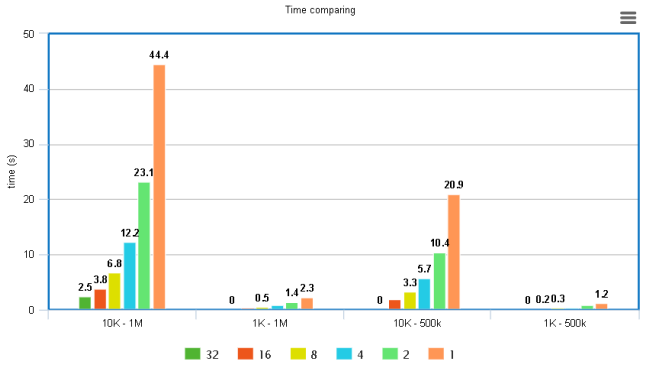

Fig. 5. Bar Graph comparing Time Execution for Different Threads

Starting by the heaviest test scenario, with one million elements and a range of ten thousand for each bucket, we quickly realized that from 16 threads onwards, continuing to increase did not compensate for the time gained when taking into account the overhead of creating these threads, so only this scenario contains a measurement of the execution time for 32 threads. Looking now at the graph in general, as expected, as we increase the number of threads the algorithm's execution time decreases.

Since we concluded that it was not rewarding to increase the number of threads after 16 threads, for the remaining measurements we set a limit of 16 threads for the various scenarios.

Just as we did to compare the different versions in terms of memory accesses and cycles and instructions executed, we will now compare between the different number of threads the misses at the L1 and L2 cache levels, as well as the CPI.

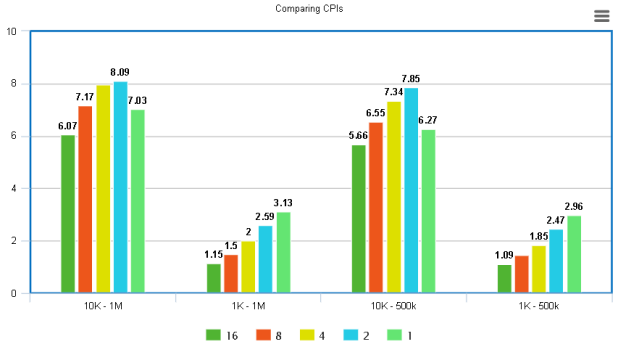

Fig. 6. Bar Graph comparing CPI for Different Threads

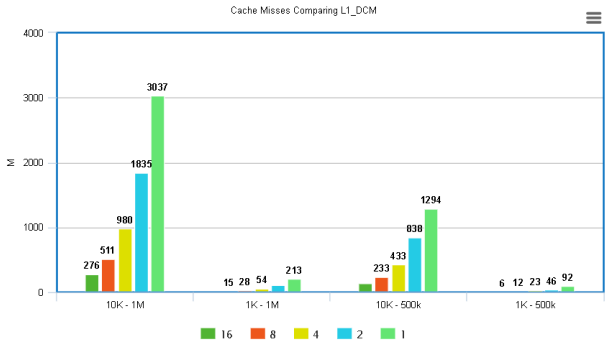

Fig. 7. Bar Graph comparing L1 Misses for Different Threads

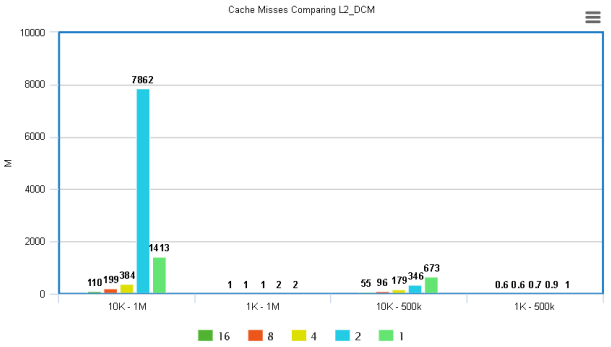

Fig. 8. Bar Graph comparing L2 Misses for Different Threads

As expected, as we decrease the number of threads we see, both for level L1 and L2, an increase in the number of cache misses. Note the particular case at the L2 level for the test scenario with one million elements and a ten-thousand interval for each bucket, for which we were unable to find an explanation for its occurrence. There is also an increase in CPI as this decrease in the number of threads happens.

Based on the results obtained from the different tests we could verify that there were indeed improvements in the algorithm with the parallelization of the algorithm, although this is not in great abundance throughout. Nevertheless, these improvements are visible both in terms of cache, CPU performance, and execution times.

However, there were other possible measurements, such as measuring the execution time of each thread so that we could compare the distribution made by them, among others.

VII. Conclusion

In this project, having as an objective to evaluate our learning in this curricular unit relative to parallel programming in shared memory, using the C language and OpenMP, it was proposed to us to develop a sequential and a parallelized version of the bucket-sort algorithm. This project can be seen as having 4 distinct phases: the first phase consists in the development and analysis of a sequential bucket-sort algorithm; the second phase consists in the conversion of this algorithm, using OpenMP, to a parallel version of the same; the third phase is where the scalability of the two implementations is studied and compared; finally, the fourth and last phase concerns the explanation of the results obtained.

As a group, we believe we have met the objective proposed by the teachers, but there are certain aspects that we are aware of that could be improved. Some of them are aspects such as: the structure we chose to work on the BucketSort, the sorting algorithm for each *bucket*, the way we indicate the initial array to be sorted and also the parallelization applied to the algorithm.

However, there were also limitations over which we did not have much control and that could improve aspects of our work, such as the analysis done at the cache level. In order to, as intended in the statement, evaluate the cache at levels L2, L3 and RAM, we tried to find a node of the cluster that would allow us to do so. However, whenever we tried to search for the misses in the L3 level of the cache, using PAPI_L3_TCM, since the architecture of the cluster nodes does not support PAPI_L3_DCM, we could not run the program and thus could not get these values.

Nevertheless, the group considers that even with all these adversities, the project was successfully concluded in the sense of integrating and evaluating parallelization in a sorting algorithm, exploring the possibilities that the OpenMP library offers, as well as consolidating theoretical and practical concepts taught during the Parallel Computing course.