Computação Paralela

Programação Paralela em Memória Partilhada

João Santos
Dept. de Informática - UC de Computação Paralela
PG47304
Braga, Portugal
pg47304@alunos.uminho.pt

Paulo Sousa
Dept. de Informática - UC de Computação Paralela
PG47556
Braga, Portugal
pg47556@alunos.uminho.pt

***Abstract*—This document is a report of the subject Computação Paralela project, which was proposed by the professors in the 2021/2022 school year.**

***Index Terms*—parallel programming, OpenMP, PAPI, C, bucket-sort, Shared-Memory**

I. Introduction

This project has the purpose of evaluating parallel programming using the C programming language and OpenMP. Therefore, it was proposed to develop a parallel version of the bucket-sort algorithm.

This algorithm takes advantadge of the division of an array of elements that can be ordered and divides the elements into ”buckets”. This buckets contains the elements of a certain range (for instance, for int elements it can store the numbers in buckets with a range of 10). The bucket-sort algorithm can be implemented in four simple steps:

- Create an array of buckets
- Insert each number in the respective bucket
- Sort each bucket
- Insert the content of each bucket in the ordered array

The development phase was divided into four phases, which were also the parameters used in the evaluation of the project quality:

1) Using C, develop an algorithm sequential version that sorts a integer array, including the impact analysis of possible optimizations to this version.
2) Use OpenMP to develop a parallel version of the algorithm, including the study of the multiple existent commitments of the parallel exploration of this algorithm.
3) Study of the implementation scalability in one SeARCh cluster node (minimum of 16 cores). Multiple test realized will be valued: data dimension, use of other machines, etc... It is suggested to use a size of the ordering input set adequate to the memory hierarchy levels (L2, L3 and RAM).
4) Justification of the obtained results, the gathering of measurements/metrics that sustain the results will be valued.

II. Resources Used

In this project was made use of the University of Minho Informatics Department’s cluster. We were able to access different types of partitions and nodes where the multiple algorithm versions could be executed and tested. Considering this, we different nodes and partition for executing and testing, the options chosen were the partition ”cpar” and the nodes 134-[101-102].

III. Bucket-Sort Algorithm

A. Specification and Clarification

As written in the introduction the Bucket-Sort algorithm separates the elements of an unordered array into several groups(buckets). Then, the content of each bucket is ordered using sorting algorithm (the bucket-sort algorithm could be used recursively). Next and to finish, the elements, ordered in the respective bucket, are added to an array, which will be sorted. In this project the objective is to create a parallel version of this algorithm. Therefore, such thing as obtaining an unordered array will not have any impact on the results attained. On the other hand, other factors, for instance, sorting the buckets, creating the buckets and filling the ordered array will have impact on the results.

IV. Sequential Version

A. Implementation decisions

To implement the sequencial version of the algorithm we had to make some decisions.

Firstly, we had to choose between two different data structures to store the information in the bucket, an array or a linked list. Using arrays has the advantage of being continuously stored in memory, resulting in faster searching. Also, it has the advantage of having spatial cache locality, resulting in less cache misses because a continuous block of memory is loaded to the cache. By contrast, linked lists have the advantage of being more efficient at allocating memory. In the end, we decided to use arrays, because its advantages are more important to our implementation.

Another decision that we had to take was how to allocate memory, statically or dynamically. Although static memory allocation would be much faster in terms of run time, it would waste a lot of memory, since, for each bucket, we would have to allocate an array of size N. Because of that, we decided to allocate the memory dynamically, by using the function *realloc*. Every time we add a new element to a bucket, we reallocate the memory to the previous array size plus 1.

Furthermore, we had to choose the algorithm to order each bucket. In the beginning we used the InsersionSort algorithm, but soon realised that, for large arrays, it was very inefficient. Therefore, we chose QuickSort, as it is a much more efficient algorithm for very large arrays and improves the execution times. Here, we could have gone further and try to implement the RadixSort algorithm, which has O(N) time complexity, instead of O(N log N), like QuickSort.

Lastly, we decided to use various array sizes to test our implementation (as it is going to be explained ahead) in order to explore the different cache levels. As the array size increases, the cache space fills out and it is necessary to explore lower hierarchy cache levels. For instance, if we are in the cache's first level, L1, and it fills, we will start to use a lower hierarchy, slower cache level, L2, this goes on until it fills the L3, third level cache, then it starts using RAM. Also, as the array size increases, the number of cache misses also increases.

V. Parallel Version

A. Objective

Having tested the sequential version we faced with the possibility of improve the algorithm performance. So, we decided to use the OpenMP library, which provides a deeper exploration of the hardware at our disposal and the use of threads to execute the bucket-sort algorithm implementation.

The algorithm should functional, that is the array is still ordered after the optimizations. With that in mind we used different OpenMP library instructions to improve/optimize the algorithm.

Taking the goal into consideration, it is explained below the decisions taken in order to improve the algorithm. Consequently, the reasons for implementing or not implementing some modifications to the algorithm are also clarified.

B. OpenMP Implementation

To implement OpenMP, in this case is implementing a parallel version of the bucket-sort algorithm, we must first identify which are the regions that can be parallelized. Therefore, we notice some code regions that can potentially be parallelized, this regions are the *for cycles* present in the algorithm, the cycle used to initialize the the buckets, the cycle to add elements to the respective buckets, the cycle to order the buckets and the cycle to orderly add the elements to the original array of integers. Since the identified regions were *for cycles* we used the instruction *pragma omp parallel for* to parallelize the bucket-sort algorithm. However, we noticed that implementing the instruction specified above in the cycle that adds the elements to the respective bucket would not have any improvement, this is due to the fact that the operations inside the cycle are critical and the use of an OpenMP instruction(*pragma omp critical, in this case*) is not relevent, once the performance improvement would not be substancial.

In the third *for cycle* besides using the instruction *pragma omp parallel for* we used the instruction *pragma omp schedule(dynamic)*, once different iterations take different amounts of time/computacional cost this was the best solution to improve performance the most. This way the loop distribution is not always the same and the first thread available is the one chosen to execute the iteration. On the other hand the first and fourth loop, as their iterations carry the same computacional costs and amounts of time, the schedule assigned is *static*, this is the default for OpenMP *for* instructions.

In addiction, the last *for cycle* has another OpenMP instruction, *pragma omp private(size)*. The variable size is a variable that is shared by all threads and its value has uttermost importance when it comes to obtain the correct output of the algorithm. Therefore, by using the clause *private* for each thread the variable size will be local, thus ensuring its integrity.

We also tried another solution that we chose not to implement, which was parallelizing the Quick-Sort algorithm instead of the for loop in the bucket-sort algorithm that calls it. Therefore, right before calling quick-sort we used the directives *pragma omp parallel* and *pragma omp single nowait*, inside the quick-sort algorithm when it calls itself (recursion) we added another two OpenMP directives (both were *pragma omp task*) in order to create tasks to execute the recursion in parallel, one for each recursive call. Unfortunately, the execution times were worse than the one mentioned before.

VI. Test Scenarios and Obtained Results

In order to test our algorithm's performance we used various test scenarios that, consist in the combination of three array sizes (N=10000000 (ten million), N=10000000 (hundred million) and N=1000000000 (one billion)) and three different bucket sizes (M=10, M=100 and M=1000). We chose big array sizes because for smaller arrays the results were almost instantaneous and could not be properly tested.

We ran the test sequentially (represented by T=1) and in parallel for a large range of threads (T=2, T=4, T=8, T=16, T=32 and T=64).

For each test represented in the following graphics, we tested the combination of one array size with all the bucket sizes and ran the test once for each Thread.

On the first tests, we ran the algorithm for an array size of 10000000 (ten million):

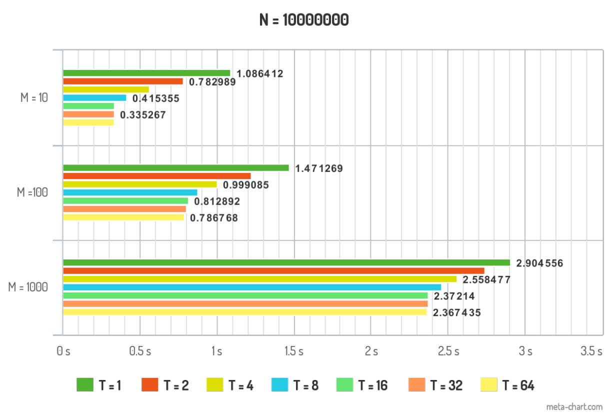

Fig. 1. Results for N=10000000 (ten million)

As we can see from the chart above, the bucket number that delivers the best execution times for this array size is 10. Also, we can see a big run time improvement until the number of threads is 16. For 32 and 64 Threads, the results are almost the same.

After that, we ran the test for 100000000 (hundred million) elements:

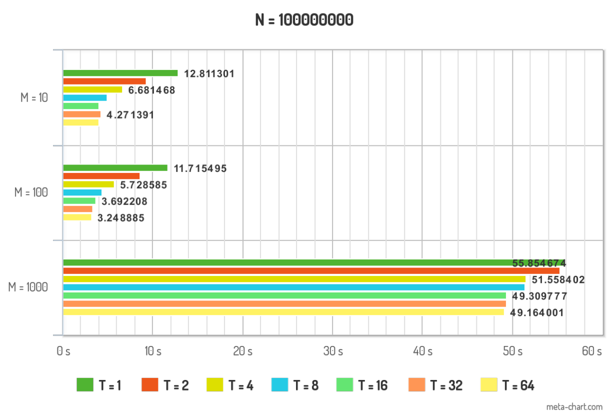

Fig. 2. Results for N=100000000 (hundred million)

In this test, we noticed that the execution time between 10 and 100 buckets is relatively closer than in the previous test. On the other hand, the run time for 100 buckets is much higher.

Looking at the results relative to the number of threads, we can again see a big improvement in paralyzing for threads 2, 4, 8 and 16. As for 32 and 64 threads, results show no significant improvement (100 and 1000 buckets) or even slower run times (10 buckets).

Lastly, we ran the test for 1000000000 (one billion) array elements:

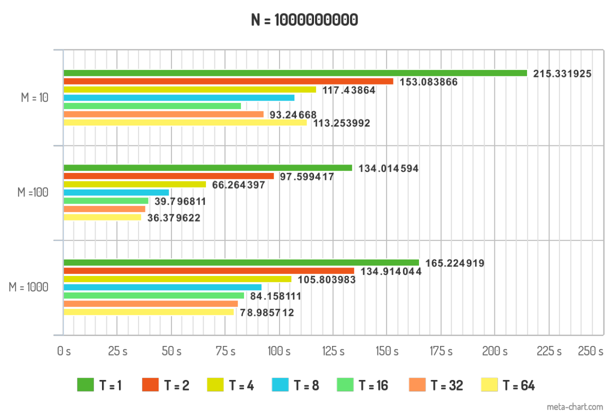

Fig. 3. Results for N=1000000000 (one billion)

In this last test, we can notice a convergence of execution times for 100 buckets, because for 10 and 1000 buckets the results are significantly higher. Also, we can notice that for an array size of 1000000000 (one billion) 10 buckets is no longer viable.

We can also see the pattern observed in the previous charts, in which more than 16 threads doesn't result in a big improvement, in terms of run time. Furthermore, for 10 buckets, the execution time for 32 and 64 threads is much worse than 16 threads.

VII. Conclusion

In this project we ought to develop sequential and parallelized versions of the bucket-sort algorithm in order to consolidate the learning, theoretical and practical, of the concepts of parallel programming in shared memory using the C language and the library OpenMP, which were taught during this semester's lessons. The project was split into 4 main phases: analysing and development of a sequential version of the bucket-sort algorithm; optimize the algorithm by implementing a parallel version of the algorithm, thus enhancing the performance of the algorithm; Studying and comparing both implementations of the algorithm (sequential and parallel); Explaining obtained results.

As a group, considering the overall perception of the work done, we achieved the goals proposed for by the professors. By development of the solutions and testing of the same we were able to face some conclusions. Firstly, we noticed that multiple factors should be taken into consideration when designing/developing a solution for the given problem. The analysis of which parts of the code could be modified in order to enhance the performance. The type of hardware used also has its own influence in the outcome of the results, since the limitations of cores stabilizes the speed up (comparing the parallel and the sequential versions). As for the use of threads we concluded that the more threads used does not mean that it is better, once communication and synchronization overhead between threads should be avoided.

On the other hand we also faced some difficulties, despite having reached a conclusion about the L3 cache misses we could not measure it (test) to justify ourselves.

Nevertheless, we consider that despite some of the adversities encountered, the project was successfully concluded once we met the goals set by the professors in terms of studying and evaluating parallelization, explore the different directives offered by the OpenMP library in order to optimize the bucket-sort performance and to apply such concepts (theoretically and practically) which were taught in the Parallel Computinh course.