Parallel Computing Practical Work Report

1st Luís Sousa
Departamento de Informática (of Aff.)
Universidade do Minho (of Aff.)
Braga, Portugal
a89597@alunos.uminho.pt

2nd Pedro Barbosa
Departamento de Informática (of Aff.)
Universidade do Minho (of Aff.)
Braga, Portugal
pg47577@alunos.uminho.pt

***Abstract*—This report serves to document the different phases of development of the practical work of the curricular unit of Parallel Computing. It begins with the study of the sequential version of the algorithm *Bucket-Sort* in the C programming language. It procedes with the analysis of the parallelized version of the same algorithm, through the use of *OpenMP*. Next, the study of its implementation on two of the nodes of *cluster SeARCH* is performed. It ends with an explanation of the results obtained using appropriate metrics.**

***Index Terms*—Parallel Computing, Algorithm, *Bucket-Sort*, *OpenMP*, *Cluster SeARCH*, Metrics, Parallel Programming, Shared Memory.**

I. Introduction

The purpose of this practical work is to evaluate the learning of parallel programming in shared memory using the imperative programming language, *C*, and also *OpenMP*.

In this way, the group was asked, using the knowledge acquired in the theoretical and theoretical-practical classes, to analyze the implementation of the parallel version of the *Bucket-Sort* algorithm. This should be done as efficiently as possible, always aiming to minimize the total execution time of the algorithm.

II. Study of the Sequential Algorithm of Bucket-Sort

A. Algorithm's Implementatiomn

Initially, we started by implementing the *Bucket-Sort* algorithm. As requested in the document of the practical project, this algorithm was divided into four distinct implementation phases, these being:

- Initialize a vector (array) of numbers.
- Put each number in a bucket.
- Sort each bucket.
- Put the numbers in the final vector (array).

Thus, the implemented sequential algorithm works as follows. First an array of random numbers is initialized, being each number obtained by dividing the function *rand* by the total number of elements of the array. Next, the range of numbers that each existing bucket will contain is calculated by dividing the total number of elements of the array by the total number of buckets. Using the chosen sorting algorithm, *Insertion Sort*, the numbers in each bucket are sorted. Finally, each bucket is traversed and the sorted numbers are placed in the final array.

B. Optimization Impact Analysis

One possible optimization could be the use of arrays instead of linked lists. This way we could still allocate memory at runtime if they were dynamic, we could access data directly and we would have a better cache locality. On the other hand, the *insertion* function is more effective with linked lists and in terms of allocating space they can be proportional to the space needed while in arrays we end up allocating more space than we need, in case they are full.

Other optimization we noticed as we were studying the sequential version of the algorithm is that when the number of array elements is equal or close to the total number of buckets it may happen that memory space is allocated to a bucket that is empty. A possible optimization, could be to free up the memory space used by this bucket in order to increase the computation efficiency. An example of this is as follows:

```
Initial array: 3 6 7 5 3 5 6 2 9 1
-------------
Bucket[0]:
Bucket[1]: 1
Bucket[2]: 2
Bucket[3]: 3 3
Bucket[4]:
Bucket[5]: 5 5
Bucket[6]: 6 6
Bucket[7]: 7
Bucket[8]:
Bucket[9]: 9
-------------
Buckets after sorting
Bucket[0]:
Bucket[1]: 1
Bucket[2]: 2
Bucket[3]: 3 3
Bucket[4]:
Bucket[5]: 5 5
Bucket[6]: 6 6
Bucket[7]: 7
Bucket[8]:
Bucket[9]: 9
-------------
Sorted array: 1 2 3 3 5 5 6 6 7 9
```

Fig. 1. Empty Buckets

Last but not least, we could have used a sorting algorithm other than *insertionSort*.By far, this algorithm is not the best in terms of time complexity and in the test section you will be able to analyze in more detail the time disparity found with this algorithm when increasing the data dimensions. Therefore, the use of a sorting algorithm such as a *quickSort*, a *radixSort*

or even recursively with a *bucketSort* would increase the efficiency and optimize the *bucketSort* algorithm.

III. Study of the Parallel Version of the Bucket-Sort Algorithm

In this phase of the project, the main goal is, through the use of *OpenMP*, to parallelize the sequential version of the bucketSort algorithm in order to work with shared memory.

The *OpenMP* OpenMP is a shared-memory application programming interface (API) whose features are based on prior efforts to facilitates shared-memory parallel programming . It consists in the implementation of multithreading, a method of parallelization that is used so that certain parts of code are executed by several threads simultaneously (concurrently), and these are allocated to different processors. This way, the algorithm will be improved in terms of performance, efficiency and total execution time while maintaining its original goal.

Next, we will procede to present and explain all the decisions made, possible mistakes and flaws in them, and other possible choices that could have been made.

We first started by analyzing the sequential version of the algorithm and trying to figure out which sections of it could be parallelized in order to improve performance and efficiency. In an overview of it, we quickly realized that almost all the instructions that were to be executed were done inside for-cycles and thus could be parallelized via a *pragma omp parallel for*.

Consequently, our approach was to test our sequential version and its improve by trying to put the above pragma into one for-cycle at a time. The first for-cycle aims to initialize all buckets allocated to the algorithm to *NULL*. By parallelizing this section of code we actually improved the overall performance of the algorithm significantly. In addition, we tried to figure out if we could use other pragmas in this section of the code but after some research we realized that we couldn't since it is a sequence of instructions that guarantees atomic accesses and updates of shared single word variables, in other words, an atomic operation.

Next, we move on to the second for-cycle. Here, the buckets are filled with their respective elements from the original array. When we tried to apply the same technique as in the first for-cycle, that is, to parallelize with a *pragma omp parallel for*, in fact the performance increased but at the end of the program the array did not appear all sorted, and the algorithm could not achieve the intended purpose. We also tried to apply *schedules* to this section of the code, either *static*, *dynamic* or *guided* in order to divide the cycle, its iterations and the execution of instructions by several *threads* but, again, without success. After further research and analysis we realized that we could not parallelize this section because of the *synchronizing overhead* that happens everytime one task spends time waiting for another.

The third for-cycle is responsible for applying the chosen sorting algorithm which, in our case, was the *InsertionSort*. We reapplied the technique used for both previous for-cycles and again obtained an improvement in performance and the purpose of the algorithm, to sort the elements of an array through bucketSort, was successfully achieved this time. In order to try to improve the algorithm further we analyzed our *insertionSort* to see if we could parallelize it but we then realized that this sorting algorithm is not parallelizable.

Finally, we analyze the last for-cycle that iterates through the different buckets and puts their numbers, already sorted, into the final array and conclude that it would not be parallelizable for the exact same reasons mentioned above when studying the parallelization of the second for-cycle.

Having finished this step concerning the study and parallelization of the sequential version of the bucketSort algorithm we move on to the testing phase where we run the algorithm on different machines, subject it to different test scenarios, and analyze and explain the results obtained.

IV. Tests and Results

A. Resources

The resources used to perform the tests made by the group were two nodes of the cluster provided by the teachers of the curricular unit, the node compute-134-115 and the node compute 145-1, and the latter was used to perform tests on a 16 cores machine but we could only get times from it and not run the algorithm with the *PAPI* instructions. Thus, we performed most of the work on compute node 134-115, having only used the other machine to see if there were noticeable improvements from using 12 to 16 cores.

B. Time - Sequential Version vs Parallelized Version

The first metric used to study scalability for both the sequentialized version and the parallelized version was time. This metric was tested by using the *PAPI*'s instruction *omp_get_wtime()*. It was analyzed on both a 12-Core Machine and a 16-Core Machine for the Sequential version and for the Parallelized version with 2,4,8 and 16 threads. In terms of array sizes and number of buckets the following were used and compared, respectively:

- 500000 array size 1000 buckets
- 500000 array size 10000 buckets
- 1000000 array size 1000 buckets
- 1000000 array size 10000 buckets
- 10000000 array size and 10000 buckets
- 10000000 array size and 100000 buckets

The results were obtained in the following graphs:

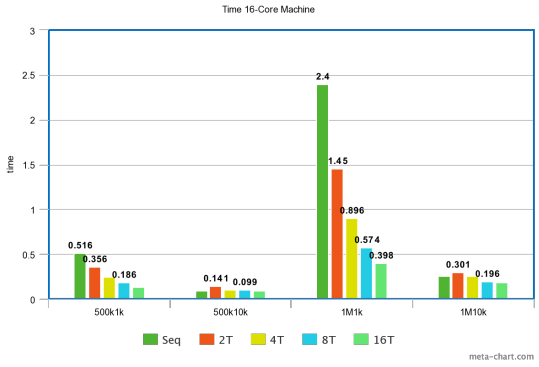

Fig. 2. Time 12-Core Machine

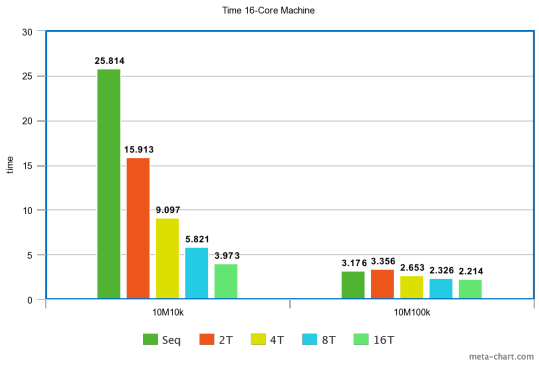

Fig. 3. Time 12-Core Machine

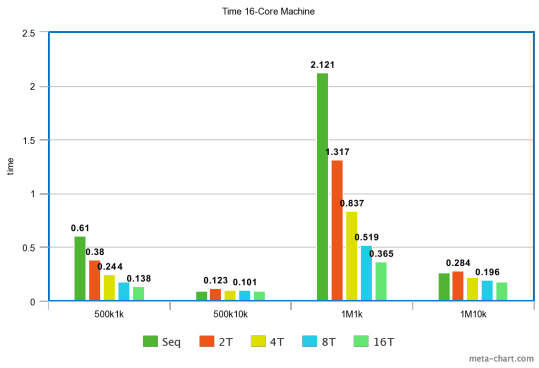

Fig. 4. Time 16-Core Machine

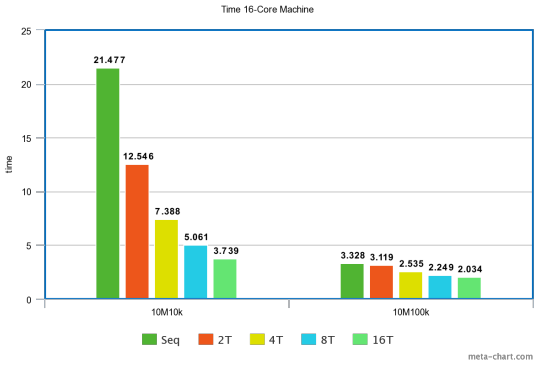

Fig. 5. Time 16-Core Machine

As we can compare both the graphs of the 12-Core Machine and the graphs of the 16-Core Machine, the improvement in the execution time of the algorithm from the sequential version to the parallelized versions is noticeable. Within each parallelized version, however, this improvement is already more subtle and the gain between these versions is increasingly smaller as we increase the number of threads used. When we came across this conclusion, we also realized that using a number of threads larger than 16 would be useless since the gain would be almost unnoticeable and would eventually start to diminish. Hyperthreading would eventually start to exist when the number of threads is greater than the number of physical cores on the machine.

In order to compare the difference in terms of execution between the use of a 12-Core Machine and a 16-Core Machine, we also made the following chart that makes this comparison in the scenario where we have 10000000 array size and 1000 buckets that will store them. The graph is presented below:

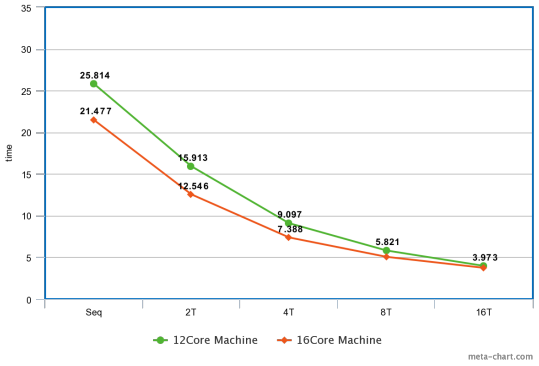

Fig. 6. Time - 16-Core Machine vs 12-Core Machine

We can thus conclude that, although not extreme, there is some difference between using both machines in terms of the algorithm's execution time. It is also possible to conclude that this difference in gain is increasingly smaller as the number of threads in the parallelized versions of the algorithm increases.

C. L1_DCM, L2_DCM and CPI metrics

In this topic we will cover the study done for cache misses, both for L1 level cache and L2 level cache, and also the differences at the CPI level for the various test scenarios. Again, for both the sequential version of the algorithm and the parallelized versions of it with 2,4,8 and 16 threads, the same test scenarios were used as for the study of the algorithm's execution time, and they were:

- 500000 array size 1000 buckets
- 500000 array size 10000 buckets
- 1000000 array size 1000 buckets
- 1000000 array size 10000 buckets
- 10000000 array size and 10000 buckets
- 10000000 array size and 100000 buckets

Therefore, we put together the following table with all the results for the different test scenarios and for the metrics mentioned above:

Array Size	Buckets	Metric	Sequential	2 Threads	4 Threads	8 Threads	16 Threads
500k	1k	L1_DCM	12 392 302	6 721 409	3 868 842	2 396 903	1 647 935
		L2_DCM	1 077 915	828 871	697 263	622 931	585 941
		CPI	2.3	2.08	1.74	1.48	1.39
500k	10k	L1_DCM	2 166 032	1 743 054	1 529 403	1 417 201	1 349 935
		L2_DCM	1 172 048	883 773	734 073	657 760	606 905
		CPI	0.92	1.18	1.24	1.22	1.18
1M	1k	L1_DCM	213 229 839	108 251 667	55 219 234	28 500 746	15 449 337
		L2_DCM	2 858 752	2 131 832	1 664 934	1 440 206	1 319 580
		CPI	3.77	3.47	2.94	2.46	1.9
1M	10k	L1_DCM	4 708 128	3 777 257	3 302 018	3 070 984	2 950 675
		L2_DCM	2 454 385	1 852 955	1 553 254	1 376 616	1 286 420
		CPI	1.21	1.33	1.34	1.19	1.15
10M	10k	L1_DCM	2 490 154 811	1 265 647 145	646 754 435	338 962 929	184 451 424
		L2_DCM	51 461 935	37 003 714	28 346 883	23 786 573	21 805 168
		CPI	4.42	4.2	3.75	3.34	2.66
10M	100k	L1_DCM	54 988 974	43 968 894	38 013 662	35 100 030	33 620 954
		L2_DCM	48 244 201	38 171 967	32 652 089	29 995 256	28 662 171
		CPI	2.02	2.11	2.01	2.02	1.97

Fig. 7. Table for L1_DCM, L2_DCM and CPI

Among all these metrics the difference between them is striking. The main target from which we can infer the differences is the bucket range, which in turn is calculated by dividing the array size by the total number of buckets. The larger it is, the more noticeable the difference and the better conclusions can be drawn.

Regarding the CPI metric, we can observe the impact that the parallelization of the algorithm had because the difference between the number of cycles per instruction from the sequential version to the parallelized versions of the algorithm is large. Comparing the different parallelized versions the CPI decrease gain is smaller and smaller as we increase the number of threads used in parallelizing the cycles. However, there is some disparity as we also see that when the interval is smaller the same is not true for some of the parallelized versions being that it has a lower CPI at times.

Regarding the cache misses at both L1 and L2 levels, once again we notice the difference between the sequential and parallelized versions, thus also showing the performance improvement of the algorithm through the use of OpenMP. This difference is again not so impactful when we analyze and compare the various sequential versions.

Through the study and analysis of the algorithm with these metrics it is evident the improved performance, efficiency and execution times of the parallelized algorithm compared to its sequential version.

D. L3 and RAM metrics

For the L3 and RAM metrics we were not able to test them because it was necessary to be able to use the *PAPI* instructions on the 16-Core Machine and we were unable to do so. Consequently we were not able to analyze our sorting algorithm with respect to these metrics.

V. Conclusion

Having concluded this project, the group feels that the main objectives proposed by the teaching team were achieved, having managed to develop a version of the *Bucket-Sort* algorithm, both sequential and parallel. We were also able to study the scalability of the implementation, analysing its results.

However, we were unable to analyse the L3 cache and RAM metrics, as explained above, which limited our analysis of the algorithm.

On a final note, the group would also like to have measured the time each thread takes to execute the parallelized part of the algorithm, allowing us to understand the existing balancing.