BUCKET SORTING ALGORITHM

Ana Filipa Pereira
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG46978

Carolina Santejo
Informatics Department
Master in Informatics Engineering
Univerity of Minho
Viana do Castelo, Portugal
PG47102

Raquel Costa
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG47600

***Abstract*— This report looks into comparing and presenting two versions of a sorting algorithm called bucket sort. These two versions consist in a sequential and parallel algorithm, and, in both cases, there will be a performance evaluation.**

Keywords*— *algorithm, sequential, parallel, optimization, bucket sort, OpenMP, performance, metrics, analysis.

I. INTRODUCTION

The aim of this project is to evaluate and explore shared memory parallelization through the usage of the C programming language and the OpenMP application programming interface. For that, it'll be studied the Bucket Sort algorithm, which is an example of a sorting algorithm, the main goal is to implement it in the most efficient and adequate way possible. Firstly, it's developed a sequential version of the algorithm, and it's implemented the due optimizations accordingly to the impacts that those have in the algorithm in study. However, to improve even more the execution time of the bucket sort, it shall be converted to a parallel algorithm and be implemented using OpenMP API, while preserving the integrity of it.

Lastly, through the analysis of some metrics like the speedup or run time execution, the final experiment results allow to evaluate the performance of the algorithm. Besides, to check the quality of the solution, it's studied its scalability through some evaluation tests using other machines or changing the data dimension.

This paper is divided into seven sections. Section 2 explains the bucket sort algorithm and its implementation. Section 3 and 4 introduces the sequential and parallel versions, respectively, and the optimizations and improvements that were made throughout the execution of the project. Section 5 provides which metrics were considered and why, and the obtained results for each version of the algorithm. Section 6 compares both versions' results and discusses them, considering the performance and efficiency of the algorithm. Finally, section 7 concludes the paper and provides some future work in consideration.

II. BUCKET SORT ALGORITHM

Bucket sort, also known as bin sort, is a sorting algorithm, which is commonly used in computer science. The underlying idea of this algorithm is to distribute the elements of an array into several groups called buckets. To do this, it is necessary to initialize a vector of buckets and then place each value of the array into its respective bucket. Afterwards, each bucket is sorted individually using a different sorting algorithm as for example quicksort. To conclude the process, all the elements in the sorted buckets are placed in a final sorted array.

Furthermore, is also important to highlight that initially, the number of elements that are going to be put in each bucket is unknown and, therefore, the size of memory that needs to be allocated for each bucket is also unknown. As a result of this, we need to be aware of the fact that if we allocate way too much memory, most of it could end up being unused but we don't allocate a reasonable amount of it, the overall performance could be impacted due to the fact that it becomes necessary to reallocate the buckets size very frequently. In the algorithms presented in this report, each bucket is going to have an initial size of half of the size of the input array.
In conclusion, this algorithm, when correctly implemented, can result in an increase of the overall performance of an application due to the fact that putting data into small buckets that can be sorted individually reduces the number of comparisons that need to be carried out.

III. SEQUENTIAL BUCKET SORT

After implementing the algorithm according to what was proposed in the previous section, the team tried to optimize the performance of the solution by using techniques that were analyzed in the theoretical and practical classes of this subject. In this section we will refer loop unrolling and vectorization, both being automatically done by gcc compiler when the respective optimization is activated. To activate loop unrolling the flag *-funroll-loops* needs to be added and vectorization is also enabled by using the *-O3* flag.

A. Complexity

TABLE I. COMPLEXITY OF THE SEQUENTIAL ALGORITHM

Complexity				
Time	***Space***	***Best Case***	***Worst Case***	***Average***
$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(\frac{n+n^2}{k+k})$*

** when k=O(n); Note: n is the num. of elements and k is the num. of buckets*

B. Optmization and Improvements

1) Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program by reducing the number of iterations. This consists in replicating the body of a loop a certain number of times and adjusting the loop control accordingly. One of the advantages of loop unrolling includes the reduction of the loop overhead.

2) Vector Instructions

Vectorization is a parallel computing method that compiles repetitive program instructions into a single vector (combination of multiple datasets), which is then executed simultaneously and maximizes computer speed. It's also an example of single instruction, multiple data (SIMD) processing.

IV. Parallel Bucket Sort

A. Required Modifications

After implementing the sequential algorithm, the team had to implement its parallelized version. In this development phase, in addition to applying the parallelization clauses provided by the OpenMP API, it was necessary to make some changes to the sequential code since there were some parts that were not possible to parallelize.
The most important modification required in the bucket sort function was applied to the fraction of the code where, for each element of the unsorted array calculates the bucket index where it will be inserted and inserts into that bucket array.

Once we are iterating through the different elements of the unsorted array, if we parallelized it without any modifications, in case two threads are reading an element that must be inserted in the same bucket, there could be a conflict between both writing at the same time to the same destination bucket array. Therefore, in order to allow parallelization in this fraction of the code, we added a new loop that for each bucket iterates through all elements of the unsorted array and only inserts the ones that match the current bucket.

Furthermore, this loop will also include the *qsort* function that will sort each bucket array after being filled.

B. Compromises of the parallelism

As we discussed in the previous topic, in order to parallelize the sequential code, we had to make some changes. Even though these modifications allowed us to parallelize some instructions, they made the algorithm less efficient, so we had to consider some compromises between performance and parallelism ability.

While in the sequential version we only iterate through the elements of the unsorted array one time, in the parallel version it will iterate as many times as the number of buckets. Therefore, considering *n* the size of unsorted array and *n_buckets* the total number of buckets, the total number of iterations will increase from *n* (sequential) to *n*n_buckets* (parallelized).

Nevertheless, using parallelism will compensate this performance drop once it will, not only fill multiple buckets, but also sort each one at the same time, as opposed to the sequential version.

C. Optimization with OpenMP

In order to implement parallelism in our bucket sort algorithm, the team had to use some clauses from the OpenMP API.

Therefore, in this bucket sort algorithm were used 4 different clauses:

• ***#pragma omp parallel num_threads(num_threads):*** used to delimit a parallel zone and set number of threads to use within. This clause was used so that we could run the algorithm with different number of threads.

• ***#pragma omp single:*** used to delimit a zone where only one thread is allowed to run.

• ***#pragma omp barrier:*** used to define a synchronization point at which threads in a parallel region will wait until all other threads in that section reach the same point.

• ***#pragma omp for schedule(dynamic):*** used to distribute loop iterations within the group of threads. In each thread it will be dynamically assigned several iterations on a "first-come, first-do" basis until all work has been assigned. We considered the use of this clause because, as opposed to other clauses, ensures the work is always distributed to some thread, which means a better performance.

V. Exploration and Evaluation of the Solution

This topic analyses and explores the impact of different testing variables on both sequential and parallel implemented algorithms. In order to perform this evaluation and compare both versions of bucket sort, it was used the Performance Application Programming Interface, PAPI, which is a library that provides a consistent interface (and methodology) for hardware performance counters, found across the system.

A. Testing variables

For the purpose of evaluating both sequential and parallel versions of the sorting algorithm, it was taken into account that the variation of different elements would lead to distinct behaviors in the performance of the implemented solutions. Therefore, the underlying testing variables associated with the bucket sort algorithm are the input size of the array, the number of buckets and the number of threads used.

In addition, there's also available a cluster with multiple nodes, each having different hardware specifications including the number of CPUs, the number of cores per CPU and the frequency in GHz of the processors. However, all processors of the cluster are provided by Intel. Therefore, the node used to execute the implemented solutions is also a testing variable.

TABLE II. Hardware Specifications of each Cluster Node

Cluster Node	**Hardware Specifications**			
	Model name @ clock rate	***Num CPU's***	***Num Cores p/CPU***	***Num Threads p/CPU***
113-2	Intel® Xeon® CPU E5520 @ 2.27 GHz	16	4	8
134-18	Intel® Xeon® CPU E5-2650 v2 @ 2.60 GHz	32	8	16
134-115	Intel® Xeon® CPU E5-2695 v2 @ 2.40 GHz	48	12	24

B. Evaluation Metrics

For the different experiment tests that were made, it was considered an appropriate group of evaluation metrics, so that in the end, it would be possible to conclude if the algorithm's implementation was efficient or not.

Firstly, the execution time is one of the most important metrics since it gives a practical demonstration of the algorithm's behavior. Consequentially, the speedup is also another very important metric in the experiment tests. Speedup achieved by a parallel algorithm is defined as the ratio of the time required by the sequential algorithm to solve a problem to the time required by the parallel algorithm to solve the same problem.

Besides that, cache misses in the hierarchy levels L1 and L2, are also used to evaluate the algorithm's performance.

C. Obtained Results

1) Sequential Algorithm Results

Initially, the experiment tests were centered on the sequential version. Here it's evaluated the impact of the diverse optimizations that were made to the algorithm.
In the following tables, it's discussed the wall clock time and the cache misses in order to evaluate the impact of the methodologies previously described. That being said, the team started by defining an array of size 500000. Afterwards, the normal and the improved versions of the sequential bucket sort algorithm were compared by varying the number of buckets.

Firstly, it was used 3000 buckets and we concluded that applying vectorization or loop unrolling individually wouldn't significantly improve the overall performance of the code. However, when these two methods were applied simultaneously the performance did end up slightly increasing, for example, the wall clock time has been notably reduced, just as the number of cycles and number of instructions. However, using loop unroll and vectorization culminated in a rise of L2 data cache misses.

Eventually, the number of buckets was gradually decreased until 30 and it was observed the same behavior as before, although the gain in performance was progressively less significant as the number of buckets was being reduced.

According to what was described, it is possible to conclude that performing vectorization and loop unroll on the sequential solution is not worthy, not only because it doesn't bring remarkably preferable results but also due to the fact that it caused more L2 cache misses.

TABLE III. WALL CLOCK TIME (USECS) - SEQUENTIAL

Num. Buckets	**Flags**	
	no loop-unrolling and no vectorization (-O2)	***loop-unrolling and vectorization (-O3)***
30	18763	17786
300	19057	17477
3000	19107	17841

Note: Wall clock time in the sequential version when the size of the input array is 500000 elems;

TABLE IV. CACHE MISSES - SEQUENTIAL

Num. Buckets	**Flags**			
	no loop-unrolling and no vectorization (-O2)		***loop-unrolling and vectorization (-O3)***	
	L1	***L2***	***L1***	***L2***
30	300051	13202	299654	69937
300	221291	24023	222195	80577
3000	219442	39749	216243	96252

Note: Cache misses in the sequential version when the size of the input array is 500000 elems;

2) Parallelized Algorithm Results

Now, it will be analyzed the impact of the several metrics in the parallel version of the bucket sort algorithm.

TABLE V. WALL CLOCK TIME (USECS) IN EACH NODE WITH DIFFERENT ARRAY SIZES

Array Size	**Cluster Nodes**		
	113-2 (cores = 4)	***134-18 (cores = 8)***	***134-115 (cores = 12)***
32KB (≈8000 int)	4820	2288	2055
256KB (≈64000 int)	25422	12142	8967
25600KB (≈6400000 int)	2455759	906104	663990

Note: Wall Clock Time in each node for: 500 buckets

In the previous table, the array size was estimated accordingly to the size of the different cache levels (L1, L2, L3), keeping in mind that an integer's size is 4 bytes. Besides that, it was also considered distinct cluster nodes in order to analyze their impact in the wall clock time. As the table demonstrates, it's possible to conclude that a cluster node with better hardware specifications, per example, a greater number of cores, results in better outcomes as expected. It is important to highlight that in the node "113-2", it was used 8 threads, in the "134-18" 16 threads, and finally, in the "134-115" 24 threads.

TABLE VI. WALL CLOCK TIME (USECS) FOR (N) THREADS AND ($N \times 2^n$) BUCKETS

Num. Threads	**Num. Buckets**			
	= Num. Threads	***= Num. Threads * 2***	***= Num. Threads * 8***	***= Num. Threads * 16***
4	195420	229720	456432	798054
8	132107	194946	468275	878484
16	112973	177170	454222	874076
24	129018	198018	585309	1145959

Note: Wall clock time for correlated values between the number of threads and the number of buckets; Values considered: Array size $= 10 \times 10^6$ *; The cluster node used is "compute-134-115".*

In these experiment tests, it's analyzed the correlation between the number of threads and number of buckets, and the resulting wall clock time. Accordingly, to the final gotten results, it's possible to conclude that the algorithm has its best performance when the number of threads equals the number

of buckets. This happens since the number of buckets is larger than the number of available threads which implies that one thread will not only sort one bucket but several. The OMP's parallel *for loop* section assigns multiple buckets per thread, that is, the workload is distributed between the threads. The fact that the dynamic clause is used over the static one, allows a more efficient way of distributing workload because each thread will be assigned a new job as soon as it becomes free. Consequently, it's possible to take advantage of the parallelism, and make better use of each thread's execution life cycle.

TABLE VII. WALL CLOCK TIME (USECS) WITH DIFFERENT ARRAY SIZES AND NUM. THREADS

Array Size	**Num. Threads**			
	4	***8***	***16***	***24***
32KB (≈8000 int)	7552	3799	2133	1317
256KB (≈64000 int)	43680	24230	12566	9663
25600KB (≈6400000 int)	3546812	1993752	1046438	702304

Note: Wall Clock Time. Values considered: Buckets = 500. The cluster node used is "compute-134-115".

In this experiment it is possible to confirm that increasing the number of available threads will translate into an improvement of the wall clock time, because there are more available threads to distribute the workload. Besides this, as expected, when the array size grows, the time tends to grow as well.

TABLE VIII. CACHE MISSES WITH DIFFERENT ARRAY SIZES AND NUM. THREADS

Array Size (KB)	**Num. Threads**							
	4		***8***		***16***		***24***	
	L1	***L2***	***L1***	***L2***	***L1***	***L2***	***L1***	***L2***
32	27390	3843	19841	3845	13694	4019	10297	4148
256	519704	37981	273165	24443	147853	19638	113195	18913
25600	503335	36593	26699836	2414165	14161292	1881897	6961697	1481831

Note: Cache misses; Array sizes used: 32KB (≈8000 int), 256KB (≈64000 int), 25600KB (≈6400000 int); Values considered: Buckets = 500. The cluster node used is "compute-134-115".

Here we tested the impact of 3 different array sizes. The first one fits in the L1 cache, the second array fits in the L2 but not L1, and the last one only fits in the L3 cache.

In the first array and using multiple numbers of available threads, was noticeable that in all cases the L2 cache misses tended to the same value. This happens because an array with approximately 32 KB fits entirely in the L1 cache, therefore, there's no need to resort to the L2 storage capacity.

In the second array the number of L1 cache misses increases compared to the previous array L1 cache misses. This happens because part of the array needs to be stored in L2 cache.

Finally, in the last array, the L1 and L2 cache misses tend to stay the same just like the second array because the L1 and L2 cache will be fully occupied.

TABLE IX. WALL CLOCK TIME (USECS) AND SPEEDUP IN BOTH VERSIONS

Num. Threads/ Num. Buckets	***Time***	***Speedup***
Sequencial (4 buckets)	278471	1
2	215646	1,29
4	132039	2,11
8	92241	3,02
16	70869	3,93
24	70154	3,97

Note: Wall Clock Time and Speedup for both algorithm versions. Values considered: Array size = 6.4×10^6 elements; The cluster node used is "compute-134-115".

In this table, the sequential version of the algorithm is being compared to the parallel version with variable number of available threads. This comparison is based on two metrics: the speedup and the wall clock time. It is possible to verify that the parallel algorithm has an overall better performance that tends to gradually increase with the number of threads. It's important to highlight that in this experiment, the number of buckets it's the same one as the number of threads. Besides that, when the number of threads is greater than 16, both speedup and execution tend to stabilize. This may happen because the specification of the hardware that is being utilized has a limitation when it comes to the number of available threads that can be used.

VI. FINAL ANALYSIS

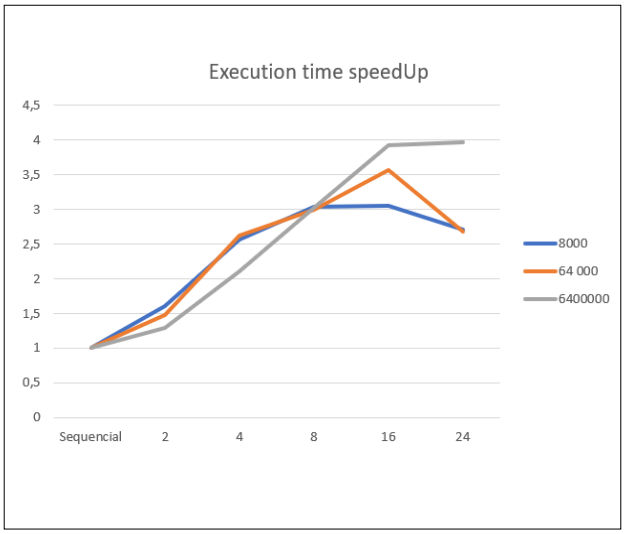

Fig. 1. *Execution time SpeedUp with num.threads = num.buckets related to the sequential version with 4 buckets*

All in all, it was possible to verify that the parallel version of bucket sort has an overall better performance, especially when the number of buckets equals the number of threads. Besides that, it is also a scalable solution. Firstly, it was created a graphic that allows us to understand the scalability of the parallel bucket sorting algorithm implemented. Now we can verify that the speedup of the algorithm tends to

significantly increase with the number of threads. That said, the speedup scales well, because there are no significant sequential elements in our parallel system. These are the attributes that make the speedup scalability possible.

VII. Conclusion and Future Work

With this practical assignment the team was able to better consolidate the knowledge acquired during theoretical and practical classes regarding not only the impact of optimizations such as loop unroll and vectorization but also the impact of using parallel programming in a shared memory environment using C and *OpenMp.*

Given that all the requirements demanded by professors were addressed in this report, the team considers that this practical assignment was successfully completed.

However, it is relevant to emphasize that some improvements or updates could be applied in the future, as for example, changing the bucket sort algorithm to make it even more parallelizable or even executing more tests using different metrics. Furthermore, using *OpenMpi* and comparing the performance gains with the ones obtained with *OpenMp* would also be an interesting approach.