Bucket-Sort Algorithm Implementation

1st Ariana Lousada
PG47034
Mestrado em Engenharia Informática
Universidade do Minho
Braga

2nd Marco Pereira
PG47449
Mestrado em Engenharia Informática
Universidade do Minho
Braga

***Abstract*—This document describes all the decisions made by the working team concerning the project proposed by the professor team in the Parallel Computing course unit. The reasoning applied to the solution will be exposed throughout this report.**

***Index Terms*—parallel, computing, algorithm, programming, bucket-sort**

I. Introduction

One of the primary objectives in programming is to obtain a product that not only does what is expected from it but also a code that can easily be parallelized in order to execute faster when given the resources. The most important aspect for that to be doable is the smart projection of how the code could be run in multiple threads at the same time.

This project's objective revolves around the development of a bucket-sort algorithm parallel version for integer arrays sorting in the C programming language. The implementation of a parallel version of this algorithm requires, for comparison, the development of a sequential version of it. Throughout this report the thought process of the group will be exposed along with the many challenges present in the implementation of a version that while running parallel can guarantee that no data races happen.

In the initial phase of the project was developed a sequential version of the algorithm, this version is explained in section II. Only in the following stage was the parallel version developed using OpenMP, the parallel version can be found in section III.

The environment that was used for testing as well as the parameters that can be tweaked will be explained in section IV.

Both versions were compared in terms of scalability and performance with the administration of various types of experiments. The results and analysis from these experiments can be found in section VI.

II. Sequential Version

The sequential version of the bucket sort algorithm was implemented by separating all the algorithm's parts into different functions: first, the creation of the buckets - the buckets are initialised with partial size of the array's total size given, their allocated size being updated if needed. After that comes the sorting of each number into the corresponding bucket, we calculate the range of each bucket by dividing the range of the numbers in the array by the number of buckets, then we pass through each member of the original array and sort it onto its bucket. Finally, we iterate through each bucket calling a sorting function. The tests that will be presented used 3 different sorting functions, them being quick sort, merge sort and radix sort. After that, the only thing left is to iterate through each bucket and copy its elements to the output array while freeing the memory allocated to each bucket.

A. Optimisations and subsequent impact in efficiency

The sequential algorithm that was implemented can however be modified to be more efficient in terms of time and space used. The first optimisation that can be done would be allocating the exact needed space to each bucket, which would require knowing exactly how many elements would be put into it. It would be a trade off in terms of time and space used. The second alternative to make the algorithm better would be to change the sorting algorithm for each bucket between the 3 implementations added.

III. Parallel Version (with OpenMP)

The parallel version of this algorithm was obtained through the usage of OpenMP. For that some changes were made to phase 3 from the algorithm presented in the sequential version. In phase 3, the sorting of the values of each bucket, this phase was changed since we added ***#pragma omp parallel for*** for the parallelism to exist. We also added ***schedule(dynamic)*** in order to give each thread the same amount of work.

A. Parallelism Optimisation

The implemented parallel version could however be better if we decide to parallelize the insertion of each value into the bucket: this could be done if the accesses to the variable that is responsible to count each new number on a bucket was changed to avoid several writes and reads from happening at the same time by different threads. A version where CUDA is used could also be explored in hopes of obtaining better results.

IV. Testing environment

In order to better understand the conditions in which testing took place, we decided to insert in this section the parameters used during testing as well as the interfaces used to obtain the information that will be shown in the following parts of this report.

The nature of an algorithm such as the one implemented allows for a lot of different changes in parameters that one can control - hyper parameters:

- Thread number : The number of threads that will be used during the parallel execution of the algorithm.
- Bucket number : The number of buckets used for the sorting algorithm. Bigger number of buckets implies that a smaller number of values will be put into each bucket but more buckets will execute in parallel (limited by the number of threads supported by the node).
- Sorting size : This parameter defines the size of the array to be created and subsequently sorted.
- Algorithm used for sorting each bucket : The algorithm used to sort each bucket can be changed between 3 implemented algorithms: quick sort, merge sort and radix sort.
- Max Number in the array : The max number in the array should be a number bigger than the size of the array to better test the sorting speed, if the number is much smaller then a lot of ”ties” will happen which will cause the array to be faster than expected.
- Node and Partition : One of the most important parts of the testing environment is the machine used for the testing. The group used 3 different machines having decided to present only the results of 2 out of those 3. The machines characteristics will be explained in the next phase.

It must also be noted that all the code was compiled with the usage of O2 flags in order to obtain better performance at the cost of a small increment in compilation time. The measurements of time, instruction number and data cache misses were obtained using the PAPI library. This library was previously used in class and the same structure of testing was applied.

V. Scalability and Experiments

Two nodes were used in the experiments:

- Node 1 - Partition=cpar, Node=compute-134-101
- Node 2 - Partition=week, Node=compute-134-7

Two types of experiments were administered using the three different sorting algorithms previously mentioned, as well as the values of each nodes L2 and L3 memory: an evaluation in performance with a fixed number of buckets and a variation in the number of threads and an evaluation with a fixed number of threads and different number of buckets.

The best results of each combination are below presented in the form of tables comparing the values obtained with the sequential versions and with their parallel counter parts.

A. Node 1

The first node used for the experiments has the following characteristics:

- L2 cache total size: `256 KB`;
- L3 cache total size: `25600 KB`;
- Number of cores: `20`;

In the I table are exposed the results from an experiment in which the sequential and the parallel versions were compared with the optimal number of buckets and threads. In this node, both values are equal to 40. For the size of the array, the team used the maximum capacity of the nodes’ L2 cache. This value was obtained through the division of the cache size for the size of each integer in the C programming language(4 bytes).

TABLE I
Algorithm comparisons (size of array = 64000)

Algorithm	L1_DCM	L2_DCM	#I	Time(ms)
Merge Sort(Parallel)	25338	7182	8242756	2.467
Merge Sort(Seq)	29229	7335	15126294	4.443
Quick Sort(Parallel)	24862	6951	5841830	2.263
Quick Sort(Seq)	29286	7762	15021190	4.443
Radix Sort(Parallel)	25145	7189	6030484	2.612
Radix Sort(Seq)	29267	7403	14943479	4.443

In the II table are exposed the results from the same experiment, using the maximum capacity of the L3 cache as the size of the array.

TABLE II
Algorithm comparisons (size of array = 6400000)

Algorithm	L1_DCM	L2_DCM	#I	Time(ms)
Merge Sort(Parallel)	2809941	972414	647944812	182.365
Merge Sort(Seq)	6654228	1116241	2024209804	599.447
Quick Sort(Parallel)	2592463	888085	598094480	165.632
Quick Sort(Seq)	6650043	1119495	2024209864	599.208
Radix Sort(Parallel)	3000753	1107244	624254549	179.831
Radix Sort(Seq)	6650783	1121560	2024209801	599.035

B. Node 2

The second node used for the experiments has the following characteristics:

- L2 cache total size: `256 KB`;
- L3 cache total size: `20480 KB`;

- Number of cores: `16`;

In the III table are exposed the results of the same experiment mentioned in the previous section. In this particular case, the team used 32 buckets and 32 threads, which is the optimal value for both in this node. In this experiment the size of the array is the maximum capacity of the nodes' L2 cache.

TABLE III
ALGORITHM COMPARISONS (SIZE OF ARRAY = 64000)

Algorithm	L1_DCM	L2_DCM	#I	Time(ms)
Merge Sort(Parallel)	25108	3453	8385221	2.701
Merge Sort(Seq)	29217	6936	15330506	6.352
Quick Sort(Parallel)	25073	3279	8000011	2.723
Quick Sort(Seq)	29203	6845	15267925	5.183
Radix Sort(Parallel)	25166	3682	7803352	2.876
Radix Sort(Seq)	290666	6966	15330506	5.870

In the IV table are exposed the results from the same experiment, using the maximum capacity of the L3 cache as the size of the array.

TABLE IV
ALGORITHM COMPARISONS (SIZE OF ARRAY = 5120000)

Algorithm	L1_DCM	L2_DCM	#I	Time(ms)
Merge Sort(Parallel)	2300148	680839	539734465	198.038
Merge Sort(Seq)	5324679	850484	1627756683	439.055
Quick Sort(Parallel)	2104502	635974	490408225	189.178
Quick Sort(Seq)	5288239	862259	1617440266	425.178
Radix Sort(Parallel)	2483519	781982	515135313	218.693
Radix Sort(Seq)	5287548	856807	1620076729	426.569

VI. RESULTS ANALYSIS

The purpose of the current section is to try to explain the values obtained and shown in the V section. Some references in this section will refer to the appendix which can be seen in section A.

Due to the lack of a command that returned the size of each type of cache for each level, the size used for L2 and L3 is the unified size and not the data size. That means that there will be some recorded misses in data from cache L2 and L3 that shouldn't happen in big numbers when having the array occupying all the cache. Another problem that the group ran into was the missing support for the PAPI flag (L3_DCM), that allows the user to know how many data cache misses happened in L3. The group could however have used the L3_TCM flag that reports the total cache misses for that level but it would bring ambiguity when compared to the other level's misses, so that approach wasn't taken.

Through the tables presented in section V we can see that the parallel execution of any algorithm obtains an increase in speed that is sometimes close to triple the speed seen in the sequential execution. Through the analysis of the tables I and III we can observe that when filling L2 cache with integers the amount of misses at that level is very small - it can be accounted, as explained above, by the inability to find the amount of Level 2 Cache dedicated to Data.

When using a bigger amount of data like the tables II and III, the difference between the parallel and the sequential execution is even bigger, close to 2.25 times faster. We can also see that the quick sort algorithm is faster than the other 2 when used even with smaller amounts of data. Through the interpretation of the tables we can also assess that the radix sort algorithm becomes better than the merge sort algorithm when using bigger amounts of data. Furthermore, the execution of the sequential implementation has a tendency of making it so that the algorithm used didn't matter on the first node. On the second node however we see that the merge sort obtains better results when compared to the radix algorithm.

Along with the tables previously presented the group decided to create some graphics with the execution time obtained throughout several tests while varying the thread number and the bucket number.

In the contents of the appendix we can see that with small amounts of data (size L2) the variation of threads produces a positive result until a number equivalent to the number of threads supported. After that the execution time slowly gets worse but it's not very noticeable. The variation of the number of buckets produces and effect that is related to the one presented above, however it less apparent.

With bigger amounts of data the proposals shown above are seen to be true - the quick sort algorithm is still better than the other presented algorithms. The performance enhancement is seen when the number of buckets/threads are raised until they reach the number of threads supported by the node. This number can be calculated by multiplying the number of cores in the node by 2 which represents the maximum number of effective parallel threads thanks to the hyper threading.

When we look at the tables that relate to the second node where tests were held we can see that the overall performance is worse than the one seen on the first node. This is due to the lesser number of cores and the same size of L2 and almost the same L3 cache. The tests held on the second node are on the same page as the analysis made to the results obtained on the first node, further securing the groups position on the matter.

On hidden tests the group found out that the time each bucket took to be sorted was the same along buckets, assuming a uniform distribution of elements and the same amount of elements in each array. The raising of the number of buckets is expected to make the performance better until the moment in which the number of buckets surpasses the number of threads. From the tests made, that was verified to be true, which lead to the opinion of the group on the matter.

The group tested different types of scheduling and got results close to one another, which lead to the group idea that scaling didn't affect the performance of the algorithm. This can be due to the fact that each iteration of the cycle is the sorting of a bucket which makes it so that, since buckets take the same time to be sorted, the amount of iterations done by each thread is almost the same.

VII. Conclusions and Future Work

In this report we discussed the implementation of 2 different models for sorting an array being that the first one was sequential and the second was parallel. Each of this implementations had a big number of hyper parameters that can be changed and swapped to try and obtain the fastest time for sorting the array. Some examples of those hyper parameters are the machine where it's being executed, the sorting algorithm used in each bucket and the number of buckets for the algorithm to use.

A very large amount of tests was done on each combination to find out the best combination of these and get the fastest time possible. After that came the analysis of the results, which was done with the help of a Google Sheet to better present the important tests and help important features stand out. In the analysis many ideas were explored and discussed to try and explain the obtained results. The test results will all be attached along with the report.

As future work the group would like to explore some alternatives for optimisation of both implementations that were explained and discussed along the report. The group would also like to use better testing mechanisms to ease the testing - an example of this would be a script that contained every test instead of changing values on the script for each experiment.

Appendix

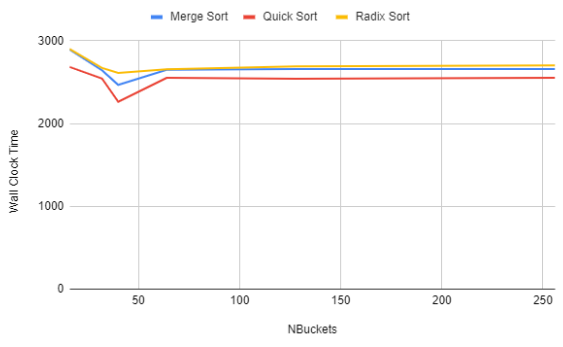

Fig. 1. Varying number of buckets experiment with fixed number of threads (40) and three sorting algorithms - Node 1, Size of the array: L2 capacity (64 000)

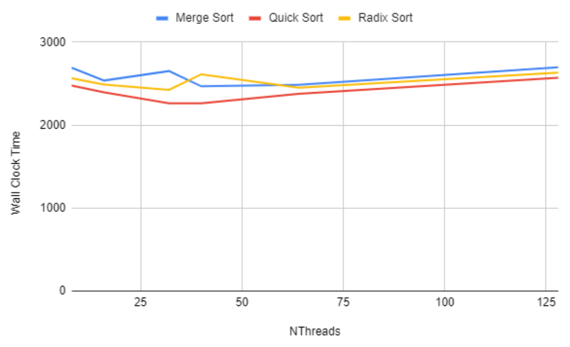

Fig. 2. Varying number of threads experiment with fixed number of buckets (40) and three sorting algorithms - Node 1, Size of the array: L2 capacity (64 000)

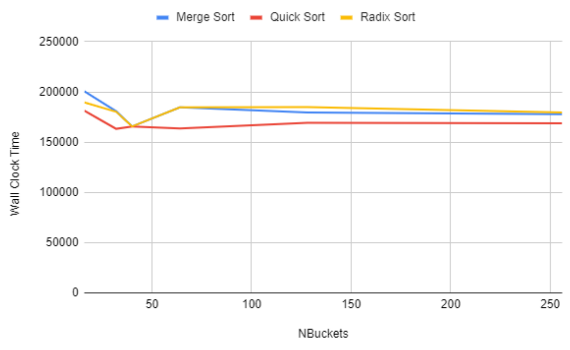

Fig. 3. Varying number of buckets experiment with fixed number of threads (40) and three sorting algorithms - Node 1, Size of the array: L3 capacity (6 400 000)

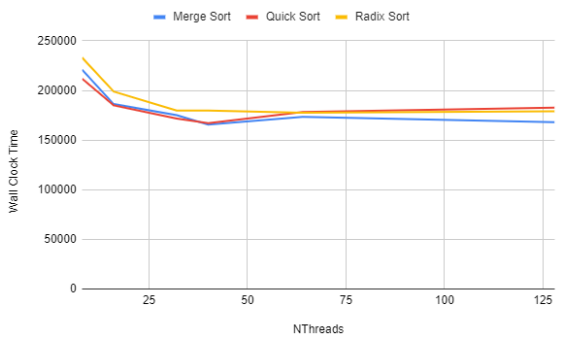

Fig. 4. Varying number of threads experiment with fixed number of buckets (40) and three sorting algorithms - Node 1, Size of the array: L3 capacity (6 400 000)

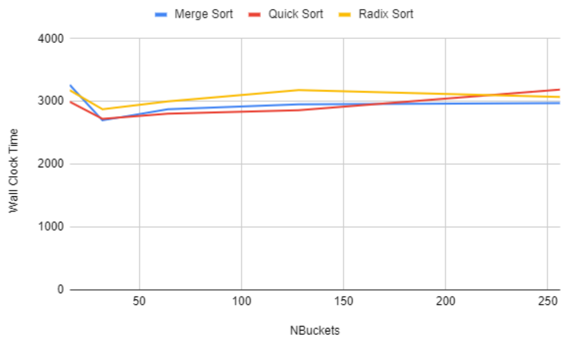

Fig. 5. Varying number of buckets experiment with fixed number of threads (32) and three sorting algorithms - Node 2, Size of the array: L2 capacity (64 000)

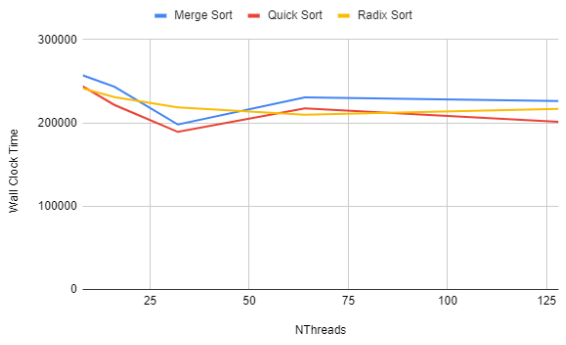

Fig. 8. Varying number of threads experiment with fixed number of buckets (32) and three sorting algorithms - Node 2, Size of the array: L3 capacity (5 120 000)

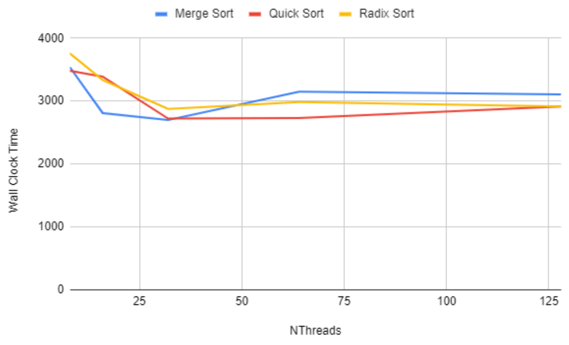

Fig. 6. Varying number of threads experiment with fixed number of buckets (32) and three sorting algorithms - Node 2, Size of the array: L2 capacity (64 000)

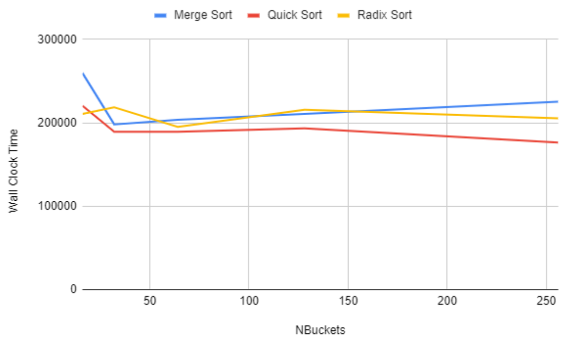

Fig. 7. Varying number of buckets experiment with fixed number of threads (32) and three sorting algorithms - Node 2, Size of the array: L3 capacity (5 120 000)