# PARALLEL BUCKET SORTING ALGORITHM

Parallel Computing's Pratical Assessment

André Martins
*Informatics Department*
*University of Minho*
Braga, Portugal
PG47009@alunos.di.uminho.pt

José Ferreira
*Informatics Department*
*University of Minho*
Braga, Portugal
PG47375@alunos.di.uminho.pt

***Abstract*—This document contains all the development, sequential and parallel, of the bucking sorting algorithm, as well as all the analysis, results and conclusions obtained during the realization of this pratical assessment. All the devolpment process and results is explained and fully justificated, making all the execution times and results fully understandable.**

***Index Terms*—Bucket sorting, Parallelism, OpenMP, PAPI, Multi-threading, Intel Xeon**

## I. Introduction

The goal of this assessment is to develop, test and analyse a parallel verison of the bucker-sort algorithm. Aiming for the lowest execution time possible, keeping the base of the algorithm, the solution presented should be able to run the bucket-sort algorithm with parallelism.
This assessment was split in four different stages:

- Develop the sequential implementation of the bucker-sort algorithm, mentioning possible optimizations to the sequential code
- Develop the parallel version using OpenMP.
- Study the scalability of the implementation.
- Results obtained and justifications

## II. Sequential Version of the Algorithm

Using bucket sort, the input set can be sorted in expected time O(n). Since bucket sort is a completely deterministic algorithm, the expectation is over the choice of the random input.

Bucket sort works in three different phases. In the first one, we place elements into n buckets, each with a range r, where the j^th bucket holds all elements in a range from $j \times r$ to $(j\times r)+r-1$. A quick example to demonstrate this distribution method:

**r = 10** (range)
Let's say we'll use n buckets. Following the previous rule for allocating values to a bucket, we'll have the following ranges for each bucket:

- Bucket 0 : [0..9]
- Bucket 1 : [10..19]
- ...

Fig. 1. Rule for Allocating Values to Buckets

In this first phase, the algorithm gives us the ability to choose how we want to set up our bucket system from a couple options but we chose to have a pre-defined range of values for each bucket and have the program determine how many buckets are needed. We used dynamic arrays as buckets so that we wouldn't have to calculate the needed capacity for each bucket. We chose to keep a range of 10 for the values in each bucket. Assuming that each element can be placed in the appropriate bucket in O(1) time, this first phase requires O(n) time.

In the second phase, each bucket is sorted individually. To sort each bucket, any conventional sorting algorithm can be used (including a recursive use of bucket sort). We decided to use Insertion Sort to sort our buckets, with complexity of O($n^2$) in the average and worst case, and O(n) in the best case.

In the last phase of the algorithm, we take the sorted buckets in order and place their values into the original array. This phase has a complexity of O(n) because it simply re-allocates a value from a bucket to the original array.

The sequential version of the bucket sort algorithm is as follows:

```
Sequential Algorithm
  1) Initialize array of empty buckets
  2) Allocate the values from the original array
     to a bucket, based on their values
  3) Sort every bucket
  4) Visit array of buckets in order and put the
     sorted elements back to the original array.
```

Fig. 2. Pseudo-code for Sequential Bucket Sorting

Looking at the actual code of the sequential algorithm, it's easy to see some possible optimizations to the algorithm such as loop unrolling that would decrease the total number of instructions to run the program by an amount that would depende on the loop-unrolling level we use. This optimization was, however, discarded in order to test the differences from a purely sequential algorithm to a parallel algorithm that runs exactly the same code but distributes it's work among the threads that are made available to run the program.

Considering the complexity of the different phases described above, we can conclude that the performance of the algorithm can be improved in the second phase (bucket sorting) with the use of parallelism, since more than one bucket can be ordered at the same time by different threads. We could also parallelize the first phase of the algorithm where we distribute the original array's values to the buckets but it would lead to synchronizing overhead and for that reason, we didn't do it.

## III. Parallel Version of the algorithm

In the parallel version of the bucket sorting algorithm, k threads are used to sort the buckets. Dividing the number of buckets equally among the k threads and having each thread sort the buckets. The first and third phase of the sequential algorithm will remain the same as in the previously explained sequential version of the algorithm because their complexity is O(n) and implementing parallelism would bring up some race dontiions in both phases. We will, however, use OpenMP to parallelize the bucket sorting. This is done by creating a *#pragma omp parallel for* before the cycle where we iterate over the buckets and sort each one individually. The work distribution among threads is the result from not using the any scheduling directive, that distributes the iterations of the cycle among the available threads so that they all have the same (or nearly the same) work load. Different scheduling options are tested in further sections of this report.

The pseudo-code for the Parallel Bucket Sorting Algorithm is as follows:

```
Parallel Algorithm
  1) Initialize array of empty buckets
  2) Allocate the values from the original array
     to a bucket, based on their values
  3) Distribute buckets to k threads
  4) Every thread sorts the buckets that had been
     allocated to them
  5) Visit array of buckets in order and put the
     sorted elements back to the original array.
```

Fig. 3. Pseudo-code for Parallel Bucket Sorting

## IV. Results

### A. CPU Runtime Measurement

OpenMP's *omp_get_wtime()* was used to measure the CPU runtime required to run the parallel part of the program. This function returns the wall clock time so there's no need to work on the value it returns since it is already measured in seconds. Since the sorting phase of the algorithm is the only thing that changes from the sequential to the parallel version, we are only interested in measuring the time difference in this specific part of the program. If more parallel regions were to be implemented, this time measurment could be used to any of those regions and/or to the program as a whole. In order to get the amount of time it took for the program to run any part of the code, we do as in the pseudo-code that follows:

```
1) t1 = omp_get_wtime()
2) /*
3) Code...
4) */
5) t2 = omp_get_wtime()
6) elapsed_time = t2 - t1
```

Fig. 4. CPU Runtime Measurement method

### B. Results and Scalability Tests

The experiments were performed using datasets stored in text files and each consists of five hundred thousand and one, five, ten, twenty and fifty million of unsorted positive integers. These integers ranging from 0 to the maximum value of its dataset, were generated randomly using a Python script, totally apart and independent from the bucket sort program. The test datasets are sorted one by one using the sequential bucket sorting with one thread only, and then using the parallel bucket sorting with two, four, eight, sixteen and thirty-two threads. The results are shown as the number of seconds that the program took to perform the different phases of the algorithm. For the purpose of this assessment we also measured the number of Data Cache Misses for the L1 and L2 cache levels. We also tried to get information on the L3 Data Cache Misses but we were unable to get this metric since the nodes available to test the program had CPUs whose architecture was incompatible with the use of the PAPI event responsible for getting the amount of L3 Data Cache Misses. This would make it easier to understand at what point of scaling the input array the program would start needing to access the system's RAM in order to fetch data. This way we are only able to know how many times it asked for values at the L3 level (happens every L2 DCM). Also, it was asked to test the program on machines that had at least 16 cores but when trying to run the program on these machines, the program resulted in PAPI errors that stated it was unable to get the number of hardware counters. For these reasons, the machines used to test the program had the following processors:

1) Intel Xeon E5520 @ 2,27GHz (4 cores, 8 threads)
2) Intel Xeon X5650 @ 2,67GHz (6 cores, 12 threads)
3) Intel Xeon E5-2650 v2 @ 2,60GHz (8 cores, 16 threads)
4) Intel Xeon E5-2695 v2 @ 2,40GHz (12 cores, 24 threads)

All of the machines used to test the program had the same cache structure with L1 Data Cache being of size 32 kB, L2 being 256 kB and L3 being 25600 kB.

We'll now present the results for testing the program on each of the four machines presented previously with all the different previously established input sizes.

The results in the following subsections represent the time it took to sort the buckets (this will be the most different section of the program between sequential and parallel versions of the same algorithm).

*1) Intel Xeon E5520 @ 2,27GHz (4 cores, 8 threads):* The results from running the program and grabbing only the time it took for the program to sort the buckets are the following:

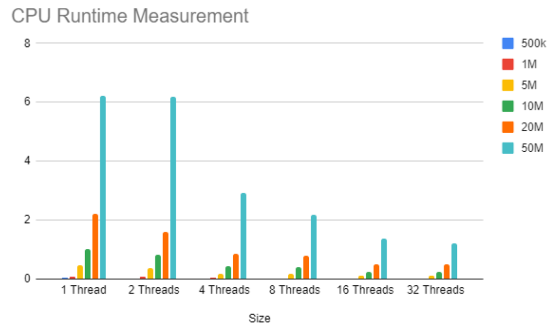


Fig. 5. CPU Runtime Comparison

Using these values we can now understand the gain from using a sequential verion to using a parallel one with either 2 or 4 threads.

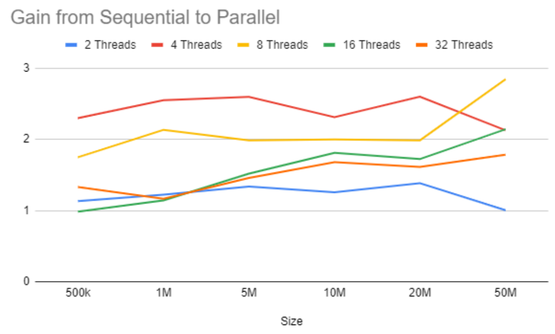


Fig. 6. CPU Runtime Gain From Sequential to Parallel

*2) Intel Xeon X5650 @ 2,67GHz (6 cores, 12 threads):* The results from running the program and grabbing only the time it took for the program to sort the buckets are the following:

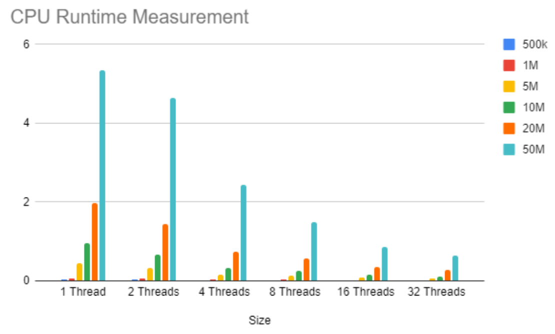


Fig. 7. CPU Runtime Comparison

Using these values we can now understand the gain from using a sequential verion to using a parallel one with either 2 or 4 threads.

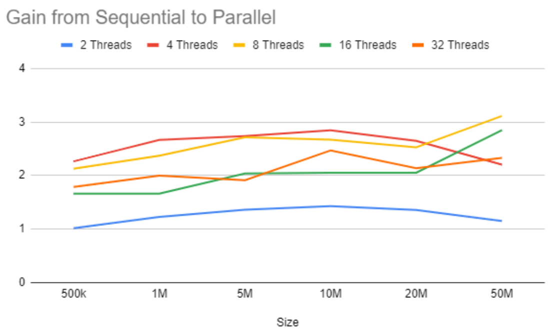


Fig. 8. CPU Runtime Gain From Sequential to Parallel

*3) Intel Xeon E5-2650 v2 @ 2,60GHz (8 cores, 16 threads):* The results from running the program and grabbing only the time it took for the program to sort the buckets are the following:

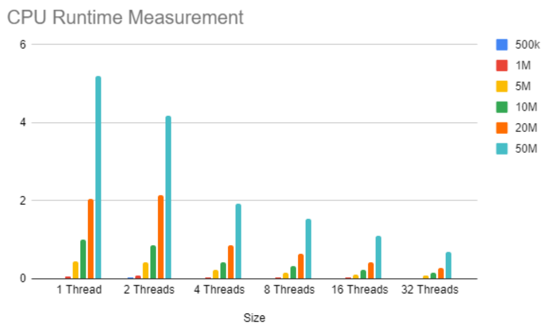


Fig. 9. CPU Runtime Comparison

Using these values we can now understand the gain from using a sequential verion to using a parallel one with either 2 or 4 threads.

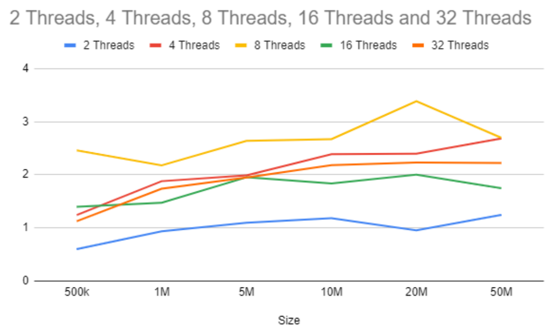


Fig. 10. CPU Runtime Gain From Sequential to Parallel

*4) Intel Xeon E5-2695 v2 @ 2,40GHz (12 cores, 24 threads):* The results from running the program and grabbing only the time it took for the program to sort the buckets are the following:

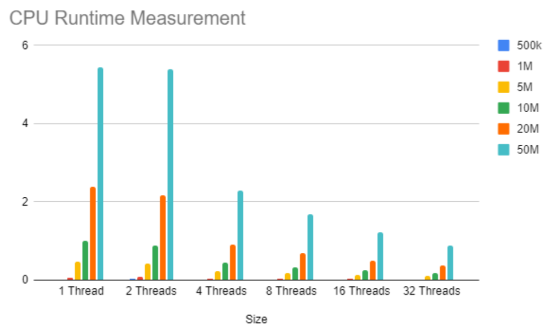


Fig. 11. CPU Runtime Comparison

Using these values we can now understand the gain from using a sequential verion to using a parallel one with either 2 or 4 threads.

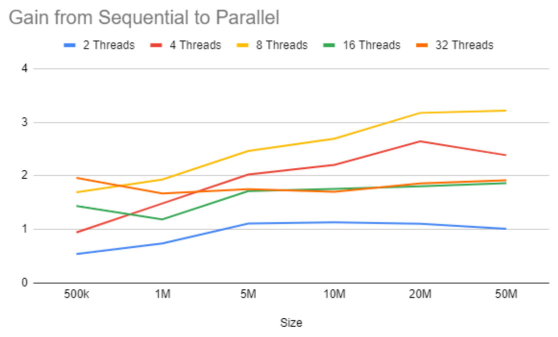


Fig. 12. CPU Runtime Gain From Sequential to Parallel

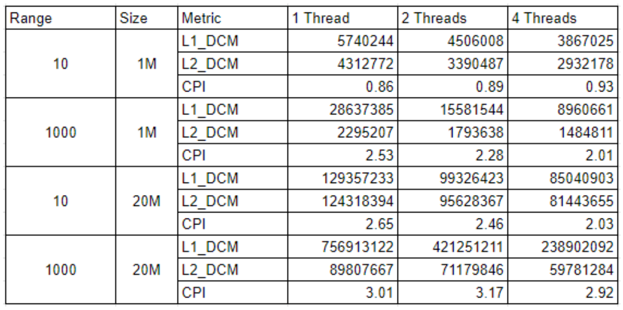
The results show some improvement when the test datasets are large enough to benefit from parallel computation. This is when the communication, synchronization overhead among the parallel threads is overcome by the gain (or speed up).

We can also see in the images above, that increasing the amount of threads used in the parallel chunk of the program does not always lead to a bigger gain in CPU Runtime. In almost every case we tested, having 16 or even 32 threads proved to reduce the gain to the sequential version we got when compared to the gain with 4 and 8 threads. This is when the work needed to create, synchronize and communicate between all threads is superior to the gain in computation parallelization.

## C. *Impact of Bucket Range on Cache and Memory Access*

Upon the first analysis we did at the algorithm, we realized right away that the amount of Memory Accesses (and Misses) would change the most with changing the range of values each bucket can contain. To avoid going on for too long in this subject, we'll only use the last machine we presented in the last subsection and test the difference it makes to have either range equals to ten or one thousand in terms of Data Misses in both L1 and L2 cache levels, for inputs of size one million and twenty million and we'll test only for the sequential version of the program and for the parallel version with 2 and 4 threads. We'll also take the number of clock cycles and total number of instructions in each execution in order to get the CPI measure for each run. We used values that differ in great amounts from each other in order to make it easier to see the impact these differences make in the metrics we get from the program.

The results from getting the values from these metrics for both range values are as follows:



| Range | Size | Metric | 1 Thread | 2 Threads | 4 Threads |
|---|---|---|---|---|---|
| 10 | 1M | L1_DCM | 5740244 | 4506008 | 3867025 |
| | | L2_DCM | 4312772 | 3390487 | 2932178 |
| | | CPI | 0.86 | 0.89 | 0.93 |
| 1000 | 1M | L1_DCM | 28637385 | 15581544 | 8960661 |
| | | L2_DCM | 2295207 | 1793638 | 1484811 |
| | | CPI | 2.53 | 2.28 | 2.01 |
| 10 | 20M | L1_DCM | 129357233 | 99326423 | 85040903 |
| | | L2_DCM | 124318394 | 95628367 | 81443655 |
| | | CPI | 2.65 | 2.46 | 2.03 |
| 1000 | 20M | L1_DCM | 756913122 | 421251211 | 238902092 |
| | | L2_DCM | 89807667 | 71179846 | 59781284 |
| | | CPI | 3.01 | 3.17 | 2.92 |

Fig. 13. L1 and L2 Data Cache Misses & CPI

By looking at the the values from the table above, we can see that when going from sequential to parallel with 2 threads and then to parallel with 4 threads, CPI gets closer and closer to the ideal value of 1. Looking at the tables in the previous section and to the table above, we can see that, as CPI gets closer to 1, the time it takes to sort the buckets gets lower and lower. We can also see that the amount of Cache Misses when the bucket range is set to 1000 is much higher than in range equals to 10. This is easy to understand considering everytime we try to sort a bucket, each bucket can have 100 times more elements when range = 1000. This will make it so that everytime the CPU tries to read a value, it's a lot less likable that the value is in the upper levels of the cache structure. Also, when the amount of Data Cache Misses gets higher, so does the CPI. This is easy to understand considering it takes more clock cycles to get a value that is in lower levels of cache (L3, RAM) than to get a value that is in L1, for example. Increasing the amount of clock cycles will also increase the CPI.

## D. *Impact of scheduling directives on the sorting phase parallelization*

As we said while explaining the parallel algorithm of bucket sorting, we used the simple *#pragma omp parallel for* before the cycle where we iterate over the buckets to sort them in order to make it a parallel region where the iterarions of the cycle are split among the available threads. However, OpenMP gives us a few additional directives to control the work balance for the threads. To test their impact on the performance of the algorithm, we took the example of a dataset with 20 million integers and tested sorting it with 4 different directives (and mantaining the non-scheduled results). We performed these tests on the 4 different machines we've been using throughout this report. The results we got were the following:

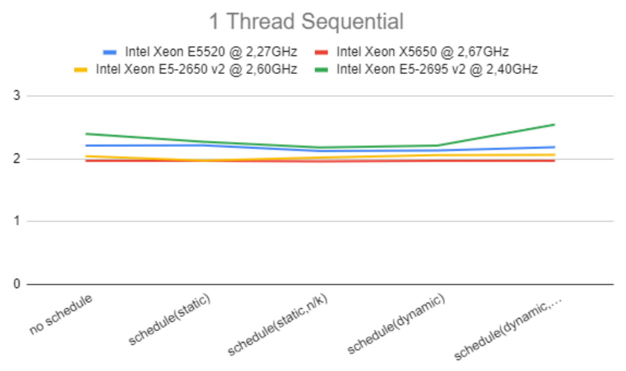


Fig. 14. CPU Runtime Comparison for Different Scheduling Directives

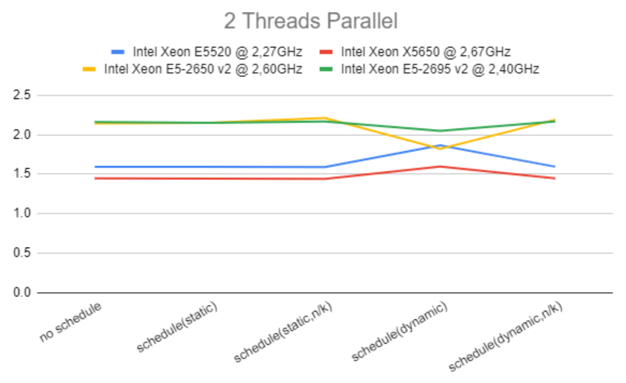


Fig. 15. CPU Runtime Comparison for Different Scheduling Directives

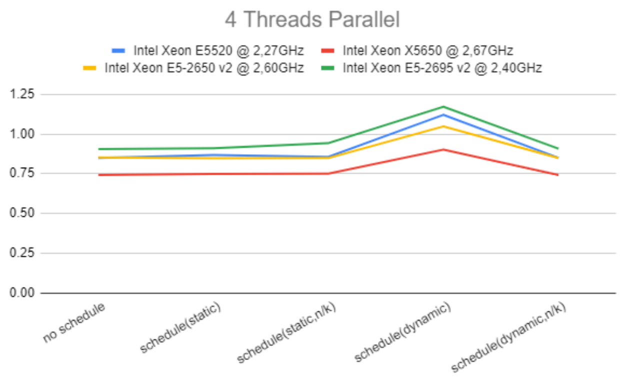


Fig. 16. CPU Runtime Comparison for Different Scheduling Directives

By the images above, we can safely say that there's no universal and clear improvement to be made to the algorithm by adding any of the directives. The *schedule(dynamic)* slighlty improved the performance on the 2 threaded parallel run on Intel Xeon E5-2650 v2 but in the 4 threaded version of the program, it decreased the performance on all CPUs. Therefore, no directive is a clear improvement when it comes to performance when comparing to the others.

## V. Conclusion

Using machines with multicore processors, this assessment shows how the bucket sort algorithm (usually a sequential algorithm) can be coverted to a parallel sorting algorithm and be implemented in C and executed using the OpenMP API. The results of the tests we made throughout the development of this project show that the algorithm benefits from parallelization when datasets are large enough to benefit from parallel computation and every advantage that comes with it. We also got to see from the tests we ran that increasing the number of threads that will run the program in parallel, doesn't always lead to a better performance. In future work, the algorithm could be tweaked in ways that would allow for the implementation of more parallel regions other than the sorting phase, thus increasing the percentage of parallelization over the full program. The testing on machines with 16 cores was something we would've liked to have a chance to try but unfortunately was not possible, such as the measurements of Data Cache Misses at L3 level, that would allows us to better understand the RAM acesses made when a miss occurs at L3. In general, we believe the objectives of the assessment were fulfilled and we feel like we got to put in practice most of the knowledge on the OpenMP API and code parallelization learned during the semester in class.