Parallel Bucket Sort

Parallel Programming Course

1st Maria Gabriela Oliveira
Physics Engineering Student
University of Minho
Braga, Portugal
mgabijo@gmail.com

2nd Miguel Caçador Peixoto
Physics Engineering Student
University of Minho
Braga, Portugal
miguelpeixoto457@gmail.com

***Abstract*—An improved sequential bucket sort was implemented and scrutinized, building up from the version provided by the professors on the basis of benchmarks performed by Perf and PAPI. This report also includes a parallel version of the algorithm which peak performed at 340x when compared to the original code. Performance, when the dimensionality or the hardware is changed, was also recorded and critically analyzed.**

***Index Terms*—Parallel Programming, Bucket Sort, Bubble Sort, Quick Sort, OpenMP, Perf, PAPI, Complexity Analysis**

I. Introduction

Traditionally, it is used sequential computing, id est, the use of a single processor to execute a sequence of discrete instructions, each executed one after the other with no overlap at any given time. This technique is significantly limited by the processor's speed and ability to perform each series of instructions. The popularisation and evolution of parallel computing in the 21^{st} century came in response to processor frequency scaling hitting the power wall. Increases in frequency resulted in an increase in power used by the processor, and scaling the processor frequency is no longer viable after a certain limit. Therefore, parallel computing has become the dominant paradigm in computer architecture. Programmers and manufacturers began designing parallel systems both in software and in hardware, hence producing power-efficient processors with multiple cores. Some applications for parallel computing are computational astrophysics, climate modelling, and financial risk management.

Parallel processing is a general term for dividing tasks into multiple threads that can execute at the same time. These threads are entities that can independently run a stream of instructions. This independence of code execution can occur both at the hardware and the software level. At the software level, an application might be adapted or rewritten to take advantage of code parallelism. The threads can execute simultaneously with the proper hardware support. However, mind that even when the hardware is capable of parallel execution (multiprocessors, multi-core, multi-threads, cluster computing, grid computing), the software must still divide, schedule, and manage the tasks.

This report will show the impact of software level parallelism applied to a sequential bucket sort algorithm step by step. The final improved algorithm results from a set of improvements to the sequential bucket sort provided by the professors that were based on different benchmarks. Moreover, it will be analysed the impact of data dimensionality as well as the impact of the hardware (within the available tools).

II. Work environment

A suite of tools was used thought-out this report. For compiling code, *GCC* version 4.8.5 was used, always with the $-std = c99$ and $-O2$ flags for consistency of the results; For benchmarking sequential code, *Papi* 5.4.1 and *Perf* 3.10.0-1160.41.1.el7 were used; Last but not least, for taking advantage of code parallelism, *OpenMP* 3.1 was applied.

As demanded, all the code executions and tests were entirely made on the computational nodes of the SeARCH cluster of UMinho's DI, SE-ARC cluster. If not referred otherwise, all the code was executed on cpar node which was contemplated by an *Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz* which has 20 cores, 40 threads, and *64GB* of RAM [1].

III. Bucket Sort Algorithm

Bucket sort is one of the many well-known sorting algorithms. It's an integer sort, meaning that it is an algorithm that sorts a collection of data values by integer keys instead of the more general comparison sorts algorithms.

This algorithm divides the array to be sorted (input), v, into several sub-arrays called buckets. Then, each bucket is sorted independently, either using various sorting algorithms or recursively implementing the same technique. On average, it is a linear sorting algorithm. However, in its worst case - when elements in the array are close in proximity, they are likely to be placed in the same bucket, and so its complexity depends on the sorting algorithm used to sort the bucket's elements.

The pseudo-code for the implementation of the algorithm is as follows:

Algorithm 1: Bucket Sort

Input: An array $[a_i]$, $i = 1, 2, \cdots, n$, where each element is an integer; K - An integer which represents the number of buckets

Output: The sorted array.

$max \leftarrow$ maximum value in the $array$
$N \leftarrow$ length of the array
$buckets \leftarrow$ new array of K empty lists

```
// Insert elements into the
   respective bucket.
```
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $elem \leftarrow array[i]$
  $x \leftarrow \frac{elem}{max}$
  insert elem into buckets[x]
end

```
// Sort the buckets individually.
```
for $i \leftarrow 0$ **to** $K - 1$ **do**
  Auxiliary_Sort(buckets[i]) `// This is an arbitrary sorting algorithm.`
end
return *Concatenation of buckets[1], ..., buckets[*$K - 1$*]*

IV. Sequential Bucket Sort

The professors provided the first version of the bucket sort algorithm, a sequential implementation. Mind that the original algorithm uses bubble sort to sort each bucket. Complexity wise, the average case is the same as its worst-case: $O(N^2)$. And the best performance occurs when the vector is already sorted, with a peak performance of $O(N)$ [2].

The first step to improve this first version of the code is to run some base benchmarks and tests to find the bottlenecks. These will also give us ground results for future comparison with future improved versions [1].

A. Perf

In this subsection, *Perf* was used to determine which section of the code is taking up more resources.

Taking a closer look at the pseudo-code step-by-step:

- The initialization of all the variables, as well as memory allocation, should be practically instantaneous;
- First loop - The corresponding bucket index for each element is computed and the current element is appended to it. Both are constant operations and will be done N times ($O(N)$);
- Second loop - The array of each bucket is sorted. Please note, in a first instance, bubble sort will be used (which has an average complexity case of $O(N^2)$);
- Concatenation of the buckets - Each bucket will be consulted and concatenated back into the original vector ($O(\#buckets)$).

[1]All of the following benchmarks were done with an initial randomly sampled array, v, generated by the rand() function from the C library..

Herewith, it is possible to theoretically infer that the second loop is where the bottleneck is located. Since it houses the bubble sort function call (which does not have a desired complexity) and will be sequentially executed for each bucket ($\#buckets$ times), it is therefore expected for the program to spend most of the computational resources executing the bubble sort code.

To prove the theoretical expectation and initialise the algorithm's practical analysis, *Perf* ran on the first implementation of bucket sort.

```
# Total Lost Samples: 95
#
# Samples: 6K of event 'cycles:uppp'
# Event count (approx.): 4542669475
#
# Overhead       Samples  Command     Shared Object     Symbol
# ........  ............  ..........  ................  ..........................
#
    99.57%          6428  bucket.out  bucket.out        [.] bubble
     0.27%            23  bucket.out  [unknown]         [k] 0xffffffffb418c4ef
     0.10%             7  bucket.out  [unknown]         [k] 0xffffffffb4196098
     0.01%             1  bucket.out  libpfm.so.4.6.0   [.] intel_x86_event_has_pebs
     0.01%             1  bucket.out  libc-2.17.so      [.] _IO_getc
     0.01%             1  bucket.out  ld-2.17.so        [.] do_lookup_x
     0.01%             1  bucket.out  ld-2.17.so        [.] strcmp
```

Fig. 1: *Perf* results for the initial algorithm.

By analyzing the results of Figure 1 our theory is confirmed - the bottleneck of the code was in fact the *bubble sort* function (99.57%) of overhead.

B. Papi

Papi will be used to see how fast code executes and extract some cache and clock cycles specific statistics. To have some base data, two different runs were done.

The first run consisted in fixing the number of buckets in 10 and varying the size of the array to be ordered to see how the different metrics evolved. The second consisted of fixing the array's length in $100k$ and varying the number of buckets with the same aim.

Through the analysis of the results (subsection X-A, Figure 6), it is possible to understand the algorithm's behavior in terms of resources infer possible optimizations. This detailed analysis is traduced in:

- **Wall Clock Time (WCT):**
  The WCP grows with the array size, which makes sense since it was expected the more you increase the input vector, the more time is needed for it to be sorted. This fact also confirms our theoretical time complexity expectation since WCP grows approximately quadratically [2].
  On the other hand, WCP has a polynomial behavior when varying the number of buckets keeping the size of the vector at $1e5$. It appears to be a parabola, i.e., there is a minimum value of WCP for approximately $\#Buckets = 1e3$ (Peak performance).
  This means there is a trade-off in performance between the $\#buckets$ and the total Wall clock time. Initially, the performance increases as the number of buckets increases. However, when the limit of $1e3$ buckets is

[2]To convince yourself, take a look at the first quadrant of the function $y = x^2$.

passed, the increasing of buckets has the inverse effect - the performance decreases as the number of buckets increases.
Two things can explain this behavior. The first is intrinsic to the algorithm - The last for loop will call the sort function a number of times proportional to the number of buckets. Naturally, there is a trade-off between the number of elements in each bucket and the number of times the sort function is called. When the $\#buckets$ increase to values closer to the array's length, the sort function won't do much of a job since each bucket shall have, on average, just one element. In this case, the second problem comes along, with a large number of buckets, each one with just an average of 1 element, and each with a total allocated space equal to the length of the original vector this will cause an increase of L1 and L2 cache misses contributing to a decrease of the overall performance of the algorithm.

- **PAPI_TOT_CYC (PTC):**
  The total clock cycles metric has similar behavior to WCT. The more time it takes for the program to run, the more cycles per clock it must take, assuming a constant clock frequency.
- **PAPI_TOT_INS (PTI):**
  The total instructions metric also has similar behavior to WCT. Since the code is full of cycles and the cycles depend on the array size, it's expected (and verified) that the number of instructions increases as the array size increase.
- **PAPI_L1_DCM:**
  As naturally expected, when the number of buckets is fixed to 10, the number of L1 cache misses increases as the length of the vector increases.
  The size of data cache L1 is 32kiB, i.e., 32 768 bytes. Since the size of each int is 4 bytes, it is only possible to keep 8192 ints at L1 cache. Considering 10 buckets, to keep all of the data information in the L1 cache, the length of the vector needs to be inferior to 800 ints. This fact was experimentally verified, the curve increases faster roughly after this point.
  When the array size is fixed at 10^5, and the number of buckets is small, the initial L1 cache misses are large. This is because the vector does not fit the cache at all. Increasing the number of buckets leads to less information in each bucket, and so it is easier to keep the information in the cache. This leads to a decrease of cache misses. At a certain point, with the increasing of the buckets, each bucket contains less significant information and due to that, the information needed is cached less often. Mind that when the number of elements per bucket is less than a line of cache, when a cache line is brought comes garbage instead of just needed information.
- **PAPI_L2_DCM:**
  Such as in L1 cache misses, when the number of buckets is fixed at 10, the number of the L2 cache misses increases as the length of the vector increases.
  In a way of thinking similar to the previous one, the cache size is now 256kiB, so it is only possible to keep less than 65536 ints at the L2 cache. Note that while the L1 cache is a data cache, the L2 cache is a unified cache, so there is more information than data in there. Considering 10 buckets, to keep all of the data information in the L2 cache, the length of the vector needs to be inferior to 6000 ints. According to the experimental observation, the curve increases faster roughly after this point.
  When the array size is fixed at 10^5, the explication is like the one for L1 cache misses.

Please note that in this and in the subsequent Papi Benchmarks, Papi started counting events on the beginning of the *bucket_sort* function and stopped on the end of it. Since only a component of the *bucket_sort* code will be tweaked at a time, this approach was the chosen one.

C. Possible Optimisations

Intending to improve the algorithm, possible optimizations will be enumerated.

Attending the results shown in subsection IV-A and subsection IV-B , possible optimizations were inferred. Videlicet, improve the *bubble sort* code to make it more efficient or even replace it by a new algorithm with a better average complexity, since the sorting function used to sort each bucket is arbitrary; Parallelize the insertion of elements into their respective bucket (1^{st} loop); Apply code parallelism to the auxiliary sort calls (2^{nd} Loop) and/or the auxiliary sort code itself; Apply parallelism to the concatenation of the buckets into the original vector.

These optimizations will be explained in detail in this report (when they are applied to the code).

V. Improved Sequential Bucket Sort

Deriving from the insight given in subsection IV-C, various modifications were implemented and tested, with the aim of doing an improved sequential bucket sort algorithm.

One of the most noticeable improvements, even though not necessarily an overall execution time of the code improvement, that was inferred by analyzing the original code is releasing the memory allocated for each bucket on the last loop.

Since this change would just affect the total allocated memory on the final execution of the program, it isn't expected significant changes in the metrics recorded by *PAPI* compared to the base benchmark.

But when the results obtained from this change (subsection X-A) are analyzed, the expected ratio closed to 1 in all metrics was not found. Instead, there were some fluctuations, sometimes as higher as 3 orders of magnitude.

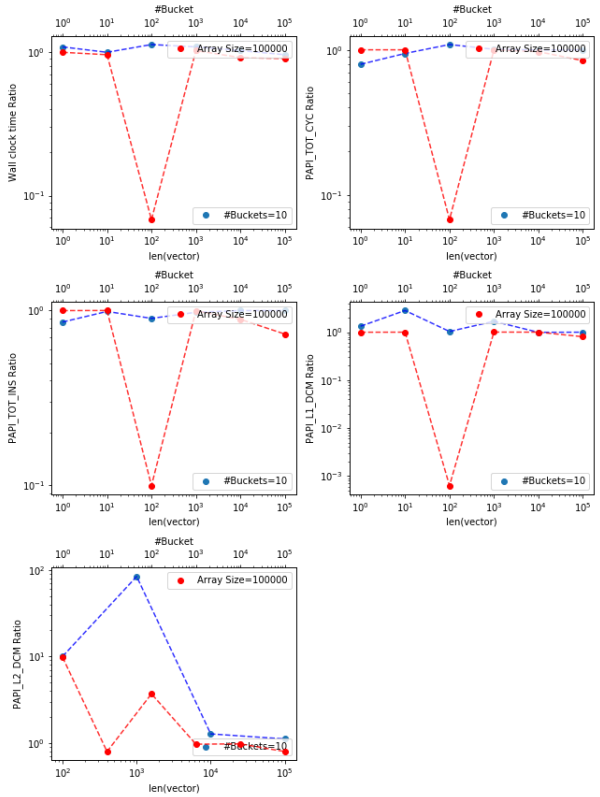

Fig. 2: Overall ratio between Base Benchmark and w/Free Benchmark Metrics

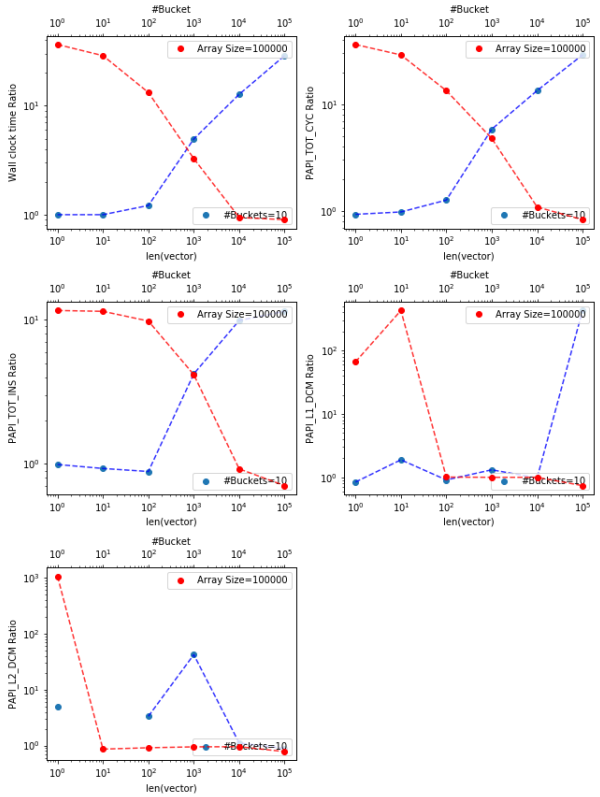

Fig. 3: Overall ratio between Base Benchmark and Quick Sort Free Metrics

It was experimentally verified that the "free version" has worse performance than the original one in some cases (when the ratio < 1). The cause of this behaviour is unknown since it could only be explained by some mechanism that the authors aren't aware nor accounted (e.g., CPU Fetcher).

Moving forward, the section IV gave an insight into how the bubble sort algorithm behaves in terms of complexity analysis.

Since there are sorting algorithms with a better average-case complexity, just as mentioned in subsection IV-C, the bubble sort on the original code was replaced by quick sort, which has an average-case performance of just $O(n \log n)$ [3]. And so, after freeing the allocated memory, the auxiliary sort was replaced, resulting in the final **improved sequential bucket sort algorithm**.

Bellow, there is a plot of the overall ratio between the base benchmark and the last sequential version of the algorithm, i.e., the first results were divided by the last ones (improved sequential bucket sort algorithm).

Attending the Figure 3, a computational improvement is easily seen with this approach. In some instances, the improved sequential bucket sort algorithm is more than 10^4 times better than the first one, and its never worse, as expected.

Note: The quick sort algorithm implemented is the same as specified on [3].

VI. Bucket Sort with Parallelism

Firstly, *Perf* was used in the last iteration of the algorithm to determine which section of the code is taking up more resources now. Behold that, for this benchmark, the loops were defined as functions and called in the main one. It was found the bottleneck is now in the partition function (92.63%) and the other loops don't have a significant overhead impact.

By inspecting the code, it's possible to reason the partition function cannot be parallelized. Although it's possible to parallelize the call of the quick sort algorithm and quick sort recursive calls, which implies parallelizing the call of the partition function.

Applying Amdahl's law [4], it's found that the theoretical maximum speedup is 8.33. Note that when this parallelization happens, the quick sort is also parallelized and so the actual fraction of parallelization is greater than 92.63

Arising from the algorithm complexity theory explained in subsection IV-A and the possible optimizations described in subsection IV-C, code parallelism was implemented in different algorithm sections.

Note that for the following sections, the improved sequential bucket sort algorithm is assumed as the starting point. All benchmarks and tests were done with a fixed vector length of 10^6, 10 buckets, and a set of $\#threads$ equal to $\{2, 4, 8, 12, 16, 24, 32, 48, 64\}$.

A. First Loop Parallelism

The first loop[3], as explained previously, computes the corresponding bucket index and appends the current element to it.

There are multiple ways to parallelize this. The pragma *omp parallel for* was chosen since each iteration uses the same overhead[4].

Also, the iterations aren't mutually independent. Suppose two threads try to append two different elements to the same bucket. In that case, they may have read the same index, resulting in just one element being added to the bucket instead of two (there is a overwrite of the 1st element written by the 2^{nd}).

Consequently, *omp critical* was applied just before putting the element in the bucket due to this problem.

At first glance, since the problem is $O(N)$, its expected a linear relation between the number of $threads$ and the total time elapsed. Considering that *omp critical* was applied, the only effective region of the code taking advantage of parallelism is the computation of the correct bucket index. But since this operation, sequentially speaking, is close to instantaneous, what is effectively being done is allocating and managing[5] threads to a operation that is by itself instantaneous. Therefore, its expected this overall change would yield non-positive performance for the code.

The results were confirmed by Figure 10, where the overall speedup is less than 1. Also, the more the number of threads is increased, the worst the performance is, as expected.

B. Second Loop Parallelism

Continuing the line of reasoning , since the sorting of each bucket is independent from one another, and the the overhead of each bucket is not the same (one bucket may have more elements that another and vice-versa), *pragma omp parallel for schedule(guided)* was chosen. It provides an intelligent balancing of work between the different threads and consequently optimises the total performance. It's expected a significant speedup since this section is the largest bottleneck.

In such a manner, this method was implemented and the speedup was recorded on Figure 12, which confirmed the theory. Analyzing the results, the more threads we use, the greater the performance until optimal performance (of 9x plus) is reached around 10 threads. After reaching this point, the speed up slowly decreases with increasing threads.

This is in agreement since buckets are fixed at 10. This way, when 10 threads are allocated, parallelism is applied to all the work simultaneously in one go. When there are less than 10 threads this wasn't possible, and when there are more than 10 threads, more threads than needed are being allocated. This is why in those regions the speedup is less than when compared $\#threads = 10$. This is also in conformity with theoretical speedup since it is near 8.33, the fact of the experimental speed being greater than the theoretical one is explained, as already said in section VI.

C. Concatenation of the buckets Parallelism

The following step of optimization of the bucket sort algorithm is to apply parallelism to the concatenation of the buckets back into the original vector. Therefrom the pragma *omp parallel for ordered* was applied to the main loop and the pragma *omp parallel for ordered schedule(guided)* was used on the nested loop.

As previously, this configuration was chosen since the first loop has a consistent workload (executing the paralleled for loop) and the second does not [6].

Also, mind that all code must run orderly due to the inherent implementation and this is why *ordered* was used. Due to this, even though *opm* pragmas are being used, they are all running sequentially and so a similar behavior to subsection VI-A is expected. This behavior is by Figure 14.

D. Quick Sort Parallelism

In this last optimization, the *quickSort* function was rewritten to call *ompQuickSort*. The main function calls it with *omp parallel* and *omp single no wait*. This is done to first initialize a parallel region on the code and then assign only one thread to execute the call. In its turn, *ompQuickSort* calls itself recursively, two times in each call. Therefore, on each call, pragma *omp task* is used and on the final line of the code, *omp taskwait* is put in place. This way, the initial thread will do its job (call the partition function) and then call the function recursively with a new assigned thread and wait for all the called thread's completion. This is done recursively until all work finish.

Since parallelism is being applied to the bottleneck (Auxiliary Sort) of the code, a considerable improvement is expected from this change. This is confirmed by Figure 16, which shows an improvement of up to 3 times faster run time.

E. Best Possible Algorithm

The preceding subsections exposed the better ways and sections to apply code parallelism to the algorithm. Putting everything together, the **best version of the bucket sort algorithm** was implemented.

The parallelism optimization for the first loop and the concatenation of the buckets were ignored, maintaining the same code sections as the original algorithm since they provided a non-positive yield in performance.

[3]Note that in this loop count, the loop where the memory for the buckets is allocated is ignored.

[4]If it didn't, pragma *omp parallel for schedule(guided)* may have been a better choice.

[5]When applied omp critical, the program has to ensure only one threads is executing that specific region of the code at a time.

[6]The for loop will be executed a number of times equal to the number of elements in the current bucket.

The second loop optimization was implemented with a slight tweak. Taking from what was learned from subsection VI-B, the *schedule* was removed and the $\#Threads$ was fixed to be the same as $\#Buckets$. And lastly, Quick Sort optimization was also implemented on this final version of the algorithm.

Looking at the performance results displayed on item 7, it's possible to see a major improvement compared to the **Improved Sequential Sort algorithm** of around 9.3 of speedup on Figure 18.

Over and above, the techniques applied showed a total speedup of around **340 times** when compared to the performance of the original code provided by the professors, as seen in Figure 4.

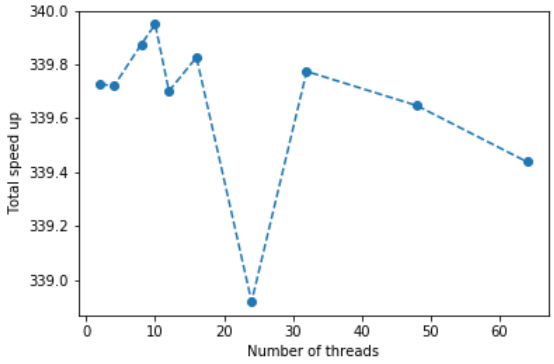

Fig. 4: Overall speedup between the result shown at subsubsection X-B1 and Figure 17

Furthermore, there is very little variation on the total speedup when the $\#threads$ vary, just around 339 and 340 of speedup. This is due because now only the auxiliary sort itself varies the $\#threads$, and since their calls are themselves being paralleled (second loop), the overall speedup remains constant.

VII. Performance when the array size is changed

In order to further explore the impact of L1 and L2 cache misses, a benchmark was ran with an array size of 100 and 1000 (these values were induced by the deductions made on subsection IV-B.

This way a performance test will be done when everything fits the L1 cache (100 ints) and another when everything fits the L2 cache (1000 ints).

The performance results, displayed on subsection X-C, show a reduction in performance as the array size decreases. For a smaller array size, each bucket has fewer elements and due to that, the code parallelization has a sub-optimal performance.

Also, please note the results appear to be quite volatile. Unfortunately, it wasn't possible to hunt down a likely culprit to this abnormal behavior by the authors.

VIII. Performance when the hardware is changed

In order to know how the performance varied when on different hardware, the best possible algorithm was runned on *week* node which was contemplated by an *Intel(R) Xeon(R) CPU E5520 @ 2.27GHz* which has 8 cores and 16 threads.

From just a number of threads perspective, *week* is a stepping down from *cpar*, from 40 threads to just 16. And so, is expected $\frac{16}{40} \approx 0.4$ the performance of *cpar*.

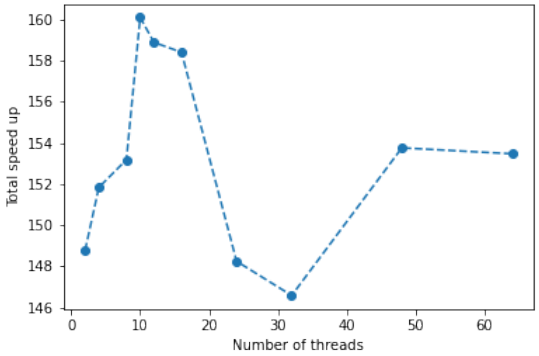

Fig. 5: Overall speedup between the result shown at subsubsection X-B1 and Figure 25

Analyzing Figure 5, it is trivial to see that the performance achieve is inferior to the one achieved in *cpar* node. Particularly, doing the performance's ratio - $160/340 = 0.47 \approx 0.4$ - , the theoretical expectation is proved.

IX. Conclusion

In each section, conclusions were blow-by-blow taken, and so, in this last one, the main results and conclusions will be wrapped.

Firstly, a maximum total speedup of 340 was achieved for 10 buckets, a vector length of 10^6, and 10 threads. This is at the same time curious and expected since there is a thread for each bucket, and thus there are no missing or excess threads (and there isn't unnecessary management of threads), therefore maximising performance.

Regarding the performance analysis for different array sizes, it is easy to see that speedup decreases as the size decreases. Attending an array that fits in the L1 or L2 cache, it's trivial to understand that each bucket has few elements and so all the extra mechanisms to manage threads don't bring as many benefits as for bigger array sizes. Hence, for this algorithm, the parallelization of the code has a greater performance with bigger array sizes.

Concerning the impact of hardware, it was experimentally verified that the speedup's ratio is the same as the ratio of the number of threads of each processor.

Despite parallelization having a great impact on the performance, it's important to be aware that the change of the bubble sort algorithm for the quick sort one is also of major importance.

References

[1] "Cluster Nodes", UMinho Informatics Department, http://search6.di.uminho.pt/wordpress/?page_id=55

[2] ”Bubble sort”, Wikipedia - https://pt.wikipedia.org/wiki/Bubble_sort
[3] ”Quicksort Algorithm”, Programiz, https://www.programiz.com/dsa/quick-sort
[4] ”Amdahl’s law and its proof”, Geeks for Geeks, https://www.geeksforgeeks.org/computer-organization-amdahls-law-and-its-proof/

X. Appendix

A. Improved Sequential Code

1) Individual Metrics: Two different variables were varied when recording PAPI Metrics: The first was the $\#Buckets$ for a fixed array size of $100k$ and the second was the length of the array to be sorted for a fixed $\#Buckets$ of 10.

A brief summary of PAPI Metrics:

- *Wall Clock Time* - Equivalent to the Elapsed real-time - is the actual time taken from the start of a computer program to the end. In other words, it is the difference between the time at which a task finishes and the time at which the task started. (usecs)
- *PAPI_TOT_CYC* - Total number of processor cycles elapsed since the beginning of the program to it’s end.
- *PAPI_TOT_INS* - Total number of assembly instructions executed by the program since its beginning of its end.
- *PAPI_L1_DCM* - Total number of L1 Cache misses elapsed since the beginning of the program to it’s end.
- *PAPI_L2_DCM* - Total number of L2 Cache misses elapsed since the beginning of the program to it’s end.

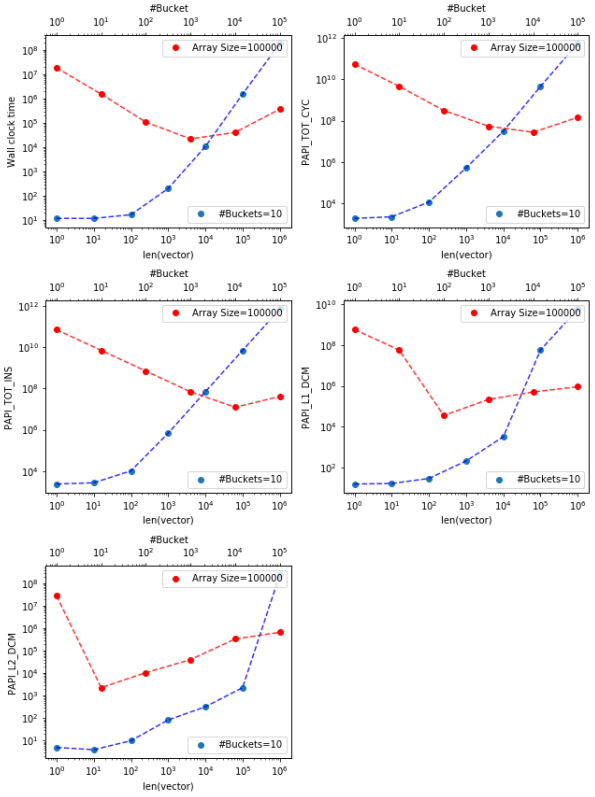

Fig. 6: PAPI Metrics for the Original Code

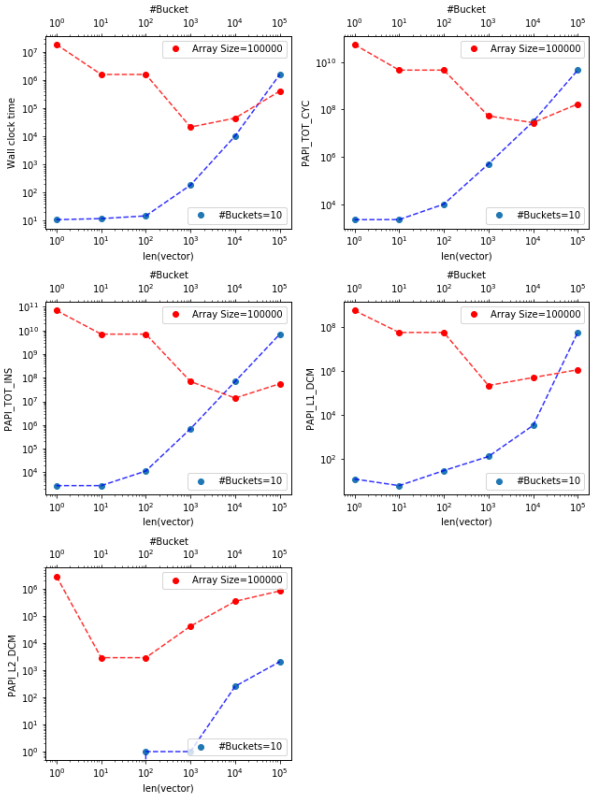

Fig. 7: PAPI Metrics for the Original Code w/Free

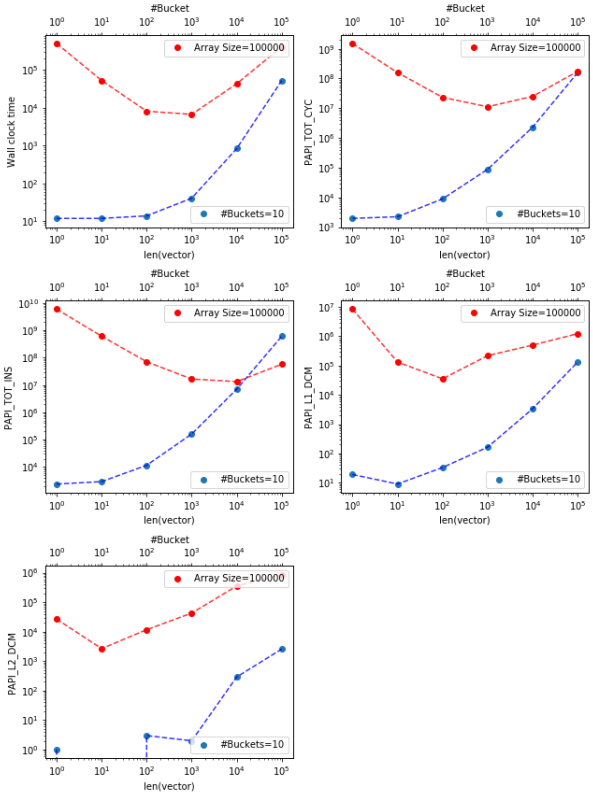

Fig. 8: PAPI Metrics for the Original Code w/Quicksort Free

Note: The reader may have noticed some missing values of the L2_DCM metric. This is not a mistake - The values are effectively 0 but since a logarithm was applied to the y-axis, the values are not defined ($\log 0 = -\infty$).

B. Bucket Sort With Parallelism

All of the following values were taken using *time* command. For consistency purposes the length of the vector to be sorted was fixed at $1M$ and $\#Buckets$ at 10.

1) Base Benchmark of the Original Sequential Sort: For length of the vector to be sorted of $1M$, and $\#Buckets = 10$, the original sequential sort algorithm took 184 seconds to finish.

2) Base Benchmark of Improved Sequential Sort: For length of the vector to be sorted of $1M$, and $\#Buckets = 10$, the improved sequential sort algorithm took 5.044181 seconds to finish.

3) First Loop

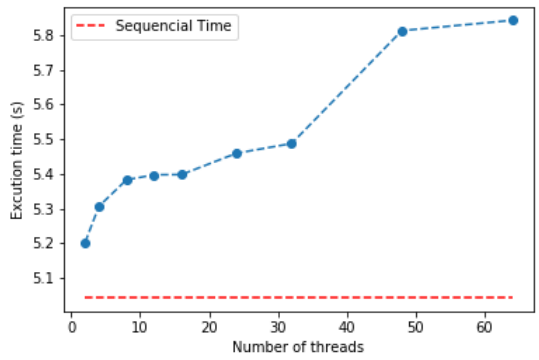

Fig. 9: Execution time when first loop parallelism is implemented in function to the nº of threads.

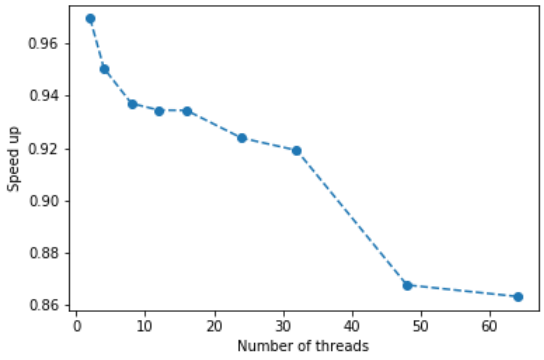

Fig. 10: Overall speedup between the result shown at subsubsection X-B2 and Figure 9

4) Second Loop

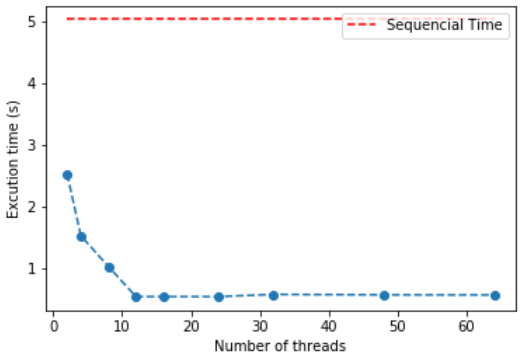

Fig. 11: Execution time when second loop parallelism is implemented in function to the nº of threads.

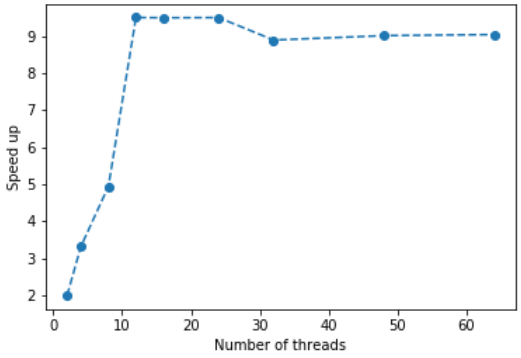

Fig. 12: Overall speedup between the result shown at subsubsection X-B2 and Figure 11

5) Concatenation of the buckets

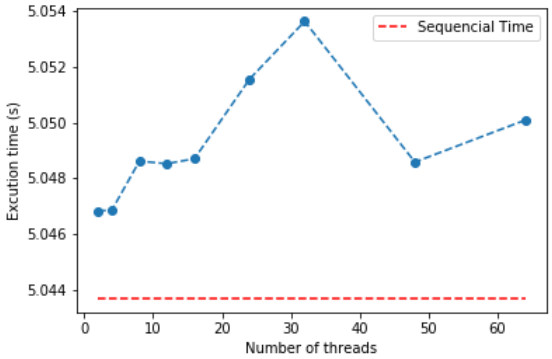

Fig. 13: Execution time when parallelism is implemented to the concatenation of the buckets in function to the nº of threads.

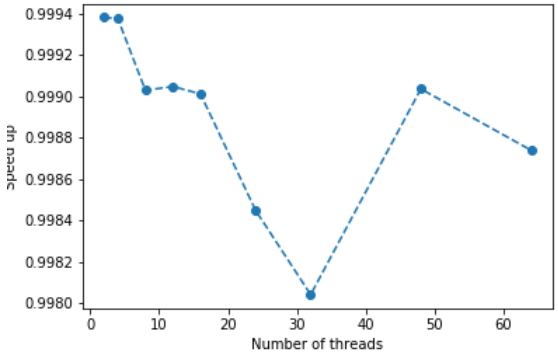

Fig. 14: Overall speedup between the result shown at subsubsection X-B2 and Figure 13

6) Quick Sort

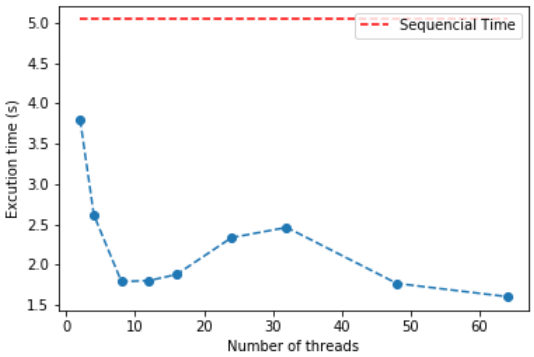

Fig. 15: Execution time when parallelism is implemented to Quick Sort in function to the nº of threads.

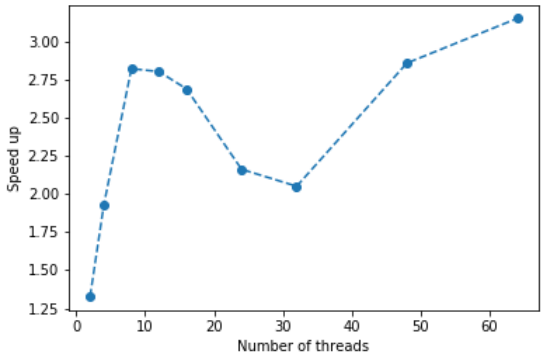

Fig. 16: Overall speedup between the result shown at subsubsection X-B2 and Figure 15

7) Best Possible Algorithm

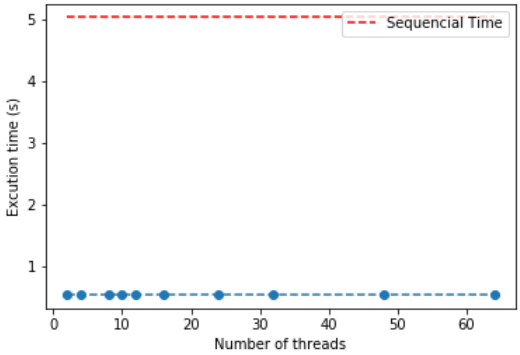

Fig. 17: Execution time of the best possible bucket sort algorithm in function to the nº of threads.

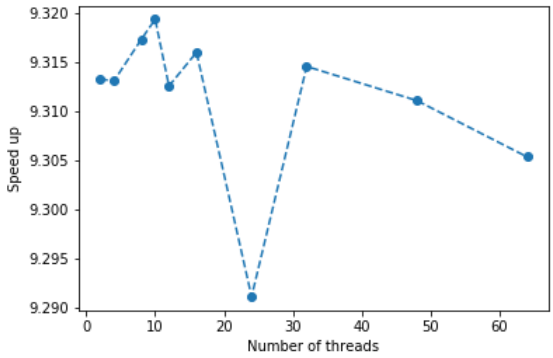

Fig. 18: Overall speedup between the result shown at subsubsection X-B2 and Figure 17

C. Impact on the dimension of data

1) For 100 ints

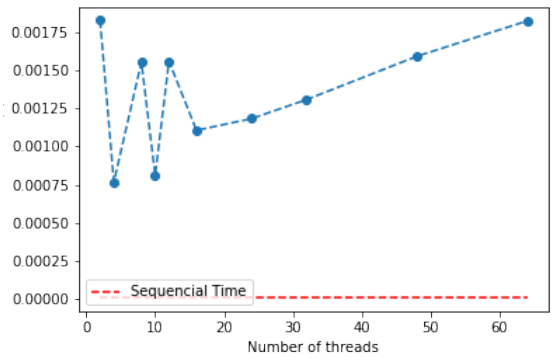

Fig. 19: Execution time of the best possible bucket sort algorithm in function to the nº of threads.

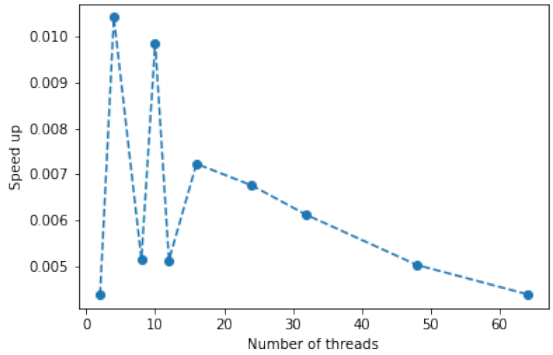

Fig. 20: Overall speedup between the result shown at subsubsection X-B2 and Figure 19

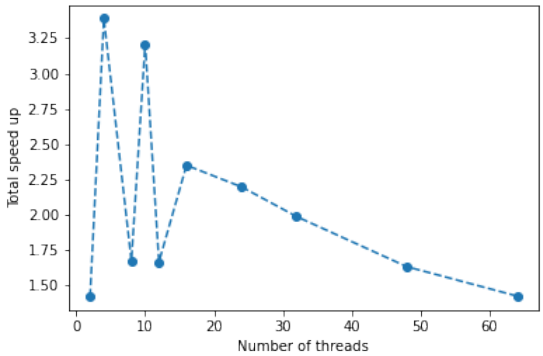

Fig. 21: Overall speedup between the result shown at subsubsection X-B1 and Figure 19

2) For 1000 ints

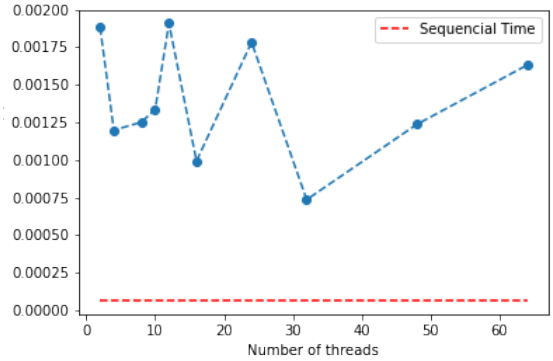

Fig. 22: Execution time of the best possible bucket sort algorithm in function to the nº of threads.

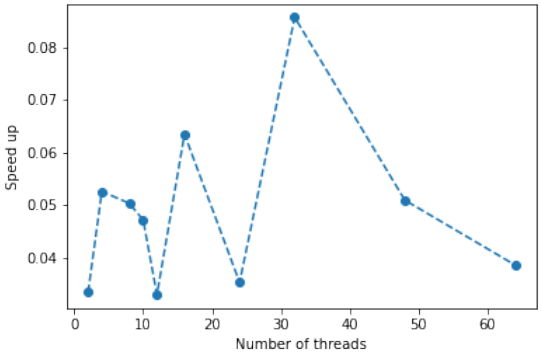

Fig. 23: Overall speedup between the result shown at subsubsection X-B2 and Figure 22

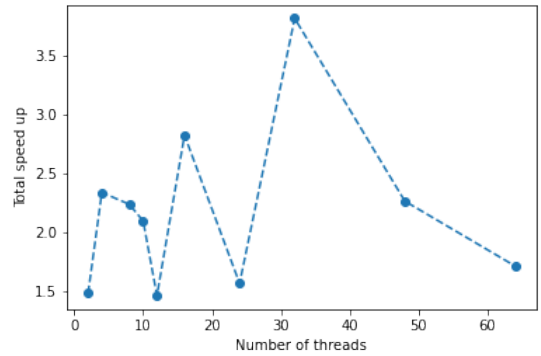

Fig. 24: Overall speedup between the result shown at subsubsection X-B1 and Figure 22

D. Week Node Performance

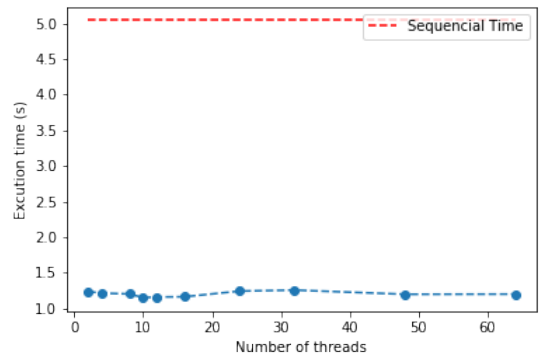

Fig. 25: Execution time of the best possible bucket sort algorithm in function to the nº of threads.

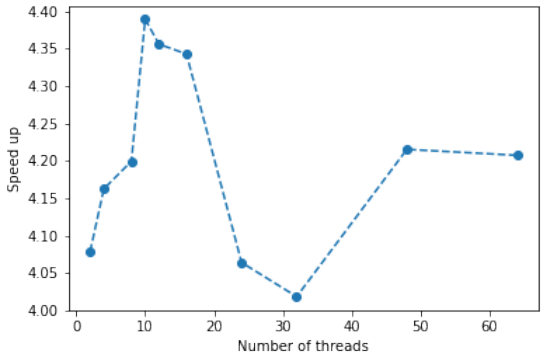

Fig. 26: Overall speedup between the result shown at subsubsection X-B2 and Figure 25