Final Parallel Programming Project

Beatriz Sousa Demétrio
Physics Engineering Student
Minho's University
Braga, Portugal
a92839@alunos.uminho.pt

Carlos Miguel Passos Ferreira
Physics Engineering Student
Minho's University
Braga, Portugal
a92846@alunos.uminho.pt

***Abstract* — This document includes the discussion of the sequential and parallel version of the Bucket-Sort function algorithm.**

***Keywords* — bucket-sort, quick-sort, wall clock time, caches misses, number of cycles and instructions, performance, CPI, OpenMP, threads, core, flag, pragma**

I. Introduction

This work aims to implement and discuss the results obtained by the improved and more efficient version of the Bucket-Sort algorithm, which was provided to us, and to implement and discuss in the same way its parallel version, in which it was created by us. For this, to implement and test the code related to the *Bucket-Sort* algorithm, we use *Cluster* (both in sequential and parallel versions).

Bucket-Sort [3] in general is a sorting algorithm that works by dividing an array/vector into a finite number of buckets. Each bucket is then sorted individually using a different sorting algorithm. Basically, *Bucket-Sort* works as follows: it starts by initializing an array of buckets, which are initially empty; then using the original array, it will include each element in a bucket; then sort all non-empty buckets; finally, puts all non-empty bucket elements into the original array.

II. Sequential version of the bucket-sort algorithm

A. *Preparing the original files*

We start by creating a folder, which we call *bucket_sort*, where we do all the work. Then, inside that initial folder, we create another one, in which we call it *sequencial*. In this second folder, we put the corresponding code of the original Bucket-Sort and is respective main (inside the *bucket_seq.c*) and we also put its respective *MakeFile*. The *MakeFile* used was practically the same as the one provided by the teacher in the second practical class, but it contains the following changes: the ***src*** name was change to *bucket_sequentialsf.c*, because it was where the code and the respective main were written; changed the C compiler flags to ***-std=c99 -O2 -I/share/apps/papi/5.4.1/include***. The compiler used throughout the implementation was the *GNU Compiler Collection*, known as ***gcc***. The *bucket_sequentialsf.c* file is also the file provided by the teacher, but it was changed to include the *Papi* in the *main* so that we can measure times, number of cycles and instructions, and caches misses.

B. *Modifying the original code so that we have a more efficient*

Before we started to change the code, we decided to start by analyzing the code itself, to know which parts took the most time. For this, we use the profiling technique. We also chose to obtain the execution time of the entire program, as well as the number of cycles and corresponding instructions and the L1 e L2 caches misses. With this we were able to verify that we had to make the following changes:

- *Changed the Bubbles function to the Quick-Sort function:* Although *Bubble-Sort* [1] is one of the simplest sorting algorithms, it is also one of the slowest, that is, in terms of efficiency it is a little bad. Basically, in a more simplified way, the *Bubble-Sort* algorithm performs two main tasks that are executed in a loop until the data is fully sorted: comparing adjacent values and changing the position of values when necessary. That's why we decided to put a function that has the same function as *Bubble-Sort* but is more efficient and faster: *Quick-Sort* [2]. We don't use other sort functions such as *Merge-Sort* and *Insert-Sort* because, compared to *Quick-Sort*, they are much less efficient for a very large array values, which is what we aim to have as well. The *Quick-Sort* algorithm uses the "divide and conquer" strategy, that is, we first choose a value that we will call the pivot and then we will split the sequence into two parts: the values smaller than the pivot and the values greater than the pivot. Finally, recursively sort the two already sorted subsequences. Therefore, by implementing *Quick-Sort* instead of *Bubble-Sort* we will greatly reduce the execution time for very large arrays.
- *Added the Free:* [8] Throughout the code of the *Bucket-Sort* and the *main* function, space was allocated in RAM using the *malloc* function, which returned an address (pointer) referring to these information spaces. The generation of disordered vectors and buckets are examples where dynamic allocation was used. Contrary to what happens with local variables, which will no longer have importance after the function is finished, dynamically allocated variables would continue to exist. To reduce memory wastage (to allow better management of the caches), a *free()* was applied that deallocates the portion of memory reserved by *malloc()*, allowing that space to be "freely" used in other applications. Thus, we reduce memory waste, allowing for greater management and use of caches. Therefore, we will have a reduction in terms of caches misses.

C. *Getting and Discussing Results*

As we said before, we did some tests to see if the *Bucket-Sort* function improved by us is more efficient or not.

1) *Unchanged Code:*

We start by analyzing the original code first, that is, without any changes. For that, we profiled it and measured the following variables, while varying program variables such as *tam_bucket*, *num_bucket* and *max*: *wall clock time* (in μsec), which corresponds to the code execution time; the total number of cycles (*PAPI_TOT_CYC*); the total number of instructions (*PAPI_TOT_INS*); number of L1 level cache misses (*PAPI_L1_DCM*); number of L2 level cache misses (*PAPI_L2_DCM*).

We did this for the purpose of seeing what changes we had to make. Therefore, the results we obtained were placed in tables, which are as follows:

	1	5	50	100	500	1000	10000	100000
wall clock time	11u	12u	13u	18u	72u	218u	11140u	153262u
PAPI_TOT_CYC	1536	1802	4507	12504	156687	554358	33236469	4589148137
PAPI_TOT_INS	2336	2492	4927	10937	171422	675462	69457607	6982527151
PAPI_L1_DCM	15	11	21	39	95	208	3199	56607164
PAPI_L2_DCM	1	3	11	10	26	80	380	2280

Table 1 - values of the variables described by varying the tam_bucket and considering num_bucket=10 e max=10 (the code is unchanged)

	1	5	10	50	100	500	900	999
wall clock time	1079u	295u	197u	140u	217u	1105u	1870u	2084u
PAPI_TOT_CYC	3163457	830101	524253	191012	165127	402210	683557	763015
PAPI_TOT_INS	6748546	1350474	675462	159146	105573	141795	219094	238354
PAPI_L1_DCM	254	140	218	498	849	3329	6288	5606
PAPI_L2_DCM	59	35	74	215	404	950	1163	1167

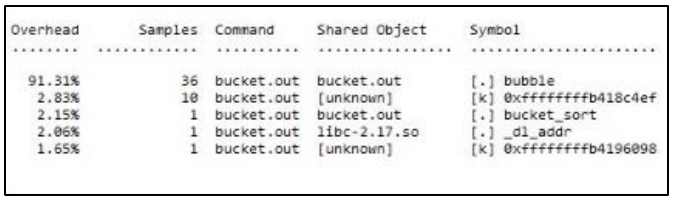
Table 2 - values of the variables described by varying the num_bucket and considering tam_bucket=1000 and max=10 (the code is unchanged)

	10	100	1000
wall clock time	218u	208u	209u
PAPI_TOT_CYC	554358	541193	522720
PAPI_TOT_INS	675462	723320	702287
PAPI_L1_DCM	208	206	208
PAPI_L2_DCM	80	77	69

Table 3 - values of the variables described by varying the max and considering tam_bucket=1000 and num_bucket=10 (the code is unchanged)

We verified with this third table that the *max* variable does not affect the execution time or the other variables at all, unlike *tam_bucket* and *num_bucket*, which greatly affect the efficiency of the code.

After the profiling, we got the following:



```
Overhead       Samples  Command     Shared Object     Symbol
........  ............  ..........  ................  ......................

  91.31%            36  bucket.out  bucket.out        [.] bubble
   2.83%            10  bucket.out  [unknown]         [k] 0xffffffffb418c4ef
   2.15%             1  bucket.out  bucket.out        [.] bucket_sort
   2.06%             1  bucket.out  libc-2.17.so      [.] _dl_addr
   1.65%             1  bucket.out  [unknown]         [k] 0xffffffffb4196098
```

Figure 1 - Profiling obtained by code without changes

With these tables and the profiling result, we can verify the following:

- that effectively what takes the longest to run is the *Bubble* function;
- that we have a lot of caches misses, both in L1 and L2, especially when we increase the values of tam_bucket and num_bucket.

2) *Code with changes to be more efficient*

As we saw in the previous point, we saw that there was a need to change the code to better the referred parts. And so, we decided to apply the changes that we also mentioned earlier. So, getting the previous tables for this new code, we will have the following:

	1	5	50	100	500	1000	10000	100000
wall clock time	11u	11u	13u	14u	26u	42u	845u	53037u
PAPI_TOT_CYC	1893	2119	5515	8852	42147	87433	2291452	158210070
PAPI_TOT_INS	2453	2688	7431	11967	62147	159944	7014667	610380807
PAPI_L1_DCM	6	16	26	35	67	238	3378	127603
PAPI_L2_DCM	0	2	4	6	6	10	293	2798

Table 4 - values of the variables described by varying the tam_bucket and considering num_bucket=10 and max=10 (the code already has the changes)

	1	5	10	50	100	500	900	999
wall clock time	90u	50u	44u	74u	151u	806u	1483u	1559u
PAPI_TOT_CYC	227997	103110	88080	89721	123288	436965	730536	807375
PAPI_TOT_INS	701235	228858	159931	111532	118369	239394	373988	407329
PAPI_L1_DCM	171	181	205	565	1175	4442	7013	7300
PAPI_L2_DCM	5	15	29	88	363	1672	2822	3061

Table 5 - values of the variables described by varying the num_bucket and considering tam_bucket=1000 and max=10 (the code already has the changes)

	10	100	1000
wall clock time	44u	48u	50u
PAPI_TOT_CYC	88080	100251	104915
PAPI_TOT_INS	159931	117920	116465
PAPI_L1_DCM	205	191	188
PAPI_L2_DCM	29	10	24

Table 6 - values of the variables described by varying the max and considering tam_bucket=1000 and num_bucket=10 (the code already has the changes)

Now, in table number 6 we continue to verify the same behavior as when the code had no changes: that increasing the max variable does not affect the code. But we also found that variables such as execution time decrease drastically.

Next, we will relate tables 4 and 5 with tables 1 and 2 through graphics:

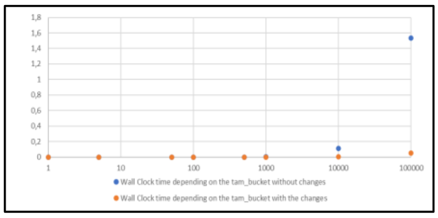


Figure 2 - wall clock time as a function of tam_bucket

With this graph, we were able to observe the variation of execution times as a function of *tam_bucket*, both in the algorithm without any change and in the already modified algorithm. Hence, we can infer that for high values of *tam_bucket*, there is a large variation in the execution time, that is, when the algorithm is already modified, the execution time is much smaller in relation to the algorithm without any change. But when the *tam_bucket* is very small, the execution time difference between the two algorithms is irrelevant. Therefore, we can say this since the algorithm with modifications has the *Quick-Sort* function, as it has exactly this behavior: for high values of arrays, it has a shorter execution time, but for lower values it will have a high runtime value.

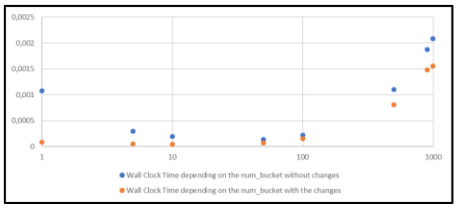


Figure 3 - wall clock time as a function of num_bucket

With this graph, we can observe the variation of execution times as a function of num_bucket, both in the algorithm without any change and in the already modified algorithm. As in the modified algorithm we took advantage of *free()*, this use will make there is a reuse of memory, that is, faster memory levels, thus allowing a reduction in execution time. But still, the execution time is not as low as expected. This is due to the behavior of *Quick-Sort*, because if we have more buckets for the same size, then the arrays that will enter the Quick-Sort are getting smaller and smaller.

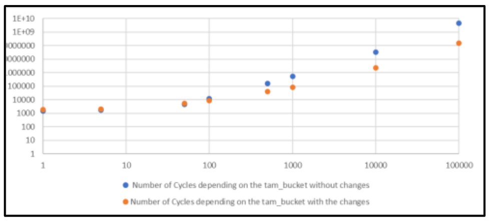


Figure 4 - number of cycles as a function of tam_bucket

As the size of buckets increases, more space is allocated in memory for each bucket, which in turn increases the amount of data requested from caches. From the moment we exceeded the maximum amount allocated in the L1 cache, slightly below 8000 integers (since the L1 cache supports 32 KiB of information), we moved to a slower memory hierarchy level, resulting in an increase in the number of cycles. The difference that can be noticed between the improved version of the function with the version lies mainly in the use of *free()* which makes better use of memories.

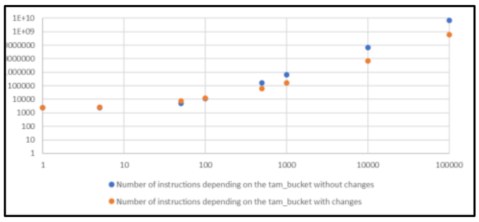


Figure 5 - number of instructions as a function of tam_bucket

As the *tam_bucket* increases, we also increase the input that will pass to the sort functions (*Bubble*, for the algorithm without change, or *Quick-Sort*, for the algorithm with modification). Now, this causes an increase in the number of instructions (as we can see in the graph for the two algorithms). But the difference in the number of instructions between the two algorithms for high values of tam_bucket is because *Quick-Sort* has a higher efficiency for larger arrays, which is why it uses fewer instructions.

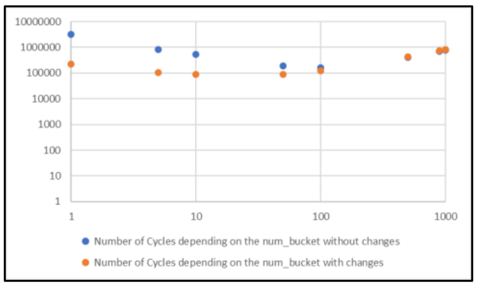


Figure 6 - number of cycles as a function of num_bucket

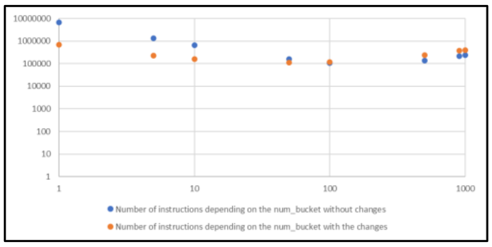


Figure 7 - number of instructions as a function of num_bucket

These graphs in figures 6 and 7 are the opposite of the graphs in figures 4 and 5, respectively. This is since as the number of buckets increases, the input we pass to the sorting functions used becomes smaller and smaller. Therefore, we will have a decrease in the number of instructions (as we can see in the graphs above). The difference in the number of instructions is due to the lack of efficiency of *Quick-Sort* for very small array values. Basically, with the use of *free()* we already know that there will be a reuse of faster memory levels, which is why the number of cycles will be less. But even so, in the corresponding graph we observe that the number of cycles increases again after a certain number (this for the modified algorithm). Now, this is because *Quick-Sort* is inefficient for small array values.

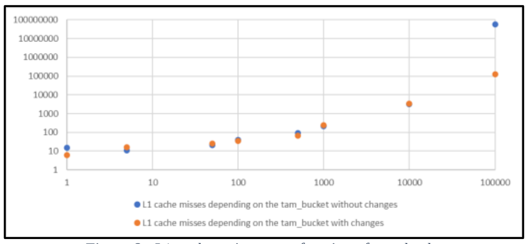


Figure 8 - L1 caches misses as a function of tam_bucket

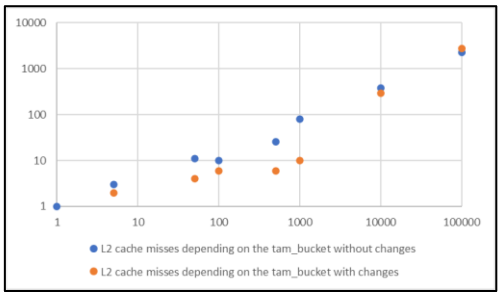


Figure 9 - L2 caches misses as a function of tam_bucket

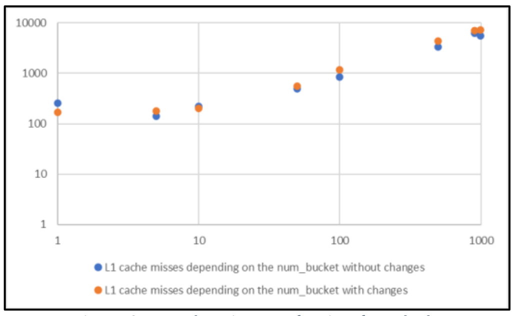


Figure 10 - L1 caches misses as a function of num_bucket

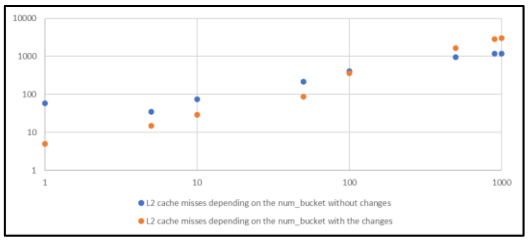


Figure 11 – L2 caches misses as a function of num_bucket

Taking into account the definition of *free()* written above, we have the following:

- *For graphs 8 and 9:* as the *tam_bucket* increases, the space allocated in memory for each bucket will increase, in turn increasing the amount of data requested from the caches, which is why there is also an increase in cache misses. With the addition of *free()*, there will be better memory reuse, which is why there are fewer cache misses;
- *For graphs 10 and 11:* with increasing *num_bucket*, we see an increase in cache misses in both algorithms. This is because increasing the numbers of buckets, there

are more allocated in memory, that is, they will occupy more memory levels, which is why there are misses. The difference between the two algorithms is due to the fact that the modified algorithm has *Quick-Sort* and *free()*.

Having all these metrics, we can now talk about *CPI* [4]. *CPI* is the average execution time of each instruction measured in cycles. This metric is very important to evaluate the efficiency of an algorithm because the smaller it is, the more efficient and faster is. *CPI* is calculated as follows:

$$CPI = \frac{number\ of\ cycles}{number\ of\ instructions} \qquad (1)$$

Which is why we will have the values of the following tables:

- *For changing tam_bucket:*

CPI (without changes)	CPI (modified)
0,6575	0,7717
0,7231	0,7883
0,9148	0,7422
1,1433	0,7397
0.9140	0,6782
0,8207	0,5466
0,4785	0,3267
0,6572	0,2592

Table 7 - CPI results of the unchanged algorithm and the modified algorithm

- *For changing num_bucket:*

CPI (without changes)	CPI (modified)
0,4688	0,3251
0,6147	0,4505
0,7761	0,5507
1,2002	0,8044
1,5641	1,0416
2,8366	1,8276
3,1199	1,9534
3,2012	1,9821

Table 8 - CPI results of the unchanged algorithm and the modified algorithm

We verified that the *CPI* is always smaller in the modified algorithm, and so we can say that it is more efficient and faster than the original one, that is, without alterations.

III. PARALLEL VERSION OF THE BUCKET-SORT ALGORITHM

A. *Preparing and modifying files*

To use parallelize the code, we proceeded to create a folder, called *bucket_sort_parallel*, placed next to the sequential version folder. Inside this new folder, three files were placed, the file "*bucket_par.c*" where the code of the sorting function under study is found, in addition to its main (where we will apply parallelization) a "*Makefile*" adapted for parallelization, like one of the Makefiles used in practical classes, adding the "***-fopenmp***" option to the compilation. By default, the inclusion of the *OpenMP* library in the ".c" file was done in the sequential version of the function, where the line ***#include omp.h*** is in the header of that file.

As in this part of the work the focus will be on reducing the execution time of the program, the "*papi*" function is replaced by one belonging to the used parallelization library known as "***omp_get_wtime()***", whose application in the Code is similar to "*papi*", where its output will be the execution time (in *seconds*) of the selected Code part.

When we are generating random numbers to fill in the array, we are interacting with memory, so we are taking advantage of caches, that is, when bucket_sort needs the array of elements, they are already in the cache, which is not what happens in "real life". Therefore, we should avoid working with the hot cache (with elements already in the cache). We can solve this problem by generating a vector "v1", with a size of 30-40MiB (ensuring that it occupies all caches) between the generation of the unsorted vector "v" and the call of the function "bucket_sort". Thus, when we call bucket_sort on the array "v", it will no longer be in cache, leading to the existence of penalties for cache misses that we intend.

B. *Brief Introduction to OpenMP*

Before moving on to the changes and applications of parallelization in sequential code, let's briefly introduce the concept of *OpenMP*.

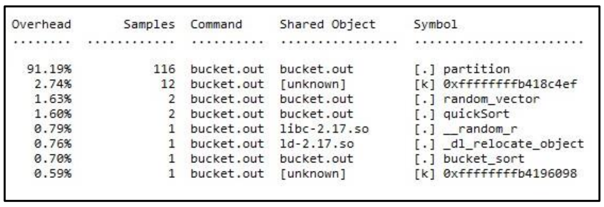
"Open multiprocessing", known as *OpenMP*, is an *API* (application programming interface) specialized in parallel programming using memory sharing through the existence of threads (thread of execution) that can be seen as an abstraction to the software to indicate the tasks that we can perform in parallel with the physical cores.

#pragma omp is the basic functionality of communicating information to the compiler so that it generates code optimized for the *OpenMP* execution environment, which is added to the code depending on the areas intended to be parallelized. Several directives are added depending on the type of parallelization intended.

C. *Modifying the sequential code*

Finished the improvement in the sequential code, we analyzed the profile again in order to check if the part that will occupy most of the execution is still the sort function (now being the *Quick-Sort*).

The following profile was obtained:



```
Overhead       Samples  Command     Shared Object      Symbol
........  ............  ..........  ................   ......................

  91.19%           116  bucket.out  bucket.out         [.] partition
   2.74%            12  bucket.out  [unknown]          [k] 0xffffffffb418c4ef
   1.63%             2  bucket.out  bucket.out         [.] random_vector
   1.60%             2  bucket.out  bucket.out         [.] quickSort
   0.79%             1  bucket.out  libc-2.17.so       [.] __random_r
   0.76%             1  bucket.out  ld-2.17.so         [.] _dl_relocate_object
   0.70%             1  bucket.out  bucket.out         [.] bucket_sort
   0.59%             1  bucket.out  [unknown]          [k] 0xffffffffb4196098
```

Since the ordering zone of the buckets continues to be responsible for the longest execution time, as well as the time that each "*for*" cycle inside the "*bucket_sort*" took to be executed, it was decided to study the parallelization in the third "*for*" cycle within the *bucket_sort* function, using different directives to find the one that will make the program as efficient as possible.

D. *Getting and Discussing Results*

This study was carried out in the *SeARCH* cluster node used in practical classes, known as *cpar*. Analyzing the characteristics of this node, we can conclude that it has 40 CPUs, which is why it supports a maximum of 40 threads [6].

To define the number of threads to be used by *OpenMP*, we used a command defined directly in the terminal, being this: ***export OMP_NUM_THREADS=N***.

Each time we want to vary the number of threads again, we just must use this command.

The third for loop of the *Bucket-Sort.* algorithm is responsible for ordering the elements within each bucket. This action calls on another sorting function: *Quick-Sort.* As mentioned before, given the need to parallelize this cycle, a possible method is to divide the various buckets (even with the elements in disarray) by the various threads. Therefore, the intention will be to sort multiple buckets at the same time.

First, we had to define the parallel region (putting ***#pragma omp parallel*** above the for loop).

Since the purpose of this parallelization is based on the division of work between the various threads, we are not interested in directives such as "*firstprivate*", "*single*", "*master*", "*critical*", among others, because we do not want data sharing between the various threads and the synchronization between them. Therefore, we are left with the following pragmas as an option: "***omp for***", "***omp for schedule(static)***", "***omp for schedule(dynamic)***" and "***omp for schedule(guided)***".

In summary, the characteristics of each pragma are:

- ***for*:** assignment of loop iterations to threads
- ***schedule(static)*:** iterations divided into chunks of size
- ***schedule(dynamic)*:** the chunks are assigned to threads in the team as the threads request
- ***schedule(guided)***: like dynamic but the chunk size decreases during execution

For the <u>scalability study</u>, it was considered that the variables vary in number and max is equal to 10 (Int). The values of the *tam_bucket* variable will be given according to the capacity of each cache. That is, considering the sizes of each cache, we will have the following values for the tam_bucket:

- • ***tam_bucket*** = 6550, considering that an L2 cache has a size of 256KiB and that we are considered integer values (4 bytes in integer);
- • ***tam_bucket*** = 100000, considering that an L3 cache has a size of 25600KiB;
- • ***tam_bucket*** = 1000000, to work outside the L3 cache (ie in RAM).

Following are the results obtained by varying the value of *tam_bucket*:

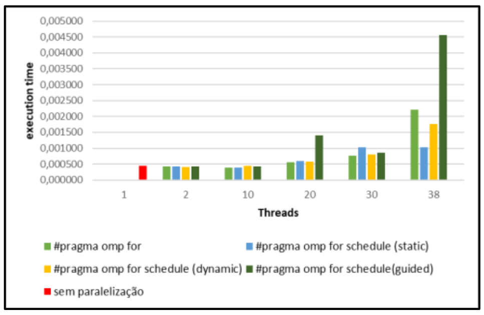


Figure 12 - Graph that shows the variation of the execution time of the various pragmas as a function of the number of threads (tam_bucket=6550)

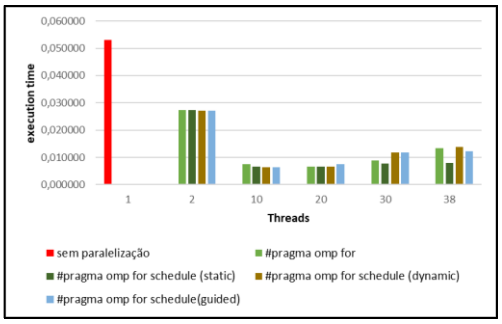


Figure 13 - Graph that shows the variation of the execution time of the various pragmas as a function of the number of threads (tam_bucket=100000)

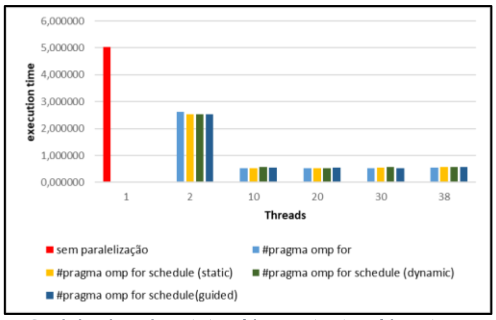


Figure 14 - Graph that shows the variation of the execution time of the various pragmas as a function of the number of threads (tam_bucket=1000000)

With the use of parallelization, there is a decrease in the execution time for high values of *tam_bucket* (which occupy the L3 cache and RAM) and that impairs the execution time for low values of *tam_bucket* (which occupy the L2 cache).

For high values of *tam_bucket*, we found that there is a shorter execution time for a number of threads between 10 and 20, but even for values greater than these, there is also a decrease in execution time compared to when using only one thread, but not it's so significant. As for the lower values of *tam_bucket*, we see that the existence of threads impairs parallelization, that is, the greater the number of threads, the longer the execution time.

But since we are looking at parallelization, the most correct term to check the efficiency of the code is not the execution time, but the speed up [7]. The speed up corresponds to the ratio between the time obtained by the sequential code (or the time obtained by only one thread) and the time obtained by the parallelized version, varying the number of threads. Since it is expected that the execution time obtained by the parallelized version is less than the time obtained by the sequential version (because we do the parallelization to make the code more efficient, that is, for it to have less execution time), then it does not sense when the opposite happens, because if we add threads and the code gets slower, then parallelization is not an option. Thus, we can conclude that the speed up must always be a number greater than 1.0 and that the higher the better, it means that the program is more efficient, that is, it has a much faster execution time compared to the time obtained by the sequential version.

Therefore, applying this speed up concept we obtain the following graphs:

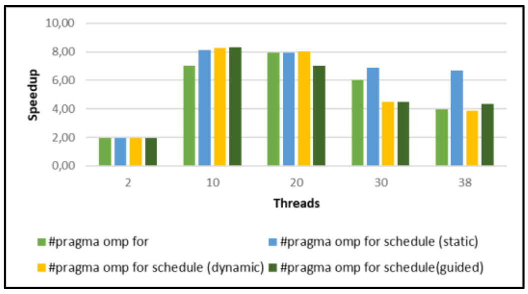


Figure 15 - Graph that shows the variation of the speed up of the various pragmas as a function of the number of threads (tam_bucket=100000)

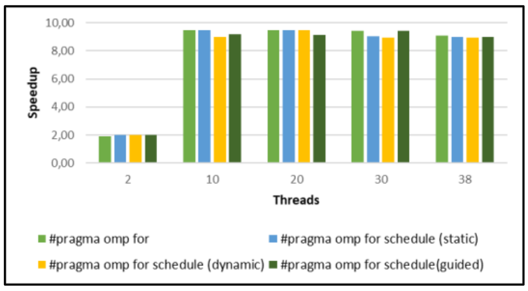


Figure 16 - Graph that shows the variation of the speed up of the various pragmas as a function of the number of threads (tam_bucket=1000000)

As we wrote earlier, it makes no sense to calculate speed up when the running time of the parallelized version is longer than the running time of the sequential version. Therefore, a speed up graph does not appear as a function of the number of threads for *tam_bucket*=6550 because we verified exactly that and with this, we can also say that with low values of *tam_bucket*, there is no logic to apply parallelization, because it will only harm.

But analyzing the graphs above (from figures 15 and 16), we can say that the speed up has very high values when the number of threads is between 10 and 20. But in the graph of figure 16, we see that for a higher number of threads at 20, there is also no big difference in speed up values, contrary to what happens in the graph of figure 15. With the observation of these graphs, we can say that the higher the tam_bucket value, the greater the speed up value, the which in turn means that the use of parallelism was noticed more significantly for larger tam_bucket values.

Regarding the use of pragmas, if we look at the two graphs presented, we see that, on average, the use of *#pragma omp for schedule(static)* is the most suitable for parallelizing this third for cycle. This is due to the fact that in this pragma there is a load sharing between the threads of the iterations but at irregular intervals (as it is done by default). But it is also worth noting that the pragmas "*omp for*" and "*omp for schedule(dynamic)*" do not fail to present interesting values, that is, they also present high values of speed up.

In conclusion, we found that the results show an improvement when the *tam_bucket* is large enough to be a beneficiary of the parallel version of the *Bucket-Sort* code. This is when it is a communication, synchronization overhead between the parallel threads is overcome by well. Otherwise, the sequential version by a single thread is preferable when the tam_bucket values are small.

As these results were obtained on the system with 4 CPU's, setting the number of threads smaller or having a performance boost above 40 does not help thread performance, using regardless of the number of CPU's can degrade overall system performance starting and terminating threads and race condition for embedded hardware resource. Cache limitation and memory-sharing architecture inducing the frequent memory of cache data and the main cache memory.

REFERENCES

[1] Ordenação de Arrays em C com o método Bubblesort - Bóson Treinamentos em Ciência e Tecnologia (bosontreinamentos.com.br)

[2] Ordenação Quicksort (up.pt)

[3] Bucket sort – Wikipédia, a enciclopédia livre (wikipedia.org)

[4] Aula06a.pdf

[5] QuickSort (With Code) (programiz.com)

[6] c - Multiple threads and CPU cache - Stack Overflow

[7] https://hmf.enseeiht.fr/travaux/CD0001/travaux/optmfn/micp/reports/s13itml/theory.htm

[8] C Dynamic Memory Allocation Using malloc(), calloc(), free() & realloc() (programiz.com)

[9] slides of theoretical parallel programming classes