Trabalho Prático – Computação Paralela

Diana Ferreira
PG46529 - MEI
Universidade do Minho

Francisco Peixoto
PG47194 - MEI
Universidade do Minho

***Abstract* - This report is elaborated under the subject of Parallel Computing, and it intends to showcase two different implementations of a sorting algorithm, a sequential version and a parallel version using C and OpenMP.**

Keywords - OpenMP, Shared Memory, Parallel Computing, Bucket-Sort, Sorting, Quick-Sort.

I. Introduction

This project is related to the subject of Parallel Computing, it allows the group to apply the knowledge gathered during the semester. It is intended to approach the concepts of parallel programming, by taking an established algorithm "bucket-sort" and trying to increase its performance while using OpenMP, the group will design and develop its own version of bucket-sort, for both the sequential and parallel versions. In the end, the group will showcase the results and study both the sequential and parallel version of the developed algorithm. To ensure correct and valid reports on the test values, all tests were done multiple times, using randomly generated numbers to fill in the arrays before sorting, after the algorithm is applied, the sorted array is then validated as either sorted or not sorted, to make sure the algorithm worked.

II. Sequential Bucket-Sort Algorithm

In this section, the group will briefly describe the development of the sequential version of the algorithm, in C, that sorts a vector of integers. The Bucket-Sort succinctly consists in defining an initially empty number of buckets, where elements from an array will be sorted and distributed among the existing buckets. Then, the content of each bucket (elements) will be sorted using the Insertion Sort algorithm. Thus, the sequential program consists in the following steps:

- Variable definition and input (if values are not input by the user, it will use predefined values);
- Allocation of memory;
- Calculate and assign a fix number of elements for each bucket;
- Distributing the numbers from the original array to the proper buckets;
- Sorting each bucket using a sorting algorithm.

To implement the algorithm the group first created the structure Bucket.

```
typedef struct bucket {
    int n_elem;
    int start;
    int index;
}*Bucket;
```

Fig. 1 - Bucket Structure

The elements present in this structure, i.e. "n_elem", "start" and "index", will be defined in the next subchapter.

After that a bucket-sort function was created with the necessary implementations for it to work. These implementations will also be explained in the next subchapters.

A. Variable definition

Variable	Type	Definition
array_size	int	Length of the array to sort given by the user.
n_baldes	int	Number of buckets given by the user or set as default: (10), in case of input value is invalid.
max	int	Max number for random generated array elements. This number is given by the user or set as default: 1000, in case of input value is invalid.
baldes	struct *	Array of struct bucket.
array	int *	Responsible for storing all randomly generated integers.
array_baldes	int *	Responsible for storing all buckets.
width	int	Width of each bucket: `(int)max/n_buckets;`
n_elem	int	Number of elements stored in each bucket.
start	int	Each bucket has a starting position in "array_baldes".
index	int	Each bucket has an index position, where it's next integer will be stored.
b_index	int	Index of bucket position in "array_baldes".

Table 1 - Variable definition

B. Allocation of Memory

To dynamically allocate a single large block of memory with the specified size (array_size) and return a pointer to the beginning of the block, we used "malloc" or memory allocation for the integer's pointers "array" and "array_baldes". In case of the struct "baldes", we used "calloc". With this, we can use our memory space more efficiently. In figure 2, we can observe the process of allocating memory with "malloc" and "calloc".

```
array = (int*)malloc(sizeof(int) * array_size);
array_baldes = (int*)malloc(sizeof(int) * array_size);
baldes = (struct bucket*)calloc(n_buckets, sizeof(struct bucket));
```

Fig. 2 - Memory Allocation

In respect to the "array", we will initialize the array with the requested size.

The "array_baldes" has the same size as the initial array to ensure resource efficiency.

The "buckets" is a structure based on the predetermined number of buckets.

C. *Calculate and assign a fix number of elements for each bucket*

To calculate and assign a fix number of elements for each bucket, we implement the following code (Illustration 3):

```
for (i = 0; i < array_size; i++) {
    j = array[i] / width;
    if (j > n_buckets - 1)
        j = n_buckets - 1;
    baldes[j].n_elem++;
}
```

Fig. 3 - Calculate nº of elements for each bucket

In this loop, we have the buckets set to the proper sizes, with the "if" serving to ensure that all elements of the array are assigned to a bucket.

D. *Distribution of the array numbers to the appropriate buckets*

```
baldes[0].index = 0;
baldes[0].start = 0;
for (i = 1; i < n_b; i++) {
    baldes[i].index = baldes[i - 1].index + baldes[i - 1].n_elem;
    baldes[i].start = baldes[i - 1].start + baldes[i - 1].n_elem;
}

int b_index;
for (i = 0; i < n; i++) {
    j = array[i] / w;
    if (j > n_b - 1)
        j = n_b - 1;
    b_index = baldes[j].index++;
    arrayb[b_index] = array[i];
}
```

Fig. 4 - Distribution of array elements

Initially, for each bucket, we have the configuration of the start and index variables, with "baldes[0].index = 0" and "baldes[0].start = 0".

In the first loop, we also have the configuration of the start and index variables for each bucket, but now with the respective values.

In the second loop the distribution and storage of elements into their designated buckets takes place.

E. *Sorting with a sorting algorithm*

Since the elements inside the buckets are unordered, to sort them we tried to use the Quick-Sort Algorithm, with the function qsort() from the standard library "stdlib.h" in C [1] and the non-local Quick-Sort algorithm, with the common "partition" and "swap" function [2] (figure 5). But we realized Insertion-Sort [3] was better.

Initially, we came up with the conclusion that there was a time discrepancy between our implementation of Quick-Sort and the qsort() function provided by the library.

Our implementation of Quick-Sort was about 20% faster for number of elements up to 20 million, and the qsort() function performed better for larger number of elements, in general.

```
if(array_size < 20000000){
    for (int i = 0; i < n_b; i++) {
        low = baldes[i].start;
        high = baldes[i].start + baldes[i].n_elem - 1;
        quickSort(array_baldes, low, high);
    }
} else {
    for (int i = 0; i < n_b; i++){
        qsort(array_baldes+baldes[i].start,
            baldes[i].n_elem, sizeof(int), cmpfunc);
    }
}
```

Fig, 5 - Sorting elements with Quick-Sort

But after testing we realized that out of all the examples we tried to simulate, the Insertion-Sort algorithm was better than both Quick-Sorts by high margins up to twice as fast, on average.

Insertion-Sort despite having better averages, sometimes it spikes worst case scenarios, being much slower, that happened around 30% of all test subjects.

III. Parallel Bucket-Sort Algorithm

Using the sequential version described in the previous chapter, we made changes to parts of the code to convert the old implementation in order to develop the new one with shared memory and OpenMP. In this section, the group will briefly describe the development and design of the parallel version of the algorithm, in C. The parallel program consists in the following steps:

- Variable definition and input (if values are not input by the user, it will use predefined values);
- Allocation of memory;
- Distributing of the data among the threads;
- Setting the right starting position for both global and local buckets in the Bucket Array;
- Sorting each global bucket using quicksort.

To implement the algorithm the group used once again the structure Bucket.

```
typedef struct bucket {
    int n_elem;
    int start;
    int index;
}*Bucket;
```

Fig. 6 - Bucket Structure

We will skip on the variable definitions of previously mentioned variables, as some variables share common definitions with the sequential version of the algorithm.

A. *Variable definition*

Variable	**Type**	**Type**
num_threads	int	Sets the number of Threads.
local	int	Helps make sure every element gets stored in a local bucket.
real	int	Helps keep track of where to store new elements in local buckets.
my_id	int	This variable is in charge to keep track of the id of the Thread running, to make sure the correct values are stored in local buckets.
global_n_elem	int *	Array that contains the number of elements for each bucket.

Variable	Type	Type
global_starting _position	int *	Array that contains the starting positions in “array_baldes”.

Table 2 - Variable definition

B. *Allocation of Memory*

The goals of using memory allocation are the same as presented in the sequential version (subchapter B in chapter 2), so in figure 7, we can observe the process of allocating memory with “malloc” and “calloc”.

```
array = (int *) malloc(sizeof(int) * array_size);
array_baldes = (int *) malloc(sizeof(int) * array_size);
baldes = (struct bucket *) calloc(n_baldes * num_threads,
         sizeof(struct bucket));
```

Fig. 7 - Memory Allocation with malloc and calloc

In respect to the “array”, we will initialize the array with the requested size. The "array_baldes" has the same size as the initial array to ensure resource efficiency. The “buckets” is a structure based on the predetermined number of buckets * number of threads, to create each local bucket.

```
memset(global_n_elem, 0, sizeof(int) * n_baldes);
memset(global_starting_position, 0, sizeof(int) * n_baldes);
```

Fig. 8 - Memory Allocation with memset()

As we can see in figure 8, concerning the variables "global_n_elem" and "global_starting_position" we use the memset() function, which sets the first num bytes of the block of memory to the specified value of 0, in both arrays.

C. *Distribution of the data among the threads*

As we can see in the following code (figure 9), the whole Bucket-Sort Algorithm is running on #pragma omp parallel, meaning that we managed to run most instructions with parallelism.

```
t = omp_get_wtime();

#pragma omp parallel
{ ... }

t_exec = omp_get_wtime() - t;
```

Fig. 9 - #Pragma omp parallel

In figure 10, to ensure the correct distribution of elements across the local buckets (meaning the buckets in each Thread), we make the variable responsible for iterations and the variable responsible for assigning each element to a bucket private.

Then the loop can safely run on multiple Threads.

In the end we calculate de global number of elements in each Thread and to make sure the code doesn’t run any further, we use #pragma omp barrier, so that all the Threads must wait for all the work to be done before proceeding any further.

```
#pragma omp for private(i, local)
for (i = 0; i < array_size; i++){
  local = array[i]/width;
  if (local > n_baldes-1)
    local = n_baldes-1;

  real = local + my_id * n_baldes;
  baldes[real].n_elem++;
}

int local_sum = 0;
for (i = my_id; i < n_baldes*num_threads; i = i + num_threads){
  local_sum += baldes[i].n_elem;
}

global_n_elem[my_id] = local_sum;

#pragma omp barrier
```

Fig. 10 – Calculation of elements across the local buckets

D. *Setting starting position for global and local buckets*

Setting the starting position for the global buckets can’t be run with multiple Threads, as each part of the loop depends on previous generated values, in order to make it work we use #pragma omp single, that allows the following instructions to run using just one Thread, we could also use #pragma omp master, to run it on the master Thread, but then we would have to use #pragma omp barrier and single already does that (figure 11).

```
#pragma omp single
{
  for (i = 1; i < n_baldes; i++){
    global_starting_position[i] = global_starting_position[i-1]
        + global_n_elem[i-1];
    baldes[i].index = baldes[i-1].index + global_n_elem[i-1];
    baldes[i].start = baldes[i-1].start + global_n_elem[i-1];
  }
}

for (i = my_id + n_baldes; i < n_baldes * num_threads;
     i = i + num_threads){
  int previous_index = i - n_baldes;
  baldes[i].start = baldes[previous_index].start
      + baldes[previous_index].n_elem;
  baldes[i].index = baldes[previous_index].index
      + baldes[previous_index].n_elem;
}

#pragma omp barrier
```

Fig. 11 – Setting starting position for global/local buckets

```
#pragma omp for private(i, b_index)
for (i = 0; i < array_size; i++){
  local = array[i]/width;
  if (local > n_baldes - 1)
    local = n_baldes - 1;

  real = local + my_id * n_baldes;
  b_index = baldes[real].index++;
  array_baldes[b_index] = array[i];
}
```

Fig. 12 – Distribution of array elements

E. *Sorting each global bucket*

Just like in the sequential version, we started by using qsort() [1], but as we were running tests we realized Insertion-Sort [3] is faster around 66% of the time (with array_size lower than one million) with up to 2.0 speed up on average.

Each instance of the sorting for loop runs independently so we can easily use #pragma omp for, given that the iteration variable is set to private.

```
if(array_size < 2000000){
  #pragma omp for private(i)
  for (i = 0; i < n_baldes; i++){
    insertionSort(array_baldes+global_starting_position[i], global_n_elem[i]);
  }
} else {
  #pragma omp for private(i)
  for (int i = 0; i < n_baldes; i++){
    qsort(array_baldes+global_starting_position[i], global_n_elem[i], sizeof(int), cmpfunc);
  }
}
```

Fig. 13 – Sorting each global bucket using the most appropriate sorting algorithm

IV. Testing and Analysis

Overall testing was done on a machine provided with the following specs:

- CPU with 6 cores 12 threads running @4.5Ghz;
- 384KB L1 cache;
- 3MB L2 cache;
- 32MB L3 cache;
- 32GB RAM.

We set multiple tests on both the sequential and parallel versions of the code, running multiple algorithms and approaches to OpenMP, most of the test runs were saved to .csv files and the group tried to showcase the results in excel spread sheets.

The most important metric was runtime, but we also considered the resources used to achieve it.

A. *Testing on the Sequential Version*

To test the sequential version of the code developed the group used different approaches to the sorting algorithm of each bucket, the three different algorithms used were previously mention (the group's implementation of Quick-Sort, the library qsort() function, and Insertion Sort).

Most of the test runs were compiled into excel spread sheets, and they were originated using the teste.c file, that allows for quick testing of the code without manual input of the variables.

We tried to cover a wide range of inputs, testing with the range of 10000 to 50 million elements in the array, with different number of buckets, ranging from 10 buckets all the way up to 10000 buckets.

B. *Analysis on the Sequential Version*

For smaller array sizes like 500 or 1000, Insertion-Sort clearly outperforms Quick-Sort. As shown in the excel spread sheet Insertion-Sort is better around 60% of the time, but the worst cases make it, so the average is a bit higher than Quick-Sort. For thar reason we concluded that Insertion-Sort shouldn't be use with higher number of buckets where it gets its performance decreased.

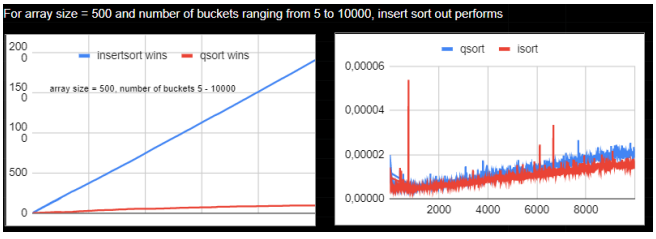

Fig. 14 – Comparison between Quick-Sort and Insertion-Sort (array = 500)

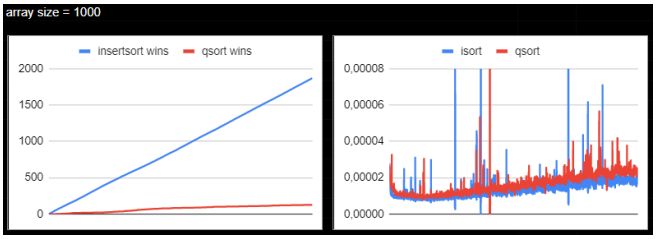

Fig. 15 – Comparison between Quick-Sort and Insertion-Sort (array = 1000)

C. *Testing on the Parallel Version*

To test the parallel version of the code developed the group first tried the same approach mentioned previously, with all the proper testing, using the basic #pragma omp parallel for the sorting of the individual buckets.

After that, the approach was introducing parallelism to the whole implementation of the algorithm, using private variables to safeguard the design.

We tried to cover a wide range of inputs, testing with the range of 100000 to 50 million elements in the array, with different number of buckets, ranging from 100 buckets all the way up to 90000 buckets.

D. *Analysis on the Parallel Version*

When comparing different sorting algorithms we quickly realized that Insertion-Sort cant process large array size, but it's most faster than Quick-Sort on smaller scales (like shown in figure 16).

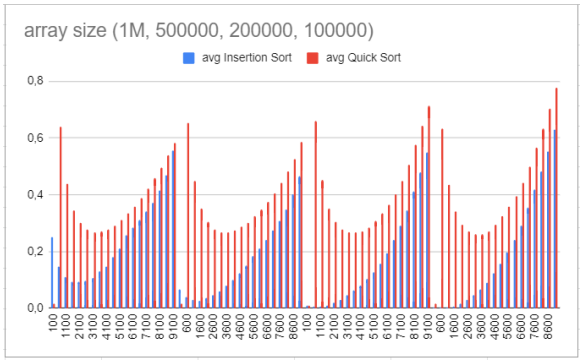

Fig. 16 – Comparison between Quick-Sort and Insertion-Sort (1M, 500000, 200000, 100000).

The group also notices the performance boost is at its highest around 2000/4000 buckets, as shown in the first 2 images the performance decreases drastically with high number of buckets, as takes the testing computer too much to deal with high number of threads.

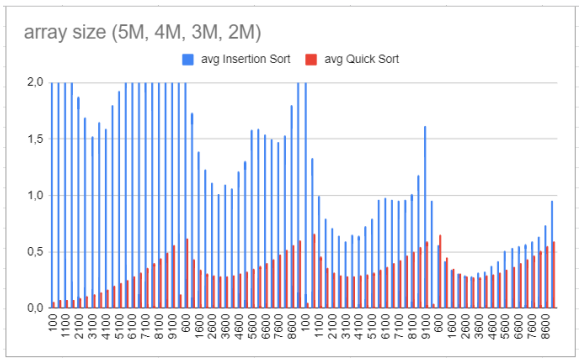

Fig. 17 – Comparison between Quick-Sort and Insertion-Sort (5M, 4M, 3M, 2M)

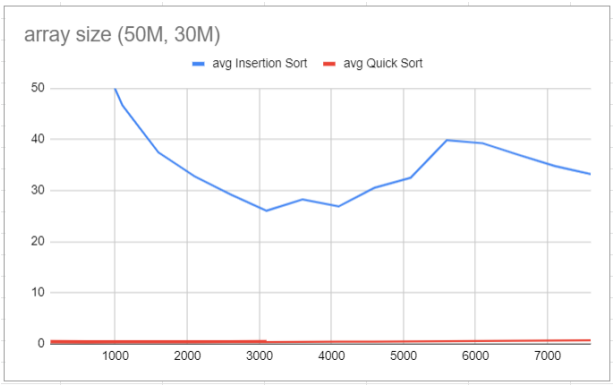

Fig. 18 – Comparison between Quick-Sort and Insertion-Sort (50M, 30M)

Even on high number array size, the Quick-Sort version of the algorithm averages away below one second.

As we were developing with the OpenMP library we thought about different ways to introduce parallelism and ultimately concluded that the whole code could be run in parallel except for the one #pragam omp single instance, previously mentioned.

It's important to note that creating high number of threads comes with a big cost, so we should only use high number of threads to sort a high number of elements as the test showcase that the speed up obtained from parallelism is already high with low number of buckets.

As we mentioned in subchapter D in chapter 3, in part of the code we could also use #pragma omp master, to run it on the master Thread, but then we would have to use #pragma omp barrier and single already does that.

We also mentioned how to introduce parallelism, with reference to the #pragma used throughput the implementation steps of parallel version in chapter 3.

V. Scalability

A. *Comparison between sequential and parallel version*

As expected, there is a significant increase in performance when switching to the parallel version of the code, in the yellow line we can see the speedup that is calculated by how many times faster the parallel version is comparing to the sequential version.

In the first image, the speedup rate is around 9.0 as the number of buckets is really close to how many threads there was in the testing computer, taking close to full potential in terms of how much faster this version could be.

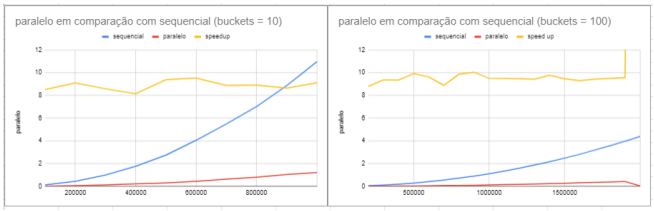

Fig. 19 – Comparison between sequential and parallel version (10/100)

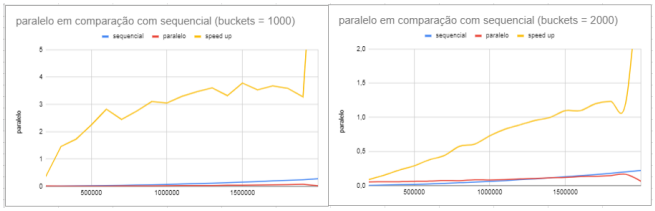

Fig. 20 – Comparison between sequential and parallel version (1000/2000)

In the figure 20, we can see that in really big number of elements the speedup increases a lot, but using buckets equal to 2000 the speedup is below 1.0 for the majority of the graph, which was surprising.

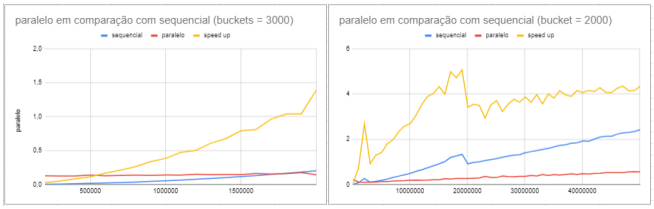

Fig. 21 – Comparison between sequential and parallel version (3000/2000)

In the previous figure, the test showing on the right was made with a really large number of elements in the array, which increases the speedup, showcasing that the parallel version is superior.

B. *Scalability tests on cluster*

To further improve our testing data, we used the cluster to run multiple tests with the purpose of taking advantage of the clusters.

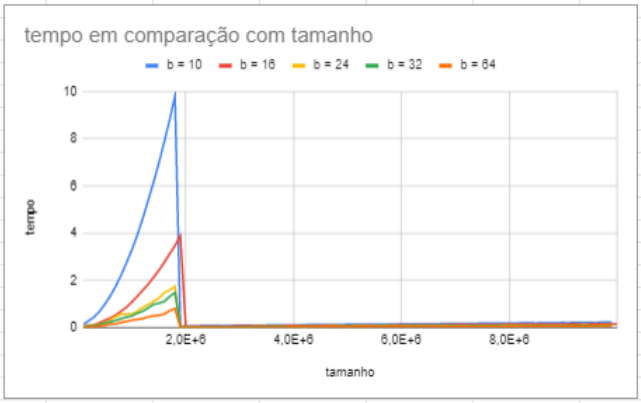

Fig. 22 – Tests on cluster (time vs size)

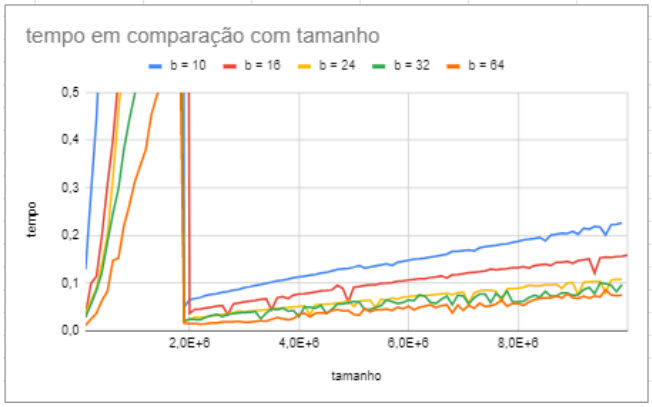

Fig. 23 – Tests on cluster (time vs size)

We ran tests on cluster for number that will be possible threads counters, like 10,16,24,32,64. As it is shown in the previous images the time significantly drops after hitting the to 1000000 marc as that is were quick-sort start sorting the algorithms, which shows that parse results might have been due to hardware limitations, in the cluster the quick-sort version is far superior.

REFERENCES

1. www.tutorialspoint.com/c_standard_library/c_function_qsort.htm *(references)*
2. https://www.geeksforgeeks.org/quick-sort/ *(references)*
3. https://www.geeksforgeeks.org/insertion-sort/ *(references*)