Parallel Computation

Guilherme Martins, *PG47225*,
João Pereira, *PG47325*

Practical Work

January 14th, 2022

Abstract: Report about the parallel computation in Bucket Sort algorithm. It's made a comparison between the sequential and parallel version of it. Then some analysis and results are shown and conclusions about the scalability of the implementation.

Index Terms:

1. Introduction

The purpose of this assignment is to learn about parallel computation applied to the Bucket Sort algorithm, using OpenMP and the C programming language. Bucket Sort receives an array of numbers and returns the same array with its elements sorted.

In this report we present the implementation of the sequential version of Bucket Sort algorithm and also its parallel version, based on four main tasks: initialize an array of buckets; put each of the array elements in some bucket according to a specific criterion; sort the elements of each bucket and finally copy the sorted elements of each bucket to the final array.

Before the parallel version implementation we consider some theoretical analysis about which parts of the code should be parallelized and why we think it should work.

Finally we compare the results between the sequential and parallel implementations based on some metrics like wall clock time, wall clock time in the parallel section, cache misses, number of threads, node of the cluster and the size of the inputs (number of buckets and array size).

2. Development of the sorting algorithm

In order to develop the sequential version of the Bucket Sort algorithm, we started by including a function from the practical lessons that allows the count of the hardware events from the PAPI library.

That function *main.c* was adapted to include the initialization of our array of numbers and to call our main function *sequential_bs.c* that has the bucket sort algorithm. The array was initialized according to this function, that generates multiple numbers in the range 0-size.

```
int init_array(int **arr, int size)
{
    int i;
    *arr = (int *)malloc(size * sizeof(int));
    for (i = 0; i < size; i++)
        (*arr)[i] = (int)rand() % size;

    return 1;
}
```

It is important to consider the main variables used in the code: **N** is the size of the input array; **BUCKET** is the number of buckets to use; **time** is the wall clock time measured with openMP in the zone where each bucket orders its elements sequentially; **buckets** is the array of buckets, each of these have an array of elements inside and finally **arr** is the input array.

2.1. Buckets vector initialization

```
// Buckets - array of buckets, each bucket has an array with the elements inside it
    int **buckets = (int **)malloc(BUCKET * sizeof(int *));

    // 1. Initialize buckets
    for (i = 0; i < BUCKET; i++)
        buckets[i] = NULL;
```

2.2. Setting each number in a bucket

A variable was created to store the actual number of elements in each bucket, it is an array called **num_elements**.

```
//2. Divide the elements by the buckets
    for (i = 0; i < N; i++)
    {
        // Find the bucket index for this element (arr[i])
        double test = ((double)arr[i]) / range;
        int b = (int)floor(test);
        if (b > BUCKET - 1)
            b = BUCKET - 1;

        buckets[b] = (int *)realloc(buckets[b], (num_elements[b] + 1) * sizeof(int));
        buckets[b][num_elements[b]] = arr[i];
        num_elements[b]++;
    }
```

2.3. Sorting the buckets

The chosen algorithm to sort each bucket was quicksort, because it is one of the most efficient sorting algorithms. Radix Sort was also an option but was not working too well in this scenario. The time is measured here, to later compare it with the parallel version time.

```
//3. Order the elements of each bucket
    *time = omp_get_wtime();
    for (i = 0; i < BUCKET; i++)
    {
        if (num_elements[i] > 1)
            quickSort(buckets[i], 0, num_elements[i] - 1);
    }
    *time = omp_get_wtime() - *time;
```

2.4. Placing the numbers in the final vector

```
//4. Fill the final array in order
    int count = 0;
    for (i = 0; i < BUCKET; i++)
    {
        for (j = 0; j < num_elements[i]; j++)
            arr[count++] = buckets[i][j];
        free(buckets[i]);
    }
```

The performance of this algorithm depends on a lot of things. First it depends on the sorting algorithm used, that's why we thought about Radix Sort at the first place, but the Quicksort is also good and we decided that it is simpler to include in our code. Then it also depends on the CPI, the number of instructions and the time to execute one clock cycle (cc):

$$T_{exec} = CPI * \#I * cc$$

According to this well-known formula, the cc might be slightly the same if we change the code, but $\#I$ and CPI can change if we make some optimizations. Our suggestion is to explore parallelization using more than one thread and more than one core to reduce the CPI of some instructions, by sharing the same work by more units (threads).

3. Parallel version development using OpenMP

Before making any adjustment to the previous version, first it has to be considered which set of instructions can be done in a parallel way and which doesn't. Some important issues to cover are:

- **Parallelism in bucket initialization:** First we thought about this possibility. Although many threads could initialize different buckets at the same time, using a private index, for instance, it could cause a unnecessary cost to our program with aditional thread initialization. This section is really simple and we decided to not consider parallelism in this part of the algorithm.
- **Parallelism in bucket filling:** In this case we needed to control the memory accesses, because this loop has instructions that access to the array and also to the bucket. Allowing more threads to do this work might be bad because some of them could be filling the same bucket at the same time. This could be done with private variables but we decided it was not a critical section to parallelize.
- **Parallelism in bucket sorting:** This is the main section that sould use OpenMP directives to parallelize, because each independent bucket needs to sort his elements, and this work could be divided by some threads without a major problem. The performance can be improved in some cases, and might be improved in a general way. Each bucket has its own array with his elements inside, there is no concurrency problems if the number of threads is less or equal to the number of buckets. We decided to use parallelism here.
- **Parallelism in filling the final array with the sorted values:** This section could be a disaster when using parallelism, because it has fatal concurrency problems, in fact all of the elements need to be in a specific order and this work must be done sequentially.

Parallel Bucket Sort final version covers parallelism only in the individual bucket sorting section. We implemented $pragma\ omp\ for\ schedule(static)$ to force all threads to have the same amount of work (in this case same number of buckets to sort) and try to balance the parallel work, like this:

```
//3. Order the elements of each bucket
    *time = omp_get_wtime();
#pragma omp parallel num_threads(num_threads)
#pragma omp for schedule(static)
    for (i = 0; i < BUCKET; i++)
    {
        if (num_elements[i] > 1)
            quickSort(buckets[i], 0, num_elements[i] - 1);
    }
    *time = omp_get_wtime() - *time;
```

num_threads holds the number of threads used in the parallelism, we decided to use 3 possibilities, 10 threads, 16 threads or 20 threads. This issue is covered in more detail in the next section of the report.

4. Tests and Results

In this section are presented different approaches to test and measure the performance of the algorithm, comparing the two versions of it and comparing the results given according the number of threads available from the processor, number of cores, pragma omp directives used and even the node of the cluster where the tests were made. All the testes were done on the cluster, using s7edu.di.uminho.pt or s7sci.di.uminho.pt.

A script was built to run our programs and to set the input values, which are the dimension of the array and the number of buckets created, the script is Run.sh.

4.1. Approaches

- **Tests in cpar partition:** First we made some tests on the general partition cpar, that we used mainly in the practical classes. Then the command $srun --partition = cpar\ cat\ /proc/cpuinfo$ displayed the characteristics of each processor. According to that information, each processor has 20 threads and 10 cores. This approach is based on parallelism with 20 threads in the loop that sorts each bucket. In this scenario we made the tests without using the schedule(static). Using 20 threads was a good choice because it is the maximum in that processor, over that can use hyperthreading and slow the performance.
- **Tests in day partition, using the node compute-134-16:** In order to increase the number of tests and to test different characteristics that can affect the algorithm performance, we decided to choose a specific node of the partition day. We tried to use the compute-145-2, to take advantage of 16 cores in each processor, but an error ocurred while trying to allocate resources. Using the same command as in the cpar case, we checked the characteristics of this node, in this case it has 16 threads available and 8 cores by processor. We decided to add a dynamic way of allocating threads, based on the size of the input array:

```
if (BUCKET < 1000)
{
    num_threads = 10;
}
else
{
    num_threads = 16; //Without using specific node, this value is 20
}
```

  Two different tests were made here, first we measured only the time to execute the parallel section, using 16 threads and schedule(static), and in the other one we used 16 threads and schedule(static), but measuring the total time of execution, with all of the instructions.

4.2. Results

After thinking about those 3 main approaches, we measured the times for that situations and got 3 different tables attached in the attachments section. In tables I and II we can see that the difference between the sequential and parallel time is not so big, but in this scenario we measured the total time, and a lot of different factors can influence the performance, for instance, we are using more threads to do more work, but they need to be initialized and that also consumes time. The huge number of buckets can also be a problem because if it is near from the dimension of the array, the algorithm loses is effect, because the buckets can hold a few number of elements and do less useful work.

In table III we see a big improvement in the loop that covers the sorting of each bucket, because we are just considering that zone and not the overall case, which includes initialization of the variables, initialization of the threads and other time consuming aspects.

An additional test was seeing the cache behaviour with large array dimension. As we can see in 4 in cache L1 we have room for $(32 * 1024)/4 = 8192$ integers, in cache L2 $(256 * 1024)/4 = 65536$ integers and in cache L3 $(8192 * 1024)/4 = 2097152$. So if we only used the cache for storing the elements of the array we would have space for 2171240 elements. We also need to see that buckets will also store elements that possibly could go to cache, and this available quantity is too small. In all of our measurements we saw that a lot of cache misses were showing up, and this means that most of time, in large arrays, the hit is only found in RAM, and this consumes a lot of execution time. That's why the execution time for huge arrays is always high even with parallelism.

5. Implementation scalability

After performing tests with different numbers of buckets, we came to the conclusion that if we use a high number of buckets, the implementation is not scalable. Generally, the total time tends to increase. This is due to the fact that, although we are spending less time on sorting each bucket, more time is spent on handling and manipulating so many buckets.

The solution to the excessive number of buckets not being scalable would be, in future work, to find out the perfect bucket value with the perfect number of threads by performing more tests.

In 3 we can see the behaviour of the table III in a graph, and the numbers represent the pair of the inputs given to the program. We can see that from the values (N=25000000, B=10000) to the values (N=25000000, B=100000) the algorithm scales, but when we go to the next case (3), the times go up again, so maybe the case (2) is the best case for this algorithm.

6. Conclusion

In this project we were able to evaluate the learning of parallel programming in shared memory. During the development of the requested algorithm we faced some difficulties, like the choice of the area that we should parallelize, since we found that it was not possible for us to parallelize the entire algorithm.

Attachments

		Time(ms)	
		Sequential Version	Parallel Version
Input: N=Array Size B=Number of Buckets	(1) N=100 000 B=1000	7.871	6.2025
	(2) N=100 000 B=2000	7.687	6.0565
	(3) N=100 000 B=3000	7.3375	6.235
	(4) N=100 000 B=10 000	7.342	7.276
	(5) N=1 000 000 B=1000	122.2	83.93
	(6) N=1 000 000 B=300 000	90.144	96.929
	(7) N=25 000 000 B=100 000	3964.72	3161
	(8) N=25 000 000 B=300 000	4317.665	3383.7

TABLE I
SEQUENTIAL VERSION VS PARALLEL VERSION (USING CPAR) - TIME MEASUREMENT

		Time(ms)	
		Sequential Version	Parallel Version
Input: N=Array Size B=Number of Buckets	(1) N=1 000 000 B=1 000	51.49	6.04
	(2) N=1 000 000 B=2 000	46.63	6.91
	(3) N=1 000 000 B=3 000	43.75	6.55
	(4) N=1 000 000 B=10 000	35.34	5.76
	(5) N=1 000 000 B=100 000	22.12	4.09
	(6) N=1 000 000 B=500 000	16	3.53

TABLE II
SEQUENTIAL VERSION VS PARALLEL VERSION (USING COMPUTE-134-16) - TIME MEASUREMENT (ONLY IN THE PARALLEL SECTION)

		Time(ms)	
		Sequential Version	Parallel Version

Input:
 	
N=Array Size
B=Number of Buckets
	(1)
	N=25 000 000
	B=10 000
		5936.7	5042.1

	(2)
	N=25 000 000
	B=100 000
		4132.4	3449.8

	(3)
	N=25 000 000
	B=1 000 000
		5103.7	4779.9

	(4)
	N=50 000 000
	B=10 000
		16303.7	14465.7

	(5)
	N=50 000 000
	B=1 000 000
		11267.4	10375.9

TABLE III
SEQUENTIAL VS PARALLEL VERSION (USING COMPUTE-134-16) - TIME MEASUREMENT

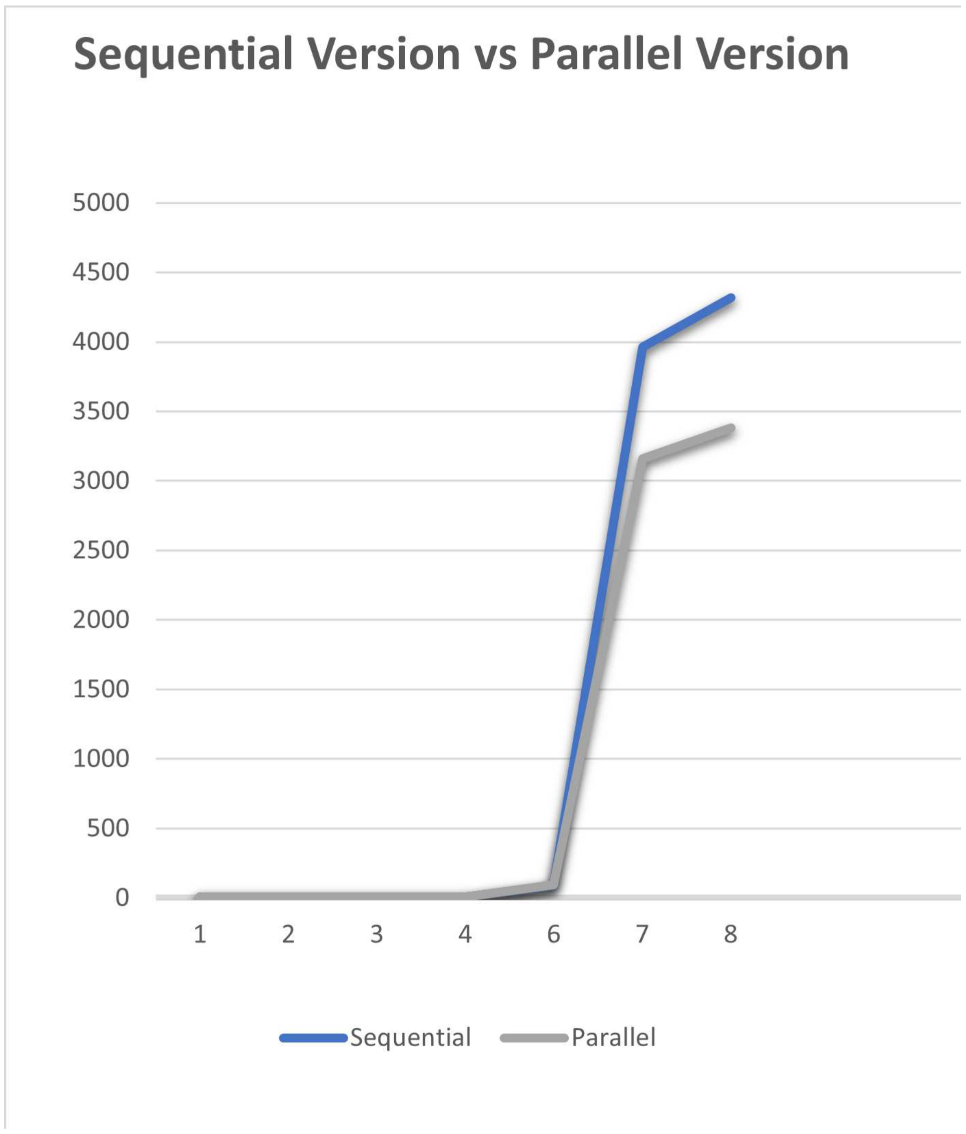


Fig. 1. Sequential vs Parallel graphic (Corresponds to table 1)

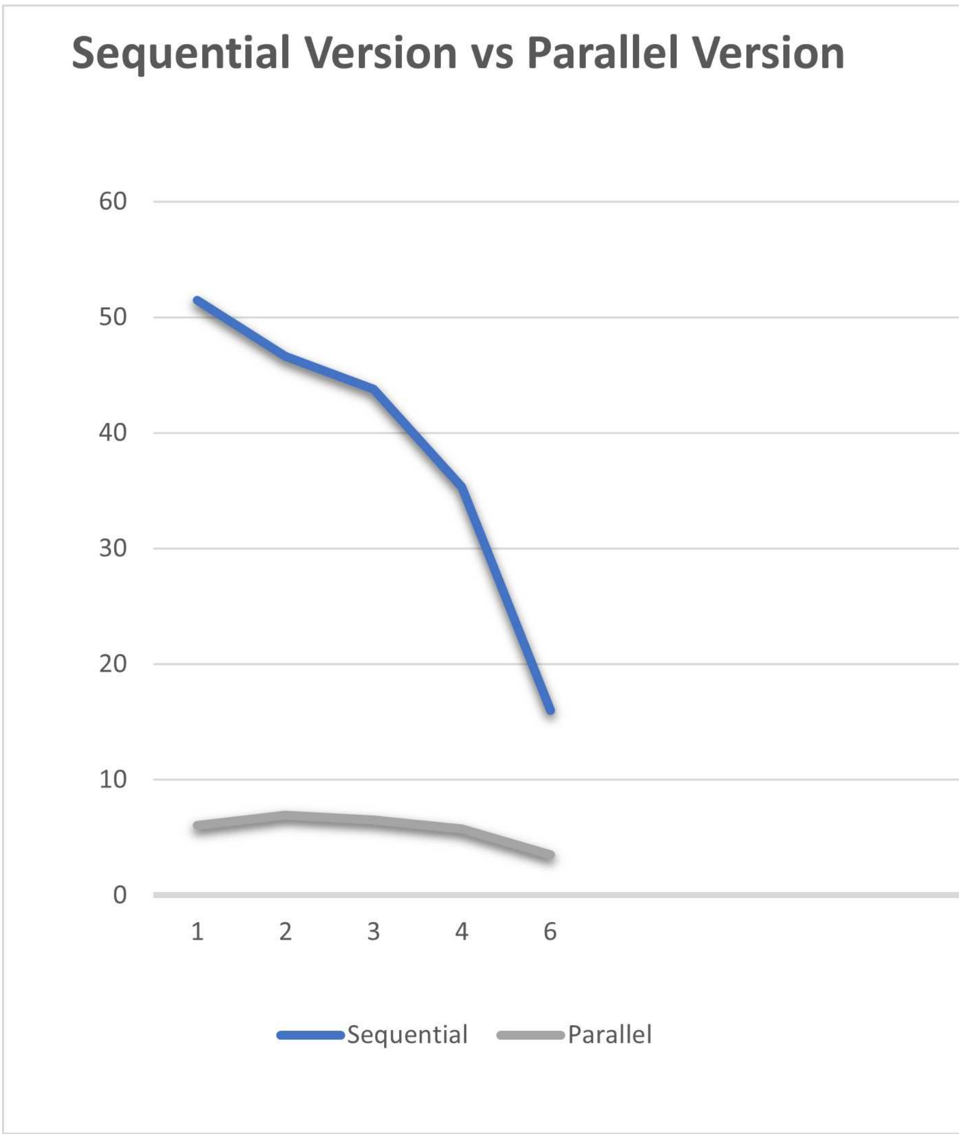


Fig. 2. Sequential vs Parallel graphic (Corresponds to table 2)

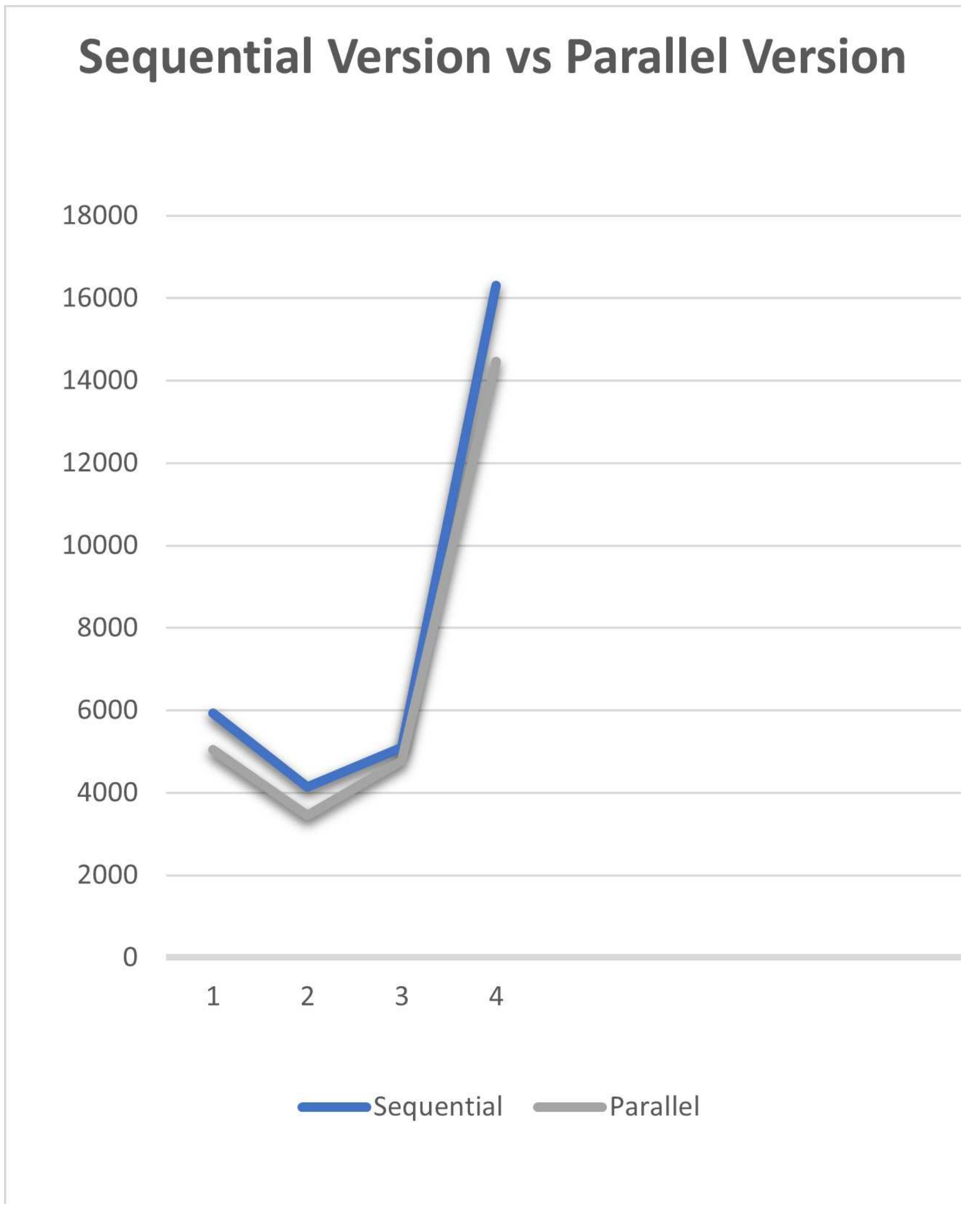
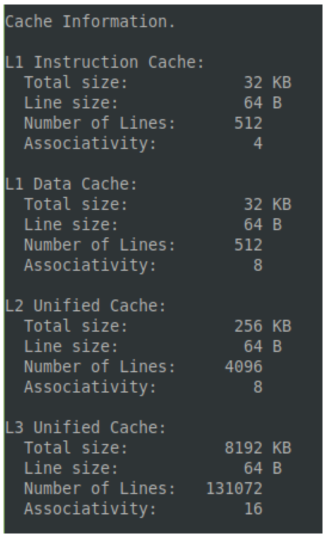


Fig. 3. Sequential vs Parallel graphic (Corresponds to table 3)



```
Cache Information.

L1 Instruction Cache:
  Total size:           32 KB
  Line size:            64 B
  Number of Lines:     512
  Associativity:         4

L1 Data Cache:
  Total size:           32 KB
  Line size:            64 B
  Number of Lines:     512
  Associativity:         8

L2 Unified Cache:
  Total size:          256 KB
  Line size:            64 B
  Number of Lines:    4096
  Associativity:         8

L3 Unified Cache:
  Total size:         8192 KB
  Line size:            64 B
  Number of Lines:  131072
  Associativity:        16
```

Fig. 4. Node compute-134-16 cache characteristics