Parallelization of Bucket-Sort

Computação/Programação Paralela

Carlos Ferreira
PG47087
Universidade do Minho
Braga Portugal
pg47087@alunos.uminho.pt

Joel Martins
PG47347
Universidade do Minho
Braga Portugal
pg47347@alunos.uminho.pt

***Abstract*— This document presents the study and analysis of the computer implementation and optimization of the algorithm Bucket-Sort using varied parallelization techniques.**

Keywords*—Parallelization, OpenMP (Open Multi Processing***), PAPI (***Performance Application Programming Interface***), Bucket-Sort, C Programming Language, Threads, CPI (***Clocks Per Instruction***), Cache, Thread (***parallelized execution task***)***

I. Introduction

This document has as its objective register the process and conclusions of the implementation, analysis and optimization of an algorithm named Bucket-Sort, a technique in the scope of computer science capable of sorting all types of elements.

During this project it was used the programming language C equipped with the different techniques mainly the ones provided by the interface OpenMP (Open Multi Processing), that can take advantage of the properties computer hardware have like the existence of several cores.

Part of the project was the posterior testing and analysis of the developed algorithm, evaluating the efficiency gained in terms of speed and determinate it's limits.

All the tests were made in the server available by the informatic department of the university Uminho [7] that gives access to many hardware resources, and all the metrics were calculated using the interface PAPI (Performance Application Programming Interface).

II. Bucket-Sort

In this study Bucket-Sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets. Each bucket is then sorted individually using a different sorting algorithm [1]. It's a very easily understandable and quite famous technique that can be readily parallelized for the simple reason that each bucket can be sorted at the same time.

It involves mainly 4 different steps:

A. Set up an array of initially empty "buckets".
B. **Scatter**: Go over the original array, putting each object in its bucket.
C. Sort each non-empty bucket.
D. Gather: Visit the buckets in order and put all elements back into the original array.

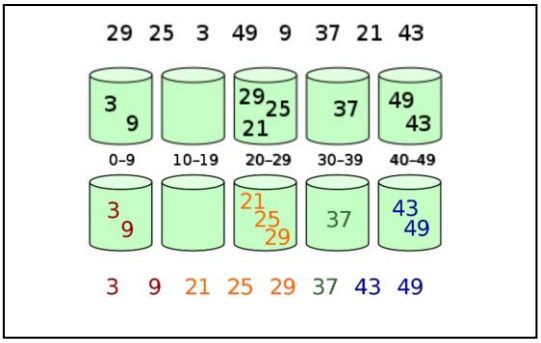


Figure. 1. Bucket Sort [1]

III. Sequencial Version Bucket-Sort

This section of the document will describe the implementation of the sequence version of the developed code, that being, without using any king of parallelization, mentioning all the decisions taken and its reason including possible future optimizations.

A. Initializing the Buckets

A bucket is essentially an array (list of elements), but the size of each bucket is dependent of the input, the bucket size is very important to optimize the usage of memory, so instead of deciding on a static size, a decision has been made of using a structure that can grow as needed. This step is basically initializing and allocation memory for each bucket.

```
typedef struct {
    int *array;
    size_t used;
    size_t size;
}Bucket;
```

Code. 1. Bucket Structure

B. Calculate Maximum, Minimum and Range

A significant important part of this algorithm is to decide in which bucket to insert each element, for that it's important to know what range of numbers each bucket is in charge of.

$$Range = \frac{Maximum - Minimum}{numBuckets} + 1 \quad (1)$$

In the language C, the division is an integer function so the result it's also an integer, because of that it's important

Identify applicable funding agency here. If none, delete this text box.

to add 1 so that the calculation of the index never surpass the number of buckets available. The calculation of the range can also give an overflow during Maximum - Minimum in case Minimum is negative.

C. *Insert Elements in the Bucket*

After knowing the range of each bucket it's easy to insert each element in the respective bucket, a function named *insertBucket* is responsible for that, and as discussed before that function increases the size of bucket if necessary.

$$BucketIndex = \frac{element - Minimum}{range} \quad (2)$$

```
void insertBucket(Bucket *a, int element) {
  if (a->used == a->size) {
    a->size *= 2;
    a->array = realloc(a->array, a->size * sizeof(int));
  }
  a->array[a->used++] = element;
}
```

Code. 2. Insert Bucket

D. *Sort each Bucket*

After inserting each element in the respective bucket, all the buckets must be sorted using a different algorithm. The quicksort was the chosen because of its popularity, averaged speed, and convenience, the C language has provided an already defined function named *qsort*.

E. *Put elements in the array*

This step it's non complicated as a simple crossing of each bucket it's already sufficient. It's important to know, for future phases, that a variable is responsible to update the number of elements inserted and can only be incremented sequentially.

F. *Freeing each Bucket*

After all the procedures are done, all the memory previously allocated in each bucket needs to be released.

G. *Extra:*

To easily test the algorithm done a function to generate arrays and a function to print arrays was made, so that we can sort those arrays using the code developed and visualize its output.

The elements are generated using the following function:

$$element = (rand\ ()\ \%\ (range)) + Minimum \quad (3)$$

This function has an advantage, it's very probable that the array generated has elements equally distributed between Maximum and Minimum, allowing each execution of the algorithm to have equally input parameters.

H. *Possible Optimizations to the Sequential Version*

Despite parallelization offers almost a guaranteed increase in efficiency, some alterations in the code developed could also give some advantages in speed.

Those modifications are related to the form the bucket Sort works.

1) *Dynamically Calculate the Number of Buckets*

Instead of using a static and already given number of buckets in the input, because the bucket sort complexity depends on the number of elements per bucket, there should be a way to determinate the best number of buckets to use taking in cause the number of elements in the array.

2) *Equalize elements per bucket*

It's already known that it's very possible to have buckets with more elements than others, in the worst-case scenery there could be a bucket with 1 element, and another with all the others.

This decreases greatly the capacity of parallelization seeing how it's intended to have no more than one thread sorting each bucket.

There could be ways to mitigate this issue, one of those could be after detecting the difference, calling the bucket sort inside the bucket, or in other words apply a recursive.

IV. Paralized Version Bucket-Sort

With the sequential version implemented it's time to try optimize it using the interface OpenMP to allow the parallelization of the code and take side of the available technology each computer has to offer.

In this section will be presented the attempt to optimize 3 different phases previously described, those who are considered computing expensive.

But before that, there is a need to understand why parallelization does not always mean best, beyond some cases where sequential version is needed because of data order modifications, the setup of parallelization comes with an overhead (not doing useful work), this overhead can include factors such as:

1. Task start-up time
2. Synchronizations
3. Data Communication
4. Task termination time

A. *Calculate Maximum, Minimum*

The Maximum and Minimum are calculated using a cycle *for*, there is a function of OpenMP that we can easily implement on cycles of this type [2], but if we look to the code available bellow if we pretend to parallelize the cycle the variables max and min need to be or in a critical zone, meaning one thread each time, or we can use the clause reduction [3], to apply a function to the value obtained in each thread.

```
#pragma omp parallel for num_threads(nThreads)
#pragma omp reduction(min:min) reduction(max:max)
for(i=0;i<tamArray;i++){
    if (array[i]>max) max=array[i];
    if (array[i]<min) min=array[i];
}
```

Code. 3. OMP maximum and Minimum

Unfortunately, by running some tests and measuring the time using a tool and method it will be referenced later, it was determined this optimization was not efficient.

Using an array of size 1 million and 4 threads this version was longer by approximately 48 milliseconds.

The reason this happens can be explained by the overhead mentioned before.

B. *Insert Elements in the Bucket*

1) *Using OMP Critical*

Using the same technique "*pragma omp parallel for" [2]* the cycle responsible for inserting the elements in each bucket was parallelized but to prevent data races the function mentioned before *"insertBucket"* has to be located in a critical zone [4], meaning only on thread at a time can execute it.

```
#pragma omp parallel for num_threads(nThreads)
for(i=0;i<tamArray;i++){
    elem = array[i];
    indiceBucket = (elem-min)/range;
    #pragma omp critical
    insertBucket(&(buckets[indiceBucket]),elem);
}
```

Code. 4. OMP critical insert

This modification proved to be very inefficient, longer 471 milliseconds using the same parameters as before, but this time there is another method that can be tried. There is not a need to restrict all the buckets at the same time when it's only inserted an element in one bucket.

2) *Using OMP Locks*

To apply this optimization there was a necessity to create an omp lock [5] in each bucket available.

Each time a thread tries to insert an element, it sets a lock ensuring that it's the only thread executing that zone until unlocks it.

```
typedef struct {
    omp_lock_t writelock;
}Bucket;

void initBucket(Bucket *a,size_t initialSize){
    omp_init_lock(&(a->writelock));
}

void insertBucket(Bucket *a, int element) {
    omp_set_lock(&(a->writelock));
    ...
    omp_unset_lock(&(a->wirtelock));
}

void freeBucket(Bucket *a) {
    omp_destroy_lock(&(a->writelock));
}
```

Code. 5. OMP lock behavior

This implementation involved some modification in the code and despite a big difference was noticed in comparison with the previous implementation using critical, the sequential version was sill faster by 135 milliseconds. So, the changes were dropped.

C. *Sort each Bucket*

Sorting each bucket was the main point of using parallelization, this phase was the one it could be expected the most gain because of the simple fact that arrange each bucket is a slow process compared with the others, and having several buckets being sorted at the same time is very efficient. The OMP function used was simple "pragma omp parallel for"[2] and the gain as expected was very considerable being 38 milliseconds faster.

```
#pragma omp parallel for num_threads(nThreads)
for(i=0;i<nBuckets;i++){
    sortBucket(&(buckets[i]));
}
```

Code. 6. OMP sort

D. *Extra*

The phase responsible to insert the elements back into the array has data race problems that could be contoured by changing the sequential version of the code, however considering that similar phases have demonstrated low to none gain, the idea was easily dropped.

Initializing and freeing the buckets are also very simple processes that it's parallelization theoretically will also get loses.

E. *Uses tools for the implementation*

As mentioned before several tools were used to implement the code presented, in this section a very short explanation of those tools will be provided including the reason they were chosen for this project.

1) *C language*

The programming language C is a language that has accompanied computer science for a long time, it holds until today because of its efficiency and speed.

When testing the optimization of coding, this is a language easy to resort to, for that reason the two authors of this project were very familiar with it.

2) *OpenMP*

OpenMP is an API available to C language that offers multi-thread and shared-memory programming, every functionality is easy to implement as demonstrated and very well documented, giving easy learning in its usage.

3) *Makefile*

Despite not being mentioned before, a *Makefile* was made to aid this project, this archive has the objective of automatize the compiling process generating an executable file of the code developed, including all the dependences necessary. It's very common in C programs.

V. PARALLELIZATION TEST

This section of the document presents one of the main goals of the project, a very important phase in every programming developing process, testing the algorithm, the objective it's to submit the algorithm to varied and numerous conditions, being parameters or hardware, generating lots of possible outputs that can be analyzed in a posterior phase.

A. Used Tools for testing

1) PAPI

In order to obtain the metrics necessary to evaluate the algorithm, the interface PAPI [6] was used. This tool is easily accessible and has the capacity to count the performance of modern micro-processors.

2) UM Cluster

The machine where the code was executed needed to have several resources, including many cores available to test the limits of parallelization, a normal computer cannot test the full capacity of the tests required, because of that the cluster of the Uminho University [7] has been provided to the authors of this project, this cluster is a group of connected computers that can be accessible via ssh.

B. Types Of Tests Done

The testes were made taken in considerable two different variables, the computation power, and the memory usage.

1) Computation power

The computation power takes in count the number of cores available in the hardware and the number of threads specified. It was decided to test the sequential version of the algorithm and the parallelized version to 4, 8, 16, 20 and 24 threads.

2) Memory usage

The memory usage takes in count the size of the array to sort, that can be specified before executing the algorithm.

In these tests was decided to vary the size of the array according to the medium different cache sizes.

$$L1 = 8KB \text{ to } 64KB \quad (1)$$

$$L2 = 256KB \text{ to } 8MB \quad (2)$$

$$L3 = 10MB \text{ to } 64MB \quad (3)$$

Considering each integer occupies 4 bytes the input of the array size decided was 10 thousand, 1 million, 10 million and 20 million integers, approximately 40 KB, 4 MB, 40 MB, and 80 MB respectively.

3) PAPI Conters

The interface mentioned PAPI count 4 different events during the application of the Bucket-Sort, those being the number of instructions (#I), the number of clocks (#C) and the number of miss caches L1, and L2.

It also counts the execution time in microseconds.

The CPI can be calculated using the number of instructions and clocks.

4) Buckets

The number of buckets was static for each run being 500.

```
Using nThreads=4,tamArray=1000000,nBuckets=500

run=0 - ComputingDone
run=1 - ComputingDone
run=2 - ComputingDone
run=3 - ComputingDone
run=4 - ComputingDone
Wall clock time:49604 usecs
PAPI_TOT_CYC=141369720
PAPI_TOT_INS=217798870
PAPI_L1_DCM=1114892
PAPI_L2_DCM=124948
CPI=0.65
```

Figure. 2. Possible Output

C. Results

1) Execution Time

Execution Time in Milliseconds (ms)						
	Seq	4	8	16	20	24
10k	0.695	0.642	0.550	0.517	0.512	0.511
1M	85.7	49.6	36.5	29.5	28.4	28.0
10M	972.7	541.1	396.9	320.1	302.5	297.0
20M	2004.2	1092.0	788.7	642.8	615.3	609.1

*All the values are presented in milliseconds and were rounded

Table. 1. Execution time to all the tests made.

2) Clocks and Instructions

Number of Instructions and Clocks							
		Seq	4	8	16	20	24
10k	#I	2.3M	1.8M	1.7M	1.5M	1.5M	1.5M
	#C	1.6M	1.4M	1.2M	1.0M	1.0M	1.0M
1M	#I	350M	217M	164M	138M	133M	131M
	#C	250M	141M	102M	81M	78M	77M
10M	#I	4.2B	2.4B	1.8B	1.4B	1.4B	1.3B
	#C	2.8B	1.5B	1.1B	857M	808M	793M
20M	#I	8.9B	3.0B	3.7B	2.9B	2.8B	2.7B
	#C	5.8B	5.0B	2.1B	1.7B	1.6B	1.6B

All the values succeeded with M are multiplied by a Million, with B are multiplied by a Billion.
All the values were rounded

Table. 2. Number of instructions and clocks to all the tests made.

3) CPI - Clocks per Instruction

$$CPI = \frac{\#C}{\#I} \quad (4)$$

Clocks Per Instruction						
	Seq	4	8	16	20	24
10k	0.70	0.77	0.71	0.68	0.67	0.67
1M	0.71	0.65	0.62	0.59	0.59	0.59
10M	0.67	0.62	0.61	0.59	0.58	0.58
20M	0.65	0.61	0.59	0.58	0.58	0.59

Table. 3. CPI to all the tests made.

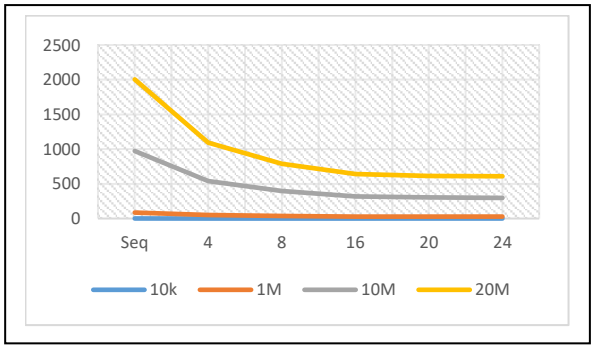
4) Cache Misses

	Array Size			
	10k	1M	10M	20M
L1 Cache Misses	16,302	1,114,892	13,408,824	28,359,547
L2 Cache Misses	2,288	124,948	1,260,888	2,793,512

Table. 3. Misses to cache L1 and L2 to all the tests made with 4 threads.

VI. Parallelization Conclusions

A. Execution Time Evolution



Graph. 1. Execution time by size and threads.

Parallelization has granted a significant improve in the algorithm speed. There have been signs of greater improve then others depending on the array size, samples where the size is small, the gained is little or none when scaled with computer time, however, with bigger input size the time saved is essential to get a useful service, with lower response time, therefore this graph can easily show how it's important the use of parallelization to process big data.

The gain however is not proportional to the number of threads used, there is an observable minimum of execution time achievable, this phenomenon is very known in the camp of computer science and it will be explained later.

B. Cache Miss Impact

To decrease average latency and reduce bandwidth requirements in memory access there is a need to construct a memory hierarchy, where in the top there is the memory closer and faster to access but with less storage capacity.

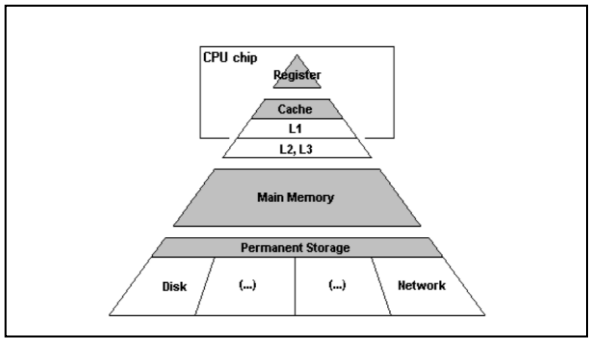


Figure. 3. Memory Hierarchy

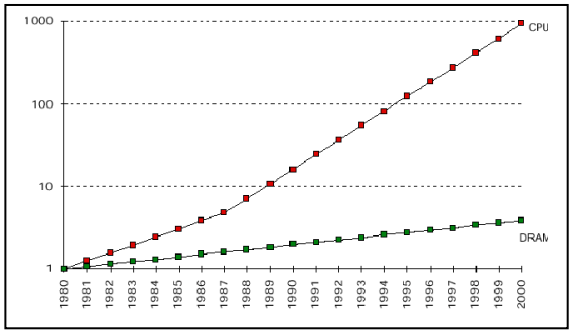
A way to analyze the impact of memory access time is by visualizing the number of cache misses presented, and deduce the losses by applying the next formula:

$$average\ memory\ access\ time = hit\ time + miss\ rate * miss\ penalty \quad (5)$$

This explains how tests made with large amount of data has disproportional bigger time of execution, and how computer processing power is not the only hardware variable in optimizing algorithms.

C. Memory and CPU speed GAP

There are some reasons why more processor power doesn't mean necessary more gain in terms of speed, the main reason and the one discussed in the field is the difference between CPU and memory speed that only have been raising throughout the time.



Graph. 2. Evolution of processor and memory speed. [8]

In the ideal world the double of processors would mean getting the work two times faster.

However, the speed of memory access simple can't follow the CPU speed. With large chunks of memory, every time more operations have to be done to get data from lower levels of memory it costs large amount of working time that could be spent in processing the data, for that reason, more cores don't mean a proportional variation in speed, and a threshold is met in terms of efficiency for every size given as input.

It's important to recognize this bottleneck so that no unnecessary processor power is given to a determined task, that could be doing useful work in other tasks.

D. CPI Metric

The CPI or also called clock per instruction is a metric used a lot to analyze the efficiency of a given code because doesn't depend on the input given but rather the speed rate or velocity of which the task is resolved.

Has it can be seen by the observation of the tests done, every input size tends to converge to a certain CPI given the number of cores available, this CPI could be used to identify easily the bottleneck previously mentioned.

Acknowledgment

Carlos F. and Joel M. thanks the informatic department of the University Uminho for the usage of their cluster.

REFERENCES

[1] https://en.wikipedia.org/wiki/Bucket_sort

[2] OpenMP 5.1 Reference Guide 2021 pp. 7

[3] OpenMP 5.1 Reference Guide 2021 pp. 9

[4] OpenMP 5.1 Reference Guide 2021 pp. 5

[5] OpenMP 5.1 Reference Guide 2021 pp. 12

[6] PAPI User's Guide version 3.5.0

[7] https://www.di.uminho.pt

[8] Patterson D., Anderson T. et al.: A Case for Intelligent RAM: IRAM. IEEE Micro (1997)