Bucket Sort

Eduardo Coelho
Department of Informatics
University of Minho
Braga, Portugal
pg47164@alunos.uminho.pt

Henrique Neto
Department of Informatics
University of Minho
Braga, Portugal
pg47238@alunos.uminho.pt

***Abstract*—In this project, we are going to implement and paralelize the bucket sort algorithm, using C and OpenMP.**

***Index Terms*—Sorting algorithm, parallelization, C, OpenMP, Optimization, PAPI**

I. Introduction

This project can be divided in three chapters. In the first chapter, we describe two implementations of sequential versions of the bucket sorting algorithm to sort an array of integers and analyse possible optimizations that can be applied to this version. In the second chapter, we are going to develop the parallel version of both algorithms using *OpenMP*. Furthermore, we will be studying the impact of all the compromises that exist in the exploration of parallelism of this algorithm. In the third chapter of this project, test the scalability of our implementations in different nodes of *University of Minho's SeARCH* cluster. Additionally the elicited data will also be explored and studied.

II. Sequential Version

A. First Implementation

Since buckets can have an arbitrary amount of integers, we first proposed a *bucket* with the ability to grow dynamically. Therefore, we created a structure based on linked lists, that allows us to easily append new elements to our buckets. Linked lists are simple dynamic structures which makes them very easy to work with, however they are positioned in non contiguous places in memory which can lead to performance losses caused by constant misses in the upper cache levels.

```
struct BucketNode {
    int data;
    struct BucketNode *next;
};

typedef struct BucketNode *List_Bucket;
```

1) Implementation

Since we are sorting an integer array, in order to get the range of each bucket, so that we can scatter all elements across them, we need to calculate the maximum and minimum elements of the input array. This is computed with the function **void** max_min(**int** * array , **int** size , **int** * max,**int*** min);

After that, the range of each bucket will correspond to the ceiling integer of the division between $(max - min)$ and the total number of buckets.

With the value of range of each bucket, we can now fill each bucket with its values. We first initialize all the buckets with a simple call of the inbuilt function **calloc**. We can then scatter all the input array elements across them. Each element is assigned to a bucket, based on its *BucketIndex*, which corresponds to the floor integer of its value with the range of each bucket.

```
for (i = 0; i < ARRAY_SIZE; ++i) {
    int pos = (arr[i] - min) / range;
    List_Bucket current = (List_Bucket)
malloc(sizeof(struct BucketNode));
    current->data = arr[i];
    current->next = buckets[pos];
    buckets[pos] = current;
}
```

Now that we have all the elements spread across their buckets, we can apply a sorting algorithm to each bucket. In this first case, we are using the insertion sort algorithm.

```
for (i = 0; i < BUCKET_NUMBER; ++i)
{
   buckets[i] = InsertionSort(buckets[i]);
}
```

Lastly, with every bucket ordered, we just have to gather all the sorted elements back into the original array.

```
for (i = 0, j = 0; i < BUCKET_NUMBER; ++i)
{
    for(List_Bucket Bucket = buckets[i];
Bucket; Bucket = Bucket->next) {
        arr[j++] = Bucket->data;
    }
}
```

2) Time Complexity Analysis and Possible optimizations

The insertion sort algorithm is inefficient. It has a time complexity of:

$$Best : \Omega(n) \tag{1}$$

$$Average : \Theta(n^2) \tag{2}$$

$$Worst : \mathcal{O}(n^2) \tag{3}$$

It is also notoriously difficult to parallelize. Therefore, a better alternative would be to use the merge sort algorithm

that, not only is much easier to parallelize, but it also has a much better time complexity:

$$Best : \Omega(n \log(n)) \tag{4}$$
$$Average : \Theta(n \log(n)) \tag{5}$$
$$Worst : \mathcal{O}(n \log(n)) \tag{6}$$

After implementing this sorting algorithm, we needed to test it, in order to confirm its efficiency. The results obtained are discussed in detail on sec. IV.

B. Second Implementation

As mentioned before, the use of Linked Lists could cause some performance issues that would derive from the excessive amount of cache misses that are likely to occur. Additionally, it was reveled, through the overhead tests, that the allocation of the lists was also causing a major drawback to performance.

In order to study and correct this issues a new version, based solely on arrays, was implemented. Arrays, in *C*, are contiguous sets of data. This characteristic benefits the machine's cache hierarchy due to the spacial locality that it would introduce.

1) Implementation

To gain the most from this structure, each bucket corresponds to a distinct interval of indexes on the unordered array. The limits of their intervals are defined in another array, which maps the index of each bucket to its offset along with the size of the array. A visual representation of these structures is presented on fig. I.

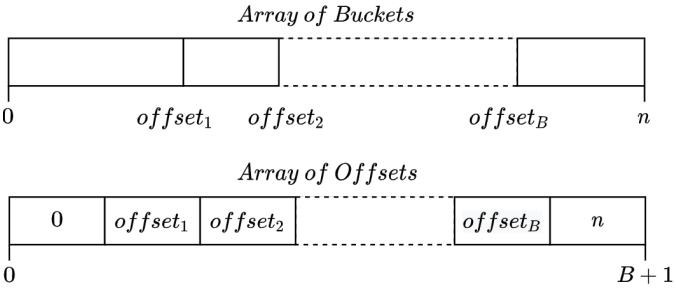

FIGURE I: Array visualization

This structure provides many advantages. In addition to having the content of each bucket contiguous, the buckets, themselves, will also be contiguous in memory, removing the need to regroup them when each one is ordered. It also means that all the buckets can be allocated before any sorting procedure with a single memory allocation call.

To separate each element into each bucket, we first create an array for the offsets, with a size of $B+1$ (where B corresponds to the number of buckets). With this array, we can then sort each element into the array of buckets. There are two functions implemented that do this, one stores the array of buckets in the original array (**sort_self_into_bucket_array**), while the other stores it in an new distinct array (**sort_into_bucket_array**). From a general point of view, the difference between each one is that one requires more memory than the other. However it can take a better advantage of the cache's spacial locality as a result.

Both implementations require the same auxiliary structures and can be separated into 3 different computing steps, in which only the last one differs. The auxiliary structures used are 2 arrays. One is labeled *counting* and it is first used to measure the size of each buckets and then it is reused as an array of indexes for each bucket. The other is labeled *locations* and will contain the bucket index of each element of the array, to avoid being recalculated each time this value is needed.

The first computing step involves a single traversal of the array in order to calculate the size of each bucket. To achieve this, we must calculate the *bucket index*, similar to what is done in the list implementation. This value is used to increase the size of each bucket on the array *counting*, and is also stored in the *locations* array.

```
for (i = 0; i < n; i++) {
    j = (input[i] - min) / range;
    locations[i] = j;
    counting[j]++;
}
```

The second computing step calculates the values of the *offset array*. For this, the first value of the array is always set to 0. Although this 0 is redundant among all possible *offset* arrays, it provides a crucial structural property that allows for a simple traversal of all intervals mapped, without the need of variables with dependencies or if statements. The remaining values of the *offset array* simply correspond to a cumulative sum of the values of the *counting* array. Additionally, this step also sets every element of *counting* to zero so that it can be used as an array of indexes on the final computing step.

```
offsets[0] = 0;
for (i = 0, j = 0; i < bucket_count; i++){
    k = counting[i] + j;
    offsets[i + 1] = k;
    j = k;
    counting[i] = 0;
}
```

The final computing step involves the sorting of each element to its respective bucket. Each element of the array is written onto its respective bucket position, based on its *bucked index* (stored in the *locations* array). The position of each element is calculated based on its bucket's offset (stored in the array *offsets*), and the position of its bucket's index (stored in the array *counting*). After each placement, the index of the corresponding bucket is incremented. The difference between each method is the order in which each element is accessed on the original and *locations* array. In the simple implementation, the output is written in a different array, therefore each element of the original array (and as consequence the *locations* array) is accessed contiguously. As a result, this implementation makes great use of the cache spacial locality on both these structures.

```
for (int i = 0; i < n; i++) {
    j = locations[i];
```

```
        output[offsets[j] + counting[j]++] =
    input[i];
    }
```

On the other hand, the *sort_self* implementation usually cannot do this. Although the algorithm tries to process each element contiguously, the non contiguous writing that occurs as a result of placing an element on each bucket can displace an element that has yet to be processed. In order not to lose data, the algorithm proceeds to process this displaced element and marks it as processed (by assigning an invalid value to its location). This element can then cause another displacement, which repeats this process. Even though each element is still processed exactly once, the array's elements and locations are no longer processed contiguously, which makes the cache spacial locality less significant. Additionally, it requires a few more operations and a additional write on the array *locations* to ensure this step's functionality.

Based on the physical attributes of the system in which the tests are taking place, it was decided that the benefits from the simple implementation greatly outweigh its downsides. As a result we used the **sort_into_bucket_array** in our performance tests.

Finally, with each element assigned to its respective bucket, we must simply invoke a sorting algorithm to each bucket interval individually.

```
for (i = 1; i <= BUCKET_NUMBER; i++){
    a = offsets[i - 1];
    b = offsets[i];
    MergeSort_Array(bucket_array+a, b-a);
}
```

After each bucket has been ordered, the output array will contain the ordered version of the original array.

2) Time Complexity Analysis

In order to keep the data-sets consistent, the tests also involved sorting each bucket with *MergeSort*. This function was modified to work with arrays instead of lists, however it still operates in the same way and so the complexity of the algorithm remains the same.

Similar to lists, after the implementation we needed to test it, in order to confirm its efficiency. The results obtained are discussed in details in sec. IV.

III. Parallel Version

As assessed before, the first sequential implementation was not the best. However, we still wanted to parallelize it and test it, so that we can further prove the potential and efficiency of the second implementation.

To parallelize the program we will use *OpenMP API* for *C*. For each implementation there are three sections of code that are candidates to parallelization.

The first section implements the scattering job associated to initializing each bucket. In both implementations, this area involves dependencies when writing to the buckets and updating their indexes. This dependencies result in a lot of critical sections in the heaviest operation, which made the parallelization with *OMP* unreliable. So we decided to keep this step sequential.

The second section consists of parallelizing the sorting of each bucket over multiple threads. The only downside to this method was the different workload that each bucket can impose, based on their size. With a normal static scheduling of the buckets this could result in the starvation or overload of the different threads. To resolve this we decide to implement this parallelization with guided scheduling, which splits the buckets into consistently decreasing lengths. This distribution results in a more distributed workload on each thread, without the performance drawback presented by dynamic scheduling. This parallelization approach is the one that is studied in this paper.

The third section consists of parallelizing the bucket sorting algorithms themselves, such as the insertion sort and merge sort. As established before, parallelizing insertion sort is extremely difficult and futile. The merge sort parallelization can be accomplished with tasks. However, based on the number of buckets, it would be expected an excessive number of tasks running concurrently across all buckets. This would also result in a lot of drag from the task's management and would, overall, not result in much better results than the previous mentioned approach. Thus, we decided that it wouldn't be worth exploring this form of parallelization.

A. Parallelizing the first implementation

As mentioned previously, our idea to parallelize this algorithm was to distribute the buckets by the threads. As such, using the OpenMP pragma *pragma omp parallel for* with a guided scheduling, we parellelized the implementation like so:

```
#pragma omp parallel for num_threads(
NUM_THREADS) schedule(guided)
for (i = 0; i < BUCKET_NUMBER; ++i)
{
    MergeSort(&buckets[i]);
}
```

B. Parallelizing the second implementation

In the second implementation, we also used a *pragma omp parallel for* to distribute the buckets across multiple threads and kept the same guided scheduling.

```
#pragma omp parallel for num_threads(
    NUM_THREADS) schedule(guided)
    for (i = 1; i <= BUCKET_NUMBER; i++){
        a = offsets[i - 1];
        b = offsets[i];
        MergeSort_Array(bucket_array+a, b-a);
    }
```

IV. Tests and Results

In order to consistently test all the implementations, all versions of this algorithm were tested over the same computing nodes in the *University of Minho's SEARch cluster*. The implementations were then ran in a machine containing two *Intel® Xeon® CPU E5-2695 v2 @ 2.40GHz*. These CPUs contain 12 cores, which are capable of running 2 threads each.

Additionally, each core contains $64KB$ of *L1 cache*, split among data and instructions. Additionally, each core possesses a total of $256KB$ of *L2 unified cache* and the *L3 unified cache* is shared among all cores and possesses a total size of $25,6MB$. To keep the results consistent, the CPU frequency has been virtually fixed to $2Ghz$ with the help of software, to keep the real clock speed as consistent as possible.

For each test, we generated an array of random integers which were seeded with the value of the system's clock time. The generated array would then be sorted with the target algorithm which would be benchmarked on 2 distinct runs. Additionally, for each run we measured 5 ordering operations which were aggregated into a single result to reduce data inconsistencies.

For the first run we used the module *PAPI* (Performance Application Programming Interface) to measure events in real time. The available *PAPI* events vary based on the system and, in this node, we were able to measure metrics such as: execution time, number of clock cycles, number of instructions and Level 1 & 2 data cache misses. Unfortunately the module wasn't able to measure Level 3 cache misses in this system.

For the second run we used the *perftool* to measure the overhead of each function had on the PUs. The *perftool* periodically stops the program's execution and samples data from the current stack, such as the top level function. This allows to analyze what functions are worthy of future optimizations.

A. MergeSort and InsertionSort

Using the process previously described, we first explored the differences between sorting each bucket with insertion sort and merge sort. Since the focus is the methods used in each bucket and not the bucket algorithm itself, we used the first sequential implementation to test both options.

N	B	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
10000	10	9430	27141750	18878521	1.44	934975	28298
10000	100	1413	3207849	5425093	0.59	45443	14168
10000	1000	1276	2008500	3445928	0.58	48832	16294
100000	100	185169	506419971	182980114	2.77	11790379	405073
100000	1000	18244	44616439	54156946	0.82	518601	397429
100000	10000	17548	38089886	38868162	0.98	1180674	906903
1000000	1000	2378943	6887191775	1890752803	3.64	215979985	4764159
1000000	10000	338146	965658136	585490694	1.65	15374112	10619778
1000000	100000	247592	691310291	388555243	1.78	16145295	12149460

TABLE I: Sequential Version Tests for Insertion Sort

N	B	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
10000	10	2190	5351535	6616575	0.81	109975	25845
10000	100	1662	3950597	5778919	0.68	45475	14907
10000	1000	2189	2637399	4324958	0.61	46420	16924
100000	100	29823	60798041	59737036	1.02	1224348	453266
100000	1000	19344	47897534	57695402	0.83	517464	393417
100000	10000	20148	44314685	47680449	0.93	1128579	865165
1000000	1000	401544	970854453	660341080	1.47	17130755	5230367
1000000	10000	363130	1006108938	621171484	1.62	15334228	10477743
1000000	100000	262090	783713108	476736911	1.64	15544956	11532689

TABLE II: Sequential Version Tests for Merge Sort

As expected, the merge sorting algorithm is generally better, as the execution times are lower in each case. However, to fully understand these results we must view them differently. As described before, the performance of both algorithms depends on the size of the bucket being sorted, therefore we must compare each element based on the average length of the buckets. Since the arrays used in these tests have random integers, we can assume that the elements will be evenly spread across all buckets and that $AVG \approx \frac{N}{B}$. To ensure that this property is consistent, we tested multiple bucket lengths with multiple array lengths. As expected, the results were extremely consistent across all arrays so we aggregated them. The final results are presented on the table III.

AVG	T Gain	#CC Gain	#I Gain	L1_DCM Gain	L2_DCM Gain
10	$-5,84\%$	$-13,21\%$	$-11,78\%$	$+1,96\%$	$+8,77\%$
100	$+5,03\%$	$-6,54\%$	$+1,33\%$	$+37,56\%$	$-0,86\%$
1000	$+447,98\%$	$+583,18\%$	$+192,65\%$	$+614,68\%$	$+833,93\%$

TABLE III: MergeSort Gain in relation to InsertionSort

With these tests between *merge sort* and *insertion sort*, we can confirm that the *merge-sort* algorithm scales a lot better with the bucket size than the *insertion sort* algorithm, as was predicted by the complexity analyses of the average and worst cases that was done previously.

Lastly, we observed the report generated by the *perf* tool with the values $N = 1000000$ and $B = 1000$. The objective was, not only to observe the work distribution across the program, but also to confirm the previous established results.

Overhead	Samples	Command	Shared Object	Symbol
29.71%	407	main.out	main.out	[.] InsertionSort_List
25.49%	350	main.out	main.out	[.] BucketSortSequential_Lists_aux
23.32%	332	main.out	libc-2.17.so	[.] malloc_consolidate

TABLE IV: Insertion Sort's Overhead

Overhead	Samples	Command	Shared Object	Symbol
24.78%	352	main.out	main.out	[.] BuckerSortSequential_Lists_aux
22.25%	318	main.out	libc-2.17.so	[.] malloc_consolidate
19.08%	273	main.out	main.out	[.] MergeSort_List
12.87%	184	main.out	main.out	[.] SortedMerge_List

TABLE V: Merge Sort's Overhead

As we can see, the insertion sort's overhead percentage was 29.71%, while the merge sort's was 19.08 + 12.87 = 31.95%. As expected these are very similar since the gain obtained in this particular case was negligible. Additionally, we can also conclude that the lists implementation is causing some drawback to the performance, since their own allocation has a large overhead.

With these tests we can confirm that the merge sort performs better than the insertion sort. As a result there is no need to do any additional tests between these algorithms, thus the remaining tests were made using only the merge sort's implementation.

B. Sequential Tests

Since the tests performed in the previous section used the sequential list's version, we can reuse the tests presented on the table II. Then, we only need to gather the rest of the data

regarding the implementation with arrays. The resulting values were:

N	B	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
10000	10	973	2326385	3055090	0.76	6460	381
10000	100	778	1776529	2519712	0.71	6395	435
100000	1000	6671	18747616	25106102	0.75	124892	15667
1000000	10000	73058	213671240	250852033	0.85	3429509	999184
1000000	1000	81844	240281757	304507804	0.79	1285953	131949

TABLE VI: Sequential Version Tests for Merge Sort - Implementation with Arrays

When we compare the results from the list implementation (table II) and the array implementation (table VI), its extremely clear the major decrease in scopes such as: data cache misses and execution time. With N=1000000 and B=1000 there was a gain of 4,906 in the execution time, most likely due to the 13,32 times reduction in L1 data cache misses and 39.64 times reduction in L2 data cache misses. The CPI in all the tests was also lower which was to be expected, since fewer data cache misses means fewer clock cycles wasted on pulling data from the memory and/or L3 cache.

Overhead	Samples	Command	Shared Object	Symbol
70.17%	218	main.out	main.out	[.] MergeSort_Array
9.44%	30	main.out	main.out	[.] sort_into_bucket_array
6.01%	29	main.out	[unknown]	[k] 0xffffffffb418c4ef
4.95%	16	main.out	lib-2.17.so	[.] __random_r

TABLE VII: Merge Sort's Overhead - Array Implementation

With this optimization, the overhead introduced by the merge sort algorithm jumped to 70.17%. This is a result of most of the work load being only presented in the sorting algorithms instead of the allocation and scattering of the buckets. Additionally there were a lot less samples from the array implementation since the algorithm finishes a lot faster.

C. Parallel Tests

N	B	Th	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
10^6	10^3	2	366032	826885066	519903267	1.59	11673921	4581416
10^6	10^3	4	310765	668753395	446351921	1.50	8549459	3998351
10^6	10^3	8	277782	596919180	409839810	1.46	6939573	3699643
10^6	10^3	16	265652	560355933	392442220	1.43	6132375	3567591
10^6	10^3	32	264618	546194115	384646765	1.42	5839844	3528933
10^6	10^3	64	265967	550756387	379804225	1.45	5784388	3491203
10^6	10^3	128	276568	430078866	314867992	1.37	5738474	3500370

TABLE VIII: Parallel Version Tests with Lists

As we can observe, there was a significant gain in performance with every increase in the number of threads until we used 64 threads. Every gain in performance would also be smaller the more threads we used. The best performance was obtained when 32 threads were used. The machine where this tests were executed possess 48 PUs (1,517 gain in the execution time when compared to the sequential version). Therefore, it was expected that the jump from 32 to 64 threads was going to degrade performance, since the number of threads we were using was bigger than the number of available processing units. However, we can also conclude that this implementation with lists is far worse than the implementation with arrays, since the best performance obtained with the list's parallelized version is worse than the performance obtained with the array's sequential version. Therefore, the proceeding tests will be executed only to the array's version of the bucket sort algorithm.

Overhead	Samples	Command	Shared Object	Symbol
26.58%	702	main.out	main.out	[.] MergeSort
24.05%	623	main.out	main.out	[.] SortedMerge
14.25%	348	main.out	main.out	[k] BucketSortParallel_aux

TABLE IX: Merge Sort's Overhead - Parallel Version with Lists

The merge sort algorithm's overhead jumped to 26.58 + 24.05 = 50.63%.

N	B	Th	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
10^6	10^3	2	64639	187352180	196986742	0.95	1193560	149840
10^6	10^3	4	52290	143584950	141708189	1.01	1396479	139152
10^6	10^3	8	42730	117851585	113680730	1.04	1119272	137259
10^6	10^3	16	38216	106521009	100010988	1.07	1106203	132992
10^6	10^3	32	36431	102602012	95000403	1.08	1099941	133980
10^6	10^3	64	36374	101538883	91365192	1.11	1104157	132158
10^6	10^3	128	38996	94579170	86012872	1.10	1100541	131886

TABLE X: Parallel Version Tests - Implementation with Arrays

By analysing this results, we can observe that, with the increase in the number of threads, the execution time decreased, with the exception of 128 threads. The performance gain decreased with the increase of the number of threads. The machine were this tests were conducted has 48 PUs. The performance with 32 and 64 threads was so close that we can say the difference is negligible due to the error margin. However, with 128 threads, the performance degradation is very apparent, which is to be expected, since the number of threads we are using is much higher than the available processing units. The best gain in execution time obtained was 2.2501 with the number of threads equal to 64.

Overhead	Samples	Command	Shared Object	Symbol
62.38%	367	main.out	main.out	[.] MergeSort_Array
8.60%	189	main.out	[unkown]	[k] 0xffffffffb4196098
7.04%	33	main.out	main.out	[k] sort_into_bucket_array

TABLE XI: Merge Sort's Overhead - Parallel Version with Arrays

In the parallelized version, the merge sort's overhead was 62.38%.

We will now conduct more tests in different machines, to analyse the impact that different hardware will have on these metrics.

D. Tests on different machines

To evaluate the impact of different hardware on the performance of the bucket sort algorithm, we performed more tests

on a series of machines in the *day* partition of University of Minho's *SEARCH* cluster. These machines vary in the number of processing units, but they all have the same cache available:

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

As we can see, these machines have smaller cache than the machine we were using in the *cpar* partition before.

N	B	Th	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
$2.5E7$	10^3	2	1975519	4875582472	4871633452	1	40517188	2888607
$2.5E7$	10^3	4	1222962	2975983982	2844792943	1.05	33760291	2979344
$2.5E7$	10^3	8	858574	2003133676	1798832382	1.11	30142308	2826125
$2.5E7$	10^3	16	754345	1752968369	1293538747	1.36	28830498	2793988
$2.5E7$	10^3	32	755416	1461161709	1093846120	1.34	27952551	2630288
$2.5E7$	10^3	64	753317	1095933900	839391618	1.31	26784718	2534698
$2.5E7$	10^3	128	756326	1073090740	810161074	1.32	26822219	2615876

TABLE XII: Parallel Version Tests - Implementation with Arrays - Partition: day, Compute-113-3 - 16 PUs

The node *Compute-113-3* has 16 PUs instead of the 48 PUs of the machine used in all the previous tests. The impact of this reduced number of PUs is very easily observable by the diminishing performance gains with the increase of the number of threads we are using. In fact, the execution time improved until we hit 32 threads. After that, the execution time had no significant improvement.

Another characteristic of this machine is that it only has 8192 KB of L3 unified cache. Since the first machine we used had more Cache capacity, it was expected to have an higher number of data cache misses.

N	B	Th	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
$2.5E7$	10^3	2	1857947	4842511490	4874334307	0.99	40485119	2972623
$2.5E7$	10^3	4	1140537	2933363163	2831445135	1.04	33490527	2861606
$2.5E7$	10^3	8	785631	1978375036	1812081415	1.09	30041723	2848437
$2.5E7$	10^3	16	638353	1583984200	1325309219	1.20	28400199	2707308
$2.5E7$	10^3	32	610025	1440440437	1104287526	1.30	27831965	2549930
$2.5E7$	10^3	64	610638	1226195716	934551256	1.31	27200420	2655466
$2.5E7$	10^3	128	598814	1125854802	880096492	1.28	26945797	2556253

TABLE XIII: Parallel Version Tests - Implementation with Arrays - Partition: day, Compute-113-17 - 24 PUs

The node *Compute-113-17* has 24 processing units. By observing these results we can see that the execution time improved with the increase of the number of threads until we used 32 threads. A number of threads greater than this didn't give us any performance gains, which was to be expected, since the number of threads would be greater than the available processing units. Even though the execution time with 128 threads was slightly better, we believe this was a result of an higher CPU frequency, since it is not locked on the machines of the *day* partition of the *SEARCH* cluster. The amount of cache of this machine was the same as the amount of cache in the previous machine, therefore it was also expected that the L1 and L2 data cache misses would be very similar.

N	B	Th	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
$2.5E7$	10^3	2	1527345	4871841244	5326651059	0.91	42006316	4667995
$2.5E7$	10^3	4	968615	2770035229	2824696931	0.98	33670809	3883534
$2.5E7$	10^3	8	681835	1911698050	1805904432	1.06	30302339	3395223
$2.5E7$	10^3	16	556859	1535409168	1349560062	1.14	28646878	3320512
$2.5E7$	10^3	32	524434	1439767628	1095585180	1.31	28190619	3077985
$2.5E7$	10^3	64	521876	1180860984	903641110	1.31	27215699	3024324
$2.5E7$	10^3	128	527105	1056184486	792510233	1.33	26699745	3333879

TABLE XIV: Parallel Version Tests - Implementation with Arrays - Partition: day, Compute-134-6 - 32 PUs

The node *Compute-134-6* has 32 processing units. As expected, the best performance was obtained when we used 32 and 64 threads, degrading when using 128.

N	B	Th	T (μs)	#CC	#I	CPI	L1_DCM	L2_DCM
$2.5E7$	10^3	2	1585630	4843487897	5223117670	0.93	41806438	4805196
$2.5E7$	10^3	4	1050882	3088290591	3133835881	0.99	34647037	4019803
$2.5E7$	10^3	8	747130	1953763365	1803693639	1.08	30162251	3474165
$2.5E7$	10^3	16	630979	1622802012	1420575010	1.14	28964509	3339527
$2.5E7$	10^3	32	550135	1385222348	1064831415	1.30	27971868	3226380
$2.5E7$	10^3	64	538544	1341853121	987204216	1.36	27601780	3124284
$2.5E7$	10^3	128	533771	1285464477	967310389	1.33	27479685	3095313

TABLE XV: Parallel Version Tests - Implementation with Arrays - Partition: day, Compute-134-113 - 48 PUs

This machine has 48 available PUs. As we can see, there were significant execution time improvements with every increase in the number of threads, with the exception of 128 threads.

N	B	Th	T (μs)
$2.5E7$	10^3	2	1170049
$2.5E7$	10^3	4	910171
$2.5E7$	10^3	8	625320
$2.5E7$	10^3	16	448425
$2.5E7$	10^3	32	363174
$2.5E7$	10^3	64	327263
$2.5E7$	10^3	128	328057

TABLE XVI: Parallel Version Tests - Implementation with Arrays - Partition: day, Compute-165-1 - 64 PUs

The node *Compute-165-1* has 64 processing units. As expected, the best performance was obtained when we used 64 threads, degrading when using 128. This machine doesn't support PAPI's counters that measure metrics such as data cache misses, CPI, #CC and #I, therefore we only have the execution time to compare to.

A visual representation of all the execution times across the node is present in the next figure.