Parallel Bucket-Sort Algorithm - Parallel Computing

1st Leonardo Marreiros
IT departement
University of Minho
Braga, Portugal
pg47398@alunos.uminho.pt

2nd Pedro Fernandes
IT departement
University of Minho
Braga, Portugal
pg47559@alunos.uminho.pt

***Abstract*—The following report refers to a project proposed by the course unit of Parallel Computing. The project goal is to: not only create two versions of the well-known bucket-sort algorithm, one sequential and another parallel; but also to try to optimize them while analyzing its results.**

***Index Terms*—bucket-sort, parallel computing, optimization, OpenMP, sorting algorithm**

I. Introduction

This project is intended to develop a parallel bucket-sort algorithm. To achieve this, many steps need to be taken. First, a sequential version of this algorithm is needed. After that, some optimization to this method should be done or at least tried. Techniques such as loop unrolling and vectorization will be considered. It is also important to measure the impact of the CPU's cache levels in terms of performance. Only after all that the main purpose of the project of creating a parallel algorithm, should be done and compared with the algorithms made before.

II. Sequential Algorithm

A. Algorithm Description

The sequential version of this algorithm receives the size (*dim*) of the array that we intend to sort and the number of buckets (*n_buckets*). Firstly, two arrays of size *dim* (*A* and *B*) are allocated: the first one is where the generated values will be stored and the second one is a temporary array where all the buckets are sorted. After that it is allocated an array of struct bucket with size equals to the number of buckets given. The bucket struct consists of three pieces of information: *n_elem, index* and *start*. *n_elem* refers to the number of elements stored in a bucket, *start* is the position where a bucket begins in B and *index* is the position in B where the next integer of a bucket will be stored.

After A is initialized randomly with the designated amount of elements, the following cycle is executed to count the number of elements for each bucket:

```
for (i=0; i<dim;i++){
        j = A[i]/w;
        if (j>n_buckets-1)
                j = n_buckets-1;
        buckets[j].n_elem++;
}
```

Following this, the bucket struct is initialized with the right values. With the exception of the first bucket where its start and index is 0, for each of the following, their start and index will be placed according to the previous one's start and number of elements.

Afterwords, all the items of each bucket are stored in B in the right position.

```
int b_index;
for (i=0; i<dim;i++){
        j = A[i]/w;
        if (j > n_buckets -1)
                j = n_buckets-1;
        b_index = buckets[j].index++;
        B[b_index] = A[i];
}
```

Finally each bucket will be sorted according to the given sorting algorithm and then A and B are swapped.

This algorithm uses 3 structures that are allocated, two int arrays and an array of buckets with the given size. Since an int uses 4 bytes of storage, and each bucket struct contains 3 ints, they will occupy 12 bytes. Since we have two arrays containing *dim* ints and also *n_buckets* buckets, then the total storage taken by our structures is given by the equation below:

$$totalspace = 4 * dim + 4 * dim + 12 * n\_buckets \quad (1)$$

B. Optimization - Sorting

Some search and testing was done to choose the best sorting algorithm to apply to sort each bucket. In total 5 famous algorithms were tried in order to determine the fastest one. To test this, the sequential code was executed with size = 100000 and number of buckets = 32 since the size of the data is big enough to observe any improvement while not too big which means some algorithms with higher complexity (N*N) don't take too long to run. The results from Fig. 1 and Fig. 2 determined quick-sort as the winner, doing so this is the sorting algorithm that will be used every-time from here on out.

C. Optimization - Loops

In order to optimize the original sequential code, we tried to use two well-known methods, Loop-Unrolling and Vectorization. In testing, each set of values was executed 10 times and PAPI was used to evaluate several metrics like number of cycles, number of instructions, CPI, level 1 data cache misses, level 2 data cache misses, and finally wall clock time. Level 3 data caches misses was not used since PAPI does not

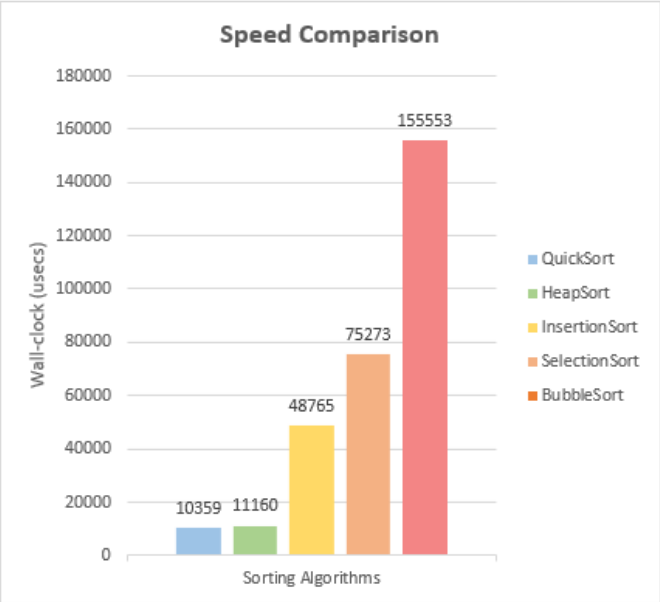

Figure 1. Sorting algorithms - Graph

QuickSort		HeapSort		InsertionSort		SelectionSort		BubbleSort	
Wall clock usecs	10359	Wall clock usecs	11160	Wall clock usecs	48765	Wall clock usecs	75273	Wall clock usecs	155553
Cycles	30164730	Cycles	32326899	Cycles	144858584	Cycles	224288910	Cycles	464527084
Instructions	36839431	Instructions	41540038	Instructions	526254080	Instructions	774531444	Instructions	986197321
CPI	0.82	CPI	0.78	CPI	0.28	CPI	0.29	CPI	0.47
L1 DCM	34933	L1 DCM	32000	L1 DCM	32996	L1 DCM	33134	L1 DCM	34523
L2 DCM	7580	L2 DCM	7416	L2 DCM	8275	L2 DCM	8092	L2 DCM	4539

Figure 2. Sorting algorithms - Table

include exactly that option. These tests also took into account data size (memory occupied given the number of buckets and size of array), so the 3 algorithms were tested for 4 different conjugations of size and buckets in order to fit in each level of cache:

- *Cache L1 :* size = 4000, buckets = 10;
- *Cache L2 :* size = 32000, buckets = 80;
- *Cache L3 :* size = 320000, buckets = 800;
- *RAM :* size = 8000000, buckets = 20000.

		Cycles	Instructions	CPI	L1_DCM	L2_DCM	Time (us)
Cache L1	O2	1000016	1306727	0.765	723	174	358
	Loop Unrolling	992653	1261118	0.787	791	127	358
	Vectorization	1008493	1306456	0.771930321	650	148	363
Cache L2	O2	7887500	10425191	0.756580863	10638	1624	2855
	Loop Unrolling	7911753	10068330	0.785805888	10685	1575	2789
	Vectorization	8031203	10426891	0.77023947	10749	2097	2792
Cache L3	O2	78336576	104154250	0.752120782	307183	31582	28220
	Loop Unrolling	79095562	100534351	0.786751605	308014	31368	27649
	Vectorization	79113354	104155378	0.759570514	308309	33381	27592
RAM	O2	1908840382	2509639528	0.76060341	25080252	13389459	650954
	Loop Unrolling	1898719780	2418473874	0.785090052	24853298	12847749	649377
	Vectorization	1914747022	2509547589	0.762984942	24852347	13237434	675278

Figure 3. O2 vs Loop-Unrolling vs Vectorization

As you can see from the tests above (Fig. 3), neither of them improved too much our original algorithm.

1) Loop-Unrolling: In order to test this optimization attempt we added the flag *-funroll-loops* into the Makefile flags. As we can see the results were not very good since some metrics only improved marginally in comparison with the O2 version. It is to be expected since using this flag implies unrolling all loops in the code, but not every loop benefits from it. In simple loops where the condition is simple and executed many times, the branch predictor should quickly pick it up and always predict correctly the branch until the end of the loop, making the ”rolled” code run almost as fast as the unrolled code. This happens in most of the sequential version for loops. Moreover, in the for loop where each bucket is sorted, the loop body is much slower than the looping itself, which means unrolling can't speed this process up. As a result, we may see a slight or non existent improvement using this technique in terms of wall-clock time. However CPI is always greater than in the O2 version.

2) Vectorization: In order to test this optimization the flags *-ftree-vectorize* and *-msse4* were added to Makefile's flags. There is not much arithmetic or simple operations done in loops. Therefore the runtime of the code is memory bound. E.g. most of the time is spent moving the data between the CPU and memory.As a result, the values of these tests fluctuated a bit from each level of cache where the data could be fitted. In some cases it was better than the O2 version, in others it was worse. However it never gained or lost much in comparison with the original O2 code version.

D. Cache Comparison

In order to test the influence of cache in this algorithm we made several tests. Each one of them was made with each code optimization referred before.

First things first, we need to find exactly the size of each level of cache in the processor that the tests are being runned on, so they are not misleading. The processor being used is an *Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz*, and according to [1] it has the following characteristics:

- *Level 1 cache size:* 10 x 32 KB;
- *Level 2 cache size:* 10 x 256 KB;
- *Level 3 cache size:* 25 MB shared;

From this we can conclude each level of cache size. It is important to notice that it was considered that data sized more than 25 MB would be tested as data fitted in RAM since it doesn't really make sense using the entire RAM for these tests.

1) Test 1 - Comparing L1 and L2 cache misses: In this comparison the objective is to observe the impact that the size of the input makes in cache misses.

- *Cache L1 :* size = 4000, buckets = 10;
- *Cache L2 :* size = 32000, buckets = 80;
- *Cache L3 :* size = 320000, buckets = 800;
- *RAM :* size = 8000000, buckets = 20000.

If we look closely all size/buckets ratio is equal, so the buckets would have the same number of elements to sort. In the table below we can compare each metric increase in proportion for the data from the L1 example. As we can see from Fig. 4, Fig. 5 and Fig. 6, all ratios excluding cache misses is somewhat proportional. This massive increase in L1 and L2 caches in bigger data happens because since it is bigger it can't fit

entirely in each level of cache, which causes it to miss when trying to access a level and the data is missing. For example in the RAM test (yellow one) if all that data fitted in L1 cache it would be expected the ratio to be 2000 but, since it only fits entirely in RAM, its proportion is actually 32700, about 16 times higher. Results from optimization to optimization don't vary much since they don't affect the data size.

Ratio	Proportion						
400	Size and Buckets porportion	Wall clock usecs	Cycles	Instructions	CPI	L1 DCM	L2 DCM
Cache L1	1	1	1	1	1	1	1
Cache L2	8	7.974860335	7.887	7.978094124	0.989542484	14.71369295	9.333333333
Cache L3	80	78.82681564	78.34	79.70620489	0.983006536	424.8727524	181.5057471
RAM	2000	1818.307263	1909	1920.553817	0.994771242	34689.14523	76950.91379

Figure 4. O2 cache misses test

Ratio	Proportion						
400	Size and Buckets porportion	Wall clock usecs	Cycles	Instructions	CPI	L1 DCM	L2 DCM
Cache L1	1	1	1	1	1	1	1
Cache L2	8	7.790502793	7.97	7.983654186	0.998729352	13.50821745	12.4015748
Cache L3	80	77.23184358	79.68	79.71843317	1	389.3982301	246.992126
RAM	2000	1813.902235	1913	1917.722112	0.997458704	31420.09861	101163.378

Figure 5. Loop-Unrolling cache misses test

Ratio	Proportion						
400	Size and Buckets porportion	Wall clock usecs	Cycles	Instructions	CPI	L1 DCM	L2 DCM
Cache L1	1	1	1	1	1	1	1
Cache L2	8	7.691460055	7.964	7.981050261	0.997809581	16.53692308	14.16891892
Cache L3	80	76.01101928	78.45	79.72360187	0.983988442	474.3215385	225.5472973
RAM	2000	1860.269972	1899	1920.881828	0.988431182	38234.38	89442.12162

Figure 6. Vectorization cache misses test

2) Test 2 - Comparing wall-clock time: The objective of this test is to observe how increasing data size, and successively, the need to use larger cache levels, influence the time taken by the algorithm to perform its purpose. So we chose the following data sizes fixing the number of buckets in order to each one fit in each level:

- *Cache L1 :* size = 4000, buckets = 10;
- *Cache L2 :* size = 32000, buckets = 10;
- *Cache L3 :* size = 320000, buckets = 10;
- *RAM :* size = 8000000, buckets = 10.

From the tables in Fig. 7, Fig. 8 and Fig. 9 we can observe that, according to expected, the necessity for data to be stored in a lower level of cache causes the algorithm to run slower, once again due to cache misses. If all the data could be stored in L1 cache it would be expected for the duration to follow size proportion, since this isn't the case the time's proportion is always bigger than the size's one. In this test, again we can see that the loop-unrolling and vectorization versions don't really impact the test, since the important part of it is data size.

	Proportion						
	Size porportion	Wall clock usecs	Cycles	Instructions	CPI	L1 DCM	L2 DCM
Cache L1	1	1	1	1	1	1	1
Cache L2	8	9.69	9.68	9.17	1.06	12.91	4.86
Cache L3	80	112.92	117.54	107.58	1.09	474.06	94.31
RAM	2000	3168.41	3351.18	3247.83	1.03	24549.68	24440.64

Figure 7. O2 - test 2

	Proportion						
	Size porportion	Wall clock usecs	Cycles	Instructions	CPI	L1 DCM	L2 DCM
Cache L1	1	1	1	1	1	1	1
Cache L2	8	9.46	9.74	9.33	1.04	13.04	8.16
Cache L3	80	112.65	116.98	109.44	1.07	458.78	146.84
RAM	2000	3216.98	3395.03	3303.91	1.03	23658.56	34151.04

Figure 8. Loop-unrolling - test 2

	Proportion						
	Size porportion	Wall clock usecs	Cycles	Instructions	CPI	L1 DCM	L2 DCM
Cache L1	1	1	1	1	1	1	1
Cache L2	8	9.22	9.93	9.35	1.06	14.55	5.94
Cache L3	80	114.57	120.55	109.69	1.1	499.7	124.83
RAM	2000	3132.66	3444.13	3311.25	1.04	25706.8	30443.59

Figure 9. Vectorization - test 2

III. Parallel Algorithm

A. Algorithm Description

In order to parallelize the algorithm it was taken into account some how long each loop takes to run. It was found that the cycle that took more to run was the one that organizes data within each bucket. With this in mind two options were made. First one, simply parallelize this cycle and leave the others in sequential mode. The second approach was to parallelize the entire algorithm.Since we believed we could improve the first one , we developed a second version creating an entire parallel section around the code placing some barriers for synchronization and some master sections instead of creating parallel sections in each loop.

1) Sorting parallelization: In this first version, we added a parallel section to the loop where data is sorted with the number of threads being the number of given buckets:

```
#pragma omp parallel for private(i)
    for(i=0; i<n_buckets; i++)
        qsort(B+buckets[i].start,
              buckets[i].n_elem,
              sizeof(int), cmpfunc);
```

Using this annotation results is dividing the loop in *n_buckets* threads, each one will have a private i not initialized and discarded after its run.

We also added the following code to guarantee that the number of threads is equal to the number of buckets:

```
int num_threads = n_buckets;
omp_set_num_threads(num_threads);
```

2) Full parallelization: In this version, we added an entire parallel section with necessary barriers and master directives when necessary to handle data races. Once again each bucket corresponds to a thread.

Firstly, we divide the data among the threads. Each thread will handle an average of *dim/num_threads* ints of the array. Each element will be assigned to the right bucket by summing the local index with the multiplication of the thread that computes it by the number of threads. The number of elements in this bucket is also incremented at this moment.

```
#pragma omp for private(i,local_index)
for (i=0; i< dim;i++){
    local_index = A[i]/w;
    if (local_index > n_buckets-1)
```

```
        local_index = n_buckets-1;
    real_bucket_index = local_index +
                        my_id*n_buckets;
    buckets[real_bucket_index].n_elem++;
}
```

The next step is to set the right starting position for both local and global buckets in B. To reach this, we first need to compute the number of elements for the global bucket of the thread that is executing this and store it in *global_n_elem[i]*, *i* being the thread.

After that, we wait for all the threads to reach this point and use a master thread to run once the initialization of the global buckets' starting position and the first *n_buckets* local buckets. The next loop is done in parallel and its purpose is to set up the proper starting position in B for each of the rest of the local buckets.

```
#pragma omp barrier

#pragma omp master{
for (j=1; j<n_buckets; j++){
        global_starting_position[j] =
            global_starting_position[j-1] +
            global_n_elem[j-1];
        buckets[j].start =
            buckets[j-1].start +
            global_n_elem[j-1];
        buckets[j].index =
            buckets[j-1].index +
            global_n_elem[j-1];
}
}

#pragma omp barrier
for (j=my_id+n_buckets;
    j< n_buckets*num_threads;
    j=j+num_threads){
        int prevoius_index = j-n_buckets;
        buckets[j].start =
            buckets[prevoius_index].start +
            buckets[prevoius_index].n_elem;
        buckets[j].index =
            buckets[prevoius_index].index +
            buckets[prevoius_index].n_elem;
}
```

We can now make each thread pass all of its local bucket's elements in B, using *index* and incrementing it by one in each step.

```
#pragma omp barrier

    #pragma omp for private(i, b_index)
    for (i=0; i< dim ;i++){
        j = A[i]/w;
        if (j > n_buckets -1)
                j = n_buckets-1;
        k = j + my_id*n_buckets;
        b_index = buckets[k].index++;
        B[b_index] = A[i];
    }
```

Finally, the last step is to sort locally all the buckets using *global_starting_position[i]* and *global_n_elem[i]* to determined where that bucket starts and how many elements it has.

```
#pragma omp for private(i)
    for(i=0; i<n_buckets; i++)
        qsort(B+global_starting_position[i],
              global_n_elem[i],
              sizeof(int), cmpfunc);
```

B. Parallel vs Sequential

For this comparison it was used the O2 version of the sequential code and the full parallelized version of the code, i.e. the version where all the loops are parallel using OpenMP. To compare them in *cpar* partition, it was executed a sample with size of 10 million ints and number of buckets getting duplicated after each iteration beginning at 1 in order to check if the time reduced in half after each run. Analysing the results (Fig. 10 and Fig. 11) they are somewhat expected. Using only 1 bucket it is predictable that the sequential version is faster since the parallel version will only use one thread as well but also has all the overhead from OpenMP to create parallel algorithms. After that number of buckets, the time to run will lower, but not linearly as some would expect, this is due to the parallel slowdown. This phenomenon is caused by communication bottlenecks, data cannot reach each processor at the same rate as they desire, each processing node spends progressively more time doing communication than useful processing.

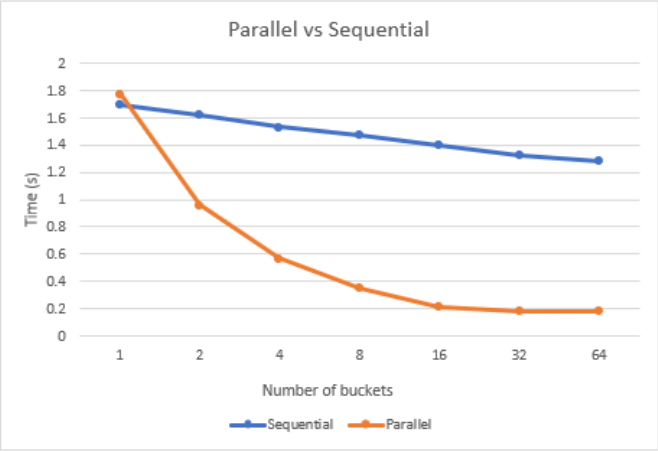

Figure 10. Parallel vs Sequential - Graph

Buckets	1	2	4	8	16	32	64
Sequential	1.698523	1.620371	1.539657	1.46858	1.400796	1.325388	1.28446
Parallel -cpar	1.779629	0.968359	0.576543	0.34527	0.216113	0.179658	0.1786
SpeedUp -cpar	0.954425332	1.673316404	2.670498124	4.25343	6.48177574	7.3772835	7.19168

Figure 11. Parallel vs Sequential - Table

C. Full Parallel vs Sorting Parallel

The following test was made to verify which part/parts of the algorithm should be parallel. With the sorting loop clearly being the one that takes longer due to its complexity, we

decided to compare the full parallel algorithm against another option that was to parallelize only the sorting part. Given the results we conclude that only using more than 8 threads it is useful to parallelize everything. This may be due to the fact that adding more parallelization will increase its overhead, and for fewer threads that overhead is not compensated by the speed increase. However using a large number of threads as 64, parallelizing everything is worth it. The test (Fig. 12 and Fig. 13) was also made for size equal to 10 million and buckets duplicating after each iteration, beginning at 1,performed on partition *cpar*.

Although the sorting section takes the most time to run, it is favorable to make the entire algorithm parallel (for a certain number of threads) since there are other parts that can be parallelized successfully like the division of data among the threads since assigning each element in the proper local buckets can be an independent task.

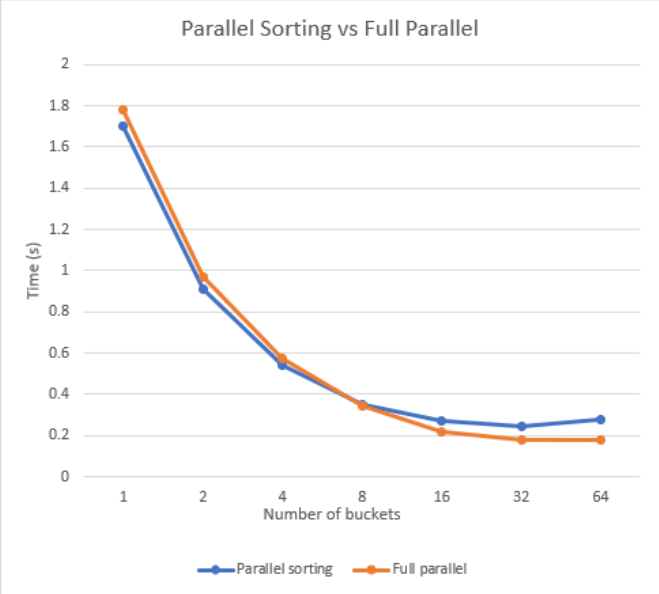

Figure 12. All Parallel vs Parallel Sorting - Graph

Buckets	1	2	4	8	16	32	64
Parallel sorting	1.702079	0.910602	0.543799	0.351524	0.268378	0.246488	0.276555
Full parallel	1.779629	0.968359	0.576543	0.345269	0.216113	0.179658	0.178603

Figure 13. All Parallel vs Parallel Sorting - Table

D. Parallel partition cpar vs Parallel partition day

For context, this test was made with size 10 million and buckets starting at 1, duplicating after each iteration. The same version of the code (full parallel code from previous test) was executed in two different partitions from the cluster, the default one *cpar* and another available *day*. Since the partition *cpar* is more powerful in terms of resources than *day* it is to be expect that it will run faster in it. The results in Fig. 14 and Fig. 15 confirm this initial suspicion.

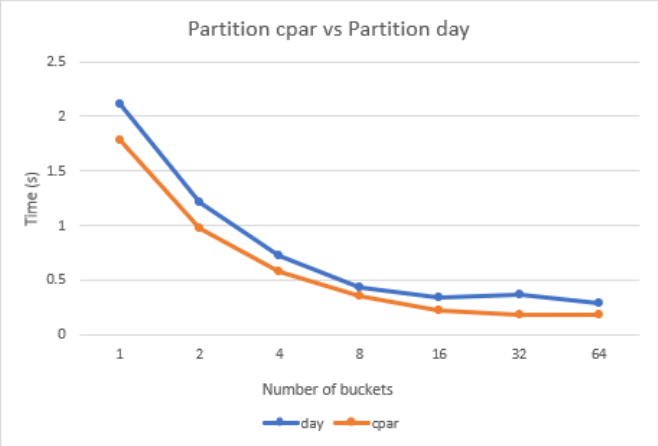

Figure 14. *cpar* partition vs *day* partition - Graph

Buckets	1	2	4	8	16	32	64
day	2.11229	1.212629	0.719851	0.433302	0.332522	0.368788	0.289296
cpar	1.779629	0.968359	0.576543	0.345269	0.216113	0.179658	0.178603

Figure 15. textitcpar partition vs *day* partition - Table

IV. Conclusion

Wrapping up we think that the developed project was a great success since we could get everything done. It was also a great way for us to explore OpenMP and practise more about optimizing code, whether parallel or sequential.

Some difficulties were encountered during the develop stage, like trying to create the parallel version, since the first draft made it slower. But in the end we managed to overcome obstacles.

It was a pleasure develop this project since its analysis helped us to consolidate the subject taught in the course unit of Parallel Computing.

References

[1] CPU-World, https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2670%20v2.html. Last accessed 12 Jan 2022

[2] Wikipedia, https://en.wikipedia.org/wiki/Sorting_algorithm. Last accessed 13 Jan 2022

[3] Wikipedia, https://en.wikipedia.org/wiki/Parallel_slowdown#:~:text=Parallel%20slowdown%20is%20a%20phenomenon,result%20of%20a%20communications%20bottleneck. Last accessed 15 Jan 2022

[4] OpenMP, https://www.openmp.org/wp/. Last accessed 16 Jan 2022

[5] Geeksforgeeks, https://www.geeksforgeeks.org/bucket-sort-2/. Last accessed 2 Jan 2022

[6] Uoregon, https://www.cs.uoregon.edu/research/tau/docs/newguide/bk03ch01s04.html. Last accessed 16 Jan 2022