Parallelization of Bubble sort and Bucket sort algorithms

Democritus University of Thrace
Department of Electrical and Computer Engineering
Computer architecture
and high performance lab
Parallel Processing Technology
Evangelos Lourmpakis
evanlour@ee.duth.gr

March 26, 2025

Contents

1 Introduction

Nowdays there is a constant visible increase on the number of processors both in the professional and the consumer market. As it is currently not possible to automatically bind tasks to muliple cores, a need is created to reconstruct all the current programs with parallelization in mind in order to fully utilize modern hardware. This scientific report is created in order to specify the improvements and speedup values of the *Bubble sort* and *Bucket sort*, using *Merge sort* to order the containers of the algorithm. For the parallelization of the mentioned algorithms the C programming language will be used along with the OpenMP [1] library.

2 Analysis of the algorithm's process pipeline

2.1 The Bubble sort algorithm

The *B*ubble sort algorithm is one of the mosot common and simple algorithms that exist. The algorithm starts from the start of the array and compares sequentially all of the numbers with each other in order to verify if the current number's position is the correct one. Afterwards, the algorithm continues with the second number and this goes on until the last array number is reached. As expected, the algorithm shows the rather high time complexity of O(N^2).

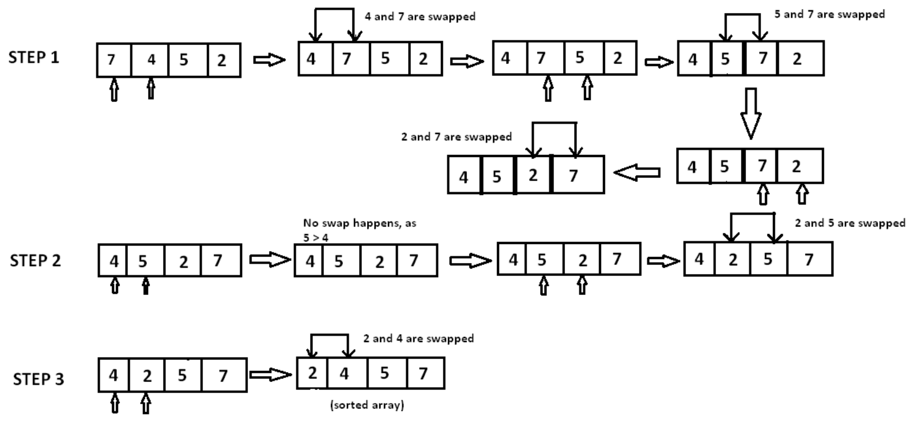

Figure 1: How bubble sort works [2].

Algorithm 1 The Bubble sort algorithm

```
$i \leftarrow 0$
$N \leftarrow$ array size
$array[N]$
for $i \leftarrow 0$ to $N-1$ do
    for $j \leftarrow 0$ to $N-i-1$ do
        if $array[j] > array[j+1]$ then
            $swap(array[j], array[j+1])$
```

Another problem that occurs is that because of how the algorithm i constructed, it is rather hard to efficiently parallelize it without sacrificing time to synchronization overhead. In order to avoid pitfalls, a variation of the *B*ubble sort algorithm will be used: The *O*dd even sort algoritghm.

2.2 The Odd even sort algorithm

The *Odd-Even Sort algorithm* is a favorable variation of *Bubble Sort*, in which each element of the array is alternately compared with its right and left neighbor. While its theoretical time complexity remains the same, the fact that each element is only compared with its adjacent ones makes it much easier and more efficient to parallelize the process. N cores check elements in pairs on their left side first and then on their right side. Since, at any given moment, the elements being processed by each thread are independent of the others, no time is lost on synchronization. This allows us to observe significant improvements in execution time.

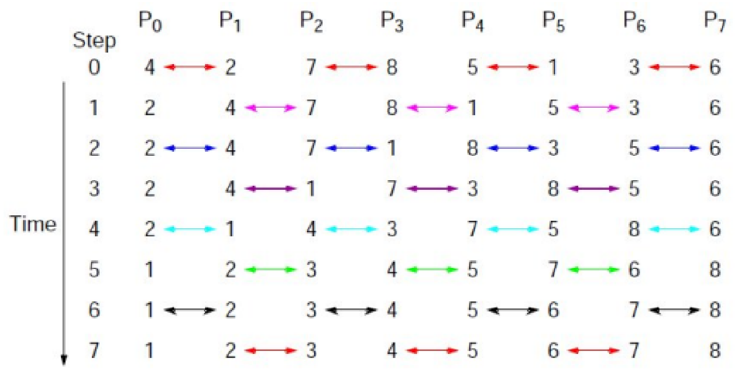

Figure 2: How Odd even sort works [5].

Algorithm 2 The Odd even sort algorithm

```
$i \leftarrow 0$
$j \leftarrow 0$
$N \leftarrow$ array size
$array[N]$
**while array is not sorted do**
    **for** $i \leftarrow 1$ **to** $N - 1$ **with jumpsize:** 2 **do**
        **if** $array[i] > array[i + 1]$ **then**
            $swap(array[i], array[i + 1])$
    **for** $j \leftarrow 1$ **to** $N - 1$ **with jumpsize:** 2 **do**
        **if** $array[j] > array[j + 1]$ **then**
            $swap(array[j], array[j + 1])$
```

It is quite clear that this variation offers a much more efficient parallelization approach in order to utilize the multiple cpu, or even GPU cores [3]

2.3 The Bucket sort algorithm

The *Bucket Sort* algorithm divides the elements of the array into N buckets. Then, within each bucket, it rearranges the elements, and finally, it stores the elements from the buckets into the output array in order. Various sorting algorithms can be used internally within the buckets. In this case, *Merge Sort* was chosen because it exhibits logarithmic time complexity. The parallelization process is relatively simple: the numbers are distributed across N threads, and the elements are placed into the corresponding local buckets. Then, it is enough to merge the corresponding buckets together and connect them sequentially to finally form the output array. This approach avoids the extensive use of synchronization instructions between threads and allows for the full utilization of the available hardware.

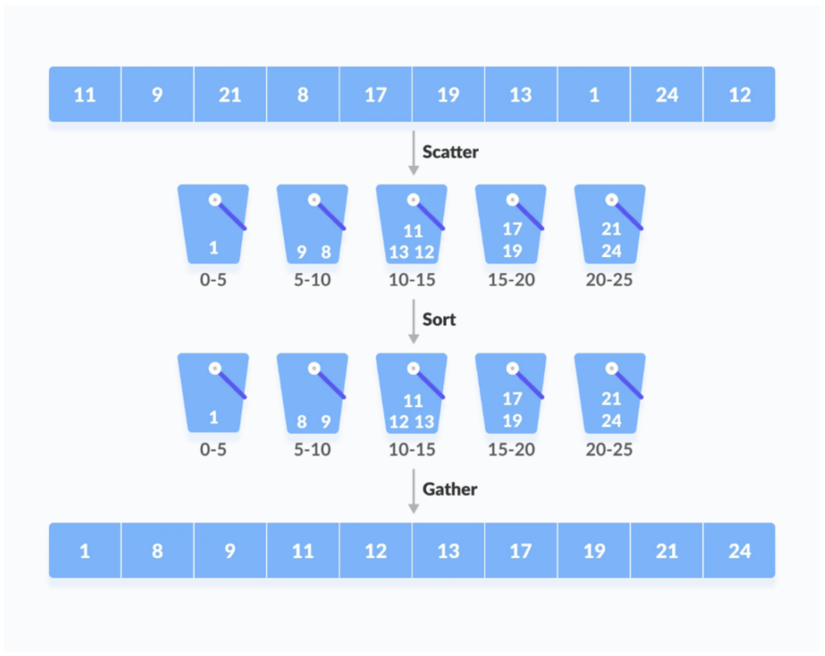

Figure 3: How the Bucket sort algorithm works [4].

Algorithm 3 The bucket sort algorithm

```
$N \leftarrow$ array size
$M \leftarrow$ number of containers
$array[N]$
$containers[number\ of\ containers]$
$i \leftarrow 0$
$j \leftarrow 0$
for $i \leftarrow 0$ to $N - 1$ do
    Send $array[i]$ To the corresponding container
for $j \leftarrow 0$ to $M$ do
    merge sort ($containers[i]$)
$array \leftarrow$ Concatenate(containers)
```

2.4 The Merge sort algorithm

For the internal rearrangement within each bucket, the *Merge Sort* algorithm was chosen. *Merge Sort* is ideal because it is a stable sorting algorithm, which is a requirement for *Bucket Sort*, and it also offers fast execution times. It is worth noting that, in the experimental data, it was decided not to implement it in parallel. This choice was made because the speedup of the algorithm when using parallel methods is already well-known [6], and, additionally, for a reasonable number of buckets, the overhead would significantly reduce any speedup gained from parallelization. *Merge Sort* operates as follows: it breaks down the initial large array into pairs of two, rearranges them internally, and then gradually builds the rest of the array by repeatedly selecting the smallest available number. This method achieves very fast execution times.

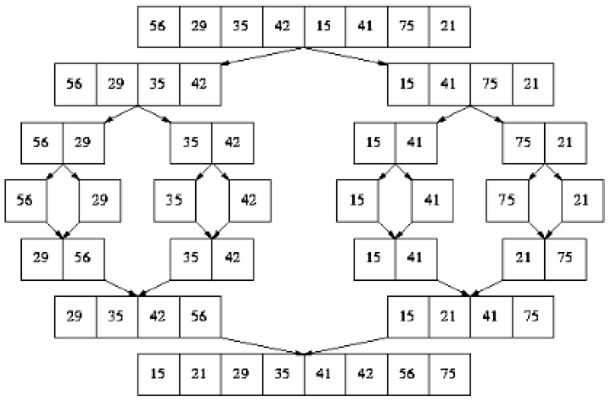

Figure 4: How Merge sort works [4].

Algorithm 4 The Merge sort algorithm

```
$MergeSortArray(array, size)$
if size $\leq 1$ then
    return
$half \leftarrow size$ / 2
$left \leftarrow new\ array\ of\ size\ half$
$right \leftarrow new\ array\ of\ size\ (size$ - $half)$
for $i = 0$ to $half - 1$ do
    $left[i] \leftarrow arr[i]$
for $i = 0$ to $size - half - 1$ do
    $right[i] \leftarrow arr[half + i]$
$MergeSortArray(left, half)$
$MergeSortArray(right, size - half)$
$leftPointer \leftarrow 0$
$rightPointer \leftarrow 0$
while $leftPointer < half$ and $rightPointer < size - half$ do
    if $left[leftPointer] < right[rightPointer]$ then
        $array[leftPointer + rightPointer] \leftarrow leftPointer$
        $leftPointer + +$
    else
        $array[leftPointer + rightPointer] \leftarrow rightPointer$
        $rightPointer + +$
while $leftPointer < half$ do
    $array[leftPointer + rightPointer] \leftarrow leftPointer$
    $leftPointer + +$
while $rightPointer < size - half$ do
    $array[leftPointer + rightPointer] \leftarrow rightPointer$
    $rightPointer + +$
```

3 The experimental performance of the algorithms

3.1 The working environment

The execution code was written in the C programming language, and the parallelization of the results was implemented using the OpenMP library [1]. The execution hardware consisted of an AMD Ryzen 5800H processor running at 3.4GHz and 16GB of RAM at 3200MHz. During the execution and collection of experimental data, no other processes were running to ensure fairness in the execution environment. Additionally, timing was performed internally within the code using the *omp_get_wtime()* command, and only the execution time of the algorithm was measured. Finally, it should be mentioned that the processor has 8 physical cores and 16 threads, meaning that the same level of speedup will not be observed when using 16 threads instead of 8, compared to variations such as 8 instead of 4.

3.2 The Bubble sort results

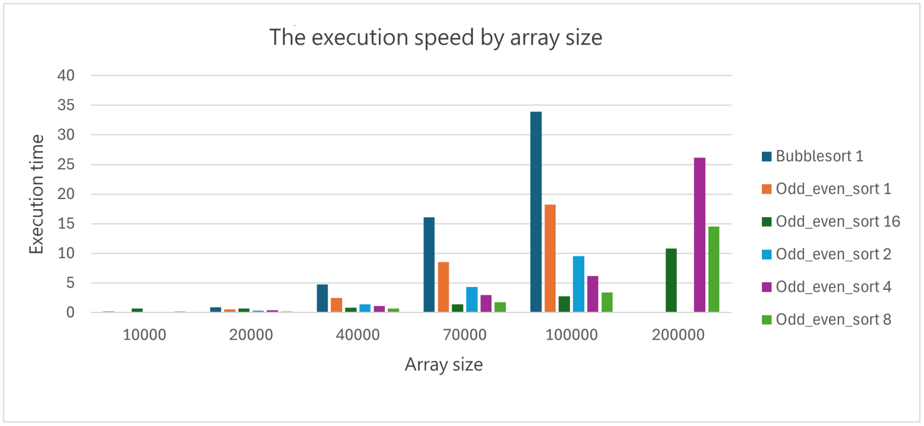

Figure 5: The execution times of Bubble sort and Odd even sort.

In Figure 5, the execution times of the algorithms are shown, along with the number of threads utilized depending on the array size. Several interesting results can be observed: for very small array sizes, parallelization provides no speedup, which is expected due to the overhead introduced by the additional functions required to utilize the system's extra resources. Moreover, the *Odd-Even Sort* algorithm exhibits lower execution times even when using only a single thread. This implies that its overall execution time is lower, making the algorithm more efficient. For larger array sizes, we observe that the additional threads provide a significant speedup and are not affected by the overhead introduced during parallelization.

3.3 The Bucket sort results

In this case, beyond inspecting the speedup provided by the additional hardware, emphasis was also placed on observing the importance of the number of buckets in the speed of the *Bucket Sort* algorithm. Starting with the number of buckets, it is observed that, generally, additional buckets in large array sizes offer a small percentage of speedup, provided that there are enough available threads to utilize them effectively. Moving on to the comparison of methods, it is initially observed that *Bucket Sort* is faster even when using a single thread compared to *Merge Sort*, which is the expected outcome since their complexities are $\mathcal{O}(N)$ and $\mathcal{O}(N \log N)$ respectively. Next, we observe an approximately 40% reduction in execution time for each doubling of available threads. Considering that the ideal reduction would be 50%, this indicates that both the implementation of the algorithm and the algorithm itself achieve a very high level of parallelization efficiency.

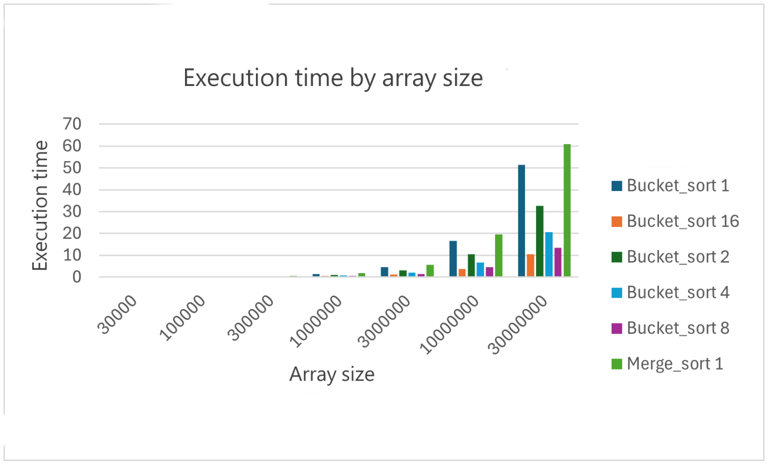

Figure 6: The execution times for the Bucket sort and Merge sort algorithms.

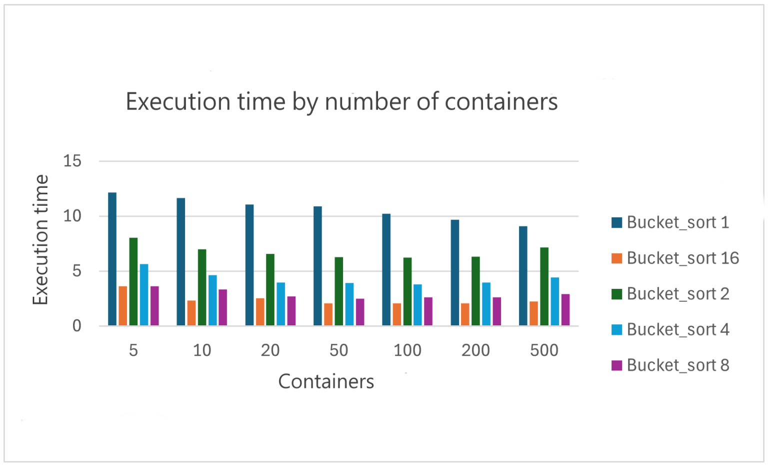

Figure 7: The execution times of the Bucket sort for different container sizes.

4 Conclusions

According to the experimental data as well as the theoretical background, it is clear that by utilizing modern methods, libraries, and tools, we can successfully achieve a significant speed-up in the performance of the *Bubble Sort* and *Bucket Sort* algorithms, reaching execution times that are unattainable through serial code execution. At the same time, it is worth noting that the programmer still plays a significant role in the design and performance of the program, as they are responsible for the optimal utilization of resources and, when necessary, as in the case of the *Bubble Sort* algorithm, for modifying the mathematical method itself to favor parallel processing. Finally, it is important to mention that serial execution still plays a role in the development of programs, especially in fields where there is no significant need for intensive resource usage or in domains where parallelization cannot be effectively applied.

References

[1] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[2] HackerEarth. *Bubble Sort Algorithm Tutorial.* URL: `https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial`.

[3] Yernar Kumashev. *GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm.* URL: `https://essay.utwente.nl/80585/1/Final_Research_Paper.pdf`.

[4] Rashita Mehta. *Bucket Sort Algorithm.* URL: `https://www.scaler.in/bucket-sort-algorithm/`.

[5] Ricardo Rocha and Fernando Silva. *Parallel Sorting Algorithms.* URL: `https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf`.

[6] Christopher Zelenka. *Parallel Merge Sort.* URL: `https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf`.