Final Project - MC970

Name: Renan Matheus da Silva Florencio. **RA**: 244808.

Name: Gabriela Martins Jacob. **RA**: 186087.

Name: Felipe Gabriel Brabes da Silva. **RA**: 247085.

System Description. SO: Linux Ubuntu 22.04 LTS. Compiler: gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0. Processor: Intel® Core™ 7 150U × 12. RAM: 32 GB. Graphics: Mesa Intel® Graphics (RPL-U). **Obs**: Every execution time plot is a mean of ten runs.

Project Description

The project chosen was the **Sorting-Visualizer** by Dipesh Mann available at Github[1]. It is a visualizer for sorting algorithms which allows for a better understanding of how each algorithm works. By default, the sorting algorithms implemented are: merge sort, quick sort, bubble sort, heap sort, insertion sort and selection sort. None of these algorithms had parallel versions.

Our contribution to the project is the implementation of parallel versions of bubble, selection, merge, and quick sort, and also a new sorting algorithm and its parallel version: bitonic sort. Not only did we improve the code to enable the visualization of these algorithms running in parallel, but we also implemented routines to compare execution times, providing both a visual understanding of how parallelism reduces execution time and a clear time comparison between the serial and parallel versions. Videos of the visualization running for both the serial and parallel versions of the algorithms can be found in the README file on the Github repository, which was forked from the original project[2].

Implementation

All the parallel sorting algorithms were implemented using OpenMP, since sorting algorithms are easier to parallelize on the CPU and the main objective of the project is not to achieve the greatest speedup possible, but rather to enable a better comprehension of how parallelization works, which would not be possible with the thousands of threads usually used in the GPU implementation of parallel programs.

It is important to note the difference between execution times when using the visualizer and when not using it. The visualizer must make use of sleep time between iterations in order to make sure the visualization does not break, making it inappropriate for time comparisons. Therefore, the time comparisons between the serial and

[1] https://github.com/dipesh-m/Sorting-Visualizer

[2] https://github.com/RenanFlorencio/Sorting-Visualizer

parallel versions should not be done while visualizing the algorithms. For this reason, non-visual versions of the algorithms were also implemented and compared, as detailed in the Performance Analysis section.

Each subsection below is dedicated to a concise explanation of how parallelization is implemented for each of the sorting algorithms and the changes made to the visualizer.

Bubble Sort

Bubble sort is one of the simplest sorting algorithms: it compares each two elements, sorting them until the entire array is sorted. Since these comparisons are made sequentially as the algorithm moves along the array, we can't directly parallelize it. A possible solution is to first compare even-starting pairs and odd-starting pairs. In this manner, in the same iteration, one shift doesn't need to be sequential with the others, and OpenMP may be used for parallel computing.

Selection Sort

Selection Sort is another simple algorithm, but which cannot be directly parallelized. At each iteration, the smallest element of the remaining array is placed to where it's sorted. Because of this, the swap is not parallelizable. However, the comparisons to choose the smallest element are parallelizable, and we explore this possibility when writing a parallel version. Basically, each thread will hold a portion of the array and have a local minimum index and, at the end of the comparisons, we make a final comparison between the threads to get the global minimum.

Merge Sort

Merge Sort is a recursive divide and conquer algorithm, meaning we cannot parallelize it using `#prama opm parallel` and must use tasks, since the number of iterations is unknown beforehand. It is also important to note that we are using a size threshold (5000) for the creation of tasks at each recursion, since small arrays tend to flood the task scheduler and worsen performance.

Besides that, we only need to create a task for each recursive call and always wait before starting the next iteration.

Bitonic Sort

Bitonic Sort is another divide and conquer algorithm that, like Merge Sort, creates tasks for recursive calls when the input size is above a given threshold. However, while Merge Sort builds a monotonically ordered sequence, Bitonic Sort constructs a sequences that first increase and then decrease. In this implementation, the merge steps do not involve OpenMP parallelization; parallelism is applied only to the recursive

division of tasks.

Quick Sort

This is also a divide and conquer algorithm. It sets a pivot of each array and does recursive calls to partition the array in such a way that elements smaller than the pivot are to its left and elements greater than the pivot are to its right. We are able to parallelize the sorting of each subarray using OpenMP tasks, as before.

A single thread begins the execution and each recursive calls creates two tasks, for the subarray to the left of the pivot and to the right. The same over-flood control was applied using a size threshold for task creation. A portion of the `partitionArray` subroutine was also parallelized to faster compute the number of values smaller than the pivot in a given array.

Visualizer

The visualizer had to be modified to use a sort of busywait to avoid work-steal from happening on OpenMP. When a thread is set to sleep, the scheduler will wake up another thread, not leaving enough time for the visualizer to display the operation on the screen. By keeping the thread awake and busy, we avoid this problem and enable the visualization.

Performance Analysis

This section is related to the comparisons made without the visualizer. We compared each parallel algorithm to its serial counterpart for several input sizes; the code can be found in the repository under `no-visualizer/runner.py`.

Bubble Sort, Insertion Sort and Heap Sort, the slowest algorithms, were compared for arrays of sizes 2^{10} up to 2^{16}. Bitonic Sort, Merge Sort, Quick Sort and Heap Sort were compared for arrays of sizes 2^{10} up to 2^{21}. The algorithms were executed ten times for each array size in order to obtain a better estimate of the time needed, and a 95% confidence interval is shown in each plot.

Figures 1 to 5 show the performance of each algorithm, and Table 1 shows the speedup between the parallel and serial versions of each algorithm. We can see the parallel versions always beat their serial counterparts for some large array size, showing the relevance of these improvements when working on large data. We can also see that merge sort is the best algorithm, having the best serial times and the best speedups achieved.

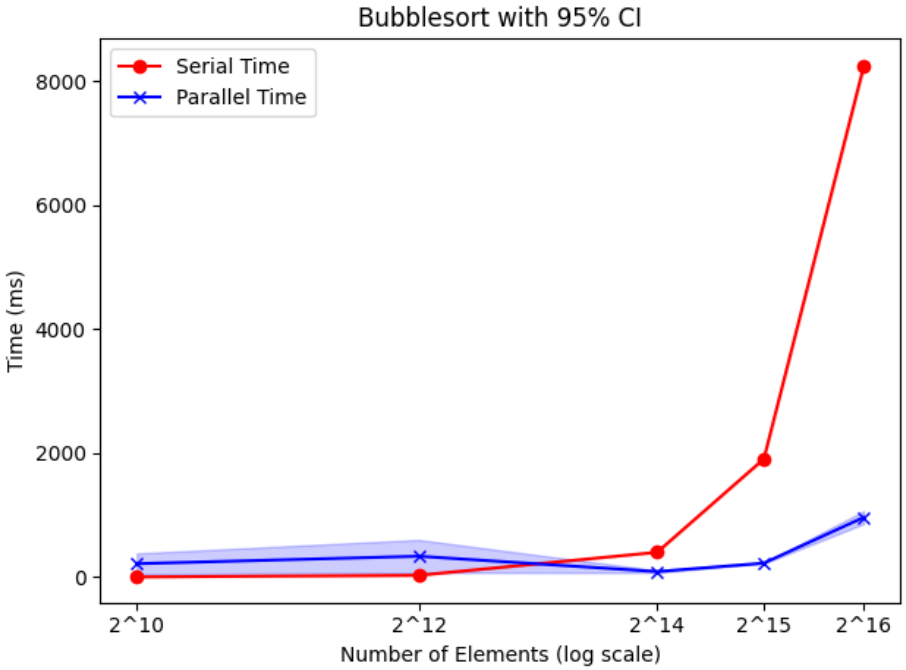

Figure 1: Bubble Sort time comparison between serial and parallel versions. A smaller maximum size array was considered since the serial version time explodes too quickly.

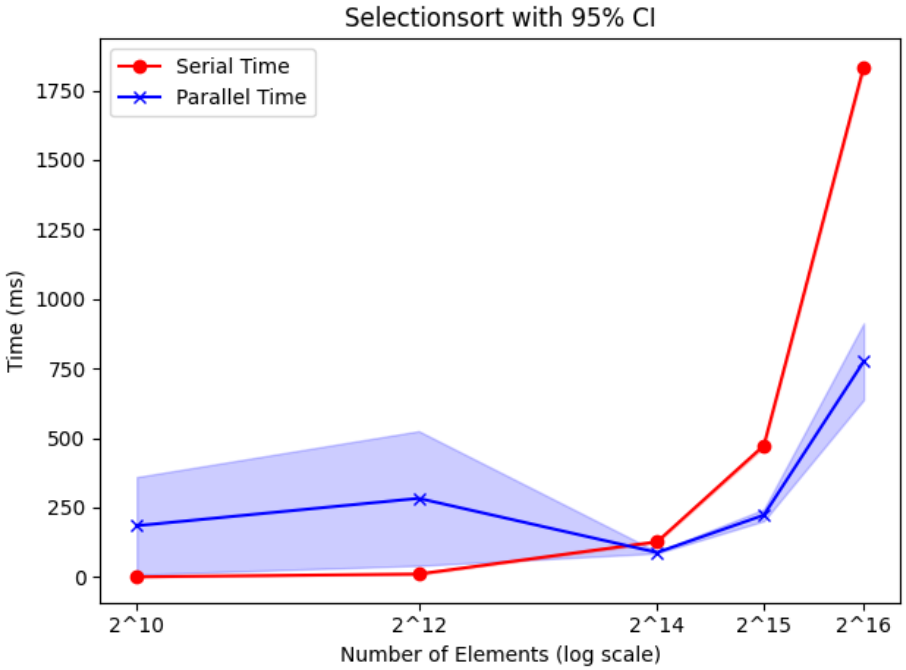

Figure 2: Selection Sort time comparison between serial and parallel versions.

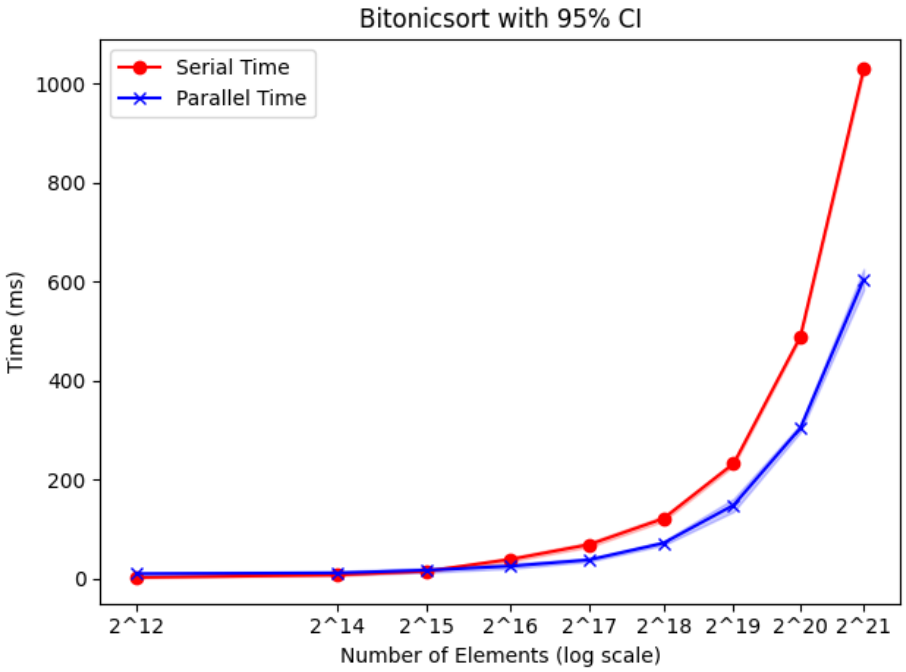

Figure 3: Bitonic Sort time comparison between serial and parallel versions.

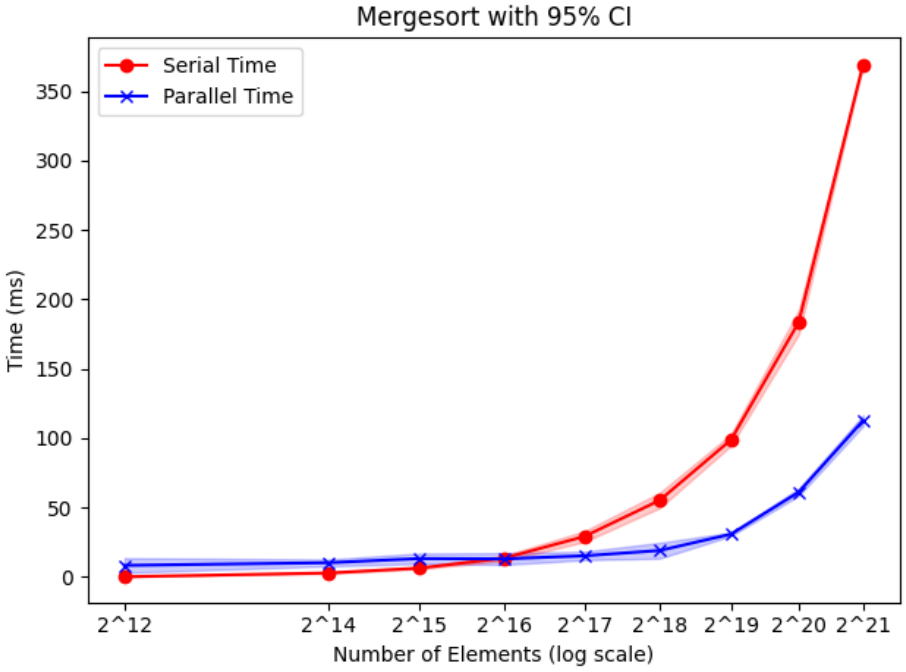

Figure 4: Merge Sort time comparison between serial and parallel versions.

Array size	BubbleSort	SelectionSort	BitonicSort	MergeSort	QuickSort
2^{10}	0.01x	0.00x	–	–	–
2^{12}	0.43x	0.04x	0.14x	0.01x	0.06x
2^{14}	2.63x	1.11x	0.51x	0.25x	0.35x
2^{15}	4.98x	1.29x	0.95x	0.46x	0.57x
2^{16}	5.97x	1.84x	1.73x	1.14x	1.58x
2^{17}	–	–	1.88x	2.06x	2.55x
2^{18}	–	–	1.49x	3.17x	3.55x
2^{19}	–	–	1.62x	2.91x	3.17x
2^{20}	–	–	1.66x	2.82x	2.69x
2^{21}	–	–	1.79x	2.99x	3.03x

Table 1: Speedup of parallel sorting algorithms relative to the serial version for varying array sizes.

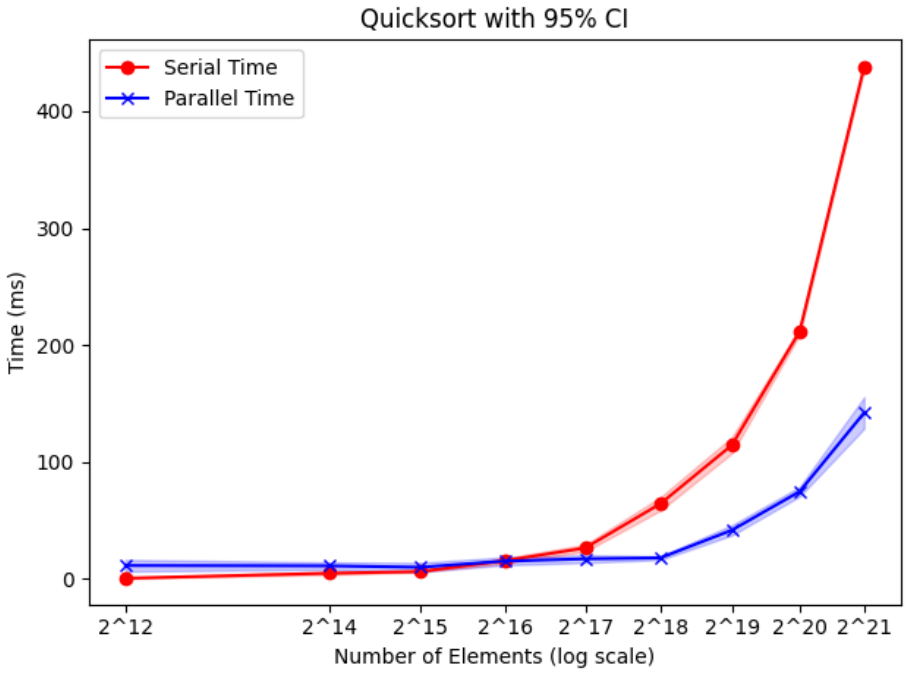

Figure 5: Quick Sort time comparison between serial and parallel versions.