Parallel Algorithms and Programming

Lab2 report

Ahmad Ghalawinji: ahmad.ghalawinji@grenoble-inp.org
Emile Wansa: emile.wansa@grenoble-inp.org

March 20, 2022

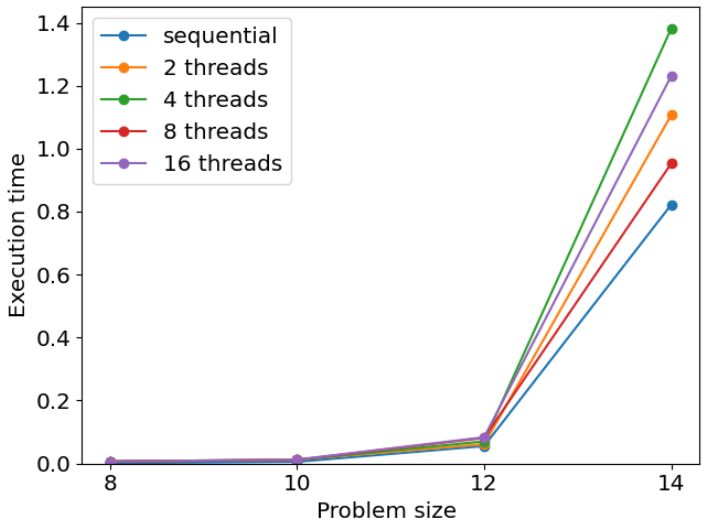
1. Introduction:

The purpose of this lab is to implement different sort algorithms in both sequential and parallel implementations, the parallel implementation was done using the OpenMP library.
For each of the Sorting Algorithms, we ran the code on the processor described below:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    2
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 78
Model name:            Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
Stepping:              3
CPU MHz:               1900.031
CPU max MHz:           2800,0000
CPU min MHz:           400,0000
BogoMIPS:              4793.46
Virtualisation:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0-3
```

2. Bubble sort:

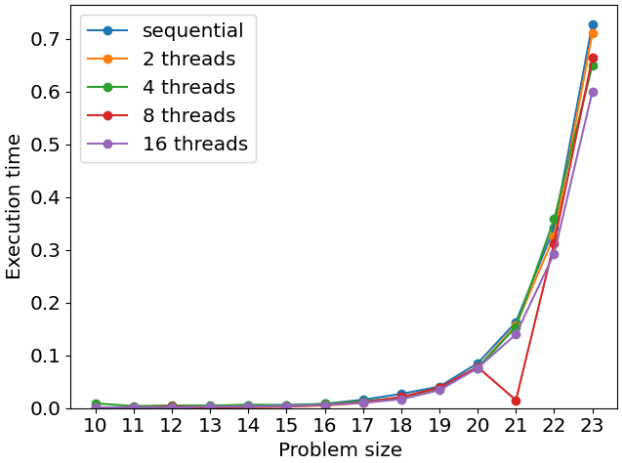
Concerning the scheduling policy for the parallel bubble sort we decided to use a static scheduler since we wanted to divide the workload between the threads equally and assign to each an equal chunk size s = size/nb_Threads. At first, we were having an error where the edges of the chunks while sorting were being swapped. We then created private variables i and k where i iterated for the maximum number of threads and k is used to iterate through the chunk (s) in each thread. We noticed that our method is always slower than the sequential time, hence we didn't plot the speedup.

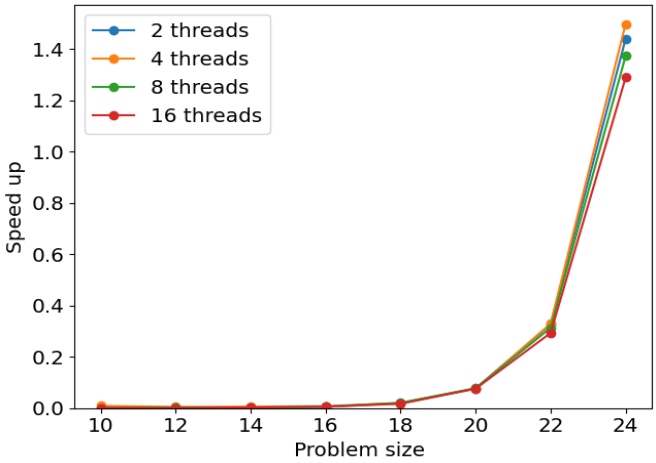


3. Merge Sort with tasks:

Merge Sort is a Divide and Conquer algorithm. It divides the input array in half, calls itself for each half, and then combines the two sorted parts. The merge() method is used to join two halves together. For the parallel_merge_sort we divide the work by task and at the end, we wait till the "Division" of the array is done and then merge in parallel.

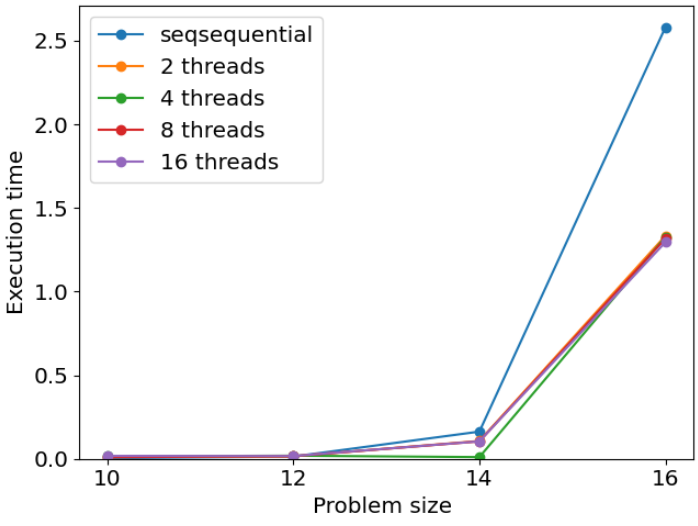
For a small number of elements, we have basically a very small speedup. We can add an if statement in #pragma that only executes the code in parallel for the case of (N > 20).

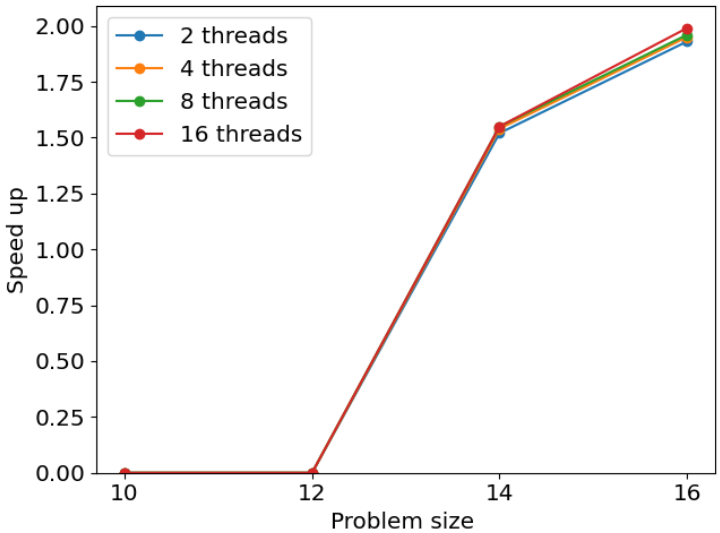




4. Odd-even sort

In the bubble sort algorithm, every computational part is related to the previous part, thus it's not efficient to use a parallel algorithm with it. While in the odd-even sort algorithm we can divide our array depending on the index(odd or even) and implement the sort algorithm on each party in a parallel manner because we can compute each part separately. We can see here that contrary to the bubble sort, we have a speedup of roughly 2 (1.97~1.99) for larger problem sizes.





5. Quick Sort:

We Tried to implement the Quick Sort algorithm but the compare function wasn't working properly for unit64_t. And the sequential code isn't working. However, we tried to implement the parallel qsort algorithm even if we couldn't check our result. We adjusted the makefile and created a main similar to the previous exercises with the necessary changes.