Parallel Computing Challenge N.1 - Parallel MergeSort with OpenMP

Davide Villani
Francesco Virgulti
Emanuele Caruso

1 Experimental Setup

In the following implementation we are going to solve the MergeSort Algorithm by exploiting parallelism, our input is defined by an array containing random elements ranging from 0 to 95, the output is going to be the sorted vector.
All the test were run on the following architecture specifics:
CPU(s): 16
Model name: 12th Gen Intel(R) Core(TM) i7-12650H
CPU family: 6 - Thread(s) per core: 2 - Core(s) per socket: 10 - CPU max MHz: 4700
RAM size: 16GB - DDRAM4 - 3200 [MT/s]
L1d: 416 KiB (10 instances) L1i: 448 KiB (10 instances) L2: 9.5 MiB (7 instances) L3: 24 MiB

2 Design Choices

In this section we are going to talk about our main design choices in the implementation.
Firstly we create our parallel region and inside of it the *master/single thread* that has the goal of creating the tasks which are going to be executed by our threads.
MergeSort is implemented in a recursive way by splitting the array in two sub-arrays and solving it by the divide-and-conquer technique, in this case we are going to exploit this by creating different tasks for each division, that are going to be executed by different threads in parallel.
At this point we need to briefly discuss the choice of the *DEPTH MAX* parameter, since the division in sub-arrays is going to last until we find single element sub-arrays, we don't want to overload our threads so we will stop the task creation once we reached a certain depth in the division; this depth is chosen depending on the number of threads available on the maching, in particular we will consider:
$DEPTH\_MAX = \log_2(NUM\_OF\_THREADS)$
This mean that if we have N threads, at level depth max we will have N threads that are going to work on different tasks, and from this point on each thread is going to work sequentially on the problem, the synchronisation of these is implemented by the openMP directive *taskgroup* so that only after every child of this level of recursion is solved we can go on with the next instances of the level above avoiding partial sorted arrays, this synchronisation will also avoid synchronising the tasks created in the parallel merge that will be explained after.
At this point it's pretty clear how our recursion tree is divided by splitting the arrays in half and creating parallel tasks until we reach a certain depth, but if at the depth max level we have that every thread of our cpu has some workload, all the other levels above are going to use half of the threads each time so for example at level 2 we will have 4 working threads and N-4 non working threads.
To overcome this problem we decided to parallelize the merge function by dividing it in two halves, and one thread is going to work on each half of the array.
Before explaining how this work we need to say that also this function have a cut-off, but differently from the recursive one we can start the parallel approach only if our depth level is less than (depth max - 1), this is fundamental because if we started it as before in (depth max) we would have an overload of threads in this level, since all the threads are already busy doing tasks, but by starting from 1 level above we know that at least half of the threads will be IDLE, so we can exploit this by using them to help us merging the subarrays.

In this way we don't guarantee that at each level all the threads are going to work, after (depth max - 1) all the above levels will still have half of the previous number of threads, but we are still using at least two threads to compute a merge at each level.
The main idea is the following: Since we assume that the subset of the array we want to merge are sorted, we can use two threads, one that take the minimum of the minimum values of each subset and one that take the maximum of the maximum values of each subset, the first one is going to write the taken values starting from *outputarray[begin]* and writing from left to right, while the second thread is going to to the opposite starting from *outputarray[end-1]* from right to left, this way we are sure there are not going to be any shared memory problems and the threads will exactly divide the merging load in two, improving parallelism in these levels of depth.

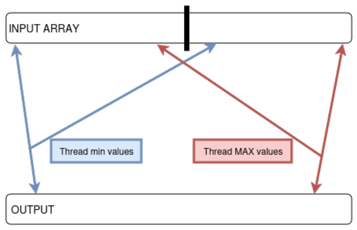

Figure 1: Visual example of two threads working on the same array.

2.1 Results

The following results are computed by running the algorithm with different input size array, ranging from 10^5 to 10^9 elements. In fig5. we can clearly see how the minimum time is reached when depth max is exactly the log2 of the number of threads (which in this case is 16 as shown before)

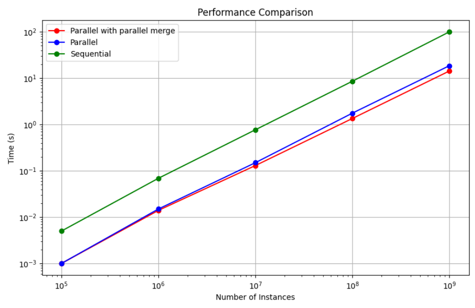

Figure 2: Performance of all the algorithms with and without parallelism

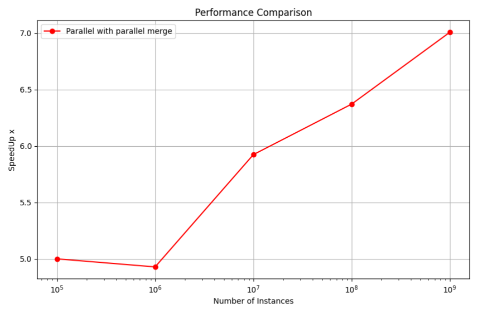

Figure 3: Speedup of the fastest parallel implementation

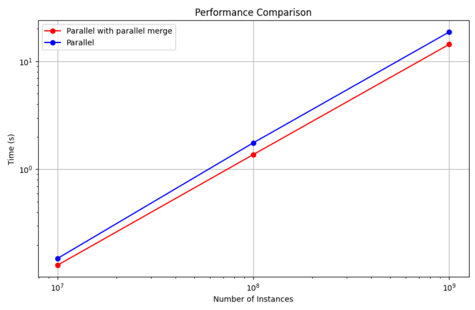

Figure 4: Speed difference between parallel with and without parallel merge, on average 1.35x speedup

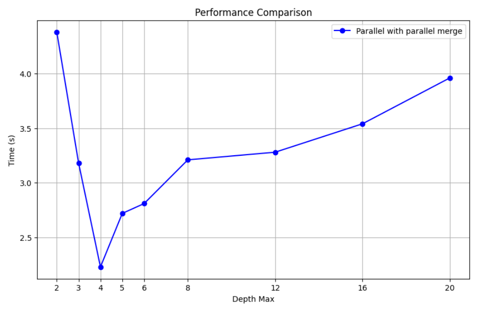

Figure 5: Speed at different depth levels, with array size = 160milion