MergeSort Challenge Report

Daniele Kopyshevskiy
280470

Giacomo Pauletti
271544

Lorenzo Pettenuzzo
273597

Parallel Computing (2024-2025)
Politecnico di Milano

1 Introduction

In this report we overview the many approaches we attempted to fully exploit the multicore system of a laptop. We present them in order of complexity and performance.
Specifically, the task was to optimize the mergesort algorithm using C++ and OpenMP.
As benchmarking computer we used a Acer Aspire A515-54G with an Intel Core i7-10510U processor, with 4 physical cores and 8 logical cores, 16GB of RAM and running Ubuntu 22.04. The compiler used is g++ with the following flags `-O3 -fopenmp`.

Note that the first 2 solutions presented below are not present in the code: they are used as a basis for building the solutions present in the code.

2 Serial solution

The serial solution, already given to us, has a time of execution around `9.7s` with an array of roughly `380MB` of size. The speedup of the following solutions is based on the same array size and the relative serial time.

3 Recursion paralellization

As first solution, we parallelized the recursive calls of the algorithm, leaving the merge part untouched. We added a new parameter, `depth`, which is the number of levels of recursion which have to be parallelized. We set it to be the between 3 and 4, which correspond to use at maximum 16 threads at the same time, ideally 2 per logical core. We also tried to play with environmental variables, but with little success. Of course `OMP_MAX_ACTIVE_LEVELS` must be greater or equal than `depth`. We tried to use turn on `OMP_DYNAMIC`, which gave a little increase in performances. Instead setting `OMP_PROC_BIND` to true and/or `OMP_WAIT_POLICY` to active decreased the performances.

Finally we tried different configuration of the OpenMP constructs. This problem is easy to be translated in OpenMP tasks, so all our attempts were based on those. We tried using nested parallelism constructs: the 2 “recursive” tasks were generated inside a parallel region with 2 tasks. We found this being the best configuration, compared to having just 1 level of parallelism (with more than 2 threads) and assigning the tasks to the threads of that level.

The speedup `P` achieved with this improvement is between `P=3.5` and `P=4.0`. This is reasonable since for a big part of the execution, from recursion layer 3, 8 threads are run but in the previous part of the execution an exponentially decreasing (due to “reverse” recursion) number of thread is used. And those threads are used to merge the largest arrays. When using 16 threads (4 level of parallel recursion) the speedup is not improved nor degradated.

4 Merge parallelization

From the speedup achieved in the previous point we realized that the bottleneck is now the merge: the number of threads decreases exponentially, when "exiting" from recursion levels, while the dimension of the subarrays to sort and merge are increasing exponentially.
When a thread, in the first `depth` levels, needs to merge the two sorted subarrays `A` and `B`, we pickup the value in the middle (median) of array `A` and with a binary search we find the same value (or the closest higher value) in `B`. We have then subdivided `A` in `A1` and `A2` and `B` in `B1` and `B2`. The higher array have as first element the middle value of `A`, for `A2`, and the binary searched value, for `B2`. The arrays are then merged by couples (`A1` and `B1`, `A2` and `B2`) directly in the respective portions of the final array. Thanks to the binary search, final array order is guaranteed.

The speedup `P` achieved from the serial version is between `P=4` and `P=4.5`. The improvement is very little actually. This might be due the fact that even though now more threads are merging, they are still not fully exploited.
This brings to the last method

5 Merge full parallelization

Instead of doubling the number of threads which performs the array merge, why not using as many threads as needed to have always $2^{\{\text{depth}\}}$ threads used. This would mean, at recursion level 0, using 8 threads to merge the array (instead of 2), at level 1 again 8 threads, instead of 4 (2 each recursive call).
This is the third solution proposed.
In the merge phase, there is not only 1 binary search, but as many as needed to guarantee the fixed amount of threads (actually, tasks) in execution at the same moment. This solution is implemented in the function `MsMergeParallel`.
Against our predictions, this solution did not improved as we expected. Ideally, a speedup of `P=8` (number of logical processors) was what we were looking for, but instead we achieved no more than `P=5`.
We formulated also another solution, implemented in funciton `MsMergeParallel2` which is based on making also the merge phase recursive. The arrays to merge are subdivided in 2 subarrays each, as explained before. Then the `MsMergeParallel2` function is again invoked in the subarrays obtained, until a certain a certain array length is reached and the arrays are finally merged sequentially. This is an alternative solution of what proposed before in this section, but it reveals more efficient. With this solution we were capable of reaching more than `P=5` of speedup, with a time of execution of around `1.95s`

6 Conclusions

The code proposed parallelizes the mergesort in its recursive calls and in the merging steps, which are now recursive as well. The cutoff on the recursive calls is based on the number of processors whilst the cutoff on the merge is based on the array size.
The speedup achieved, roughly around `P=5` is probably close to be the best possible achievable on the benchmarking machine. The optimization which still might be obtained is in the choice of the cutoffs. Maybe a different way of selecting the cutoffs, based both on the number of logical cores and on the array size, would be better performing.
Also, we put little effort in balancing the sizes of the merging arrays, due to time reasons. This could be source of further improvements.