Parallel Computing - Challenge 1

Parallel implementation of merge sort using OpenMP

Andrea Barletta

A.Y. 2024/2025

1 Introduction

The goal of the challenge is to parallelize an existing program that performs the merge sort of a vector. Recalling that the merge sort works as follow:

1. Divide array into halves
2. Call sort on both
3. Merge

It's clear that this algorithm can benefit from parallelization, particularly in the second step. It will also be interesting to compare the performance (measured in execution time) of the serial version against the parallelized version, and to examine the effects of different design choices.

2 Experimental setup

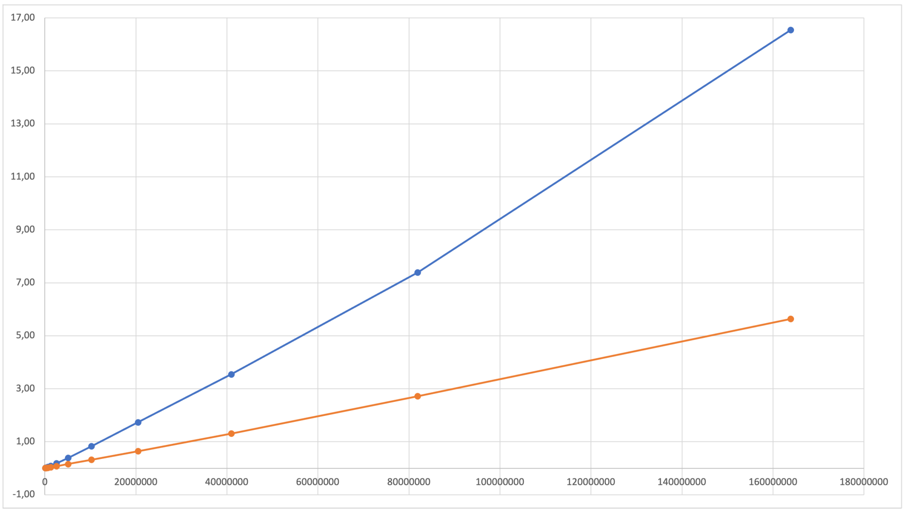
The programs were executed on an Apple M1 chip, which features 8 CPU cores. This number is relevant for the way OpenMP creates threads using the "*#pragma omp parallel*" directive

3 Performance measurements

Each version was run 10 times, and the average execution time was calculated, excluding the minimum and maximum values.
The table below shows the execution times (in seconds), where in the parallel version, we create 8 threads and a total of 7 tasks.

Vector length	Serial time (s)	Parallel time (s)
80000	0.009	0.003
160000	0.012	0.005
320000	0.020	0.008
640000	0.042	0.017
1280000	0.088	0.037
2560000	0.187	0.073
5120000	0.389	0.158
10240000	0.828	0.316
20480000	1.734	0.641
40960000	3.545	1.312
81920000	7.385	2.718
163840000	16.536	5.631

It's also interesting to have a look at a plot of these values, and comparing their growth.



Blue line: serial version Orange line: parallel version

The parallel version is approximately 3 times faster than the serial version.

4 Design choices

Each half of every layer of the merge sort is assigned to a new thread using the task directive.
Creating more tasks than threads can lead to a performance slowdown due to the overhead of handling task dispatch. Therefore, after reaching a certain depth in the merge sort tree, the algorithm proceeds sequentially.
It's also interesting to explore different configurations, particularly the impact of varying the number of threads and tasks.

# tasks / # threads	1	3	7	15	31	63	127
1	4.389	4.429	4.415	4.406	4.681	4.416	4.44
2	4.388	2.388	2.361	2.509	2.349	2.356	2.365
4	4.4	2.352	1.321	1.848	1.622	1.616	1.621
8	4.467	2.333	1.333	1.023	1.175	1.196	1.168
16	4.41	2.48	1.347	1.064	0.976	1.033	1.021
32	4.406	2.341	1.322	1	1.012	1.1	0.983
64	4.409	2.347	1.368	1.13	1.03	0.988	1.037

The red area indicates a configuration where there are more threads than CPU cores, resulting in little to no improvement. The orange area shows configurations where there are more tasks than threads, with minimal improvement despite increased resource usage.
The most efficient configuration, in terms of threads and tasks used relative to performance, is 8 threads and 15 tasks (purple value).
It's slightly faster than our initial approach since we can benefit from an extra task assigned to the remaining thread.
The execution times for all the test vector lengths are reported in the table below

# elements	80000	160000	320000	640000	1280000	2560000	5120000	10240000	20480000	40960000	81920000	163840000
time (s)	0.001	0.004	0.007	0.015	0.028	0.6	0.127	0.231	0.494	0.976	2.09	4.136