Problem Description

The problem I will be discussing involves parallelizing Merge Sort and counting the number of inversions. The original problem is provided in Polish in the file **instructions.md**.

Tests

I have generated 5,000 test cases, each consisting of an array of size in the range from 20,000 to 100,000.

Single-thread

I have introduced a few optimizations for the single-threaded case. The table below shows the average times for each program on several randomly generated tests:

	Avg Time (s)	Code Description
1	20.39	A naive merge sort implementation
2	9.72	Disabled C/C++ stream synchronization
3	10.57	Added insertion sort for small subproblems
4	9.69	Limited the number of comparisons

Unsurprisingly, adding *magic lines* speeds up the process immensely. However, employing insertion sort for small subproblems, which seemed like an obvious speed-up since it avoids the recursive stack, turned out to decrease performance. Finally, I limited the number of comparisons in the code, which resulted in a small speedup

Other potential optimizations could include using built-in functions (e.g., `memcpy` instead of manually copying arrays), avoiding repeated initialization of the h array by reusing a single buffer, and making better use of processor registers (e.g., incrementing a local counter in the while loop and updating the global counter only at the end).

std::threads

Starting from a naive parallelization, I have introduced a few optimizations, including:
- introduced a local variable instead of updating the same counter with synchronization in the loop,
- introduced a threshold for parallelization: if a subproblem size is smaller than the threshold, no parallelization is used,
- introduced a shared buffer to avoid creating the same array repeatedly,
- counted inversions without using a global atomic counter.

	Avg Time (s)	Code Description
1	11.24	A naive parallelization
2	7.66	Optimized addressing shared resources
3	7.08	Introduced a shared buffer
4	6.72	Avoided using a global inversion counter

openMP

For a long time, I struggled to speed up OpenMP parallelization—on average, the results were equal to or worse than using a single thread. It turned out that calling `#pragma omp parallel` multiple times was the bottleneck, as it spawns multiple threads repeatedly. I managed to bypass this and ended up with the following results:

×	Avg Time (s)	Code Description
1	9.56	A naive use of the library
2	6.17	Optimized addressing shared resources

More threads

The tests above were run on my local machines. For each type, I selected one and ran tests on the `student` machine. Notice that beyond a certain point, execution does not get faster - as we optimized

	Threads	Avg Time (s)
single thread	1	12.04
threads	8	9.31
openmp	8	6.08
threads	12	6.59
openmp	12	6.15
threads	16	7.25
openmp	16	6.55
threads	64	6.12
openmp	54	6.46

only the recursion and not the merging step. Additionally, performance seems to get worse if too much threads are used. Presumably, this is due to the overhead of managing threads.

Instructions

Running `python3 run.py` in the root folder generates five large tests (this may take a while, so you may want to change `TEST_CASES` from 1000 to 100), runs these C++ programs, measures the average execution time, and checks whether the results are correct.

In the `code` folder, you will find all the programs discussed above. You can modify the number of threads used. By default, it is set to 8.