Challenge 1 - Parallel Computing Merge-Sort in OpenMP

Federico Pinto - Mattia Gotti - Michele Milani
10766381 - 10766863 - 10776157

Introduction

This report documents the work carried out for the first Parallel Computing challenge using OpenMP to implement the Merge-Sort algorithm in parallel, with the goal of testing its performance under different configuration parameters.

Experimental Setup

The code was implemented in a single `.cpp` file, structured as follows:

- **Main program**: Handles input parsing, memory allocation, and calls to the Merge-Sort algorithm (both sequential and parallel versions).
- **Utility functions**: Includes helper functions such as `isSorted()` to verify the correctness of the sorted array.
- **Sorting algorithm**: Implements both the sequential and parallel versions of Merge-Sort. The parallel version uses OpenMP tasks to divide the problem and perform sorting in parallel.
- **Merge function**: Implements both sequential and parallel merging of two subarrays, using OpenMP tasks for the parallel merge.

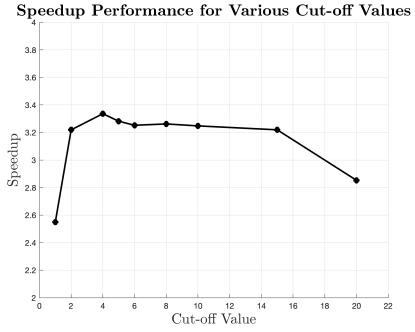
Design Choices

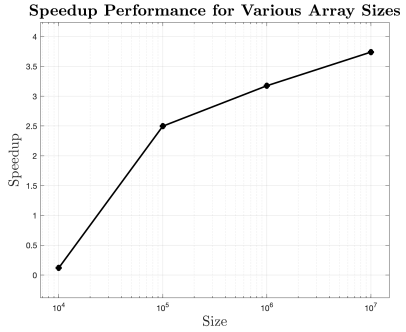
The implementation is based on a divide-and-conquer strategy, where the array is recursively split into two halves. The fact that each one of these smaller arrays can then be divided independently from the others suggests that the algorithm could be parallelized.
Parallel execution is initialized in the `MsSerial` function, which sets the maximum number of threads usable and enables their dynamic usage by the program.

Then a `single` thread is opened and the recursive function is called.
In the recursive function `MsParallel` the parallelism is managed using OpenMP `task`. It is important to note that the tasks *must be synchronized*. We observed that if the tasks are not synchronized, the verification step results in `...FAILED`.
A cut-off mechanism, based on recursion depth, was then introduced to avoid the overhead of creating too many tasks for small arrays.

Performance Measurements

Performance was measured in terms of execution time by varying the cut-off values and the size of the array.
Various cut-off values were evaluated to determine the optimal balance between parallelization overhead and computational efficiency. The results in Figure 1 illustrate the speedup trend as the cut-off values increase. The optimal cut-off value, representing the depth of recursion, falls between 4 and 6.
Figure 2 illustrates the speedup performance of the algorithm as it scales with different array sizes, highlighting its *efficiency* and *scalability* with different input sizes. As expected, the trend is monotonically increasing.





Conclusion

The implementation of Merge-Sort with OpenMP showed significant performance improvements in multi-threaded environment, especially for large arrays. The cut-off mechanism optimized efficiency by balancing the overhead of task management with the benefits of parallel computation. On the other hand, a parallel merge algorithm did not significantly improve performance. This, in our opinion, could be due to either *over-parallelization* or insufficiently large data sizes.