Parallel Computing hands-on challenge

Giovanni La Gioia (10755342) - Luca Leonzio (10789156)

Merge-sort parallelization with OpenMP

ALGORITHM

Merge Sort, is an efficient, versatile, and comparison-based sorting algorithm.
This sorting technique is based on a divide-and-conquer approach and was first introduced by John von Neumann in 1945. The algorithm works by recursively dividing the array into smaller subarrays, sorting each of these subarrays, and finally merging the sorted subarrays back together into a single, ordered array.
The average case performance of merge sort is $\theta\, n\, log(n)$, a very powerful algorithm in comparison to other sorting algorithms, even in the worst cases of sorting.

GOALS

The objective of the challenge is to improve the time efficiency of the non parallelized merge sort code, by using OpenMP library instructions.
The code provides 3 different functions that can be parallelized at some points to scale up the sorting algorithm and exploit all the advantages of parallelism.

KEY COMPONENTS

Sequential Merge Function:
MsMergeSequential() merges two sorted subarrays into a single sorted output array. It iteratively compares elements from both subarrays and copies them to the output. If the remaining portion of either subarray exceeds a specified value *(CUT_OFF)* the merging process is divided into two parallel sections. For smaller remaining subarrays, the function reverts to sequential merging.

Sequential Merge Sort Function:
MsSequential() implements the main sorting logic. It recursively divides the array until the base case is reached and invokes the merging step. If the size of the subarray exceeds the specified value cut-off, the function utilizes OpenMP tasking to parallelize the sorting of subarrays, allowing concurrent execution.
The cut-off parameter is a critical component that determines when to stop creating new tasks for parallel execution. If the size of the subarray exceeds this threshold, the function starts implementing sequential execution. This helps avoid the overhead of creating too many threads for small subarrays, ensuring that parallelism is only applied when it actually gets performance benefits.

Parallel Region:

MsSerial() initializes the parallel region. It calls *MsSequential()* from the master thread, ensuring that the sorting occurs in a parallel environment to avoid an extremely large amount of thread generated by the recursion.

Main Function: The program initializes the array with random values, and measures the execution time of the sorting process. It also validates the sorted output by comparing it to a reference sorted array.

THE CUT-OFF PARAMETER

The depth parameter plays a crucial role in controlling the parallelism. The recursive calls to sort the subarrays are made concurrently using OpenMP tasks. However, creating too many tasks for very small subarrays introduces overheads, removing the benefits of parallelism for extremely short tasks.
After conducting various experiments and analyzing the performance characteristics, we observed a logarithmic growth trend in the execution time with respect to the size of the input, as we expected to reach. Through these tests, we found that a depth value of approximately 500 ensures optimal performance for more than 10^4 array elements to be sorted.
So, in conclusion, this depth value guarantees a good switch from parallel execution to sequential execution.

RESULTS

The following screenshots represents some tests developed on the terminal with 2 different executables: oldmergesort-co represents the algorithm without using openMP, mergesort-co is instead the executable of our solution mergesort.cpp. It can be shown that in general the parallelized algorithm is nearly 4 times faster than the non parallelized one.

Anyway for arrays under 10,000 elements there is no significant difference in terms of efficiency because both methods reach a <0.01s computational time. Anyway the cut-off value can be modified in reaction to the size of arguments, in particular it can be increased for small amounts of data (under 10.000 elements). In such cases, adjusting the cut-off leads to fewer task divisions and a more efficient execution, making it competitive also with smaller input sizes.

Result shown for cutoff=512 and 10^7 array input size: