Parallel Computing - Challenge 1

Matteo Arrigo, Emanuele Severino, Nicola Noventa

21/10/2024

1 Design Choices

We started from the given sequential version of the algorithm, and we parallelized the recursive sorting calls. Initially, we created a parallel block in the function `MsMergeSort`, where a single thread starts the recursive call to the function `MsParallel`.

We compute a cutoff value (`depth`) to decide when to use the parallel version of the algorithm. Knowing that the recursive algorithm forms a binary tree of recursive calls, the value `depth` represents the maximum depth of the tree at which the parallel version of the algorithm is applied. We can compute this in two ways:

- by using the hyper-parameter `chunkSize`, which represents the maximum size of an array a single thread should handle. The default value is set to 256, so that a single thread writes a minimum of 256 contiguous cells of the input arrays, consequently minimizing false sharing issues.
- freely set by the user.

This cutoff helps to limit the overhead of managing tasks, by not creating tasks when the overhead would be too big.

If the recursive call happens at a level higher than `depth`, we create two tasks performing the sorting step, which are dynamically assigned to two threads. Otherwise, the function is called without creating tasks. In order to proceed with the merging step, the thread needs to work with the sub-arrays that are already sorted, and thus there is a barrier after task creation.

1.1 Almost-fully Parallel Version

In addition to the above choices, we also attempted to parallelize the merging step.

The idea is to run two OpenMP sections concurrently, filling the cells of the output array from the lowest positions (in ascending order) and the highest ones (in descending order) simultaneously. For the algorithm to be correct, a critical region is needed for each section to coordinate them and make the algorithm stop at the right point.

However, we noticed that the results from this implementation were significantly worse, likely due to synchronization overhead and memory issues such as false sharing. Therefore, we opted not to use this approach.

2 User Guide

The program can be executed with or without arguments. If executed without arguments, it runs a series of tests.

```
bin/mergesort-co <size> <mode [0,1,2]> <depth|nthread>
```

Where:

- `size`: Number of elements to sort.
- `mode`: Execution mode (`0` = sequential, `1` = parallel with threads, `2` = parallel with cutoff depth).
- `depth|nthread`: Parameter for parallel mode.

2.1 Usage Examples

- Sequential execution to sort 10,000 elements: `bin/mergesort-co 10000 0`
- Parallel sorting with 8 threads on an array of 50,000 elements: `bin/mergesort-co 50000 1 8`
- Parallel sorting with a maximum recursion depth of 4: `bin/mergesort-co 50000 2 4`

3 Performance Measurements

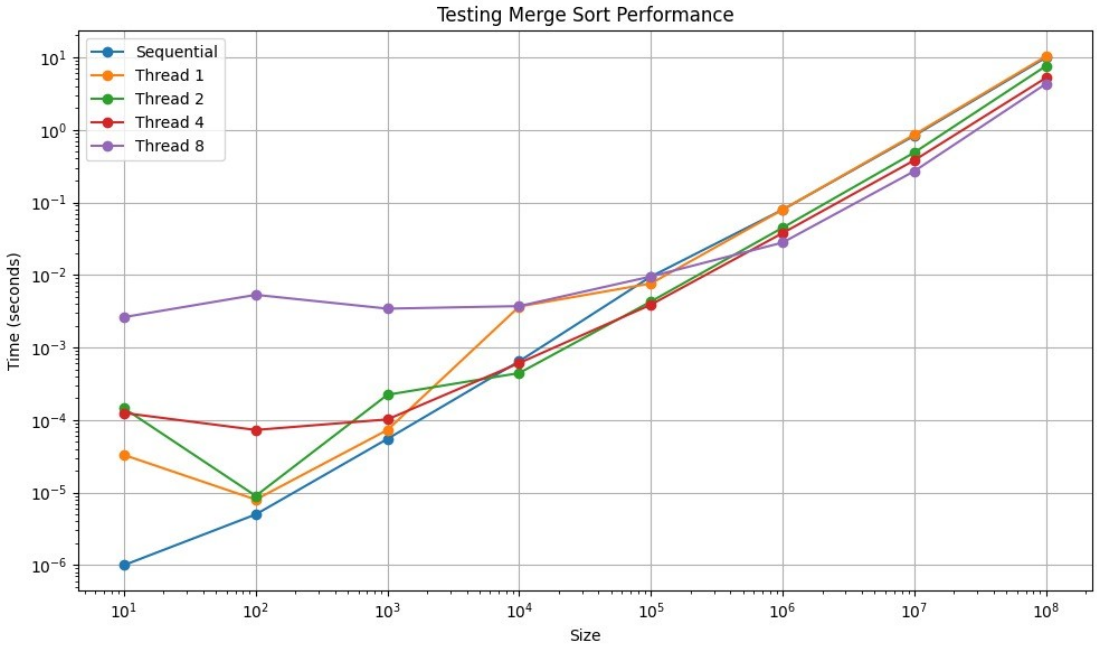

Figure 1: Performance Plot

Here is a logarithmic plot of the execution times of

- the completely sequential algorithm
- the parallel algorithm with respect to different number of threads used

As we can expect, for small arrays the sequential algorithm performs better then the parallel one, while for larger arrays the asymptotic performance seems better as the number of thread increases, with similar times for the sequential version and the parallel one with 1 thread

In particular we report the performance parameters for this run on the array with size $n = 10^8$

Sequential time	$T^* = 9.938$				
Parallel times with p threads	$T_1 = 10.321$ $T_2 = 7.601$ $T_4 = 5.220$ $T_8 = 4.320$	Speedup	$SU_1 = 0.963$ $SU_2 = 1.307$ $SU_4 = 1.904$ $SU_8 = 2.300$	Efficiency	$E_1 = 100.00\%$ $E_2 = 67.89\%$ $E_4 = 49.43\%$ $E_8 = 29.86\%$

Table 1: Performance parameters

Since a significant portion of the algorithm remains sequential, we can notice that the speedup is not constant and the efficiency decreases as the number of threads in the parallel team increases.