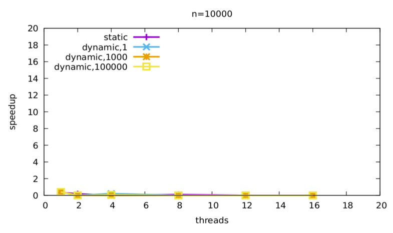
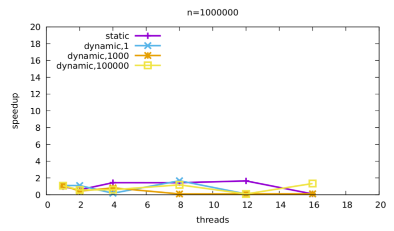
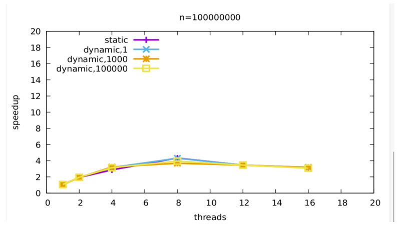
Assignment: OpenMP Task

1. Reduce

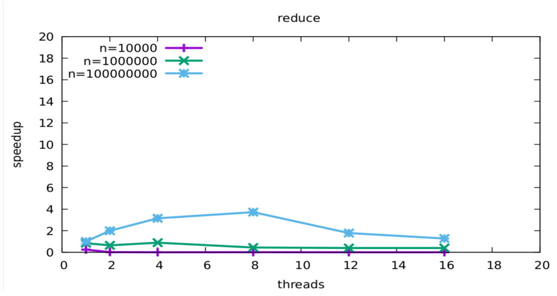
Question: Implement computing the sum of an array using the OpenMP's tasking construct (15 pts). Write the code in directory reduce/ in file reduce.cpp. Test the code is running correctly with make test
-- done

Question: Benchmark the code on hpc-cluster using make bench (10 pts). And plot results using make plot (10 pts). What speedup do you achieve with 16 threads; is it better than your previous Reduce implementation or not, why?

Reduce implementation without Task:

n=10000 	n=1000000 	n=100000000 

Reduce implementation with Task:



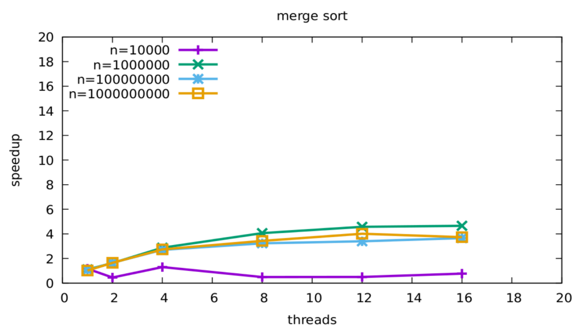
Speedup across no of threads follows a similar trend for the reduce implementation with and without Task.
For n=10000 → speedup for 16 threads is same while using task.
For n=1000000 → speedup for 16 threads is slightly higher using task.
For n=100000000 → speedup for 16 threads is slightly lower than while using task.
Default static scheduling tries to divide the array into equal chunks. The implementation tends to follow a similar path as static schedule implementation, in spite of being implemented using task. The overhead of the creation of tasks reduces the speedup. Hence the decrease in speedup for higher values of n.

2. Merge Sort

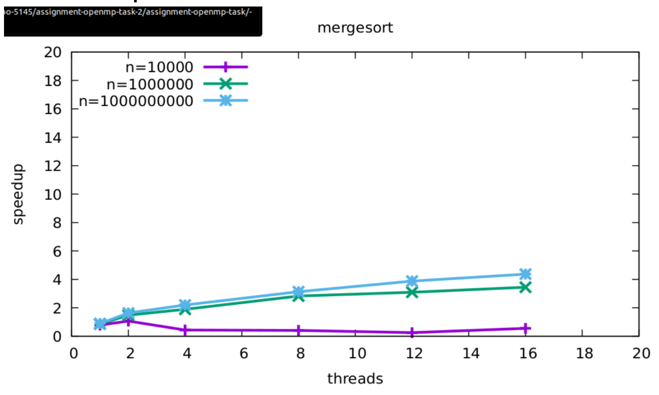
Question: Implement a parallel function using OpenMP's tasking construct to perform merge sort on an array of integer (15 pts). Write the code in directory mergesort/ in file mergesort.cpp. Test the code is running correctly with make test
--done

Question: Benchmark the code on hoc cluster using make bench (10 pts). And plot results using make plot (10 pts). What speedup do you achieve with 16 thread; is it better than your previous loop-based Merge sort? Why?

Reduce implementation without Task:



Reduce implementation with Task:



For n=10000 → speedup for 16 threads is same for while using task.
For n=1000000 → speedup for 16 threads is lesser using task.
For n=100000000 → speedup for 16 threads is slightly higher than while using task.

Speedup depends on how the tasks are divided. Here the speedup has increased slightly or remains the same with and without task. Tasks were implemented to make nested parallelism easier. Nested parallelism leads to oversubscription which could explain the increase in the speedup of merge sort code using task.
When implementing pruning the working tends to be similar like dynamic scheduling. Due to pruning at 100000 speedup increases for higher values of n for 16 threads.