Assignment 5: OpenMP - advanced

The purpose of this assignment is for you to learn more about

- the tasks construct of OpenMP,
- how parallel loops can be implemented with recursive tasks in OpenMP,
- how easy some parallel recursive algorithms are to write with OpenMP tasks.
- see complex parallelism structures in OpenMP

As usual all time measurements are to be performed on the cluster.

Activate OpenMP in GCC by passing `-fopenmp` to the compiler and linker. (Note that if you omit this parameter, the code will probably still compile. But its execution will be sequential.)

You can control the number of threads in OpenMP in two ways: the environment variable `OMP_NUM_THREADS` or by calling the `omp_set_num_threads` function.

When recursively decomposing the workload, the granularity of the decomposition is set by the size of the smallest task that will not be further decomposed. Traditionally, a threshold is used under which all tasks are processed sequentially.

Grade Threshold: $A \geq 70$; $B \geq 54$; $C \geq 45$; $D \geq 27$

1 Reduce (19 pts)

Question: Implement computing the sum of an array using OpenMP's tasking construct. Write the code in directory `reduce/` in file `reduce.cpp`. Test the code is running correctly with `make test`.
Question: Benchmark the code on `mamba` using `make bench`. And plot results using `make plot`. What speedup do you achieve with 16 threads? (grade will depend on achieved speedup.)

Hint: Think Divide and Conquer: The sum of an array is the sum of the left part plus the sum of the right part.

2 Merge Sort (27 pts)

Question: Implement a parallel function using OpenMP's tasking construct to perform merge sort on an array of integer. Write the code in directory `mergesort/` in file `mergesort.cpp`. Test the code is running correctly with `make test`.
Question: Benchmark the code on `mamba` using `make bench`. And plot results using `make plot`. What speedup do you achieve with 16 threads? (grade will depend on achieved speedup.)

Hint: A basic implementation will call `merge` in parallel so that multiple merges run at the same time. But to achieve the highest speedup, one will need to make `merge` parallel. This is not trivial. A recursive task decomposition of `merge` is the easier to write.

3 Longest Common Subsequence (27 pts)

Question: Implement a parallel version of the Longest Common Subsequence algorithm. Use the construct (looping or tasking), granularity, parameters, etc. that you deem appropriate. Write the code in directory `lcs/` in file `lcs.cpp`. Test the code is running correctly with `make test`.
Question: Benchmark the code on `mamba` using `make bench`. And plot results using `make plot`. What speedup do you achieve with 16 threads? (grade will depend on achieved speedup.)

Hint: Extracting the dependencies of the LCS algorithm will help you identify where there is parallelism that can be leveraged. Think about how granularity can be implemented on such a problem.

Note: When running the code manually, if the two strings are of size less than 10, the LCS will always be of length 0.

4 Bubble Sort (27 pts)

Question: Implement a parallel version of the BubbleSort algorithm. Use the construct (looping or tasking), granularity, parameters, etc. that you deem appropriate. Write the code in directory `bubblesort/` in file `bubblesort.cpp`. Test the code is running correctly with `make test`.
Question: Benchmark the code on `mamba` using `make bench`. And plot results using `make plot`. What speedup do you achieve with 16 threads? (grade will depend on achieved speedup.)

Hint: Extracting the dependencies of the BubbleSort algorithm will help you identify where there is parallelism that can be leveraged.