MSc in High Performance Computing
Coursework for Threaded Programming Part 3

The object of this assessment is to implement an alternative loop scheduling algorithm in OpenMP.

You are provided with a piece of code which contains the same two loops which you experimented with in Part 2. Instead of using the work sharing loop directive, the loop is scheduled "by hand" using a parallel region. The implementation you are provided with corresponds to the STATIC schedule kind. The code measures the execution time for 1000 repetitions of each loop, and includes a verification test for each loop.

The code can be found on the course pages on Learn. You may choose to work with *either* the C (`loops2.c`) *or* Fortran 90 (`loops2.f90`)version.

You should always compile the code with the `-O3` option to ensure a high level of sequential optimisation, but you must *not* alter the routines `loop1chunk` and `loop2chunk` which contain the body of the parallel loops.

We will use the term *chunk* in the same sense as in the OpenMP standard, i.e. a contiguous, non-empty subset of the iterations of a loop.

Affinity scheduling

Affinity scheduling can be described as follows:

- Each thread is initially assigned a (contiguous) *local set* of iterations.
- For a loop with n iterations, and p threads, each thread's local set is initialised with (approximately) n/p iterations.
- Every thread executes chunks of iterations whose size is a fraction $1/p$ of the *remaining* iterations in its local set, until there are no more iterations left in its local set. (Note that once a chunk is assigned to a thread, that thread must complete all the iterations in the chunk.)
- If a thread has finished the iterations in its local set, it determines the thread which has most remaining iterations (the "most loaded" thread) and executes a chunk of iterations whose size is a fraction $1/p$ of the remaining iterations in the "most loaded" thread's local set.
- Threads which have finished the iterations in their own local set repeat the previous step, until there are no more iterations remaining in any thread's local set.

Modify the code to implement this algorithm. You should take great care with the implementation of this algorithm to ensure that threads are correctly synchronised and there are no race conditions. You are encouraged to experiment with modifications/optimisations of the algorithm, but you should only submit these *in addition to* (and not instead of) a correct implementation of the algorithm as given.

Once you have implemented the algorithm, run your code on 1, 2, 4, 6, 8, 12, 16, 24 and 32 threads and compare the results to the best built-in OpenMP schedule which you determined in Part 2 of the coursework.

Submission

You are required to submit the following:

1. A written report.
   (Guideline length: 6-8 pages including figures and tables.)
2. Source code - include all versions for which you include results in your report.

The deadline for both report and source code is 16:00 on Friday 3rd December 2020. Your report should contain the following sections:

1. a *short* introduction
2. a pseudocode description of your implementation of the affinity scheduling algorithm, with particular attention to the shared data structures used and how the threads are synchronised.
3. a discussion of the results of running the affinity scheduling algorithm, including appropriate graphs and/or figures and a description of the experimental environment.
4. some *brief* conclusions

Your report should *not* contain any background material.

The code will be marked on design and readability as well as correctness and performance.

The maximum available mark for this assessment is **60**. Marks will be allocated as follows:

- Report content and presentation out of **30**.
- Source code out of **30**.