Tutorial Session: Parallel Monte Carlo using OpenMP

Monte Carlo integration is a method to estimate the value of an integral as the mean over a set of random variables. For instance,

$$\int_{\Omega} f(x)\,dx \;\approx\; \frac{|\Omega|}{N}\sum_{i=1}^{N} f(X_i),$$

where X_i are samples from a uniform distribution on the domain Ω. The algorithm can be applied for calculation of volume of arbitrary shapes, in which case the integrand is the indicator function. For a unit circle centered at $(0,0)$, it has the form

$$f(x,y) = \begin{cases} 1, & x^2+y^2 < 1, \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, the number π can be estimated as

$$\frac{4}{N}\sum_{i=1}^{N} f(X_i, Y_i) \approx \pi$$

where X_i and Y_i are samples from uniform distribution on the square $[0,1]\times[0,1]$.

a) Given a serial implementation of the algorithm provided in the skeleton code, write a parallel version using OpenMP by splitting the sampling work among multiple threads. Make sure you do not introduce race conditions and the random generators are initialized differently on each thread. For storing the thread-local data, you may need to use arrays indexed by the thread id or rely on data-sharing attributes of OpenMP. Provide three versions of the implementation, one for each of the following cases

   1. use any OpenMP directives, arrays are not allowed;
   2. the only available directive is `#pragma omp parallel for reduction`, arrays are allowed but without additional padding, this may cause false sharing;
   3. the only available directive is `#pragma omp parallel for reduction`, arrays allowed and must include padding to avoid false sharing.

b) Run the program both on your computer and on Euler. The makefile provides various tools

   - `make` builds the executable,
   - `make run` runs the executable for all available methods (`m=0,1,2,3`) varying the number of threads between 1 and `OMP_NUM_THREADS` (if set, otherwise 4) and writes the timings to new directory `out`,
   - `make plot` plots the timings collected in directory `out`.

   Compare the plots for the methods you implemented, see if the results can be explained by false sharing.

c) Answer the following questions:

- Is the amount of computational work equal among all threads (for large number of samples)?
- Do you observe perfect scaling of your code? Explain why.
- Do you get exactly the same numerical results if you run the same program under the same conditions twice? Are there reasons for slight changes in the results? Consider cases of (a) serial program, (b) OpenMP with one thread, (c) OpenMP with multiple threads.