Parallel Programming WS15 HOMEWORK #8

Submit by email to: `fabregat@aices.rwth-aachen.de`
Due by: Friday, January 15th, 5pm.

OpenMP – Part 1

1. In this task you will estimate the value of π using the Monte Carlo method. Given a probability P, the Monte Carlo method relies on the generation of random samples (events) to compute a numerical approximation of P.

   Consider a circle inscribed in a square (Fig. 1); the method simply consists in generating random points (x, y) within the square range. Given the probability that the points lie within the circle, and the actual number of generated random points that do so, we can estimate π.

   First, what is the probability of random points ending up within the area of the circle? The answer is to find the relationship between the geometry of the square and the circle. The area of the square is $4R^2$ and the area of the circle is πR^2. Now, the ratio of the area of the circle over the area of the square is $\frac{\pi}{4}$. That is, the probability that a random point within the square lies also within the circle is $P = \frac{\pi}{4}$, and thus $\pi = 4P$.

   Write a program that generates np random points using the `drand48` function, computes the ratio q that approximates the probability P, and uses it to estimate the value of π. Base your code on a for loop with $np = 50,000$ iterations. Do not expect more than 1 or 2 decimals of accuracy.

   Parallelize your code using the `parallel` and `for` OpenMP constructs. Pay attention to shared and private variables. You may need to use some of the clauses presented in class.

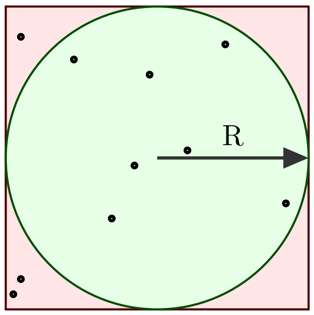

Figure 1: Circle of radius R inscribed in a square of side R^2. Some random points are shown.

2. In this task you will work again on the heat problem introduced in `Homework 6`. The goal of this task is to take the code given in `HW8.c` and produce multiple parallel versions using OpenMP. After testing the code for its correctness, you will collect timings to study the attained speedups, using 1, 2, 4, and 8 threads. You have to code the following 4 versions of the function `compute_*`.

   (a) Parallelize the outermost loop in the matrix update (lines 63–68).

   (b) Parallelize the innermost loop in the matrix update (lines 63–68).

      - Which of these two versions do you expect to be faster? Why?
      - Run the code and look at the timings. Was your prediction correct/accurate?

   (c) Take the fastest of the two versions above and parallelize also the computation of the convergence. Use two separate parallel regions.

   (d) Rewrite version "(c)" using one single parallel region.

   Create the 4 different functions (`compute_outer`, `compute_inner`, `compute_conv` and `compute_single_region`, respectively), and add the necessary code to the `main` function to evaluate the timings and speedups. Run experiments for $n \in [100, 500, 1000]$ using 1, 2, 4, and 8 threads.