Chapel Tutorial Exercise: A Monte Carlo Approximation of π

The Monte Carlo method of approximating π relies on computing the ratio of the area of a circle to the area of a square where the diameter of the circle is equal to the length of each side of the square:

$$\frac{\pi \cdot r^2}{(2 \cdot r)^2} = \frac{\pi}{4}$$

By computing random points in a square and determining how many of these points are also in the circle, a Monte Carlo simulation can be used to approximate the value of π.

The following serial Chapel code uses this technique to compute an approximation of π using a unit circle:

```
use Random;

config const n = 100000,        // number of random points to generate
             seed = 314159265;  // seed for random number generator

writeln("Number of points    = ", n);
writeln("Random number seed  = ", seed);

var rs = new RandomStream(seed, parSafe=false);

var count = 0;
for i in 1..n do
  count += (rs.getNext()**2 + rs.getNext()**2) <= 1.0;

delete rs;

writeln("Approximation of pi = ", format("#.#######", count * 4.0 / n));
```

To get experience with Chapel, try writing the following variations of estimating pi, either from scratch or using the provided starting codes:

1. **A Serial Variant.** Using the concepts presented in the *Language Basics* lecture, modify the serial Chapel program above (provided online with additional comments) to determine the number of random points needed to compute π within a user-specified tolerance, `epsilon`, of a hard-coded value of π specified to 20 decimal places.

2. **A Data-Parallel Version.** Using the concepts presented in the *Data Parallelism* lecture, parallelize the original serial Chapel program using forall loops or promoted functions/operators. Measure the speedup. (Tip: See the other side of this document for notes on the RandomStream class that will be useful for this exercise).

3. **A Task-Parallel Version.** Using the concepts presented in the *Task Parallelism* lecture, parallelize the original serial Chapel program using multiple explicit tasks. Measure the speedup. Compare this code to your data-parallel implementation in terms of performance, effort to write, readability, and maintainability.

4. **A Multi-Locale Task-Parallel Version.** Using the concepts presented in the *Locales* lecture, extend the task parallel version from the previous step so that it runs using multiple locales. Measure the speedup across locales, varying the number of tasks per locale.

5. **A Multi-Locale Data-Parallel Version.** Using the concepts presented in the *Domain Maps* lecture, extend the data parallel version from step 2 to run using multiple locales. Measure the speedup across locales. Compare the differences between this code and your single-locale data-parallel code with the differences between your single-locale and multi-locale task-parallel codes.

See the reverse side for additional notes on the RandomStream class

Here are some notes on the `RandomStream` class that may be useful for this exercise.

- The seed value must be an odd 64-bit integer in the range $(1, 2^{46})$.
- The `getNext()` method returns the next value in the stream as a `real(64)`.
- A method `fillRandom(X: [])` can be used to fill the argument array of `real(64)` elements with random values.
- An (undocumented) iterator method `iterate(D: domain)` will generate random values for all indices in a domain D (either serially or in parallel).

This should be all you need to complete these exercises. For additional notes on the `RandomStream` class, refer to the Chapel language specification in the *Optional Modules* section of the *Standard Modules* chapter.