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 try
             seed = 589494289; // 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
  if (rs.getNext()**2 + rs.getNext()**2) <= 1.0 then
    count += 1;

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

delete rs;
```

To get experience with Chapel, try writing the following variations of estimating π. The program above (with additional comments) is available online as a starting point for each.

1. **A Serial Variant.** Using the concepts presented in the *Language Basics* lecture, modify the serial Chapel program above to determine how soon (i.e. after trying how many random points) the computed approximation arrives within a user-specified tolerance, `epsilon`, of a hard-coded value of π, e.g., 3.1415926535897932384.

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. (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. Compare this code to your data-parallel implementation in terms of 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.

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. 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.

6. **Examine performance.** Measure the speedup achieved by your parallel versions. Check how performance varies with the number of tasks per locale and/or the number of locales. (Tip: see the other side of this document for notes on performance timings. For data-parallel versions use the `dataParTasksPerLocale` configuration variable to vary the amount of parallelism used.)

Notes on Random Number Generation

Here are some notes on Chapel's `RandomStream` class that should be useful for this exercise. For this tutorial, assume it generates a stream of `real(64)` values between 0.0 and 1.0. For further documentation on the `RandomStream` class, refer to the Chapel language specification's description of the `Random` module (*Standard Modules* chapter, *Optional Modules* section).

- The seed value must be an odd integer in the range $(1, 2^{46})$.
- A `RandomStream` object includes a cursor, which is placed initially at the 1^{st} position in the stream.
- The `parSafe` argument of the `RandomStream` constructor determines whether it is safe to use the RandomStream object by concurrent tasks. `parSafe` defaults to `true`. Requesting `parSafe=false` eliminates synchronization overhead, but requires the programmer to guard against concurrent calls to the methods below on the resulting object. Note that, with some cleverness, all variants of these exercises can be computed using `parSafe=false` to avoid these overheads.
- The `getNext()` method returns the stream value at the current cursor position and advances the cursor.
- The `skipToNth(n: int)` method moves the cursor to the n^{th} position in the stream.
- The method `fillRandom(X: [] real(64))` fills the argument array `X` with random values. It operates in parallel when possible. It advances the cursor by the number of elements in `X`.
- The (undocumented) iterator method `iterate(D: domain)` generates a random value for each index in the domain `D`. Like any iterator, it can be invoked in a loop (serial or parallel, possibly multiple times for zippering) or evaluated as an expression. It advances the cursor by the number of indices in `D`.

Notes on Performance Timings

- When doing performance runs of Chapel, always compile with the `--fast` flag. It turns off a number of runtime safety checks and turns on optimizations for the generated C code (see the **chpl** man page for details)
- For timing-related calls, use the Chapel Time module (`use Time;`). The simplest way to do a short-lived timing is to use the `getCurrentTime()` routine which by default returns the number of seconds that have passed since midnight as a `real(64)`. For example:

```
const startTime = getCurrentTime();
timeThis();
const stopTime = getCurrentTime();
writeln("Elapsed time was: ", stopTime - startTime);
```

- For further documentation on timing routines, refer to the Chapel language specification's description of the `Time` module in the *Optional Modules* section of the *Standard Modules* chapter.