ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

A python script

Recall the code for the π estimator using Monte Carlo integration

```python
# get access to the random munber generator functions
import random
# sample size
N = 10**5
# initialize the interior point counter
interior = 0
# integrate by sampling some number of times
for i in range(N):
    # build a random point
    x = random.random()
    y = random.random()
    # check whether it is inside the unit quarter circle
    if (x*x + y*y) <= 1.0: # no need to waste time computing the sqrt
        # update the interior point counter
        interior += 1
# print the result:
print("pi: {0:.8f}".format(4*interior/N))
```

simple, fast but inflexible

Accuracy and cost of the python script

N	π_N	Δ	t(sec)
10^0	0	1	.014
10^1	3.6	1.46×10^{-1}	.014
10^2	3.36	6.95×10^{-2}	.014
10^3	3.076	2.09×10^{-2}	.015
10^4	3.156	4.59×10^{-3}	.027
10^5	3.14496	1.07×10^{-3}	.144
10^6	3.144028	7.75×10^{-4}	1.265
10^7	3.142112	1.65×10^{-4}	12.624
10^8	3.14170136	3.46×10^{-5}	130.430

The equivalent C++ code

```
#include <iostream>
#include <gsl/gsl_rng.h>

int main(int, char*[]) {
   // the sample size
   const int N = 1.0e7;
   // initialize the counters
   int interior = 0;
   // allocate a random number generator
   gsl_rng * generator = gsl_rng_alloc(gsl_rng_ranlxs2);
   // integrate by sampling some number of times
   for (int i=0; i<N; ++i) {
      // create a random point
      double x = gsl_rng_uniform(generator);
      double y = gsl_rng_uniform(generator);
      // check whether it is inside the unit quarter circle
      if ((x*x + y*y) <= 1.0) { // no square roots
         // update the interior point counter
         interior++;
      }
   }
   // print the result
   std::cout << "pi: " << 4. * interior / N << std::endl;
   // all done
   return 0;
}
```

Accuracy and cost of the C++ code

N	π_N	Δ	t(sec)
10^0	0	1	.002
10^1	3.2	1.86×10^{-2}	.002
10^2	3.16	5.86×10^{-3}	.002
10^3	3.2	1.86×10^{-2}	.002
10^4	3.1456	1.28×10^{-3}	.004
10^5	3.13528	2.01×10^{-3}	.026
10^6	3.140756	2.66×10^{-4}	.230
10^7	3.141948	1.13×10^{-4}	2.277
10^8	3.1417769	5.86×10^{-5}	22.749
10^9	3.1415631	9.41×10^{-6}	227.735

- it appears that C++ is about six or seven times faster
- but you have to compile and link every time you make a change
- where does the speed difference come from?

The hunt for objects

perhaps you can be productive without any extra complexity, but that's very rare

```python
# get access to the random munber generator functions
import random
# sample size
N = 10**5
# initialize the interior point counter
interior = 0
# integrate by sampling some number of times
for i in range(N):
    # build a random point
    x = random.random()
    y = random.random()
    # check whether it is inside the unit quarter circle
    if (x*x + y*y) <= 1.0: # no need to waste time computing the sqrt
        # update the interior point counter
        interior += 1
# print the result:
print("pi: {0:.8f}".format(4*interior/N))
```

what flexibility do we need?

Base class for the random number generator

the base class, in `PointCloud.py`

```python
class PointCloud(object):
    """
    The abstract base class for point generators
    """

    # interface
    def point(self, box):
        """
        Generate a random point on the interior of {box}

        parameters:
            {box}: a pair of points on the plane that specify the
                   major diagonal of the rectangular region
        """
        raise NotImplementedError(
            "class {.__name__!r} should implement 'point'".format(type(self)))
```

Encapsulating the built-in python RNG

an example implementation, in `Mersenne.py`

```python
import random
from PointCloud import PointCloud

class Mersenne(PointCloud):
    """
    A point generator implemented using the Mersenne Twister random number
    generator that is available as part of the python standard library
    """

    # interface
    def point(self, box):
        """
        Generate a random point in the interior of {box}
        """
        # unpack the bounding box
        tail, head = box
        intervals = tuple(zip(tail, head))
        # build the point p by caling random the right number of times
        p = [ random.uniform(left, right) for left, right in intervals ]
        # and return it
        return p
```