# ACM/CS 114
# Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Spring 2012

# Amortizing cost: containers

- the flexible solution is more expensive
  - this is sensible and, at some level, inevitable: flexibility implies runtime decisions
  - so we should be prepared to tolerate some slow down
- part of bad performance is caused by executing the *overhead* multiple times
- can we amortize the overhead over $N$, the total number of points?

# Using a container for the sampling points

the base class, in `PointCloud.py`

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

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

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

# Filling a container with random points

in `Mersenne.py`

```
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 points(self, n, box):
        """
        Generate {n} random points in the interior of {box}
        """
        # unfold the bounding box
        intervals = tuple(zip(*box))
        # create the container for the sample
        sample = []
        # loop over the sample size
        while n > 0:
            p = [ random.uniform(left, right) for left,right in intervals ]
            sample.append(p)
            n -= 1
        # return the samples
        return sample
```

# Shape requires little modification

in Shape.py

```
class Shape(object):
    """
    The abstract base class for representations of geometrical regions
    """

    # interface
    def interior(self, points):
        """
        Examine each point in {points} and return a container of only the
        interior points
        """
        raise NotImplementedError(
```

# The updated interior of a circle, in `Disk.py`

```python
from Shape import Shape

class Disk(Shape):
    """
    A representation of a circular disk
    """

    # interface
    def interior(self, points):
        """
        Build a container with members of {points} that fall within my disk
        """
        # precompute the frequently used values
        r2 = self.radius**2
        x0, y0 = self.center
        # initialize the container of interior points
        keep = []
        # iterate over the given points and save the interior ones
        for point in points:
            x, y = point
            dx = x - x0
            dy = y - y0
            # save the point if it is interior
            if dx*dx + dy*dy <= r2:
                keep.append(point)
        # and return them to the caller
        return keep

    # meta methods
    def __init__(self, radius=1.0, center=(0.0, 0.0)):
        self.radius = radius
        self.center = center
        return
```

# The driver for the container based solution

```python

def gauss():
    """
    The driver for the container based implementation
    """
    from Disk import Disk
    from Mersenne import Mersenne

    # inputs
    N = 10**5
    box = [(0,0), (1,1)]
    # the point cloud generator
    generator = Mersenne()
    # the region of integration
    disk = Disk(center=(0,0), radius=1)

    # the integration algorithm
    # build the point sample
    sample = generator.points(N, box)
    # count the interior points
    interior = len(disk.interior(sample))

    # print out the estimate of π
    print("pi: {0:.8f}".format(4*interior/N))
    return
```

# The performance of the container solution

| $N$ | `C++` $t$(sec) | python $t$(sec) | naïve OO $t$(sec) | containers $t$(sec) |
|---|---|---|---|---|
| $10^0$ | .002 | .014 | .014 | .014 |
| $10^1$ | .002 | .014 | .014 | .014 |
| $10^2$ | .002 | .014 | .014 | .015 |
| $10^3$ | .002 | .015 | .020 | .019 |
| $10^4$ | .004 | .027 | .078 | .063 |
| $10^5$ | .026 | .144 | .625 | .504 |
| $10^6$ | .230 | 1.265 | 6.242 | 5.925 |
| $10^7$ | 2.277 | 12.624 | 61.583 | 188.318 |
| $10^8$ | 22.749 | 130.430 | | |
| $10^9$ | 227.735 | | | |

# Generators

- the container solution has a certain elegance
  - the creation of the container and its use are separated
  - we pay less overhead per point
  - with our points in containers, we can use the fast *functional* routines to operate on them
- but, we have now uncovered a new source of cost: managing memory
  - it appears that building large lists is expensive
  - can we avoid building the container all together?
- generators!

# Generating random points

in `Mersenne.py`

```
import random
from PointCloud import PointCloud

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

    # interface
    def points(self, n, box):
        """
        Generate {n} random points in the interior of {box}
        """
        # unfold the bounding box
        intervals = tuple(zip(*box))
        # loop over the sample size
        while n > 0:
            p = [ random.uniform(*interval) for interval in intervals ]
            yield p
            n -= 1

        return
```

# The modified `Disk.py`

```python
from Shape import Shape

class Disk(Shape):
    """
    A representation of a circular disk
    """

    # interface
    def interior(self, points):
        """
        Predicate that filters out points that are not in my interior
        """
        # precompute the frequently used values
        r2 = self.radius**2
        x0, y0 = self.center
        # iterate over the given points and return the interior ones
        for point in points:
            x, y = point
            dx = x - x0
            dy = y - y0
            if dx*dx + dy*dy <= r2:
                yield point
        # all done
        return


    # meta methods
    def __init__(self, radius=1.0, center=(0.0, 0.0)):
        self.radius = radius
        self.center = center
        return
```

# The driver for the generator based solution

```python

def gauss():
    """
    The driver for the generator based implementation
    """
    from Disk import Disk
    from Mersenne import Mersenne

    # inputs
    N = 10**7
    box = [(0,0), (1,1)]
    # the point cloud generator
    generator = Mersenne()
    # the region of integration
    disk = Disk(center=(0,0), radius=1)

    # the integration algorithm
    # build the point sample
    sample = generator.points(N, box)
    # count the interior points
    interior = count(disk.interior(sample))

    # print out the estimate of π
    print("pi: {0:.8f}".format(4*interior/N))
    return


def count(iterable):
    """
    Count the entries of iterable
    """
    counter = 0
    for item in iterable:
        counter += 1
    return counter
```

# The performance of the generator solution

| $N$ | C++ $t$(sec) | python $t$(sec) | naïve OO $t$(sec) | containers $t$(sec) | generators $t$(sec) |
|---|---|---|---|---|---|
| $10^0$ | .002 | .014 | .014 | .014 | .014 |
| $10^1$ | .002 | .014 | .014 | .014 | .014 |
| $10^2$ | .002 | .014 | .014 | .015 | .014 |
| $10^3$ | .002 | .015 | .020 | .019 | .018 |
| $10^4$ | .004 | .027 | .078 | .063 | .053 |
| $10^5$ | .026 | .144 | .625 | .504 | .401 |
| $10^6$ | .230 | 1.265 | 6.242 | 5.925 | 3.780 |
| $10^7$ | 2.277 | 12.624 | 61.583 | 188.318 | 38.242 |
| $10^8$ | 22.749 | 130.430 | | | |
| $10^9$ | 227.735 | | | | |

# Representing integrands

in `Functor.py`

```
class Functor(object):
    """
    The abstract base class for function objects
    """

    # interface
    def eval(self, points):
        """
        Evaluate the function at the supplied points
        """
        raise NotImplementedError(
            "class {.__name__!r} should implement 'eval'".format(type(self)))
```

# Constant functions

in `Constant.py`

```
from Functor import Functor

class Constant(Functor):
    """
    A representation of constant functions
    """

    # interface
    def eval(self, points):
        """
        Compute the value of the function
        """
        # cache the constant
        constant = self.constant
        # return the constant regardless of the evaluation point
        for point in points: yield constant
        # all done
        return

    # meta methods
    def __init__(self, constant):
        self.constant = constant
        return
```

# A non-trivial integrand

```
from Functor import Functor

class Gaussian(Functor):
    """
    An implementation of the normal distribution with mean μ and variance σ²
    """

    # interface
    def eval(self, points):
        """
        Compute the value of the gaussian
        """
        # access the math symbols
        from math import exp, sqrt, pi
        # cache the shape information
        mean = self.mean
        spread = self.spread
        # precompute the normalization factor and the exponent scaling
        normalization = 1 / sqrt(2*pi) / spread
        scaling = 2 * spread**2
        # loop over points and yield the computed value
        for p in points:
            # compute the norm |p - mean|^2
            # this works as long as {p} and {mean} have the same length
            r2 = sum((p_i - mean_i)**2 for p_i, mean_i in zip(p, mean))
            # yield the value at the current p
            yield normalization * exp(- r2/scaling)
        # all done
        return

    # meta methods
    def __init__(self, mean, spread):
        self.mean = mean
        self.spread = spread
        return
```

# The Monte Carlo integrator

```python
import operator, functools

def gauss():
    """
    The driver for the generator based implementation
    """
    from Disk import Disk
    from Gaussian import Gaussian
    from Mersenne import Mersenne

    # inputs
    N = 10**7
    box = [(-1,-1), (1,1)]
    B = functools.reduce(operator.mul, ((right-left) for left,right in zip(*box)))
    # the point cloud generator
    generator = Mersenne()
    # the region of integration
    disk = Disk(center=(0,0), radius=1)
    # the integrand
    gaussian = Gaussian(mean=(0,0), spread=1/3)

    # the integration algorithm
    # build the point sample
    sample = generator.points(N, box)
    # select the interior points
    interior = disk.interior(sample)
    # compute the integral
    integral = B/N * sum(gaussian.eval(interior))

    # print out the estimate of the integral
    print("integral: {0:.8f}".format(integral))
    return
```