Assignment 5: Monte Carlo with Python, Cython, and C++

PHYS 5v48.001 – HPC S25, Aaron Smith

ABSTRACT

The goal of this assignment is to explore the performance of Monte Carlo methods in Python, Cython, and C++. By randomly sampling points within a unit square and determining the fraction that lie within an inscribed unit circle ($x^2 + y^2 < 1$), we can approximate the area of the circle. This ratio, when multiplied by 4, provides an estimate π. The principle behind this method is that as the number of samples increases, the estimate of π converges to the true value, with the error scaling as $1/\sqrt{n}$. This exercise involves the following implementions: (1) pure Python using a for-loop, (2) Python with `NumPy`, (3) Python with `numba`, (4) Cython, and (5) C++. You will benchmark the performance of each implementation, generate a table comparing execution times and per-sample throughput, and run a larger job on Ganymede2 in C++ for $\sim 10^8$ or more samples. You will compile your results in a short PDF report discussing performance, memory usage, and solution accuracy relative to π.

Learning Objectives: Monte Carlo methods (demonstration of an embarrassingly parallel problem), performance comparison between implementations (Python, Cython, and C++), HPC workflows (building, running, and timing), plotting of results, and analysis of performance scaling and memory usage.

1. PURE PYTHON IMPLEMENTATION

1. **Serial For-Loop**: Create a file `pi_python.py`:

```python
from numpy.random import rand
import sys

def calc_pi_loop(n):
    h = 0  # Number of hits inside the circle
    for _ in range(n):
        x, y = rand(), rand()  # Random points in [0, 1)
        if x*x + y*y < 1.:
            h += 1  # Successful hit
    return 4. * float(h) / float(n)  # Estimate pi

if __name__ == "__main__":
    n = int(sys.argv[1])  # Command-line argument
    pi_est = calc_pi_loop(n)
    print(f"n={n}, pi={pi_est}")
```

2. **Timing**:

```
time python pi_python.py 10000
```

   Start with a small n to check correctness, then increase n to measure the performance.

3. **Table of Results**: Record the total time and the time per sample in a table, which will be used for comparison with other implementations. For best results, run each method with as large an n as possible (can be different for each method).

2. NUMPY IMPLEMENTATION

1. **Vectorized Approach**:

```
from numpy import sum
from numpy.random import rand
import sys

def calc_pi_numpy(n):
    h = sum(rand(n)**2 + rand(n)**2 < 1.)
    return 4. * float(h) / float(n)  # Estimate pi

if __name__ == "__main__":
    n = int(sys.argv[1])
    pi_est = calc_pi_numpy(n)
    print(f"n={n}, pi={pi_est}")
```

2. **Compare Performance**: Time the execution for a range of n values. It should be faster than the Python loop, but the memory usage will be higher. Can you tell at large enough n?

3. NUMBA-ACCELERATED PYTHON

1. **Numba-based Monte Carlo**:

```
import random, sys
from numba import jit, njit, prange

@njit
def calc_pi_numba(n):
    h = 0
    for _ in range(n):
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        if x*x + y*y < 1.:
            h += 1
    return 4. * h / n

@jit(nopython=True, nogil=True, parallel=True)
def calc_pi_parallel(n):
    h = 0
    for _ in prange(n):
        x = random.uniform(0, 1)
        y = random.uniform(0, 1)
        if x**2 + y**2 < 1:
            h += 1
    return 4. * h / n

if __name__ == "__main__":
    n = int(sys.argv[1])
    pi_est = calc_pi_numba(n)
    # pi_est = calc_pi_parallel(n)
    print(f"n={n}, pi={pi_est}")
```

2. **Compare Performance**: Note that `@njit` compiles the function at runtime so there is some overhead with this method. Time the execution for a range of n values. At what point does the overhead become

negligible? Compare the parallel version with the serial version for large enough n, reporting the speedup in additin to the time and per-sample cost.

4. CYTHON IMPLEMENTATION

1. **Setup Directory**: Include the Cython code (`calc_pi.pyx`), a build script (`setup.py`), and a driver script (`test_cython.py`). In case you need to install Cython, use `conda install cython` (or pip if you prefer).

2. **Cython Code** (`calc_pi.pyx`):

```
# cython: language_level=3
cimport cython
from libc.stdlib cimport rand, RAND_MAX

@cython.boundscheck(False)  # Disable bounds checking for performance
@cython.wraparound(False)  # Disable negative indexing for performance
def calc_pi_cython(int n):
    cdef:
        int i, h = 0
        double x, y

    for i in range(n):
        x = rand() / RAND_MAX  # Generate a random x coordinate
        y = rand() / RAND_MAX  # Generate a random y coordinate

        # Check if the point (x, y) is inside the unit circle
        if x*x + y*y < 1.:
            h += 1

    # Estimate Pi using the ratio of points inside the circle to the total points
    return 4. * h / n
```

3. **Setup Script** (`setup.py`):

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("calc_pi.pyx")
)
```

4. **Driver Script** (`test.py`):

```
from calc_pi import calc_pi_cython
from time import time

logn = 8  # Number of samples
n = 10**logn  # Number of samples

start = time()  # Start timer
pi = calc_pi_cython(n)  # Calculate pi
end = time()  # End timer

print(f"Estimated Pi: {pi}  [Samples: 10^{logn}, Time: {end - start:.3f} s]")
```

5. **Compile and Test**:

```
python setup.py build_ext --inplace
python test_cython.py 1000000
```

6. **Performance Comparison**: Compare the Cython implementation with the previous methods. Note the speedup and memory usage. Does the Cython version incur any overhead for small n like `numba`? For large n, is it faster or slower than `numba` for this simple problem?

5. C++ IMPLEMENTATION

1. **Code** (`main.cpp`):

```cpp
#include <iostream>
#include <random>
#include <cstdlib> // for atoi

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " N\n";
        return 1;
    }
    const int n = std::atoi(argv[1]);
    std::random_device rd;  // Random seed from hardware
    std::mt19937 gen(rd()); // Mersenne twister engine
    std::uniform_real_distribution<> rand(0., 1.);

    int h = 0;
    for (int i = 0; i < n; ++i) {
        // Get random points
        const double x = rand(gen);
        const double y = rand(gen);

        // Check if point is inside the circle
        if (x*x + y*y <= 1.) h++;
    }

    double pi = 4. * double(h) / double(n);
    std::cout << "n=" << n << ", pi=" << pi << std::endl;
    return 0;
}
```

2. **Compile and Run**:

```
g++ -O3 main.cpp -o mc_pi
time ./mc_pi 10000000
```

3. **Performance Comparison**: Compare the performance with the Python, Cython, and `numba` implementations. How does the C++ version stack up in terms of performance and memory usage? Can you run a job with $n = 10^8$ or more on Ganymede2? For very large n, you might spend several seconds to minutes for a single core run.

6. COLLECTING AND ANALYZING RESULTS

1. **Benchmark Table**: For the Python, NumPy, Numba, Cython (serial and parallel), and C++ implementations, determine the largest n that can be run in a reasonable time ($\sim$ 1 minute). Rank the speeds of the different methods in terms of the *samples per second.* Then for the fastest method only, run a range of $n \in \{10^3, 10^4, \ldots, 10^8, 10^9\}$, record the following statistics:
   Table: n | π Estimate | Absolute Error | Standard Deviation | Runtime [s] | time/n | samples/sec
   Specifically, for each value of n, collect at least 10 estimates of π to compute the standard deviation. Save the results to a file to separate data collection from analysis. If you have time, use Ganymede2 to practice HPC job submission.

2. **Plots**:
   - **Mean Error**: Plot the absolute error of the mean π estimate as a function of n.
   - **Standard Deviation**: Plot the standard deviation of the π estimates as a function of n.
   - **Power-law Fit**: Fit the standard deviation data to a power-law $\sigma \propto n^{-\alpha}$ to determine α.

3. **Analysis**:
   - The value of α (and its uncertainty if you can). How does this compare to the expected $1/\sqrt{n}$ scaling?
   - Based on α, estimate the number of samples (n) neded to achieve 12 digits of accuracy for π.
   - Can you fit that many points in memory with the NumPy approach?
   - Assuming perfect linear scaling, how long would it take to run that many samples on a single core?
   - With perfect weak scaling, how many cores would you need to run that many samples in one year?

7. REPORTING GUIDELINES

1. **Submission Format**:
   - Provide a short PDF (LaTeX or word processor) including:
     (a) All relevant code modifications if needed.
     (b) A benchmark table comparing each method.
     (c) A plot of error and standard deviation vs. n.
     (d) A power-law fit to the standard deviation data.
     (e) Discussion of the analysis questions.

2. **Evaluation Criteria**: Correctness & Completeness, Clarity & Organization, and Discussion & Analysis.

This completes Assignment 5, practicing Monte Carlo methods in Python, Cython, and C++. Happy coding!