Homework 9

For this homework you will run the same algorithm under different parallelization methods and evaluate the speedups (execution time). You will compare simple serial (no parallelization), threading, and multiprocessing.

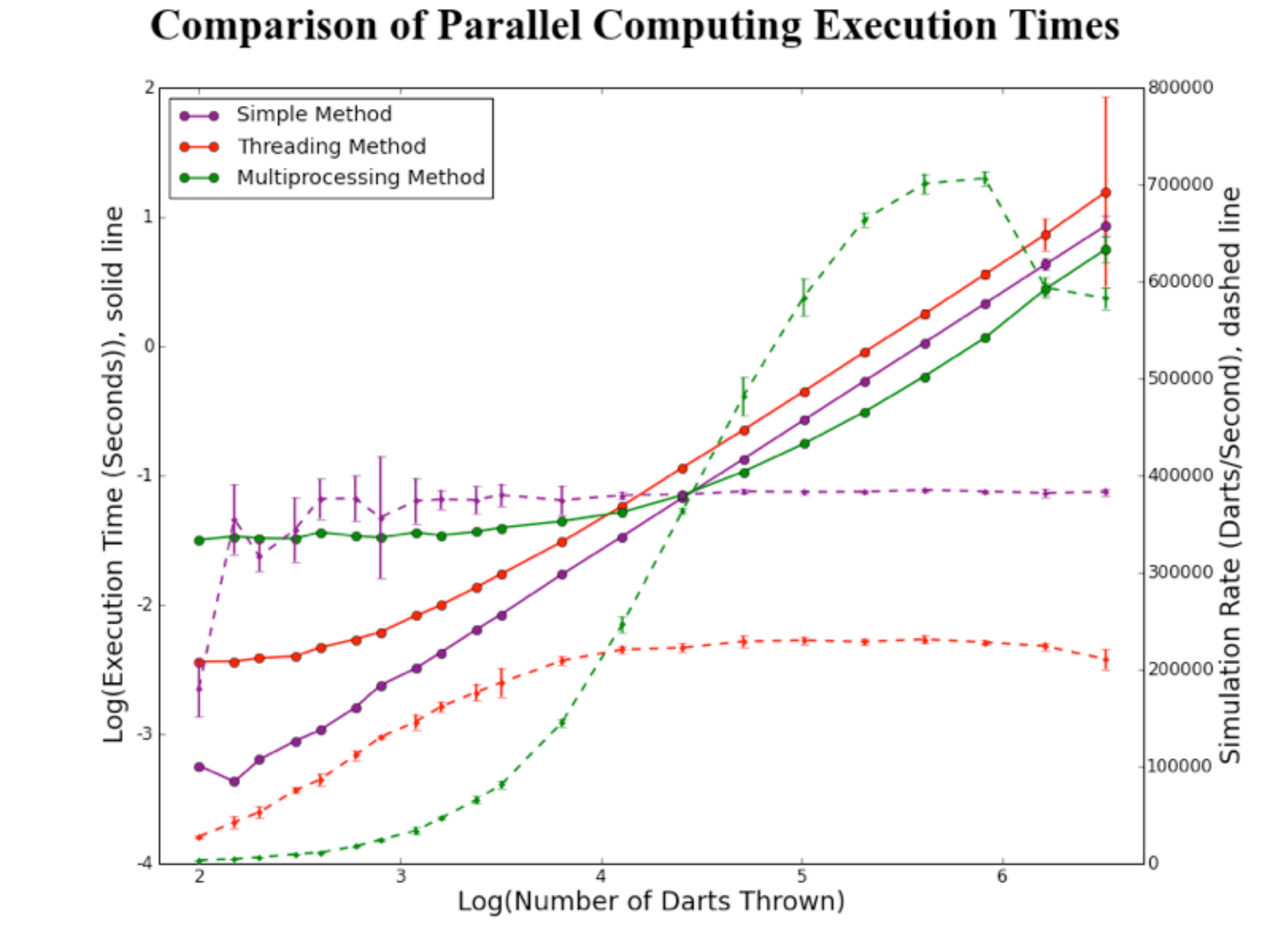
The algorithm uses a Monte Carlo dart-throwing simulation to calculate a numerical approximation to Pi. A simple Python version of this program is on the next slide to show how it can be implemented without parallelization. It is deliberately not copy-pastable so that you have to read it and (hopefully) understand it.

```python
#!/usr/bin/env python
"""
AY 250 - Scientific Research Computing with Python
Homework Assignment 9
Author: Christopher Klein
"""
from random import uniform
from math import sqrt
from time import time

# Define the total number of darts we'll throw.
number_of_darts = 200000
# Define a variable to store the number of darts that fall inside the circle.
number_of_darts_in_circle = 0

# We will use time() to record the execution time of the loop that runs the
# dart throwing simulation.
start_time = time()

# This loop simulates the dart throwing. For each dart, find a random position
# in the unit square for it to fall. Test if it falls within the circle by
# calculating the distance from the origin (0.5, 0.5) to the dart. Darts that
# fall within 0.5 of the origin are within the circle.
for n in range(number_of_darts):
    x, y = uniform(0, 1), uniform(0, 1)
    if sqrt((x - 0.5)**2 + (y - 0.5)**2) <= 0.5:
        number_of_darts_in_circle += 1

# Record the time after the conclusion of the loop.
end_time = time()
# The total time required to run the loop is the difference.
execution_time = end_time - start_time

# We can calculate an approximate numerical value for pi using the formula for
# the area of a cirlce (which defines pi). A = pi * r**2. Here, r=0.5 and the
# area of the circle can be approximated by the ratio of the number of darts
# that fall inside the circle over the total number of darts thrown. Thuse we
# have pi = 4 * Area.
pi_approx = 4 * number_of_darts_in_circle / float(number_of_darts)

# Print out some summary info about the run. Execution time should increase
# with increasing number of darts. Darts thrown per second should stay
# relatively constant and is sort of a measure of the speed of the processor
# (although it is highly dependent on the efficiencies of the various involved
# operators and functions.
print "Pi Approximation:", pi_approx
print "Number of Darts:", number_of_darts
print "Execution Time (s):", execution_time
print "Darts Thrown per Second:", number_of_darts/execution_time
```

Homework 9

Write a large program that runs this algorithm under the three different parallelization methods. Run several trials with different numbers of darts thrown (up to execution times ~10 seconds). Keep track of the execution times as a function of number of darts and method of parallelization. Also, keep track of the simulation rate (darts thrown per second). If you want to be awesome, you can run each simulation multiple times for each number of darts and calculate the standard deviation for the execution time and the simulation rate, but this is not required.

Plot execution time and simulation rate as a function of number of darts for all three methods. If you calculated standard deviations, use errorbar plots. See the example plot on the following page and try to emulate it. In your readme file, explain the behavior you measure and illustrate in the plot. The grader should be able to simply run your submitted program and reproduce the plot.


Comparison of Parallel Computing Execution Times
Simple Method
Threading Method
Multiprocessing Method
Log(Execution Time (Seconds)), solid line
Simulation Rate (Darts/Second), dashed line
Log(Number of Darts Thrown)
2
1
0
-1
-2
-3
-4
2
3
4
5
6
800000
700000
600000
500000
400000
300000
200000
100000
0

Advice

Put your plotting imports after running all the simulations. They can introduce overhead that slows down seemingly unrelated code. Do some investigation online to see examples of threading and multiprocessing.

To use threading, create a class that inherits from threading.Thread. Use Thread.start() and Thread.join() to start and end the threaded parallelization.

You can use multiprocessing in a very similar fashion as threading (the former was written to preserve the syntactical functionality of the later), but try experimenting with Pool() and use that if you prefer or find it to be faster.

For threading, use 2*num_processors for number of threads. For multiprocessing, use num_processors for number of processes. You can experiment with other values to see how this affects performance, and if you find a better configuration, use it and note it in the readme.

Include your processor type in the readme (example plot was generated on MacBook with 2.4 GHz Core2Duo, two cores and so multiprocessing achieves best results with 2 processes) during explanation of measured behavior.

In the example plot, simulation rate of the multiprocessing method dips for the last two points because the CPU throttled down to protect against overheating.