

# Homework 9

*due Thursday Oct 6, 2013 @ 9am*

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), multiprocessing, and IPython's parallel module.

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.

```

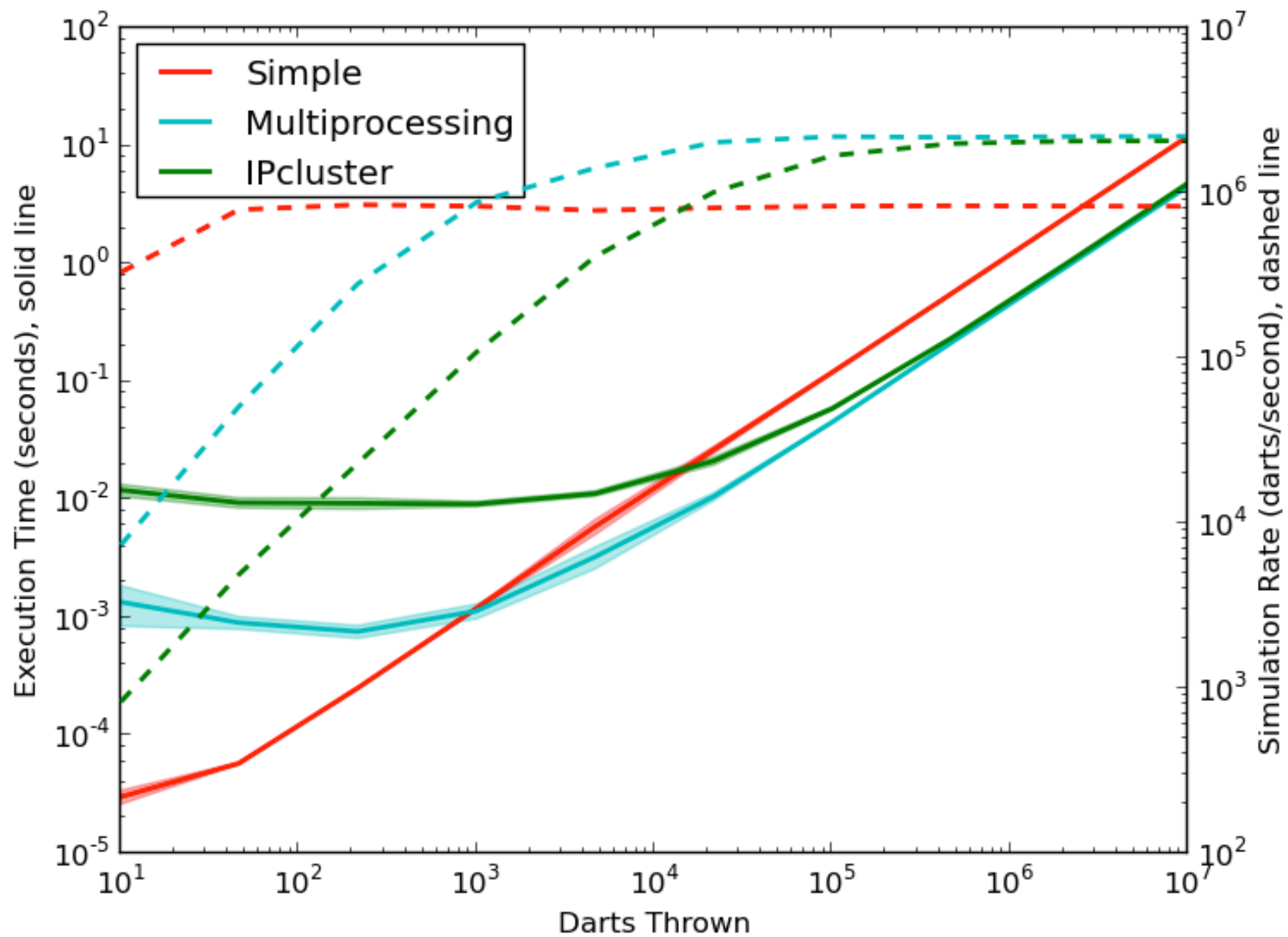
7 from random import uniform
8 from math import sqrt
9 from time import time
10
11 # Define the total number of darts we'll throw.
12 number_of_darts = 200000
13 # Define a variable to store the number of darts that fall inside the circle.
14 number_of_darts_in_circle = 0
15
16 # We will use time() to record the execution time of the loop that runs the
17 # dart throwing simulation.
18 start_time = time()
19
20 # This loop simulates the dart throwing. For each dart, find a random position
21 # in the unit square for it to fall. Test if it falls within the circle by
22 # calculating the distance from the origin (0.5, 0.5) to the dart. Darts that
23 # fall within 0.5 of the origin are within the circle.
24 for n in range(number_of_darts):
25     x, y = uniform(0, 1), uniform(0, 1)
26     if sqrt((x - 0.5)**2 + (y - 0.5)**2) <= 0.5:
27         number_of_darts_in_circle += 1
28
29 # Record the time after the conclusion of the loop.
30 end_time = time()
31 # The total time required to run the loop is the difference.
32 execution_time = end_time - start_time
33
34 # We can calculate an approximate numerical value for pi using the formula for
35 # the area of a circle (which defines pi).  $A = \pi * r^2$ . Here,  $r=0.5$  and the
36 # area of the circle can be approximated by the ratio of the number of darts
37 # that fall inside the circle over the total number of darts thrown. Thus we
38 # have  $\pi = 4 * \text{Area}$ .
39 pi_approx = 4 * number_of_darts_in_circle / float(number_of_darts)
40
41 # Print out some summary info about the run. Execution time should increase
42 # with increasing number of darts. Darts thrown per second should stay
43 # relatively constant and is sort of a measure of the speed of the processor
44 # (although it is highly dependent on the efficiencies of the various involved
45 # operators and functions.
46 print "Pi Approximation:", pi_approx
47 print "Number of Darts:", number_of_darts
48 print "Execution Time (s):", execution_time
49 print "Darts Thrown per Second:", number_of_darts/execution_time

```

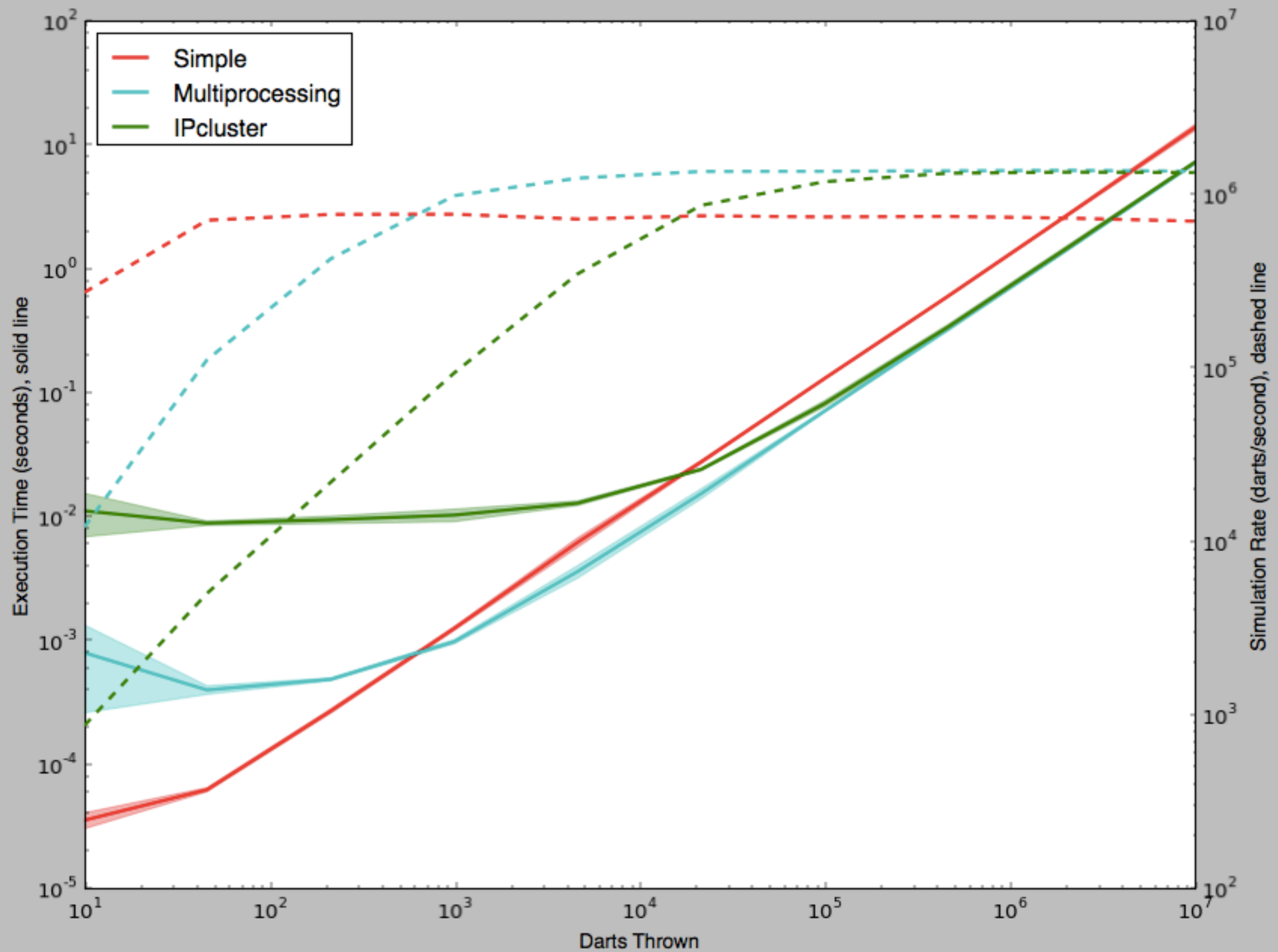
# Homework 9

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  $\sim 100$  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.



Linux box with 1.9 GHz Intel Core, 8 “cores”



MacBook Air with 2 GHz Intel Core i7, 2 “cores”

# 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 multiprocessing and IPython parallel.

The official python multiprocessing documentation is here:

<http://docs.python.org/library/multiprocessing.html>

You should investigate the pool function in particular.

You can check out IPython parallel info at these links:

<http://ipython.org/ipython-doc/dev/parallel/index.html> <https://github.com/minrk/PyData2012>

Include your processor type in the readme (the second example plot was generated on MacBook Air with 2 GHz Intel Core i7, 2 “cores”) during explanation of measured behavior.