Week 14 - Multiprocessing

Based on notes from sebastianraschka.com

How do Programs Work?

1. Computer allocates memory to the program
2. Program issues a series of instructions to the processor
3. Upon completion of one instruction, the next is started
4. Information is returned to the program as needed
5. New instructions are sent to the processor
6. Return to step 2, repeat until program is finished

How do Programs Work?

It is important to note that when a program is running, it typically has a single space in memory in which it stores all relevant information.

- This allows the information to be used by whichever part of the program requires that information to use it.
    - Things that are accessible across the program are called **globally defined values**
    - Variables with reduced **scope** are not available to all segments of a program

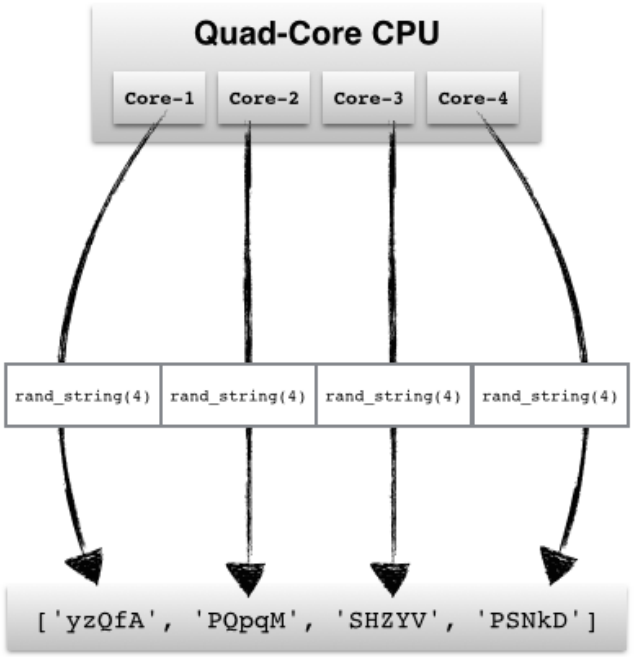
Aside: Scope

Scope is a term used to define the areas in which a given value in memory is accessible.

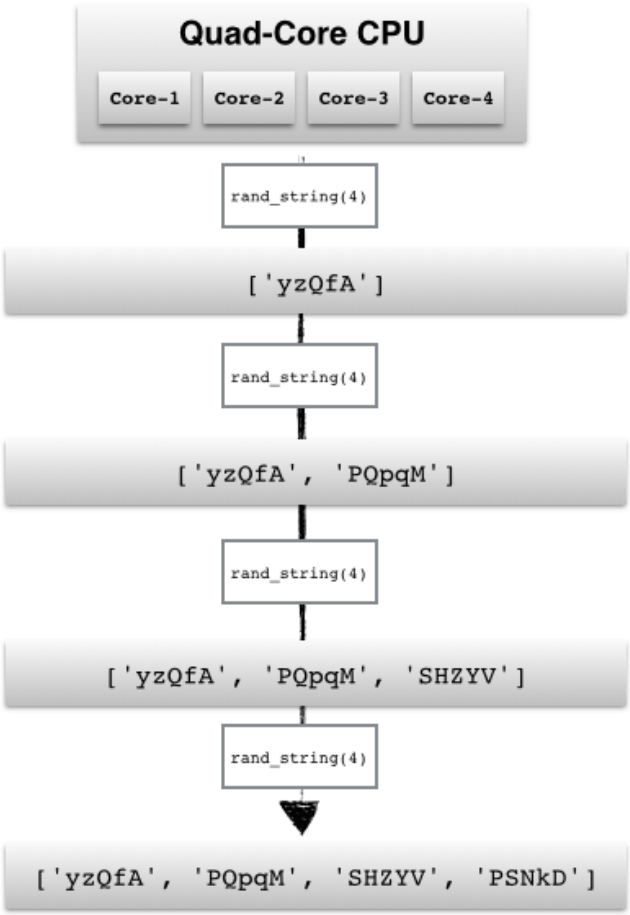
- Variables that are **Global** in scope can typically be accessed by any function or command running as part of the program
- Variables that are **Local** can only be accessed within the scope in which they are defined
    - A variable created inside a function is **local** to that function, and only available until the function is concluded.

Serial vs Parallel

[parallel processing]



[serial processing]



Serial Programs

When we perform tasks, some steps MUST be performed sequentially.

- We need to import data before we clean it
- We need to estimate regression coefficients before we can estimate standard errors on those coefficients
- We need to fit a machine learning model before we can use it to make predictions

Serial Programs

Many tasks are required to be performed sequentially, since subsequent actions or calculations are dependent on the result of prior calculations.

- It is critical that the results of one calculation be within the **scope** of the other calculations
- If one calculation cannot view the results of the other, then the second function typically cannot be completed

Parallel Programs

Some calculations can be performed independent of the results of other steps:

- Batch processing of files
- Non-sequential simulations
- Serving recommended products to many users
- Repeated random draws
- Rendering polygons

Parallel Programs

The key difference between serial and parallel programs is determining the dependency of calculations on the results of previous calculations.

- Serial programs tend to rely on previous results
- Parallel programs depend much less on the results of other calculations

Parallel programs can (obviously) occur simultaneously, allowing us to accelerate execution

Example - Numeric Integration

Used for

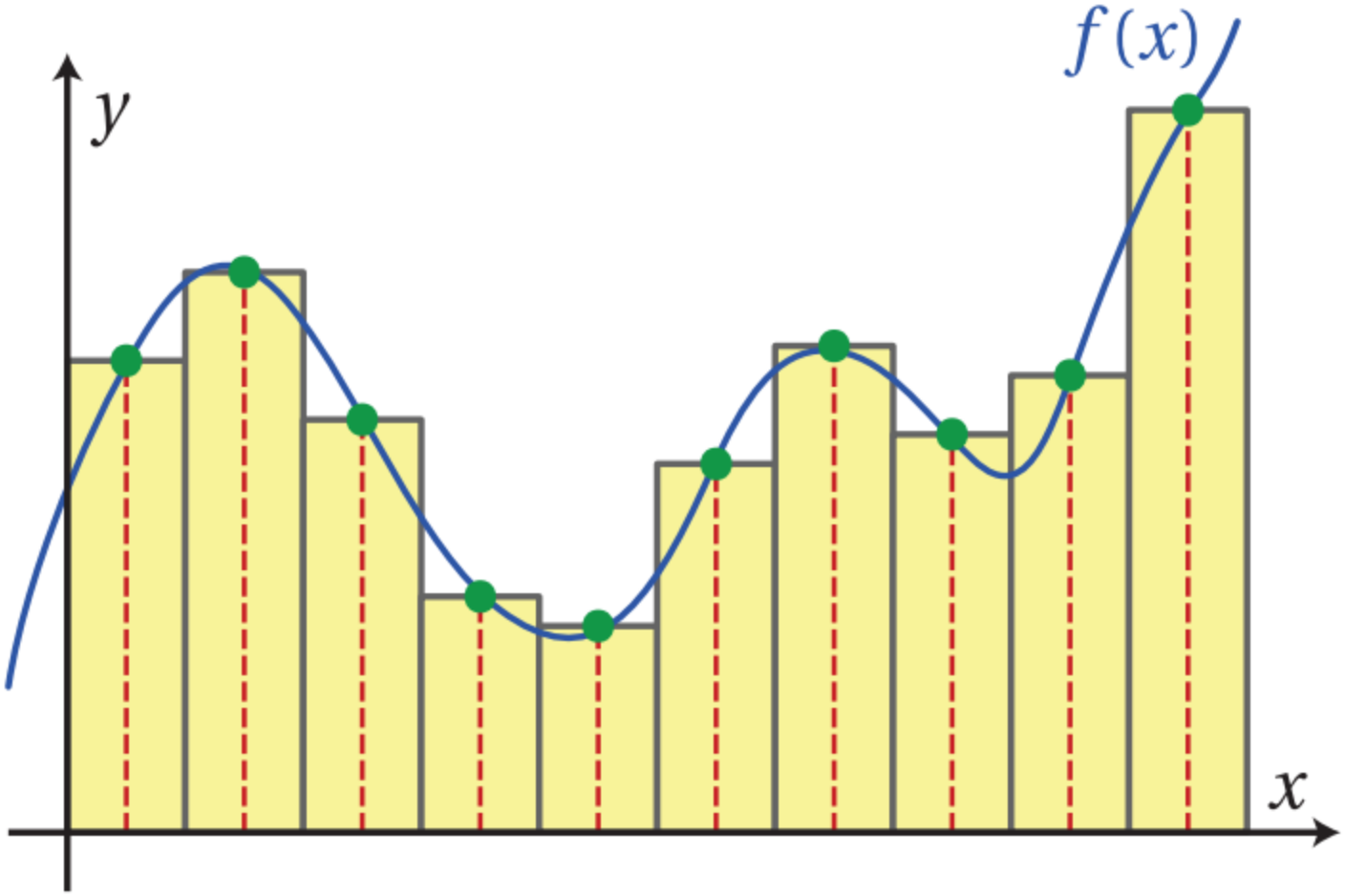
- Estimating Producer/Consumer Surplus
- Calculating probabilities from frequency tables

Example - Numeric Integration

Often, when integrating complicated functions, there is no **algebraic** solution to the integral. This means that we need to estimate the value of the integral **numerically**.

1. Choose points at which to estimate the value of the function
2. Choose bandwidth
3. Multiply function values by bandwidth
4. Add all estimates to calculate approximate integral

Example - Numeric Integration



Example - Numeric Integration Convergence

```
import numpy as np
import multiprocessing as mp # This module is part of the
                             # python standard library

# define any function here!
def f(x):
    # return the value of the function given x
    return 1/(1 + x**2)
```

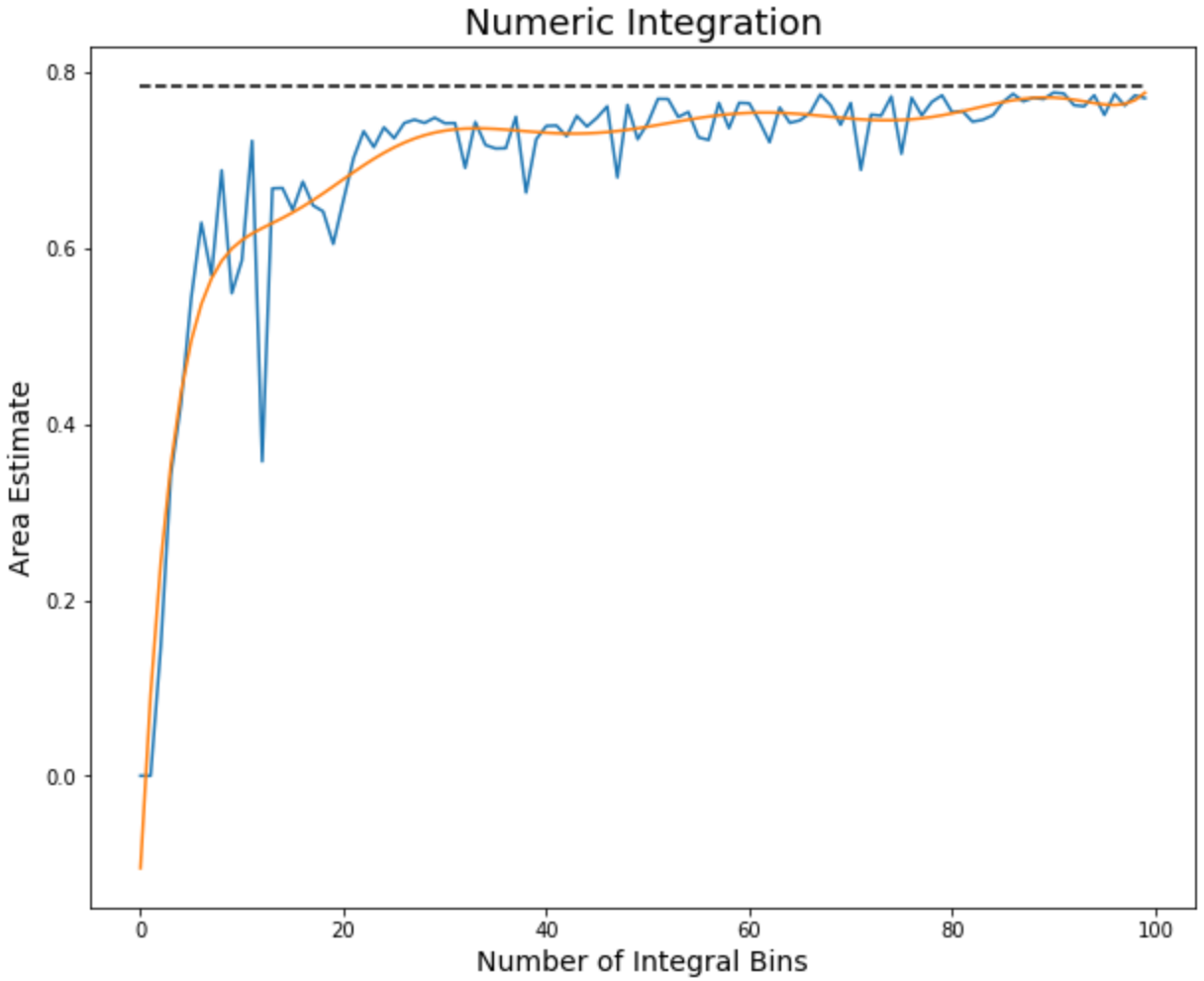
The `multiprocessing` library is designed to create separate instances of the python interpreter, each returning values that are independent of the other instances

Example - Numeric Integration Convergence

```
def integral(nSample, f, xmin, xmax):
  # determine points of estimation
  sample = np.sort(np.random.uniform(xmin, xmax, nSample))
  # Calculate height at each point
  value = f(sample)
  # Calculate areas, sum
  area = np.sum(np.diff(sample) * value[1:])
  # Return integral
  return (nSample, area)
```

This is our function for actually integrating a function `f` from `xmin` to `xmax` across `nSample` random intervals.

Example - Numeric Integration Convergence



Using Multiple Processes

Some computational power must be assigned to send off processes and to retrieve their results upon completion.

- Parallel processing is not good for simple problems
- It's reserved for computationally intensive problems

Using Multiple Processes

It's not worthwhile to make a parallel version of our single integral function above. Instead, let's create a function that can calculate the integral many times in parallel.

First, the serial version:

```
def serial_average(n_bins, n_reps, f, xmin, xmax):
  attempts = [serial_integral(n_bins, f, xmin, xmax) for i in range(n_reps)]
  return sum(attempts)/n_reps
```

Using Multiple Processes

Now, we write it in parallel:

```
def parallel_average(processes, n_bins, n_reps, f, xmin, xmax):
  pool = mp.Pool(processes=processes)
  results = [pool.apply_async(serial_integral,
      (n_bins, f, xmin, xmax)) for i in range(n_reps)]
  results = [p.get() for p in results]
  return sum(results)/n_reps
```

Let's explore what that function does.

Using Multiple Processes

```
def parallel_average(processes, n_bins, n_reps, f, xmin, xmax):
  pool = mp.Pool(processes=processes)
  ...
  return ...
```

The `mp.Pool` class provides the functionality to organize our processes. We specify how many active processes there should be at any time `proceses`.

- Performance is best when the number of processes is at or below the number of cores available

Using Multiple Processes

```
def parallel_average(processes, n_bins, n_reps, f, xmin, xmax):
  ...
  results = [pool.apply_async(serial_integral,
        (n_bins, f, xmin, xmax)) for i in range(n_reps)]
  ...
  return ...
```

The `apply_async` method passes the values that we want our pooled instances to calculate. We provide

- the function to be executed
- the arguments for the function in each iteration with each each of the arguments an element in a tuple

In our example, we are not varying the arguments, so that is a constant

Using Multiple Processes

```
def parallel_average(processes, n_bins, n_reps, f, xmin, xmax):
  ...
  results = [p.get() for p in results]
  ...
  return ...
```

The next step is to use the `get()` method to fetch the return statement values from each of the processes that we declared in the last line. This is where the code actually runs.

The rest of the function works just like the serial version.

Timing it

Next, it is time to write code that will allow us to test our parallel and serial performance.

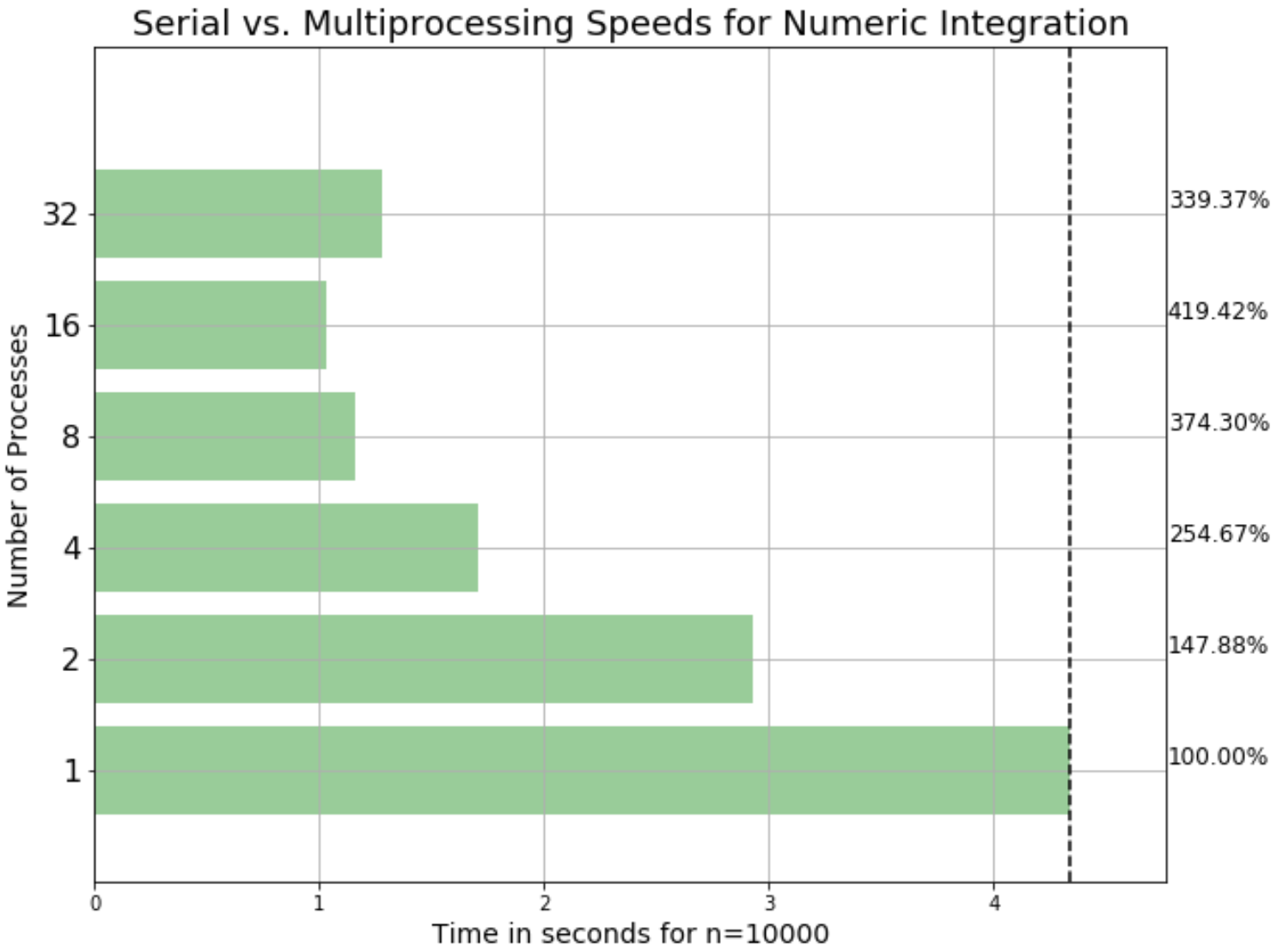
```
import timeit # library for timing execution of code

benchmarks = [] # list to store our execution times

benchmarks.append(timeit.Timer('serial_average(10000, 100, f, 0, 1)',
  'from __main__ import serial_average, serial_integral, f').timeit(number=1))
    # Note that we need to include a second line
    # that imports our functions from __main__.
    # This tells the timer what needs to be IN SCOPE

benchmarks.append(timeit.Timer(
  'parallel_average(2, 10000, 100, f, 0, 1)',
  'from __main__ import parallel_average, serial_integral, f').timeit(
    number=1))
    # Need to include number of processes
    # when timing the parallel implementation
```

Example - Timing it



Example - Timing it

The parallel version of this problem executes over 5x faster than the serial version

- This was done on a 16-core processor
- Creating too many processes (going past 16 to 32) actually started to slow the computations down
- We need to be aware of the hardware that we are utilizing when designing parallel code

```
mp.cpu_count() # Tells us the number of available CPUs
```

Lab Time!