Getting Started with the Python Multiprocessing Package

Armand R. Burks, Ph.D.
Advanced Research Computing
University of Michigan

https://github.com/arburksUMich/python-multiprocessing

Agenda

- The global interpreter lock
- Threads in Python
- Python multiprocessing package overview
- Hands-on exercises

Python Memory Management

- Reference counting
  - Tracks the number of references to every created object
  - When reference count is zero, the object can be released from memory

Reference Counting and Multithreading

- Multiple threads running concurrently can cause problems
- Race condition
  - Multiple threads change an object's reference count concurrently

Mutual Exclusion (Lock)

- Software mechanism that prevents multiple threads from executing **critical code** concurrently
- Critical code
  - Code that accesses a shared/critical resource (such as an object's reference count)

The Global Interpreter Lock (GIL)

- A mutual exclusion/lock on the Python interpreter
- Solution to protecting reference count
- Allows only one thread to execute at any time
  - Causes bottleneck for CPU-bound code

Why Use Threads in Python?

- Allows work to be done in one thread while another thread is waiting
- Examples:
  - I/O heavy process.
    - Can do other processing while waiting for disk
  - Web requests or database connection
    - Can do other processing while waiting for response from server

Get Ready for Coding Exercises

- We will use the Great Lakes cluster for today's exercises
- Make sure you are connected to UMVPN
- SSH to **greatlakes.arc-ts.umich.edu**
  - Mac or Linux
    - Use Terminal: **ssh uniqname@greatlakes.arc-ts.umich.edu**
  - Windows
    - Use PuTTY

Use the Python 3 Module

- Enter the following command on Great Lakes:

```
module load python3.7-anaconda
```

Coding Example: Threading

Concurrency vs Parallelism

- **Concurrency**
  - Two or more tasks can start, run, and complete in *overlapping* time periods.
  - Example: multitasking on a single-core machine.

- **Parallelism**
  - Tasks run at the *same time*, e.g., on a multicore processor.

True Parallelism in Python?

- The multiprocessing package
- Parallelizes code by using multiple processes instead of multiple threads
  - Completely avoids the issue of the GIL
  - Each process has its own interpreter instance
  - Multiple processes can take advantage of multicore CPU

The **multiprocessing** Package

- We will cover some of the basic functionality of the multiprocessing package
- There are MANY more features

multiprocessing.cpu_count()

- Determines the number of available cores on the machine
- Let's try it!

The Process Class

- Class representing a runnable process
- Basic way to run separate tasks in parallel

Coding Example: The **Process** Class

multiprocessing.Pool

- Represents a pool of worker processes
- Run tasks in worker processes
- Very easy way to parallelize code

Coding Example: Pool.map()

Coding Example: Pool.starmap()

Pi Estimation Exercise

- We will use the following to estimate the value of pi

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

- Sum over MANY points
  - Accuracy increases as we add more points
  - Time to compute increases with number of points

Some Rules of Thumb

- Loops can often be easily parallelized
- Use Pool for parallelizing loops
- Data can often be processed in chunks
- Always use join() after using Process.start()
- Avoid sharing data between processes

Disconnect From Great Lakes

- To end your SSH session, type:

```
exit
```

- PuTTY users, you may close PuTTY afterwards