Parallel Programming for Data Scientists

Benjamin S. Skrainka

July 11, 2016

Objectives

Today's objectives:

- Describe basic components of a computer
- Describe basic components of an operating system (OS)
- List components of a process
- State difference between a task & a process
- List issues involved in parallelizing computation

References

A couple references:

- The Design & Implementation of the FreeBSD Operating System
- Python documentation

Basic computer architecture

Draw picture

Operating system basics

An operating system is a program which manages a computer's resources:

- Synchronization to coordinate work
- Mutual exclusion to protect shared resources
- Scheduling of work
  - Usually 'fair'
  - Can change priority with `nice`

Processes

Work runs inside processes:

- Runs program *text*
  - Only one copy is in memory on the entire computer
  - Read-only
  - From OS, programs, and libraries
- Contains:
  - *data*: initialized global & static variables
  - *bss*: uninitialized global & static variables
  - One copy per process

States of a process

Processes can be in one of the following states:

- Ready/executing
- Blocked
- Delayed
- Suspended

Lifecycle of a process

The process life cycle is:

- Fork
- Exec
- Exit
- Reaped by parent

Components of a process

A process has the following main components

- OS control information:
  - PID & PPID
  - File descriptor table
  - Mapping of standard input, output, and error
  - Error status
  - Signal handlers
- Thread of execution
- Stack: local storage for thread
- Heap: general memory for process to allocate dynamically

A process should be relatively insulated from other processes

Process vs. thread

A thread is a lighter-weight concept:

- One or more run inside of a process
- Each thread has its own stack
- All threads uses the same heap/global memory
  - Easy to communicate
  - Easy to cause *race conditions* if threads do not coordinate access to shared memory

Parallelization

Use parallelization to speed up big jobs when:

- *Embarassingly parallel*: can break work up into independent chunks
- Operations can block or fail
- Application decomposes into different types of work or stages
- Have more data than fits in a single computer

Tools for parallelization

OS and computer language provide support for parallel programming:

- *OpenMP* works within one node (multi-core) via shared memory
- *MPI* works between nodes, e.g., over a network

Python provides:

- Processes: use `multiprocessing` module
- Threads: use `threading` module

Python Global Interpreter Lock

Python has *Global Interpreter Lock* (GIL):

- CPython only lets one thread in a process at a time run
- To avoid compromising shared/global data structures
- Makes parallelization difficult
- To get parallelization, must run multiple Python jobs as separate processes

When to use a process

Use a process for longer running jobs:

- Length of the job must offset the cost of launching process
- Circumvent GIL
- Need extra fault tolerance
- Common on clusters using Condor, PBS, etc.
- Robust to errors

When to use a thread

Use a thread for parallelization when:

- Quick creation/destruction vs. processes
- Easy communication via shared memory

Pro-tips

Beware of trade-offs:

- Processes vs. threads: robustness vs. speed
- Cost of launching a process/thread vs. length of work

Other issues:

- Fork, then load data; not vice-versa
- Parallel programming is hard to debug

Conclusion

- List components of an OS
- List components of a process
- Define states of a process
- When to use process vs. threads