3110: Concurrency

Logistics

- Huffman Project Out

- Project 2?

- Homework out.

Concurrency & Parallelism

- Fenyman – have to think in a new way.

- Note: GPU’s and GPGPU’s

Demo

Python

- Demo of multiprocessing package verse threads in python
- What's going on?
- Notice order is not preserved in multiprocessing. Latency! A wild creature, guaranteed to exist.

Call a Thread

- Use a class that inherits from threading.Thread
  - Have your own initializations and functions
  - Does not inherit global variables – very contained.
  - But all threads ‘share everything’ so require locks.

Call a Process

- Initialize with a function and arguments.
  - Gets a copy of the environment so it can produce no side effects. 'Shares nothing!'
  - Explicit data sharing only.

Very Similar Calls

- start(runs)
- run(executes main), only call through start()
- join(waits until finished)
  - Note: what if thread t called t.join() ?

- For Threads can add to these.

Demo

Very Different Performance

- Using processes will actually take advantage of multi-core.
- Using threads can be quick.
- Processes can be arbitrarily better than threads
- Threads can beat Processes by the initialization time.

Light-weight & Heavy-weight

- Notice for simple creation, threads are much faster than processes.
- Threads are ‘light-weight’ not much to manage, not much to create.
- Processes are ‘heavy-weight’, have to create an entirely new process

- Global Interface Lock:
  - Threads must obtain a lock on the GIL in order to operate – Python limits processing
  - Process actually spawns another process – operating system on limits.

Locks?

- Control Data modification

- Imagine a bank account balance(starting at $200) with multiple transfers.

  Transfer1 – deposit $100

  Transfer2 – withdraw $200

- Possible balance outcomes:

  Transfer1 asks for balance

  Transfer1 tells bank to set account to $300

  Transfer2 asks for balance

  Transfer2 tells bank to set account to $100

  Balance = $100

Error Down

- Balance starts at $200
- Possible balance outcomes:

  Transfer1 asks for balance

  Transfer2 asks for balance

  Transfer1 tells bank to set account to $300

  Transfer2 tells bank to set account to $0
- Balance ends at $0

Error Up

- Balance starts at $200
- Possible balance outcomes:

  Transfer1 asks for balance

  Transfer2 asks for balance

  Transfer2 tells bank to set account to $0

  Transfer1 tells bank to set account to $300
- Balance ends at $300

Prevent Race Conditions

- To prevent an unseen data update from happening, we use locks on data. (very complicated for database updates that must deal with unexpected crashes)

- The previous example is imperative with destructive updates – mitigate those in functional programming, allows for greater independence.

Demo

Locks

- Really act as flag indicators.

- A lock is either locked or unlocked. While locked, assume another thread/process really needs it.

Obtaining a Lock

- Look before you leap logic. Check whether or not a Lock is taken. Then either grab it or move on. problem.
- Act now, ask for forgiveness later logic. Just go for a Lock. Use try statements.

Demo

Python Locks

- Locks are not owned.

```
def Evil(name, std_lock):
    std_lock.release()
    std_lock.acquire()
    for i in range(10):
        time.sleep(.1)
        print name
```

Demo

Demo

Way around – Mutex (mutual exclusion lock)

- Mutex is a Queue of functions and inputs.
- mutex.lock(function, input), tries to execute, if not enqueues the call
- mutex.unlock(), unlocks if the queue is empty, otherwise executes the next function(input)
- Normal behavior – not a thread, not a process, just a queue of function(input) calls to execute.
- Can serialize your program.
- (demo)

Shared Data

- Mutex and Threads automatically share data. Processes don't.
- Value and Array allow for shared data

- Manager module. (more flexible, more overhead, slower)

Communication

- Shared Memory over Flags/Locks – good, 1 process isn't left hanging on another

- Can even use Queues to collect results

- But, for more complicated computations – need direct communication - Pipes

Latency Example

- Ping a machine in Greenland, vs on campus.

- Imagine doing this for every third operation.

- Want to avoid communication, accept bigger O() algorithms to avoid it even.

Good Banking Ex.

- Presuming don't care about crashes.

```
balance_lock = Lock()

Transfer1:
    balance_lock.acquire()
    balance += $100
    balance_lock.release()

Transfer2:
    balance_lock.acquire()
    balance -= $200
    balance_lock.release()
```

Function1:

```
std_lock.acquire()
compute1()
store_lock.acquire()
store1()
store_lock.release()
std_lock.release()
```

Function2:

```
compute2()
store_lock.acquire()
store2()
std_lock.acquire()
output2()
std_lock.release()
store_lock.acquire()
```

Break

- Locks are good, but won’t get you the entire way.

- Coming up:
  - Communication
  - Pools of Threads and Processes

Demo

Queues

- from multiprocessing import Queue
- Similar to the structure Queue – but suped up!

- put, qsize, empty, get

+ put_nowait, get_nowait, join

- get(False) is equivalent to get_nowait()
- What's going on?
- Many Process and Thread calls allow you to back out of an operation if desired.
- Say, Process is a cleaner, it tries to get work, but turned down it'll go onto the next job.
- Not implemented as a Condition Variable in Python. Implemented as a function option.

Demo

Pipes

- Simulates basic send recv in mpi(message passing interface) the foundation of parallel computation in C

- Big hang up. Waiting for other process.
  - Python recommends using files to avoid.
  - Use good coding!

```
parent, child = Pipe()
child_p = Process(target=function, args=(child, ))
```

- In function, call send(), and on the other end call recv()

Pools

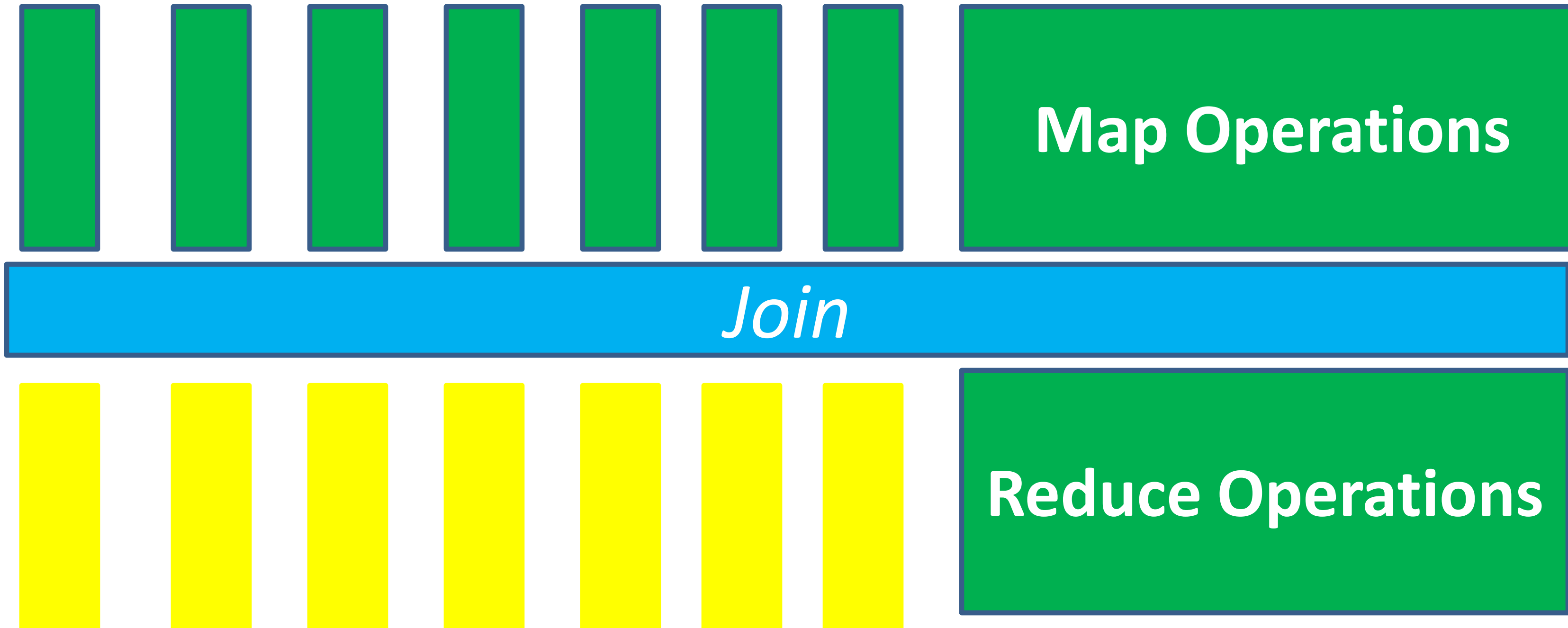
- Processes take awhile to startup. So, start them up and have them wait.

- Demo

```
pool = Pool(processes=10)
result = pool.apply_async(f, [10])
pool.map(f, range(10))
```

Great – That's how. Now what.

- Embarrassingly parallel question:
  - Simple map reduce. No dependencies. Computation model is:



Map – 1st go

Given: map(function, input)

- Create a process for each function(input)

- Enqueue result in a queue

Reduce 1st go

Given reduce(function, input)

- Each process pulls 1st two items off queue, reduces and adds back to queue

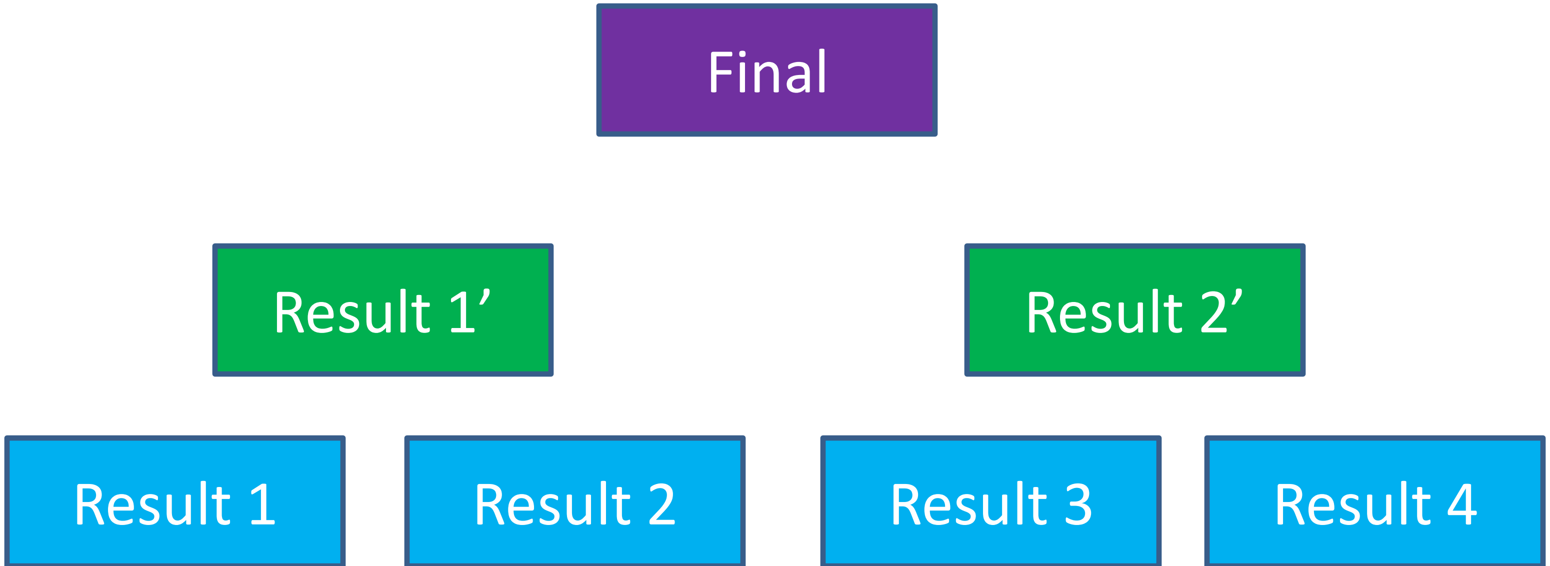
- When queue has only 1 item – you're done.

Problems?

- Bottle necking yourself.
- Too much overheads from 1000 processes.
- Solutions?

Reduce 2nd attempt

- Use Pipes for communication between neighboring processes
- Fill in Balanced Binary Tree with appropriate number of processors.



Problems?

- Load Balancing.
  - Don’t want to have to stop at each layer.
  - Number of Processes that can be created may not be enough.

- Good – O(Log(n)) wall clock time.

Reduce 3rd Attempt

- Do local and global structure. Clump the work.
- Have each process run a map-reduce on 10 inputs. Then go into next level up – 10 processors hand off to 1 – and again.

Good!

- Now you can write embarrassingly parallel programs.
- Like map-reduce. Projects 4 and 5.

Big Programs

- First attempts:
  - Spatial decomposition
    - Becomes too unevenly balanced, use quad trees
  - Multigrid
    - Levels of refinement, each providing more detail
    - Still may have high communication cost, immense storage needed.
  - Object decomposition
    - All objects may need to communicate

Big Programs Suffer from

- High communication costs from high latency
- While computation may be parallel, I/O systems are rarely designed alongside.
- Comprehending the data
- N processors does not mean 1/N time.
- Node failures.
- High energy costs.

- Only so far we can push our current understanding of algorithms.

"We need algorithms that will not work in a serial setting." -Phil Andrews

Huffman Tree Ex.

- Most representations of a file you think of are fixed length code. Ie, IEEE floating point standard. Ascii

- Huffman is variable length – yes, it works.

- See project writeup for description.
- Ex. On board.