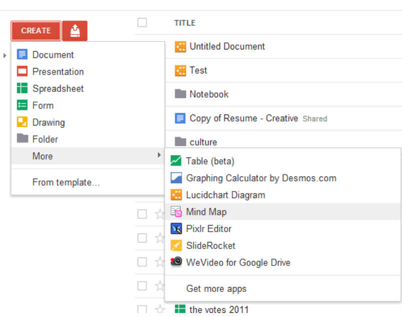
61A LECTURE 27 – PARALLELISM

Steven Tang and Eric Tzeng
August 8, 2013

Announcements

- Practice Final Exam Sessions
  - Worth 2 points extra credit just for taking it
  - Sign-up instructions on Piazza (computer based test)
  - Friday 9am-12pm
  - Friday 1pm-4pm (waiting on room...)
  - Saturday 1pm-4pm
  - Sunday 3pm-7pm
- TA led review sessions, following 2 of the exam sessions:
  - Friday 4pm-5pm
  - Saturday 4pm-5pm
- HW13 out (last true homework!)

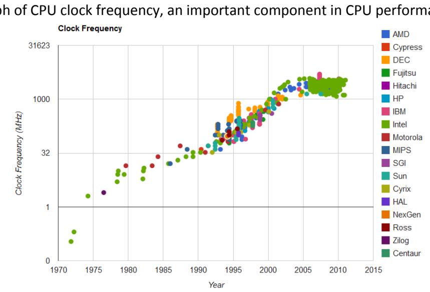
Multiple entities, one shared data



CPU Performance

Performance of individual CPU cores has largely stagnated in recent years

Graph of CPU clock frequency, an important component in CPU performance:



http://cpudb.stanford.edu

Parallelism

Applications must be *parallelized* in order run faster

- Waiting for a faster CPU core is no longer an option

Parallelism is easy in functional programming:

- When a program contains only pure functions, call expressions can be evaluated in any order, lazily, and in parallel
- Referential transparency: a call expression can be replaced by its value (or *vice versa*) without changing the program

But not all problems can be solved efficiently using functional programming

Today: Investigate what happens when you share data across different programs running in parallel

Next time: Easier case of parallelism, using only pure functions

- *MapReduce*, a framework for such computations

Parallelism in Python

Python provides two mechanisms for parallelism:

Threads execute in the same interpreter, sharing all data

- However, the CPython interpreter executes only one thread at a time, switching between them rapidly at (mostly) arbitrary points
  - Want to know more more? Look up **global interpreter lock**
- Operations external to the interpreter, such as file and network I/O, may execute concurrently

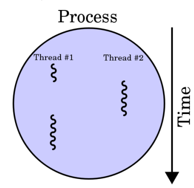
Processes execute in separate interpreters, generally not sharing data

- Shared state can be communicated explicitly between processes
- Since processes run in separate interpreters, they can be executed in parallel as the underlying hardware and software allow. Threads in Python ***switched*** between rapidly, while processes might actually be run in parallel.

The concepts of threads and processes exist in other systems as well

Terminology

- Computer programs are lines of code
- When a program is executed, it's considered a process
- You might have 20 processes running at the "same time", but only one or two processors
- Processor ***switches*** between processes very rapidly, so it looks to us like many programs are running at once
- A process can contain multiple threads



Threads

The **threading** module contains classes that enable threads to be created and synchronized

Here is a "hello world" example with two threads:

```
from threading import Thread, current_thread

def thread_hello():
    other = Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    print('hello from', current_thread().name)

>>> thread_hello()
hello from Thread-1
hello from MainThread
```

Function that the new thread should run

Start the other thread

Arguments to that function

Print output is not synchronized, so can appear in any order

Processes

The **multiprocessing** module contains classes that enable processes to be created and synchronized

Here is a "hello world" example with two processes:

```
from multiprocessing import Process, current_process

def process_hello():
    other = Process(target=process_say_hello, args=())
    other.start()
    process_say_hello()

def process_say_hello():
    print('hello from', current_process().name)

>>> process_hello()
hello from MainProcess
>>> hello from Process-1
```

Function that the new process should run

Start the other process

Arguments to that function

Print output is not synchronized, so can appear in any order

The Problem with Shared State

Shared state that is mutated and accessed concurrently by multiple threads can cause subtle bugs

Here is an example with two threads that concurrently update a counter:

```
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

Wait until other thread completes

What is the value of **counter[0]** at the end?

The Problem with Shared State

```
from threading import Thread

counter = [0]

def increment():
    counter[0] = counter[0] + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

What is the value of **counter[0]** at the end?

Only the most basic operations in CPython are *atomic*, meaning that they have the effect of occurring instantaneously

The counter increment is three basic operations: read the old value, add 1 to it, write the new value

The Problem with Shared State

We can see what happens if a switch occurs at the wrong time by trying to force one in CPython:

```
from threading import Thread
from time import sleep

counter = [0]

def increment():
    count = counter[0]
    sleep(0)
    counter[0] = count + 1

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

May cause the interpreter to switch threads

The Problem with Shared State

```
def increment():
    count = counter[0]
    sleep(0)          May cause the interpreter to switch threads
    counter[0] = count + 1
```

Given a switch at the **sleep** call, here is a possible sequence of operations on each thread:

```
Thread 0                     Thread 1
read counter[0]: 0
                             read counter[0]: 0
calculate 0 + 1: 1
write 1 -> counter[0]
                             calculate 0 + 1: 1
                             write 1 -> counter[0]
```

The counter ends up with a value of 1, even though it was incremented twice!

Practice

x = 1

What are the possible values of x if the following 2 threads are run concurrently?

```
>>> x = x * 2
>>> x = x + 10
```

Race Conditions

A situation where multiple threads concurrently access the same data, and at least one thread mutates it, is called a *race condition*

Race conditions are difficult to debug, since they may only occur very rarely

Access to shared data in the presence of mutation must be *synchronized* in order to prevent access by other threads while a thread is mutating the data

Managing shared state is a key challenge in parallel computing

- Under-synchronization doesn't protect against race conditions and other parallel bugs
- Over-synchronization prevents non-conflicting accesses from occurring in parallel, reducing a program's efficiency
- Incorrect synchronization may result in *deadlock*, where different threads indefinitely wait for each other in a circular dependency

We will see some basic tools for managing shared state

Break

Synchronized Data Structures

Some data structures guarantee synchronization, so that their operations are atomic

```
from queue import Queue      Synchronized FIFO queue
queue = Queue()
def increment():
    count = queue.get()      Waits until an item is available
    sleep(0)
    queue.put(count + 1)
other = Thread(target=increment, args=())
other.start()
queue.put(0)                 Add initial value of 0
increment()
other.join()
print('count is now', queue.get())
```

Manual Synchronization with a Lock

A *lock* ensures that only one thread at a time can hold it

Once it is *acquired*, no other threads may acquire it until it is *released*

```
from threading import Lock
counter = [0]
counter_lock = Lock()
def increment():
    counter_lock.acquire()
    count = counter[0]
    sleep(0)
    counter[0] = count + 1
    counter_lock.release()
other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

The With Statement

A programmer must ensure that a thread releases a lock when it is done with it

This can be very error-prone, particularly if an exception may be raised

The `with` statement takes care of acquiring a lock before its suite and releasing it when execution exits its suite for any reason

```
def increment():
    counter_lock.acquire()
    count = counter[0]
    sleep(0)
    counter[0] = count + 1
    counter_lock.release()

def increment():
    with counter_lock:
        count = counter[0]
        sleep(0)
        counter[0] = count + 1
```

Simple example of (possible) deadlock

```
lock1 = Lock()
lock2 = Lock()
def foo():
    lock1.acquire()
    lock2.acquire()
    print('hello')
    print('world')
    lock1.release()
    lock2.release()
def bar():
    lock2.acquire()
    lock1.acquire()
    print('boom')
    lock2.release()
    lock2.release()
```

Example: Web Crawler

A *web crawler* is a program that systematically browses the Internet

For example, we might write a web crawler that validates links on a website, recursively checking all links hosted by the same site

A parallel crawler may use the following data structures:

- A queue of URLs that need processing
- A set of URLs that have already been seen, to avoid repeating work and getting stuck in a circular sequence of links

These data structures need to be accessed by all threads, so they must be properly synchronized

The synchronized `Queue` class can be used for the URL queue

There is no synchronized set in the Python library, so we must provide our own synchronization using a lock

Synchronization in the Web Crawler

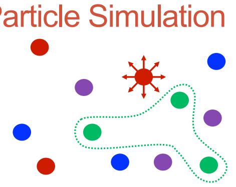
The following illustrates the main synchronization in the web crawler:

```
def put_url(url):
    """Queue the given URL."""
    queue.put(url)

def get_url():
    """Retrieve a URL."""
    return queue.get()

def already_seen(url):
    """Check if a URL has already been seen."""
    with seen_lock:
        if url in seen:
            return True
        seen.add(url)
        return False
```

Example: Particle Simulation



A set of particles all interact with each other (e.g. short range repulsive force)

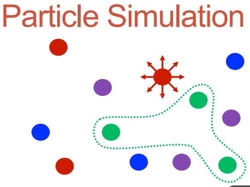
The set of particles is divided among all threads/processes

Forces are computed from particles' positions

- Their positions constitute shared data

The simulation is discretized into timesteps

Example: Particle Simulation



In each timestep, each thread/process must:

Concurrent reads are OK

1. Read the positions of every particle (read shared data)
2. Update acceleration of its own particles (access non-shared data)
3. Update velocities of its own particles (access non-shared data)
4. Update positions of its own particles (write shared data)

Steps 1 and 4 conflict with each other

Writes are to different locations

Solution #1: Barriers

In each timestep, each thread/process must:

1. Read the positions of every particle (read shared data)
2. Update acceleration of its own particles (access non-shared data)
3. Update velocities of its own particles (access non-shared data)
4. Update positions of its own particles (write shared data)

Steps 1 and 4 conflict with each other

We can solve this conflict by dividing the program into *phases*, ensuring that all threads change phases at the same time

A *barrier* is a synchronization mechanism that accomplishes this

```
from threading import Barrier
barrier = Barrier(num_threads)
barrier.wait()
```

Waits until `num_threads` threads reach it

Solution #2: Message Passing

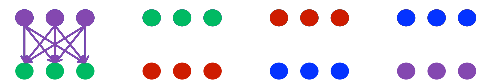
Alternatively, we can explicitly pass state from the thread/process that owns it to those that need to use it

In each timestep, every process makes a copy of its own particles

Then, they do the following `num_processes-1` times:

1. Interact with the copy that is present
2. Send the copy to the left, receive from the right

Thus, reads are on copies, so they don't conflict with writes



Summary

Parallelism is necessary for performance, due to hardware trends

But parallelism is hard in the presence of mutable shared state

- Access to shared data must be synchronized in the presence of mutation

Making parallel programming easier is one of the central challenges that Computer Science faces today

Summary

- Many start-ups are in the business of dealing with "Big Data"
- Use distributed computing and parallel programming to tackle Big Data
- **Big Data**: A buzzword used to describe data sets so large that they reveal facts about the world via statistical analysis.
- 61A gives you a starting point for thinking about computing in parallel
- 162 makes you implement the operating system that handles parallel computation

Parallel Computation Patterns

Not all problems can be solved efficiently using functional programming

The Berkeley View project has identified 13 common computational patterns in engineering and science:

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Sructured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

MapReduce is only one of these patterns

The rest require shared mutable state

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine