EECS 490 – Lecture 25

Parallel Computing

12/5/17

Announcements

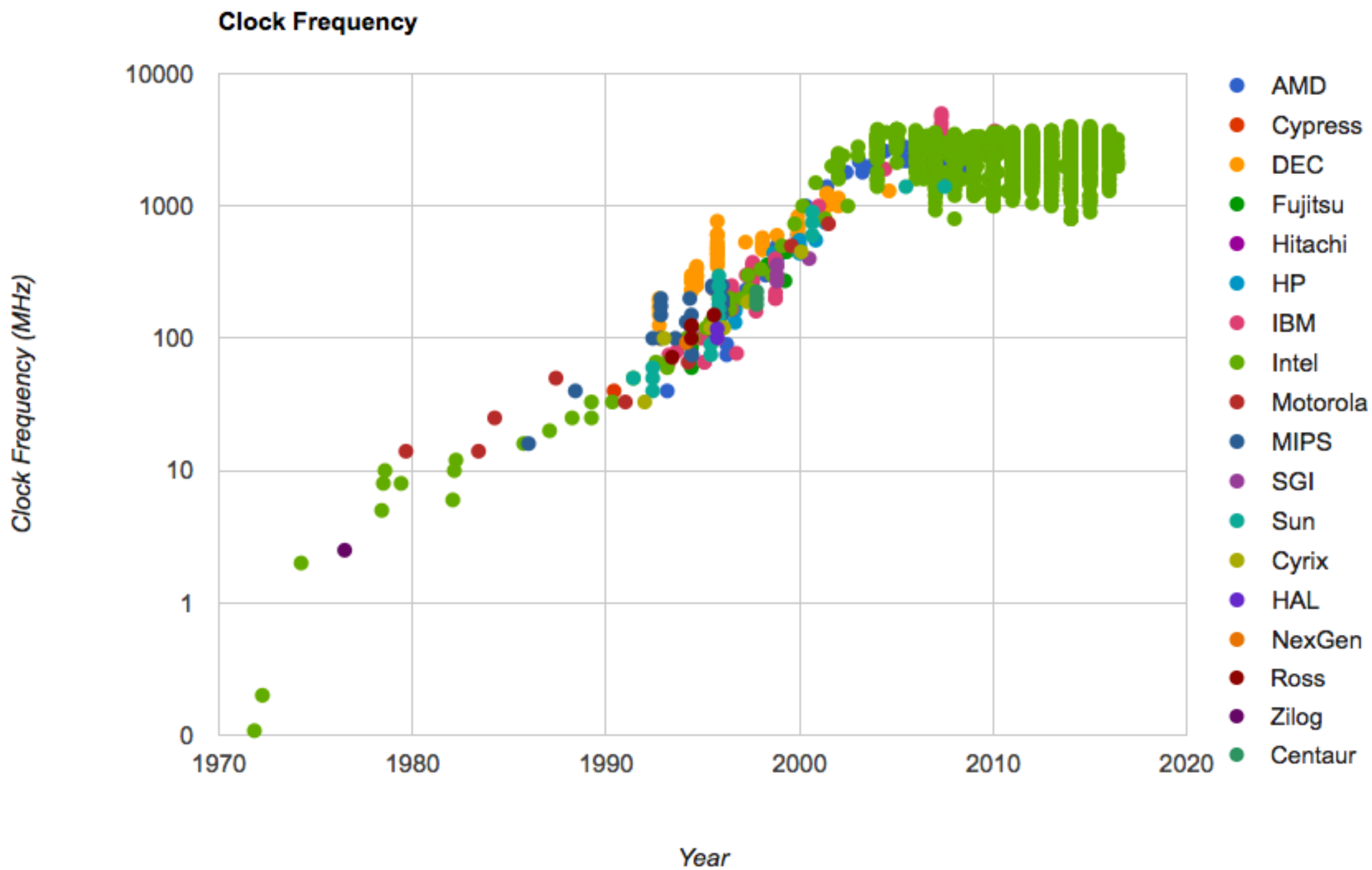
- Project 5 due Tue 12/12 at 8pm
- Final survey due Thu 12/14 at 8pm
- Office hours in discussion this week, in the discussion rooms
- No lecture on Tue 12/12
  - Office hours instead, in 2632 BBB

Final Exam

- Thursday, 12/21 10:30am-12:30pm
  - DOW 1010 for **uniqnames** that start with `a-i`
  - DOW 1017 for **uniqnames** that start with `j-z`
- Comprehensive, with emphasis on Lectures 13-25 (Operational Semantics through Parallel Computing)
- Covers all material in notes except:
  - §6.3.5 (Nested Iteration)
  - §7.2 (Asynchronous Tasks)

CPU Performance

- Performance of individual CPU cores has largely stagnated in recent years



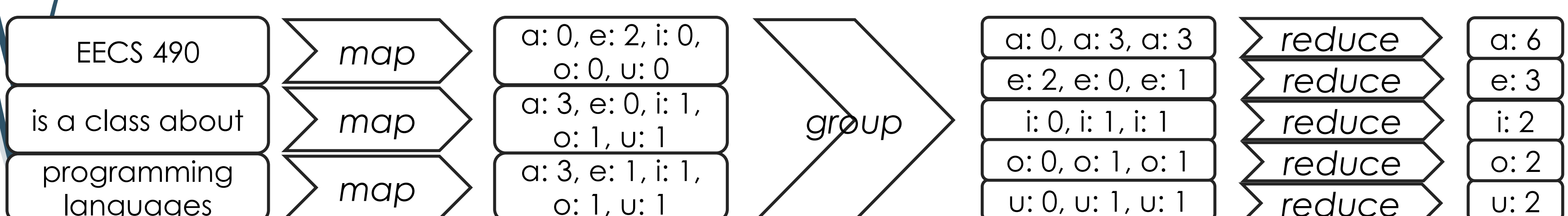
http://cpudb.stanford.edu

Parallelism

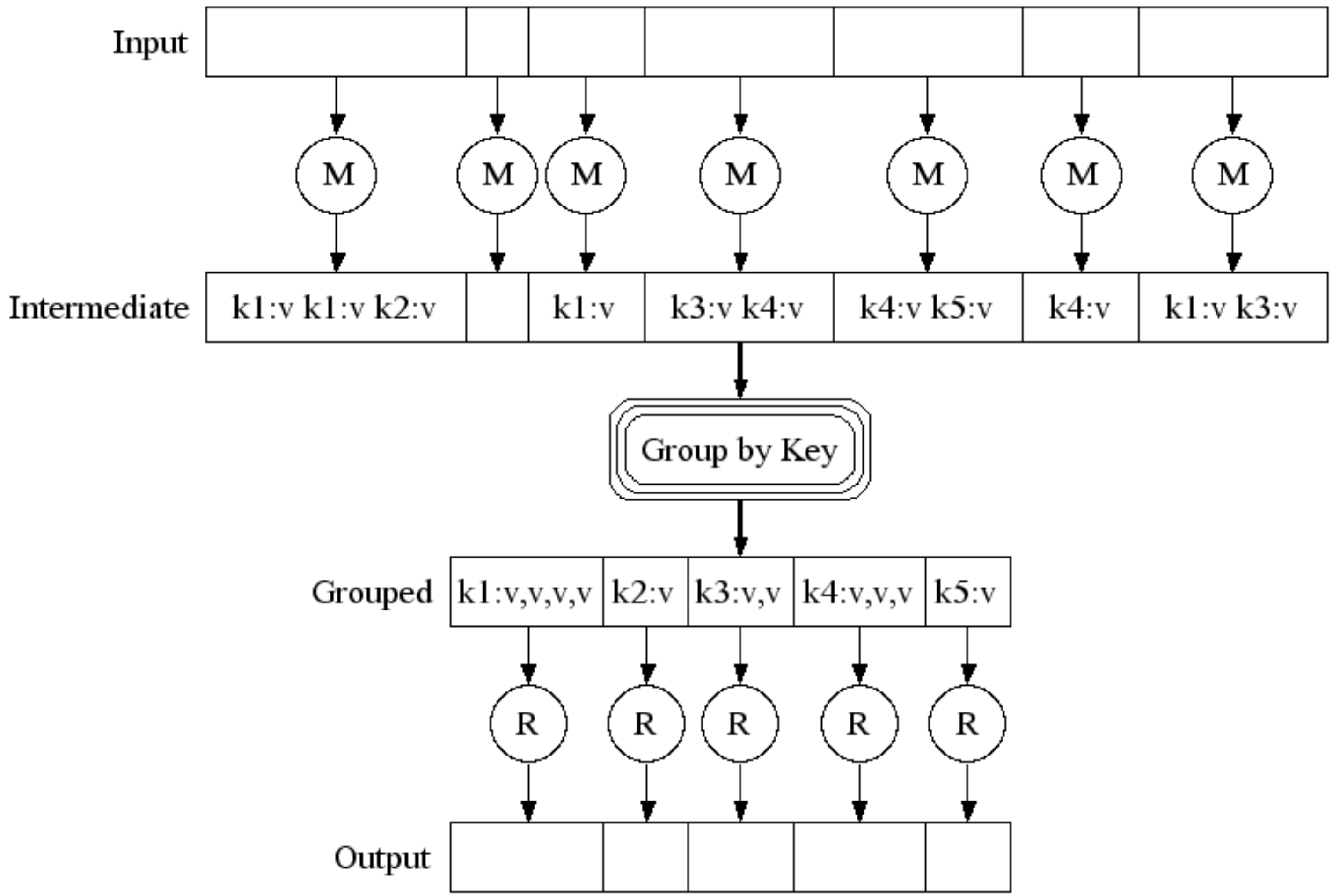
- Applications must be parallelized in order run faster
  - Waiting for a faster CPU core is no longer an option
- Parallelism is easy in functional programming:
  - When a program contains only pure functions, call expressions can be evaluated in any order, lazily, and in parallel
  - Referential transparency: a call expression can be replaced by its value (or vice versa) without changing the program
- We will look at MapReduce, a framework for such computations
- But not all problems can be solved efficiently using functional programming, so we will also look at strategies for parallelism with mutable shared state

MapReduce Evaluation Model

- Map phase: Apply a mapper function to inputs, emitting a set of intermediate key-value pairs
  - The mapper takes an iterator over inputs, such as text lines
  - The mapper yields zero or more key-value pairs per input
- Reduce phase: For each intermediate key, apply a reducer function to accumulate all values associated with that key
  - The reducer takes an iterator over key-value pairs
  - All pairs with a given key are consecutive
  - The reducer yields 0 or more values, each associated with that intermediate key



MapReduce Execution Model



http://research.google.com/archive/mapreduce-osdi04-slides/index-auto-0007.html

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. Structured 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

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

Threads in Python

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

```
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)
```

Start the other thread

Function arguments

Function that new thread should run

Print output unordered

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

Processes in Python

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

```
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)
```

Start the other process

Function arguments

Function that new process should run

Print output unordered

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

The Problem with Shared State

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

```
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 the other thread completes

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

Atomic Operations

- Only the most basic operations are *atomic*, taking effect instantaneously, in CPython or any other system
  - Even in a mostly sequential system like CPython, a non-atomic operation can be interrupted by another thread
- The increment is actually several atomic operations

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 can end up with a value of 1, even though it was incremented twice!

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

Synchronized Data Structures

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

```
from queue import Queue

queue = Queue()

def increment():
    count = queue.get()
    queue.put(count + 1)

other = Thread(target=increment, args=())
other.start()

queue.put(0)
increment()

other.join()
print('count is now', queue.get())
```

Synchronized FIFO queue

Wait until an item is available

Add initial value of 0

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 Thread, Lock

counter = [0]
counter_lock = Lock()

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

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

A lock is a context manager

```
with counter_lock:
    count = counter[0]
    counter[0] = count + 1
```

Break Time!

- We’ll start again in five minutes.

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
- 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

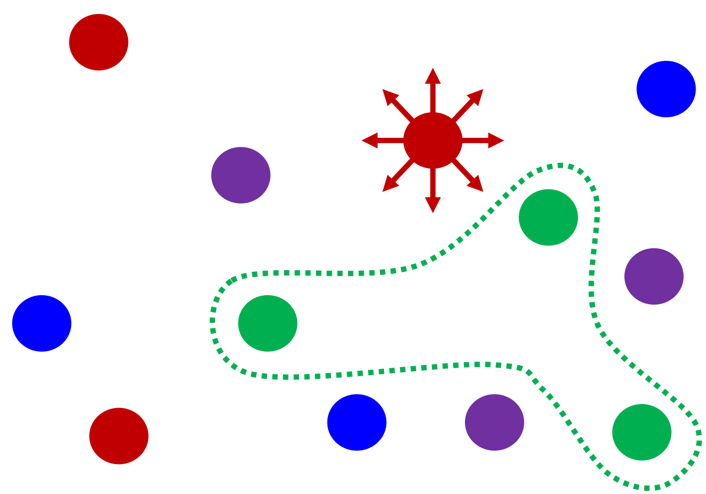
Web Crawler Synchronization

- URL coordination code:

```
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 or processes
- Forces are computed from particles' positions
  - Their positions constitute shared data
- The simulation is discretized into timesteps



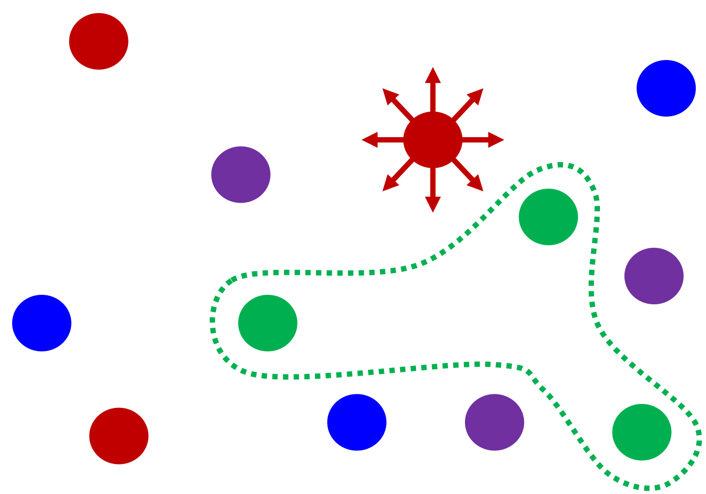
See `particle.uc` in Projects 2 and 3 for the sequential uC version.

Example: Particle Simulation

Concurrent reads are OK

Writes are to different locations

- In each timestep, each thread or 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



Solution 1: Barriers

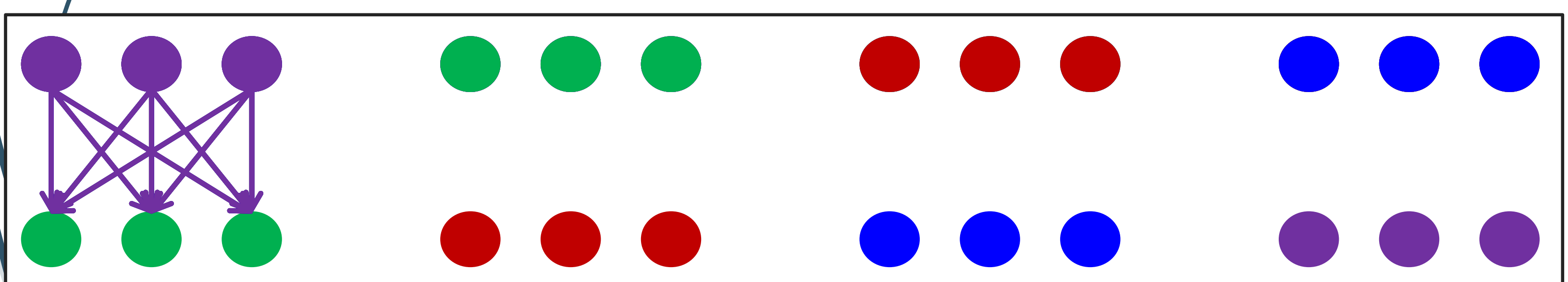
- In each timestep, each thread or process must:
  1. Read the positions of every particle (read shared data)
  2. Update acceleration of its own particles (access non-shared data)
  3. Update velocities of its own particles (access non-shared data)
  4. Update positions of its own particles (write shared data)
- Steps 1 and 4 conflict with each other
- We can solve the conflict by dividing the program into *phases*, ensuring that the phases do not overlap
- A *barrier* is a synchronization mechanism that enables 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

Where to Go from Here

- Related classes
  - EECS 483 and 583: Compilers
  - EECS 590: Advanced Programming Languages
- Read language specifications
  - Python's is very accessible
  - cppreference.com for C++
- Practice using the programming paradigms we've learned
  - Use the right tool for the job
- Keep up with new language features
  - C++17: structured bindings, template argument deduction on constructors, parallel STL algorithms, constexpr if, etc.

Good Luck on the Project and Exam!