CSE332: Data Abstractions

Lecture 15: Into to Parallelism and Concurrency

James Fogarty

Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman

Changing a Major Assumption

So far most or all of your study of computer science has assumed

One thing happened at a time

Called sequential programming – everything part of one sequence

Removing this assumption creates major challenges & opportunities

- Programming: Divide work among threads of execution and coordinate among them (i.e., synchronize their work)
- Algorithms: How can parallel activity provide speed-up (more throughput, more work done per unit time)
- Data structures: May need to support concurrent access (multiple threads operating on data at the same time)

A Simplified View of History

Writing correct and efficient multithreaded code is often much more difficult than single-threaded code

- Especially in typical languages like Java and C
- So we typically stay sequential whenever possible

From roughly 1980-2005, desktop computers got exponentially faster at running sequential programs

- About twice as fast every couple years

But nobody knows how to continue this

- Increasing clock rate generates too much heat
- Relative cost of memory access is too high
- Still making "wires exponentially smaller" (per Moore's "Law"), so we put multiple processors on the same chip (i.e., "multicore")

What to do with Multiple Processors?

- Next computer you buy will likely have 4 processors
  - Wait a few years and it will be 8, 16, 32, …
  - The chip companies have decided to do this (it is not a “law”)

- What can you do with them?
  - Run multiple totally different programs at the same time
    - Already do that? Yes, but with time-slicing
  - Do multiple things at once in one program
    - This will be our focus, and it is more difficult
    - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

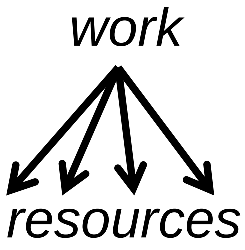
Parallelism vs. Concurrency

Note: Terms not yet standard but the perspective is essential

- Many programmers confuse these concepts

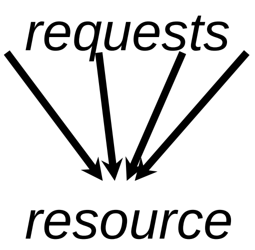
Parallelism:

Use extra resources to solve a problem faster



Concurrency:

Correctly and efficiently manage access to shared resources



There is some connection:

- Common to use threads for both
- If parallel computations need access to shared resources, then the concurrency needs to be managed

An Analogy

CS1 idea: A program is like a recipe for a cook

- One cook who does one thing at a time!

Parallelism:

- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But too many chefs and you spend all your time coordinating

Concurrency:

- Lots of cooks making different things,
  but only 4 stove burners in the kitchen
- Want to allow access to all 4 burners,
  but not cause spills or incorrect burner settings

Parallelism Example

Parallelism: Use extra resources to solve a problem faster (increasing throughput via simultaneous execution)

Pseudocode for array sum

- No ‘FORALL’ construct in Java, but we will see something similar
- Bad style for reasons we’ll see, but may get roughly 4x speedup

```
int sum(int[] arr){
  result = new int[4];
  len = arr.length;
  FORALL(i=0; i < 4; i++) { //parallel iterations
    result[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return result[0]+result[1]+result[2]+result[3];
}
int sumRange(int[] arr, int lo, int hi) {
   result = 0;
   for(j=lo; j < hi; j++)
      result += arr[j];
   return result;
}
```

Concurrency Example

Concurrency: Correctly and efficiently manage access to shared resources (from multiple possibly-simultaneous clients)

Pseudocode for a shared chaining hashtable

- Prevent *bad interleavings* (critical ensure correctness)
- But allow some concurrent access (critical to preserve performance)

```
class Hashtable<K,V> {
   …
   void insert(K key, V value) {
      int bucket = …;
      prevent-other-inserts/lookups in table[bucket]
      do the insertion
      re-enable access to arr[bucket]
   }
   V lookup(K key) {
      (similar to insert,
      but can allow concurrent lookups to same bucket)
   }
}
```

Shared Memory with Threads

The model we will assume is shared memory with explicit threads

Old story: A running program has

- One *program counter* (the current statement that is executing)
- One *call stack* (with each *stack frame* holding local variables)
- Objects in the *heap* created by memory allocation (i.e., **`new`**) (same name, but no relation to the heap data structure)
- *Static* fields in the class shared among objects

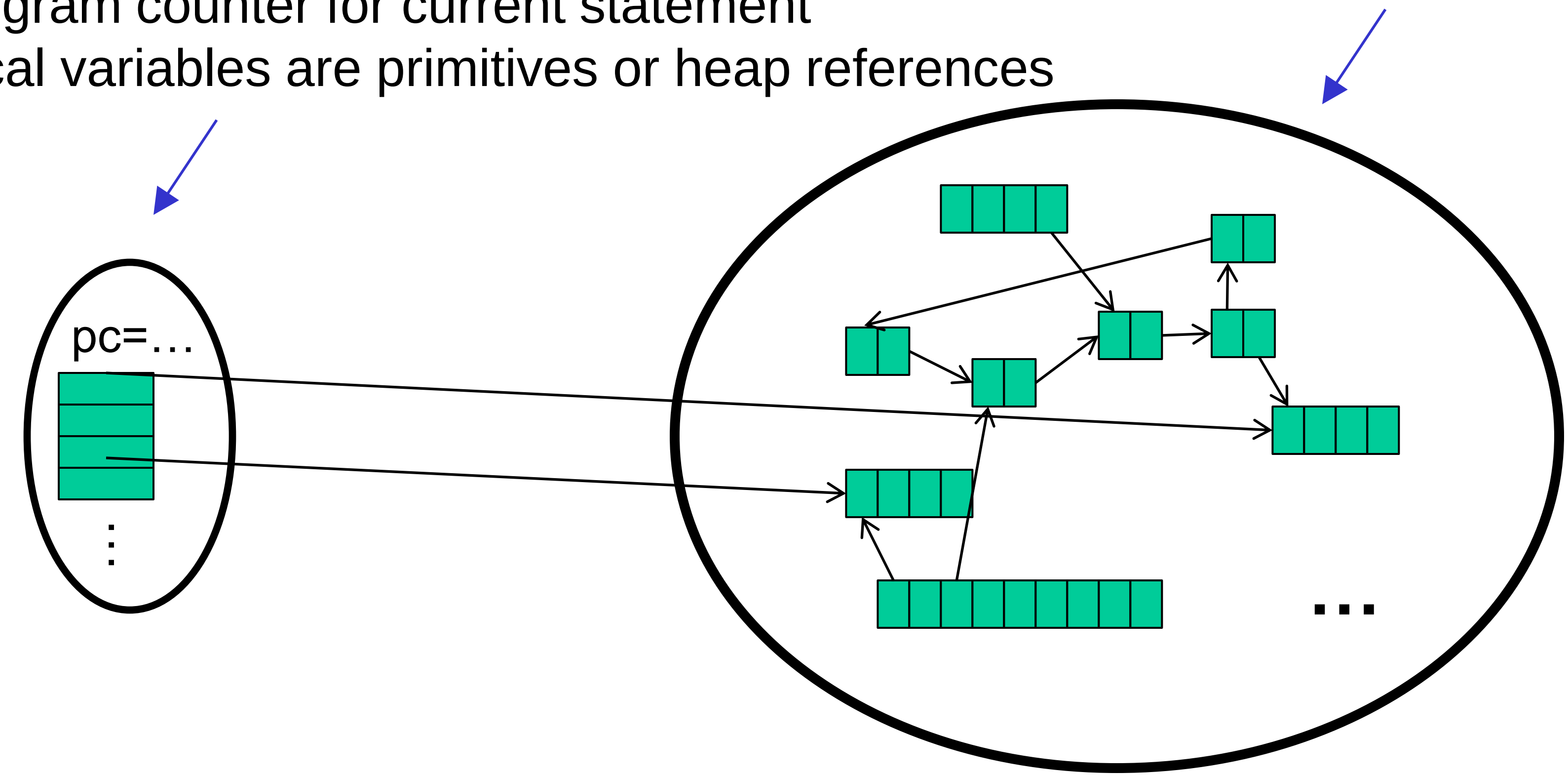
New story:

- A set of *threads*, each with a program and call stack
  - No access to another thread's local variables
- Threads can implicitly share objects and static fields
  - To *communicate among threads*, write values to a shared location that another thread reads

Old Story: Single-Threaded

Heap for all objects
and static fields

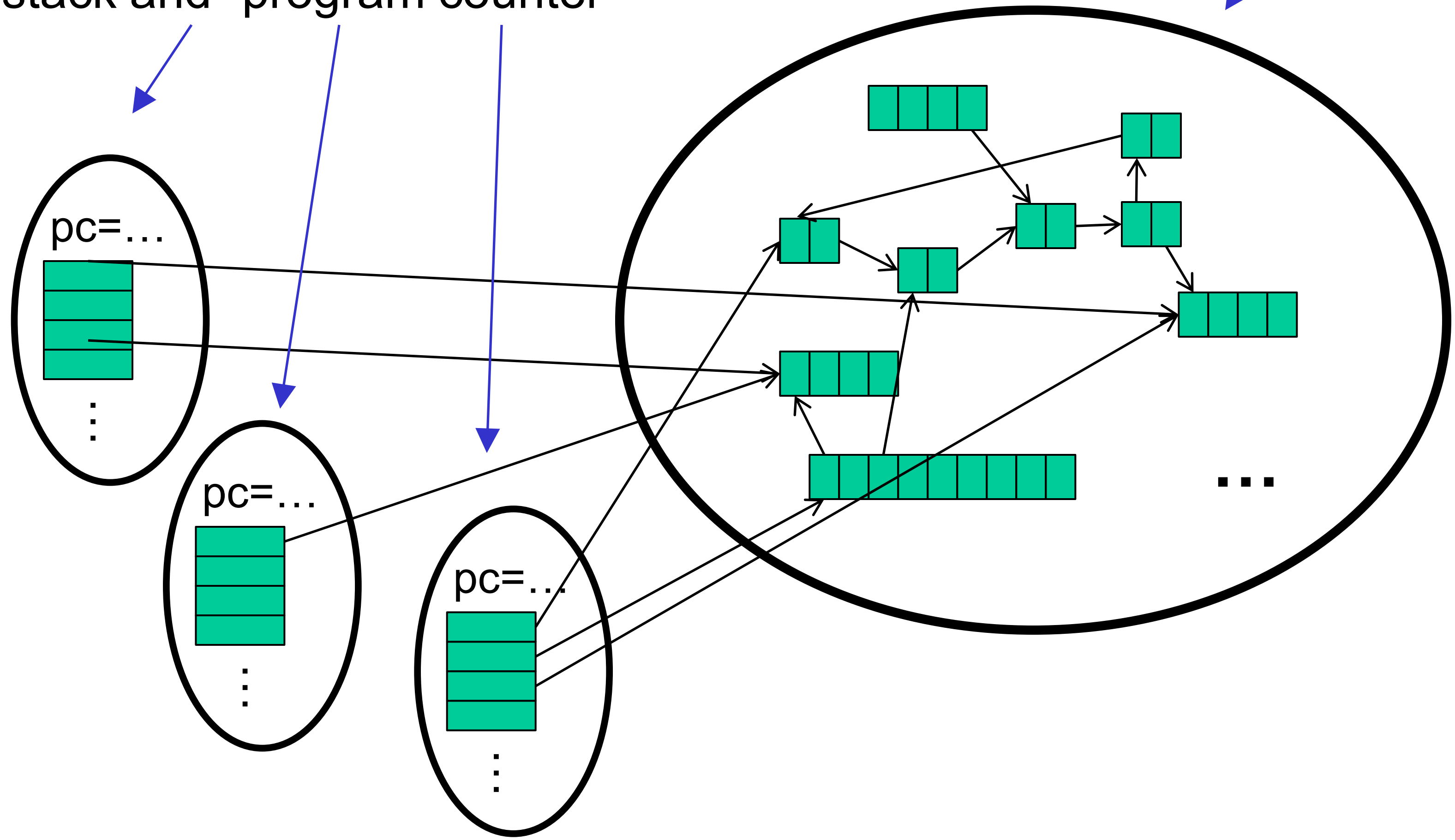
Call stack with local variables
Program counter for current statement
Local variables are primitives or heap references



New Story: Shared Memory with Threads

Heap for all objects
and static fields,
shared by all threads

Threads, each with own *unshared*
call stack and "program counter"



Other Models

We will focus on shared memory, but you should know several other models exist and have their own advantages

- Message-passing: Each thread has its own collection of objects. Communication is via explicitly sending/receiving messages
  - Cooks working in separate kitchens, mail around ingredients
- Dataflow: Programmers write programs in terms of a DAG.
  A node executes after all of its predecessors in the graph
  - Cooks wait to be handed results of previous steps
- Data parallelism: Have primitives for things like "apply function to every element of an array in parallel"

Our Needs

To write a shared-memory parallel program, need new primitives from a programming language or library

- Ways to create and *run multiple things at once*
  - Let’s call these things threads
- Ways for threads to *share memory*
  - Often just have threads with references to the same objects
- Ways for threads to *coordinate (a.k.a. synchronize)*
  - For now, a way for one thread to wait for another to finish
  - Other primitives when we study concurrency

Java Basics

First learn some basics built into Java via **`java.lang.Thread`**

- Then we will learn a better library for parallel programming

To get a new thread running:

1. Define a subclass **`C`** of **`java.lang.Thread`**, overriding **`run`**
2. Create an object of class **`C`**
3. Call that object's **`start`** method
    - **`start`** sets off a new thread, using **`run`** as its "main"

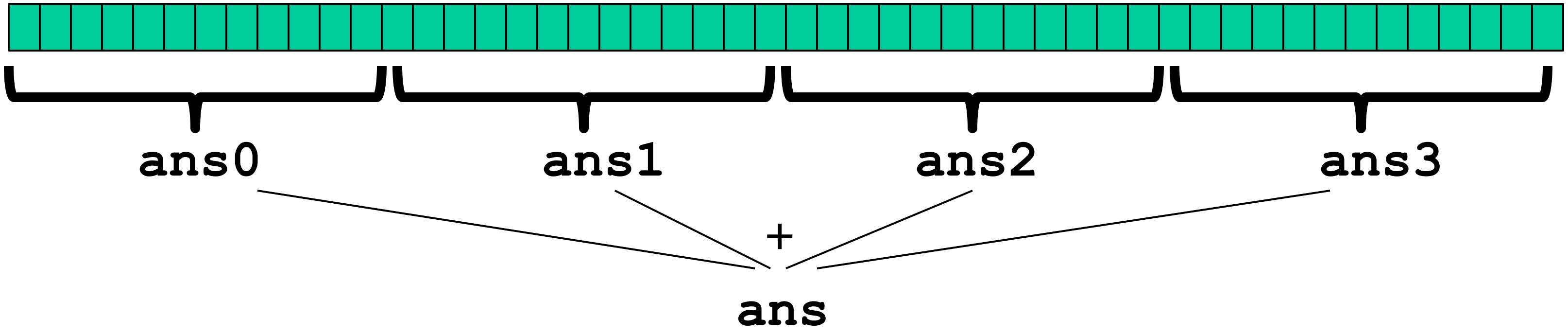
What if we instead called the **`run`** method of **`C`**?

- This would just be a normal method call, in the current thread

Then see how to share memory and coordinate via an example…

Parallelism Idea

- Example: Sum elements of a large array
- Idea Have 4 threads simultaneously sum 1/4 of the array
  - Warning: This is the inferior first approach, do not do this



- Create 4 *thread objects*, each given a portion of the work
- Call **`start()`** on each thread object to actually *run* it in parallel
- Somehow 'wait' for threads to finish
- Add together their 4 answers for the *final result*

First Attempt: The Thread

```
class SumThread extends java.lang.Thread {

  int lo; // arguments
  int hi;
  int[] arr;

  int ans = 0; // result

  SumThread(int[] a, int l, int h) {
    lo=l; hi=h; arr=a;
  }

  public void run() { //override must have this type
    for(int i=lo; i < hi; i++)
      ans += arr[i];
  }
}
```

Because we override a no-arguments/no-result **`run`**,
we use fields to communicate data across threads

First Attempt: Creating Threads (wrong)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}
```

```
int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++) // do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

Second Attempt: Starting Threads (still wrong)

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}
```

```
int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start(); // start not run
  }
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

Join: Our 'Wait for Thread' Method

- The **`Thread`** class defines various methods that provide primitive operations you could not implement on your own
  - For example: **`start`**, which calls **`run`** in a new thread
- The **`join`** method is another such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its **`run`** method returns after its execution)
  - Without join, we would have a 'race condition' on **`ts[i].ans`**
    - In short, problem if variable can be read/written simultaneously
- This style of parallel programming is called "fork/join"
  - If we write in this style, we avoid many concurrency issues
  - But certainly not all of them

Third Attempt: Correct in Spirit

```
class SumThread extends java.lang.Thread {
  int lo, int hi, int[] arr; // arguments
  int ans = 0; // result
  SumThread(int[] a, int l, int h) { … }
  public void run(){ … } // override
}
```

```
int sum(int[] arr){ // can be a static method
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// do parallel computations
    ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
    ts[i].start();
  }
  for(int i=0; i < 4; i++) { // combine results
    ts[i].join(); // wait for helper to finish!
    ans += ts[i].ans;
  }
  return ans;
}
```

Shared Memory?

- Fork-join programs thankfully do not require
  a lot of focus on sharing memory among threads

- But in languages like Java, there is memory being shared

- In our example:
  - **`lo`**, **`hi`**, **`arr`** fields written by "main" thread, read by helper thread
  - **`ans`** field written by helper thread, read by "main" thread

- When using shared memory, you must avoid race conditions
  - While studying parallelism, we'll stick with **`join`**
  - With concurrency, we'll learn other ways to synchronize