



CSE373: Data Structures & Algorithms

Lecture 26: Introduction to Multithreading & Fork-Join Parallelism

Catie Baker
Spring 2015

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 (**synchronize**) among them
- Algorithms: How can parallel activity provide speed-up (more **throughput**: 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 for single-threaded (i.e., sequential) code

- Especially in common languages like Java and C
- So typically stay sequential if 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
- But we can keep making “wires exponentially smaller” (Moore’s “Law”), so put multiple processors on the same chip (“multicore”)

What to do with multiple processors?

- Next computer you buy will likely have 4 processors (your current one might already)
 - Wait a few years and it will be 8, 16, 32, ...
 - The chip companies have decided to do this (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
 - Our focus – more difficult
 - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

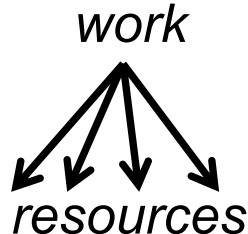
Parallelism vs. Concurrency

Note: Terms not yet standard but the perspective is essential

- Many programmers confuse these concepts

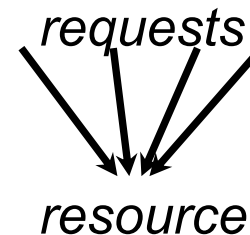
Parallelism:

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

We will just do a little parallelism, avoiding concurrency issues

An analogy

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

- One cook who does one thing at a time! (*Sequential*)

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
- Want to allow access to all 4 burners, but not cause spills or incorrect burner settings

Shared memory

The model we will assume is **shared memory** with **explicit threads**

- *Not* the only approach, may not be best, but time for only one

Old story: A running program has

- One **program counter** (current statement executing)
- One **call stack** (with each **stack frame** holding local variables)
- **Objects in the heap** created by memory allocation (i.e., **new**)
 - (nothing to do with data structure called a heap)
- **Static fields** - belong to the class and not an instance (or object) of the class. Only one for all instances of a class.

New story:

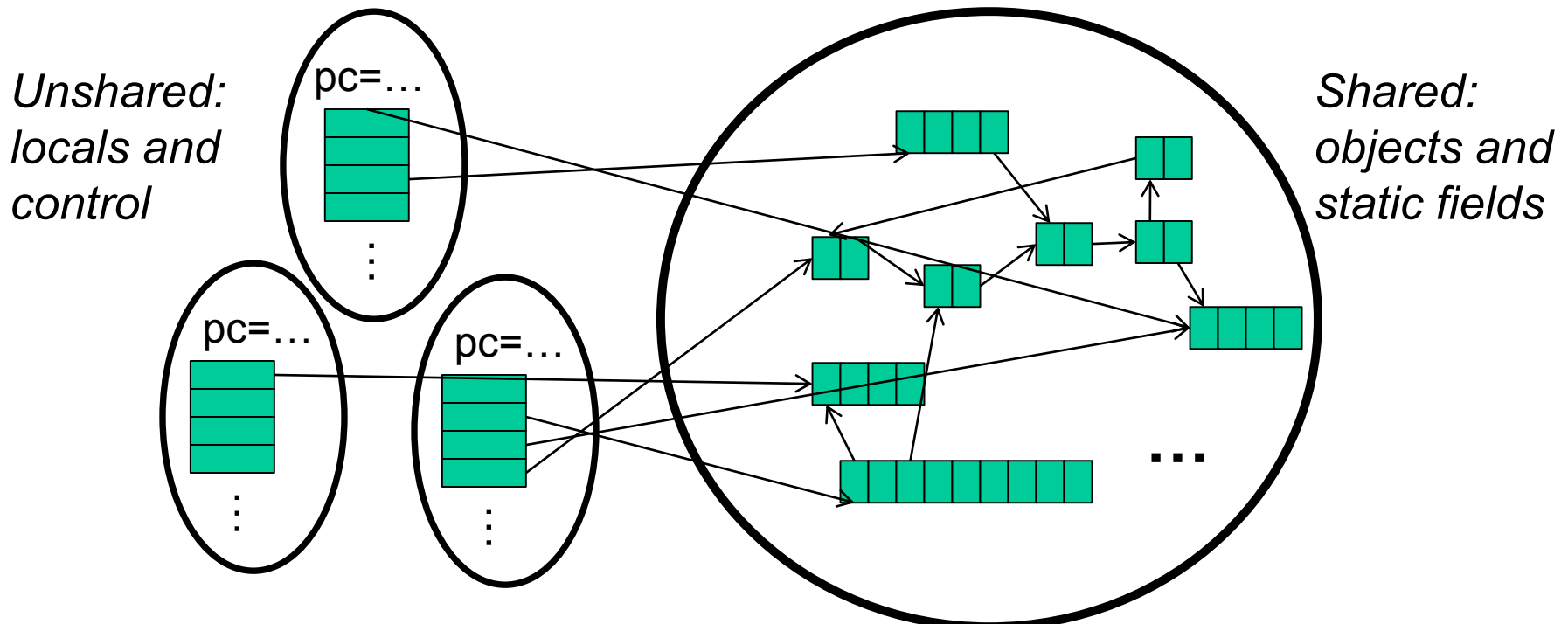
- A set of **threads**, each with its own program counter & call stack
 - No access to another thread's local variables
- Threads can (implicitly) share static fields / objects
 - To *communicate*, write somewhere another thread reads

Shared memory

Threads each have own unshared call stack and current statement

- (pc for “program counter”)
- local variables are numbers, `null`, or heap references

Any objects can be shared, but most are not



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)*
 - A way for one thread to wait for another to finish
 - [Other features needed in practice for concurrency]

Java basics

Learn a couple basics built into Java via `java.lang.Thread`

- But for style of parallel programming we'll advocate, do *not* use these threads; use Java 7's ForkJoin Framework instead

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”

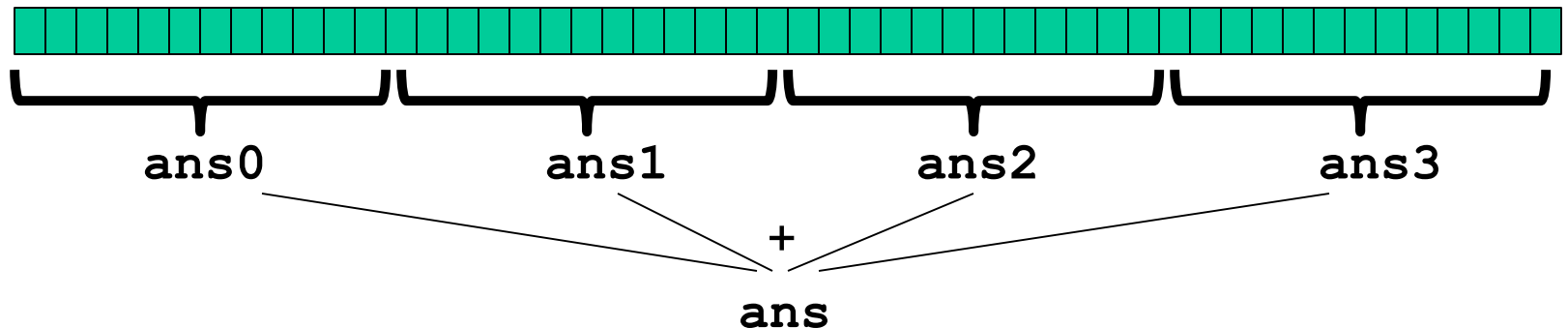
What if we instead called the `run` method of `C`?

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

Let's 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 an inferior first approach, but it's usually good to start with something naïve works

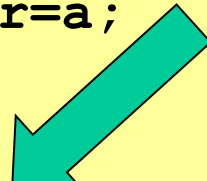


- Create 4 *thread objects*, each given a portion of the work
- Call `start()` on each thread object to actually *run* it in parallel
- *Wait* for threads to finish using `join()`
- Add together their 4 answers for the *final result*

First attempt, part 1



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

First attempt, continued (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 *(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;
}
```

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;
}
```

Join (not the most descriptive word)

- The **Thread** class defines various methods you could not implement on your own
 - For example: **start**, which calls **run** in a new thread
- The **join** method is valuable for coordinating this kind of computation
 - Caller blocks until/unless the receiver is done executing (meaning the call to **run** returns)
 - Else we would have a **race condition** on **ts[i].ans** (answer would depend on what finishes first)
- This style of parallel programming is called “fork/join”
- Java detail: code has 1 compile error because **join** may throw **java.lang.InterruptedException**
 - In basic parallel code, should be fine to catch-and-exit

Shared memory?

- Fork-join programs (thankfully) do not require much 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
 - We will stick with **join** to do so

A better approach

Several reasons why this is a poor parallel algorithm

1. Want code to be reusable and efficient across platforms
 - “Forward-portable” as core count grows
 - So at the very least, parameterize by the number of threads

```
int sum(int[] arr, int numTs) {
    int ans = 0;
    SumThread[] ts = new SumThread[numTs];
    for(int i=0; i < numTs; i++){
        ts[i] = new SumThread(arr, (i*arr.length)/numTs,
                               ((i+1)*arr.length)/numTs);

        ts[i].start();
    }
    for(int i=0; i < numTs; i++) {
        ts[i].join();
        ans += ts[i].ans;
    }
    return ans;
}
```

A Better Approach

2. Want to use (only) processors “available to you *now*”
 - Not used by other programs or threads in your program
 - Maybe caller is also using parallelism
 - Available cores can change even while your threads run

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numTs) {
    ...
}
```

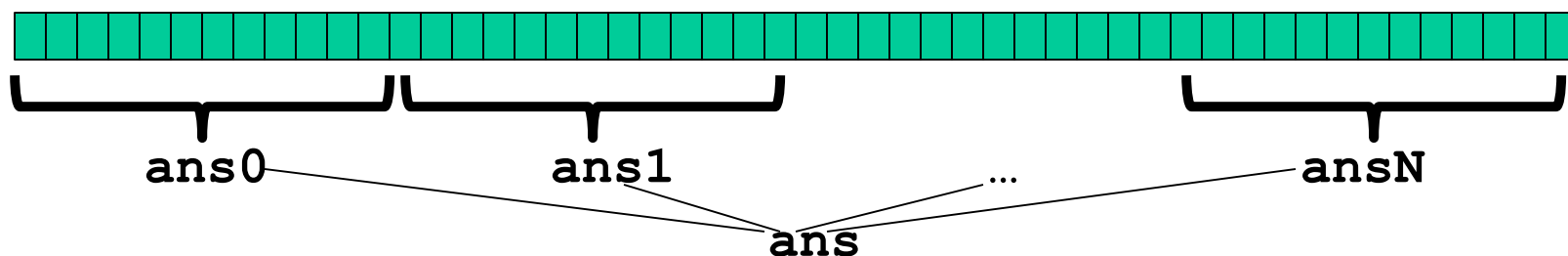
A Better Approach

3. Though unlikely for `sum`, in general subproblems may take significantly different amounts of time
 - Example: Apply method `f` to every array element, but maybe `f` is much slower for some data items
 - Example: Is a large integer prime?
 - If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup
 - Example of a **load imbalance**

A Better Approach

The counterintuitive (?) solution to all these problems is to use lots of threads, far more than the number of processors

- But this will require changing our algorithm
- [And using a different Java library]



1. Forward-portable: Lots of helpers each doing a small piece
2. Processors available: Hand out “work chunks” as you go
3. Load imbalance: No problem if slow thread scheduled early enough
 - Variation probably small anyway if pieces of work are small

Naïve algorithm is poor

Suppose we create 1 thread to process every 1000 elements

```
int sum(int[] arr) {  
    ...  
    int numThreads = arr.length / 1000;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
}
```

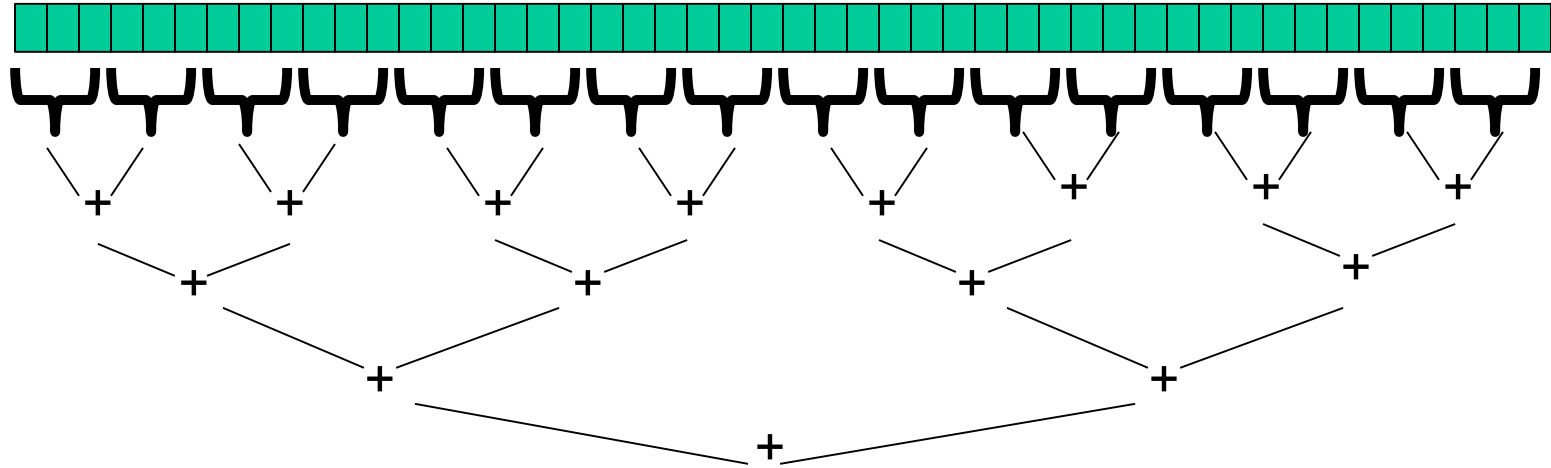
Then combining results will have `arr.length / 1000` additions

- Linear in size of array (with constant factor 1/1000)
- Previously we had only 4 pieces (constant in size of array)

In the extreme, if we create 1 thread for every 1 element, the loop to combine results has length-of-array iterations

- Just like the original sequential algorithm

A better idea



This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

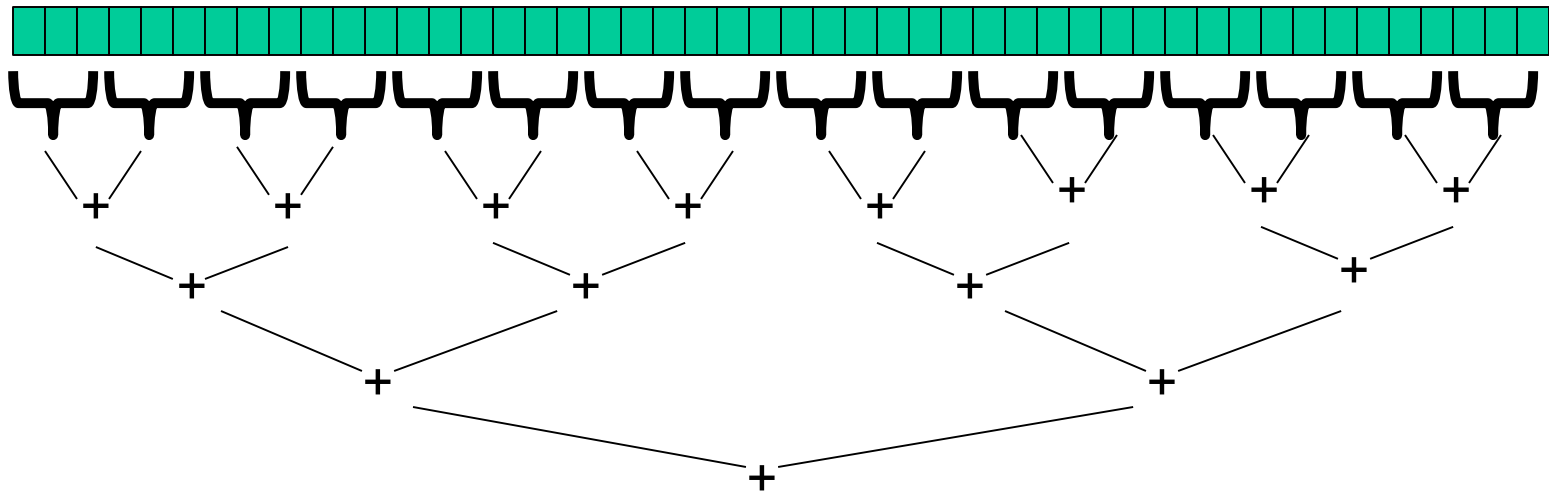
Divide-and-conquer to the rescue!

```
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
        if(hi - lo < SEQUENTIAL CUTOFF)
            for(int i=lo; i < hi; i++)
                ans += arr[i];
        else {
            SumThread left = new SumThread(arr, lo, (hi+lo)/2);
            SumThread right = new SumThread(arr, (hi+lo)/2, hi);
            left.start();
            right.start();
            left.join(); // don't move this up a line - why?
            right.join();
            ans = left.ans + right.ans;
        }
    }
}

int sum(int[] arr) {
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```


Divide-and-conquer really works

- The key is divide-and-conquer parallelizes the result-combining
 - If you have enough processors, total time is height of the tree: $O(\log n)$ (optimal, exponentially faster than sequential $O(n)$)



Being realistic

- In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup
 - Total time $O(n/\text{numProcessors} + \log n)$
- In practice, creating all those threads and communicating swamps the savings, so:
 - Use a *sequential cutoff*, typically around 500-1000
 - Eliminates *almost all* the recursive thread creation (bottom levels of tree)
 - *Exactly* like quicksort switching to insertion sort for small subproblems, but more important here
 - Do not create two recursive threads; create one and do the other “yourself”
 - Cuts the number of threads created by another 2x

Being realistic, part 2

- Even with all this care, Java's threads are too "heavyweight"
 - Constant factors, especially space overhead
 - Creating 20,000 Java threads is just a bad idea ☹
- The [ForkJoin Framework](#) is designed to meet the needs of divide-and-conquer fork-join parallelism
 - In the Java 7 standard libraries
 - Library's implementation is a fascinating but advanced topic
 - Next lecture will discuss its guarantees, not how it does it
 - Names of methods and how to use them slightly different