CSE 373: Data Structures & Algorithms
Introduction to Parallelism and Concurrency

Riley Porter

Winter 2017

Course Logistics

- HW5 due → hard cut off is today
- HW6 out → due Friday, sorting
- Extra lecture topics (last Friday, today, and Wednesday) along with more extra resources posted soon.
- Final exam in a week! Review from TAs next Monday (details TBA).

Changing a major assumption

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

One thing happened at a time

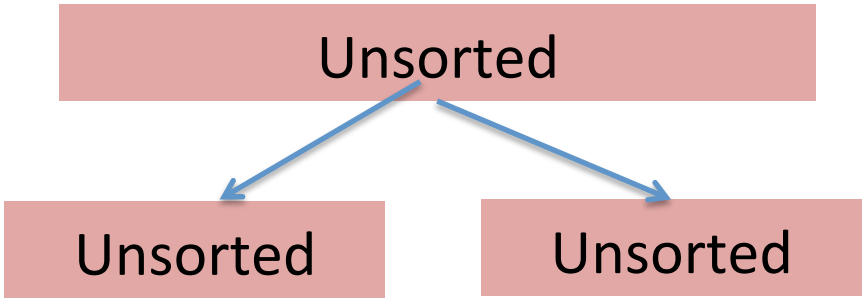
Called sequential programming – everything part of one sequence

Removing this assumption creates major challenges & opportunities

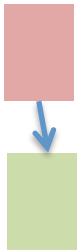
- Programming: Divide work among threads of execution and coordinate (synchronize) among them
- Algorithms: How can parallel activity provide speed-up (more throughput: work done per unit time)
- Data structures: May need to support concurrent access (multiple threads operating on data at the same time)

Review of Merge Sort

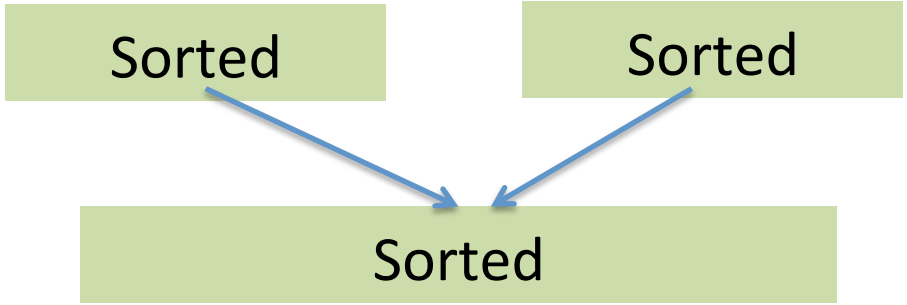
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge



Review of Merge Sort: Pseudocode

Are there any pieces of this code that we can do at the same time? Is there anything that doesn't rely on the other parts?

```
mergesort(input) {
        if (input.length < 2) {
                return input;
        } else {
                leftHalf = sort(input[0, ..., mid]);
                rightHalf = sort(input[mid + 1, ...]);
                return merge(smallerHalf, largerHalf);
        }
}
```

Review of Merge Sort: Pseudocode

The splitting! We can do the split part of the sort in parallel and if we have the sort(leftHalf) with sort(rightHalf) running at the same time, we could make this go faster.

```
mergesort(input) {
        if (input.length < 2) {
                return input;
        } else {
                leftHalf = sort(input[0, ..., mid]);
                rightHalf = sort(input[mid + 1, ...]);
                return merge(smallerHalf, largerHalf);
        }
}
```

A simplified view of the 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?

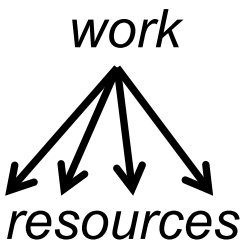
- Your current computer probably has 4 processors
  - Wait a few years and it will be 8, 16, 32, …
  - The chip companies have decided to do this

- What can you do with them?
  - Run multiple totally different programs at the same time
    - Already do that? Yes, the OS will do this for you
  - 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
    - Introduces weird bugs and is difficult... but a big payoff!

Parallelism vs. Concurrency

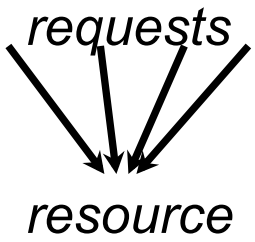
Parallelism:

Use extra resources to solve a problem faster



Concurrency:

Correctly and efficiently manage access to shared resources



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

Parallelism vs. Concurrency

Parallelism is when tasks literally run at the same time, eg. on a multicore processor.

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant.

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 (deadlocks, read-read vs read-write vs write-write conflicts, etc)

Parallelism Example

Parallelism: Use extra resources at the same time to solve faster

Pseudocode for array sum

- Bad style for reasons we'll see, but may get roughly 4x speedup

```
int sum(int[] arr){
  res = new int[4];
  len = arr.length;
  // loop that has parallel iterations (somehow)
  FORALL(i = 0; i < 4; i++) {
      res[i] = sumRange(arr,i*len/4,(i+1)*len/4);
  }
  return res[0] + res[1] + res[2] + res[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

Pseudocode for a shared chaining hashtable

- Prevent *bad interleavings* (correctness)
- But allow some concurrent access (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 table[bucket]
   }
   V lookup(K key) {
      (similar to insert, but can allow concurrent
       lookups to same bucket)
   }
}
```

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 current memory* created by allocation (i.e., **`new`**)
- *Static fields*

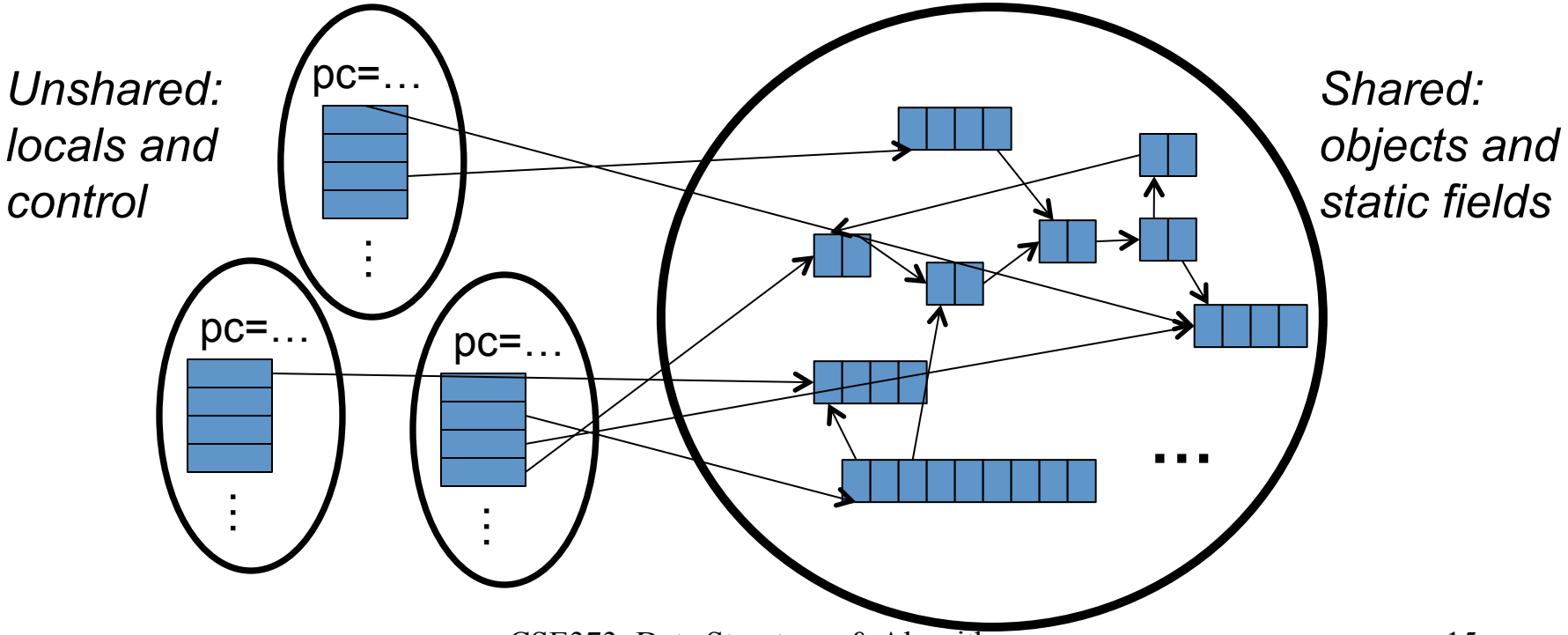
New story:

- 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 memory references

Any objects can be shared, but most are not



Parallelism Features We Need

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 - Threads

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

To get a new thread running:

1. Define a subclass **`C`** of **`java.lang.Thread`**, overriding **`run`**
2. Create a variable of type **`C (subclass of Thread)`**
3. Call that variable's **`start`** method
    - **`start`** starts the process on 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

What if we instead make C implement Runnable directly?

– slightly different idea, have to make your own thread

Java Basics – Thread continued

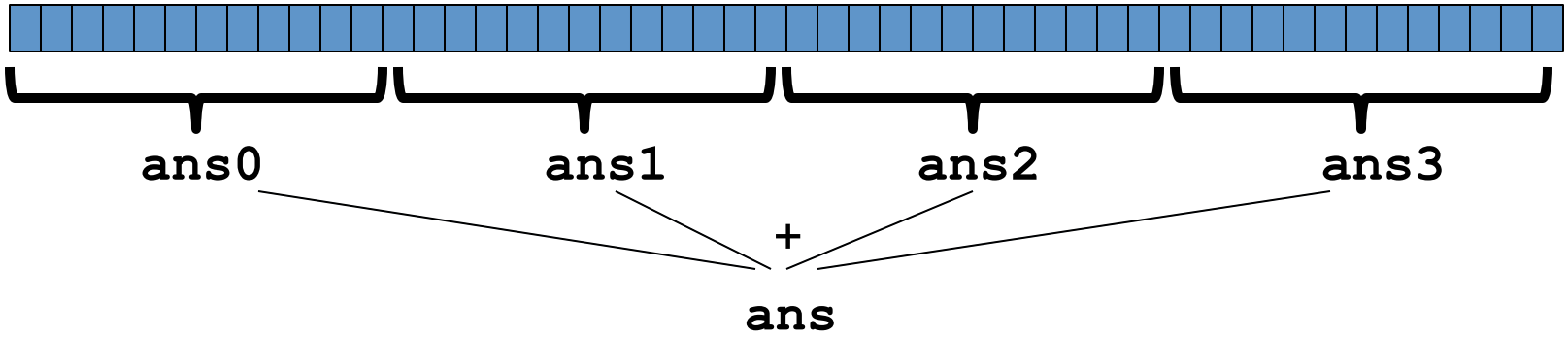
- The `start` method starts a new thread and calls `run`

- The `join` method joins results back together
  - Caller blocks until/unless the receiver is done executing (meaning the call to `run` returns)
  - Else we would have a race condition on our answer

- This style of parallel programming is called "fork/join"
  - Java has a ForkJoin Framework for this that is easier to use than implementing it on your own

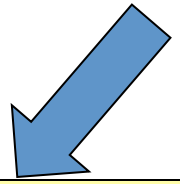
Parallelism: the basic idea

- 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



  - 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*

SumThread



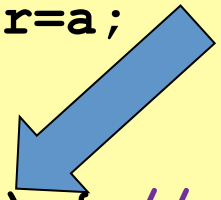
```
class SumThread extends java.lang.Thread {

  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 at sum (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){
  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);
    // threads never started
  for(int i=0; i < 4; i++) // combine results
    ans += ts[i].ans;
  return ans;
}
```

Second attempt at sum (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(); // started, but never joined
  }
  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(); // joined, but have to wait for each
    ans += ts[i].ans;
  }
  return ans;
}
```

A Better Approach, first: 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
  - If you didn't use **`join`**, you'd have to manage the answer you're computing very carefully

A Better Approach: Parameterized

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: Flexible to Computational Power

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

   – If you have 3 processors available and using 3 threads would take time **X**, then creating 4 threads would take time **1.5X**
     - Example: 12 units of work, 3 processors
       – Work divided into 3 parts will take 4 units of time
       – Work divided into 4 parts will take 3*2 units of time

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

A Better Approach: Flexible to computationally different chunks

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

Naïve algorithm for handling load balancing

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)

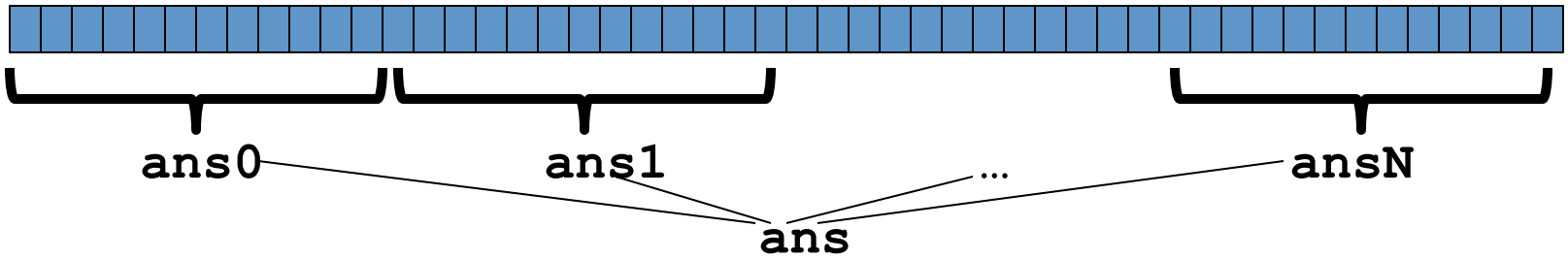
In the extreme, if we create 1 thread for every 1 element, the loop to combine results has length-of-array iterations

- Just like the original sequential algorithm

A Better Approach: Using Threads for load balancing

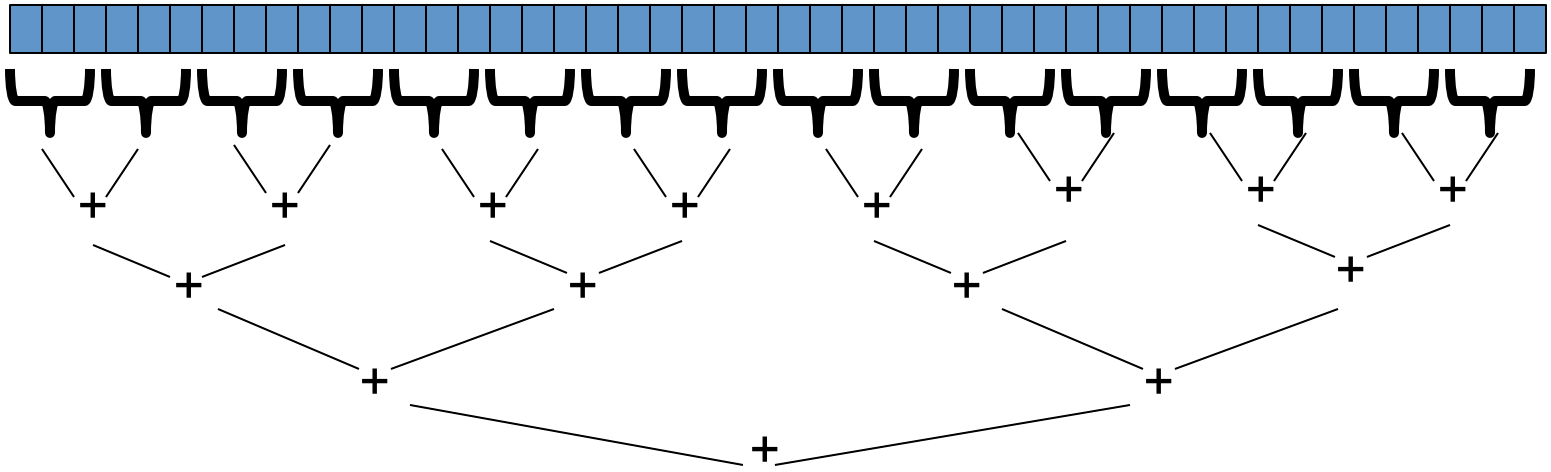
The counterintuitive (?) solution to all these problems is to use lots of threads, far more than the number of processors

- 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
   - If 3 processors available and have 100 threads, then ignoring constant-factor overheads, extra time is < 3%
3. Load imbalance: No problem if slow thread scheduled early enough
   - Variation probably small anyway if pieces of work are small

Load Balancing: The More General Idea



We can implement load balancing in a straightforward way, using divide-and-conquer

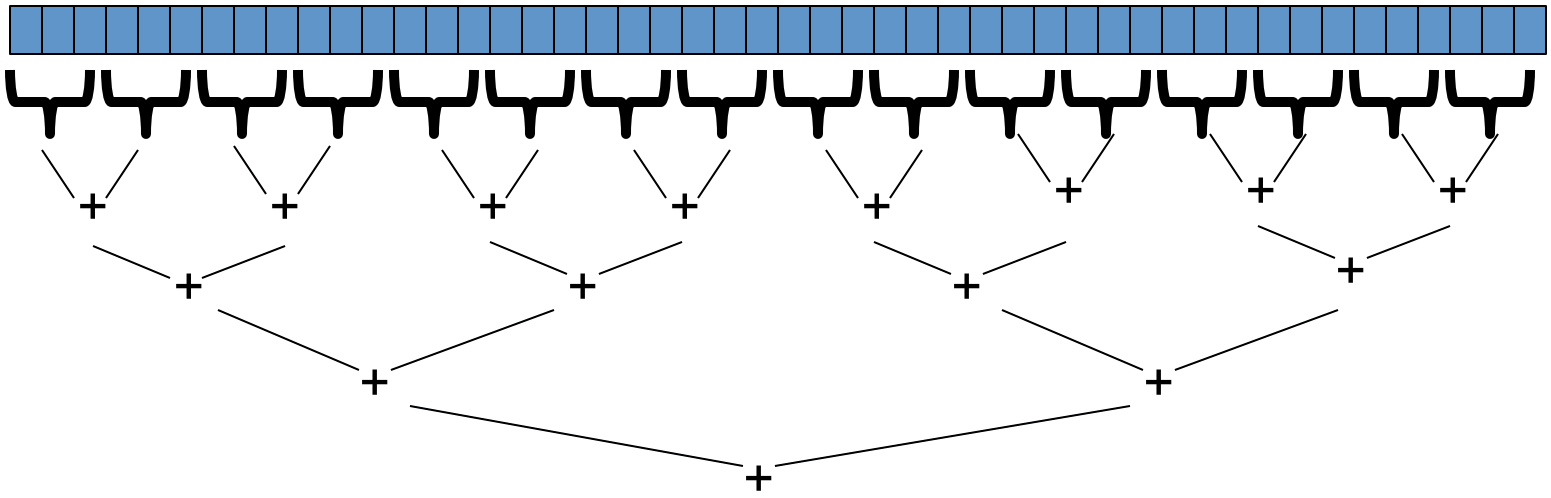
- Parallelism for the recursive calls

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: $O(\texttt{log}\ n)$ (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 $O(n/numProcessors + \log n)$

- 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
  - Can also create just one recursive thread; create one and do the other "yourself" to cut the number of threads

Being more realistic

- 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

Concurrency and Map/Reduce

- These parallelism ideas are explored further in Map/Reduce for managing and computing large jobs

- Concurrency introduces much more complexity surrounding shared memory and resources

- Applications / where you'll see this again:
  - Operating Systems (threads)
  - Database Management Systems (locks, logs, recovering from crashes with multiple threads, deadlocks, rollbacks)
  - Services with concurrent access (web apps, multiple users)
  - Distributed Systems (concurrent management)
  - Networks (threads looping and listening for packets, passing information between threads concurrently)

Today's Takeaways

- That parallelism and concurrency exist as programming constructs
  - Managing resources and conflicts is hard
  - Makes your code faster and applications usable by many users
  - There is **much much** more out there than what we discussed
- Threads exist, Java's particulars:
  - run method called by start()
  - join() waits for the Thread to be done (to die)
  - Runnable interface
- Why using divide-and-conquer for parallelism is best
  - lots of small tasks, combines results in parallel
  - Java has ForkJoin Framework