Announcements

- Homework 4 – due NOW, Monday Feb 14th at the BEGINNING of lecture
- Project 2 – Phase B due Tues Feb 15th at 11pm
  - Clarifications posted, check Msg board, email cse332-staff
- Homework 5 – due Friday Feb 18th at the BEGINNING of lecture

Today

- Graphs
  - Shortest Paths

- Intro to Parallelism
  - Multithreading & Fork-Join Parallelism

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)

Writing correct and efficient multithreaded code is often much more difficult than for single-threaded (i.e., sequential) code

A simplified view of history

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
  - 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

2011/02/11 2
Parallelism vs. Concurrency

Note: These terms are not yet standard, but the difference in perspective is essential
- Many programmers confuse them
- Remember that Parallelism ≠ Concurrency

Parallelism: Use more resources for a faster answer
Concurrency: Correctly and efficiently allow simultaneous access to something (memory, printer, etc.)

There is some connection:
- Many programmers use threads for both
- If parallel computations need access to shared resources, then something needs to manage the concurrency

CSE332: Next few lectures on parallelism, then a few on concurrency

Parallelism Example
Parallelism: Increasing throughput by using additional computational resources (code running simultaneously on different processors)
Example in pseudocode (not Java, yet): sum elements of an array
- No such 'FORALL' construct, but we’ll see something similar
- If you had 4 processors, might get roughly 4x speedup

```java
int sum(int[] arr){
    res = new int[4];
    len = arr.length;
    FORALL(i=0; i < 4; i++) {
        res[i] = help(arr, i*len/4, (i+1)*len/4);
    }
}
```

Concurrency Example
Concurrency: Allowing simultaneous or interleaved access to shared resources from multiple clients
Ex: Multiple threads accessing a hash-table, but not getting in each other's ways
Example in pseudocode (not Java, yet): chaining hashtable
- Essential correctness issue is preventing bad interleavings
- Essential performance issue not preventing good concurrency
  - One 'solution' to preventing bad inter-leavings is to do it all sequentially

```java
class Hashtable<K, V> {
    …
    Hashtable(Comparator<K> c, Hasher<K> h) { … }
    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) {
        (like insert, but can allow concurrent lookups to same bucket)
    }
}
```

A cooking analogy
CS142 idea: Writing a program is like writing a recipe for a cook
- One cook who does one thing at a time!

Parallelism: (Let's get the job done faster!)
- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But we can go too far: if we had 1 helper per potato, we'd spend too much time coordinating

Concurrency: (We need to manage a shared resource)
- Lots of cooks making different things, but only 4 stove burners
- Want to allow simultaneous access to all 4 burners, but not cause spills or incorrect burner settings

Shared memory with Threads
The model we will assume is shared memory with explicit threads

Old story: A running program has
- One call stack (with each stack frame holding local variables)
- One program counter (aka pc = current statement executing)
- Static fields
- Objects (created by new) in the heap (nothing to do with heap data structure)

New story:
- A set of threads, each with its own call stack & program counter
  - No access to another thread's local variables
- Threads can (implicitly) share static fields / objects
  - To communicate, write values to some shared location that another thread reads from

Old Story: one call stack, one pc
Heap for all objects and static fields

- Call stack with local variables
- pc determines current statement
- local variables are numbers/null or heap references
New Story: Shared memory with Threads

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

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

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 explicit messages; language has primitives for sending and receiving them.
  - Cooks working in separate kitchens, emailing back and forth
- **Dataflow**: Programmers write programs in terms of a DAG and 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”
  - ...

Some Java basics

- Many languages/libraries provide primitives for creating threads and synchronizing them
- We will show you how Java does it
  - For parallelism, will advocate not using Java’s built-in threads directly, but it’s still worth seeing them first
- Steps to creating another thread:
  1. Define a subclass of `java.lang.Thread`, overriding `run()`
  2. Create an object of class `c`
  3. Call that object’s `start()` method
     - The code that called `start()` will continue to execute after `start()` is called
     - A new thread will be created, with code executing in the object’s `run()` method
- What happens if, for step 3, we called `run()` instead of `start()`?

Parallelism idea

- Example: Sum elements of an array (presumably large)
- Use 4 threads, which each sum 1/4 of the array

Partial Code for first attempt (with Threads)

- Assume `SumThread`’s `run()` simply loops through the given indices and adds the elements

```
int sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start();
    }
    for(int i=0; i < 4; i++) {
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    }
    return ans;
}
```

Overall, should work, but not ideal

Sum elements of an array

- Each thread learns what part of the array to sum by the parameters passed to the constructor when its `SumThread` object is created:
  - `ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);`
  - `ts[i].start();` // this calls `run` on each thread
- Each thread sets its own `ans` field in its `SumThread` object

```
```
Join: Our ‘wait’ method for Threads

- The `Thread` class defines various methods that provide the threading primitives 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()` returns)
  - If we didn’t use `join`, we would have a ‘race condition’ (more on these later) on `ts[i].ans`  
- Essentially, if it’s a problem if any 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

Shared memory?

- Fork-join programs (thankfully) don’t 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

Improvements

The perhaps counter-intuitive solution to all these problems is to cut up our problem into many pieces, far more than the number of processors

- Idea: When processor finishes one piece, it can start another
- This will require changing our algorithm somewhat

1. Forward-portable: Lots of threads each doing a small piece
2. Processors available used well: Hand out threads as you go
   - Processors pick up new piece when done with old
3. Load imbalance: No problem if slow thread scheduled early enough
   - Variation probably small anyway if pieces of work are small

Complete Code (correct in spirit)

```java
public class SumThread {
    int lo, hi, int[] arr;//Fields to know what to do
    int ans = 0; // For communicating result
    SumThread(int[] a, int l, int h) {
        lo=l; hi=h; arr=a;
    }
    public void run() { //overriding, must have this type
        for(int i=lo; i < hi; i++) //sum my part of the array
            ans += arr[i];
    }
}

class C {
    static 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);
        ts[i].start(); // (start not run!)
    for(int i=0, i < 4; i++) {
        // combine results
        ts[i].join(); // wait for all 4 threads to finish their run method
        ans += ts[i].ans; // as a thread finishes, add
        // their ans to overall ans
    } return ans;
}
```

Problems with our current approach

The above method would work, but we can do better for several reasons:

1. Want code to be reusable and efficient across platforms
   - Be able to work for a variable number of processors (not just hardcoded to 4): ‘forward portable’
2. Even with knowledge of # of processors on the machine, we should be able to use them more dynamically
   - This program is unlikely to be the only one running; shouldn’t assume it gets all the resources (processors)
   - # of ‘free’ processors is likely to change over the course of time; be able to adapt
3. Different threads may take significantly different amounts of time (unlikely for sum, but common in many cases)
   - Example: Apply method # to every array element, but maybe it is much slower for small data items than others; say, verifying primes
   - If we create 4 threads and all the slow data is processed by 1 of them, we won’t get nearly a 4x speedup (‘load imbalance’)

Naïve algorithm doesn’t work

- Suppose we create 1 thread to process every 100 elements

```java
int sum(int[] arr)
    // How many pieces of size 100 do we have?
    int numThreads = arr.length / 100;
    SumThread[] ts = new SumThread[numThreads];
    ...
```

- Then combining results will have:
  - `numThreads = arr.length / 100`
  - `additions to do = linear in size of array (before we only had 4 pieces Θ(1) to combine)`
  - In the extreme, suppose we create one thread per element – If we use a for loop to combine the results, we have N iterations
  - In either case we get a Θ(n) algorithm with the combining of results as the bottleneck....
A better idea for combining... look familiar?

• Start with full problem at root
• Halve and make new thread until size is at some cutoff
• Combine answers in pairs as we return
• This will start small, and 'grow' threads to fit the problem
• This is straightforward to implement using divide-and-conquer

Remember Mergesort?

Example: summing an array with 10 elements. (too small to actually want to use parallelism)
The algorithm produces the following tree of recursion, where the range [i,j) includes i and excludes j:

Code looks something like this (still using Java Threads)

class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr;//fields to know what to do
    int ans = 0; // for communicating result
    SumThread(int[] a, int i, int j) {-
        lo = i; hi = j;
        ans = 0;
    }
    public void run() {
        if(hi - lo < SEQUENTIAL_CUTOFF)
            for(int i=lo; i < hi; i++)
                ans += arr[i];
        else // create 2 threads, each will do ½ the work
            SumThread left = new SumThread(arr,lo,(hi+lo)/2);
            left.start();
            left.join(); // don't move this up a line - why?
            right.join(); // why?
            right.start();
            right.join();
            ans = left.ans + right.ans;
    }
}

class C {
    static int sum(int[] arr)
        SumThread t = new SumThread(arr,0,arr.length);
        t.run(); // only creates one thread
        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 depth of the tree: O(log n) (optimal, exponentially faster than sequential O(n))
  – Next lecture: study reality of P < O(n) processors
• Will write all our parallel algorithms in this style
  – But using a special library designed for exactly this
  • Takes care of scheduling the computation well
  • Often relies on operations being associative like +

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 that inter-thread communication swamps the savings, so we will try to limit the creation of threads two ways:
  1. Use a sequential cutoff, typically around 500-1000
     • As in quicksort, eliminates almost all recursion, but here it is even more important
  2. Don’t create two recursive threads; create one and do the other “yourself”
     • Cuts the number of threads created by another 2x
For comparison - Java Threads Version

class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; //fields to know what to do
    public void sum() {
        int ans = 0; // for communicating result
        SumThread(int[] a, int l, int h) { - }
        public void sum() {
            for(int i=lo; i < hi; i++)
                ans += arr[i];
            else { // create 2 threads, each will do ½ the work
                SumThread left = new SumThread(arr,lo,(hi+lo)/2);
                left.start();
                right.start();
                left.join(); // don't move this up a line - why?
                right.join();
                ans = left_ans + right_ans;
            }
        }
    }
}

class C {
    static int sum(int[] arr)
    SumThread t = new SumThread(arr,0,arr.length);
    t.run(); // only creates one thread
    return t.ans;
}