Changing a major assumption

So far in 142, 143, 311, and 332, we have 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

– 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 Example**

Parallelism: Increasing throughput by using additional computational resources (code running simultaneously)

Example in pseudocode (not Java, yet): sum elements of an array

- This example is bad style for reasons we’ll see
- 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++) { //parallel iterations
        res[i] = help(arr,i*len/4,(i+1)*len/4);
    }
}
int help(int[] arr, int lo, int hi) {
    result = 0;
    for(j=lo; j < hi; j++)
        result += arr[j];
    return result;
}
```

**Concurrency Example**

Concurrency: Allowing simultaneous or interleaved access to shared resources from multiple clients

Example in pseudocode (not Java, yet): chaining hashtable

- Essential correctness issue is preventing bad interleavings
- Essential performance issue not preventing good concurrency

```java
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) {
        like insert, but can allow concurrent lookups to same bucket
    }
}
```

**Parallelism vs. Concurrency**

Note: These terms are not yet standard, but the difference in perspective is essential

- Many programmers confuse them

Parallelism: Use more resources for a faster answer
Concurrency: Correctly and efficiently allow simultaneous access

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 3-4 lectures on parallelism, then 3-4 on concurrency

**An analogy**

CSE142 idea: Writing a program is like writing 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 not too many chefs or you spend all your time coordinating

Concurrency:

- 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**

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 (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 somewhere another thread reads

**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, with telephones
- **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
- Will show you how Java does it
  - Many primitives will be delayed until we study concurrency
  - We will not use Java threads much in project 3 for reasons lecture will explain, but it’s still worth seeing them first
- Steps to creating another thread:
  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
    - Not `run`, which would just be a normal method call
Parallelism idea

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

\[\begin{array}{cccc}
\text{ans0} & \text{ans1} & \text{ans2} & \text{ans3} \\
\end{array}\]

- Steps:
  - Create 4 thread objects, assigning their portion of the work
  - Call `start()` on each thread object to actually run it
  - Wait for threads to finish
  - Add together their 4 answers for the final result

First attempt at parallelism: wrong!

```java
class SumThread extends java.lang.Thread {
    int lo; // fields to know what to do
    int hi;
    int[] arr;
    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++)
            ans += arr[i];
    }
    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);
        for(int i=0; i < 4; i++) // combine results
            ans += ts[i].ans;
        return ans;
    }
}
```

Second attempt (still wrong)

```java
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 l, int h) { ... }
    public void run(){ ... }
}
```

```java
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);
    for(int i=0; i < 4; i++) // start not run
        ts[i].start();
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}
```

Third attempt (correct in spirit)

```java
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 l, int h) { ... }
    public void run(){ ... }
}
```

```java
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);
    for(int i=0; i < 4; i++) // do parallel computations
        ts[i].start();
    for(int i=0; i < 4; i++)
        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 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 one such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its `run` returns)
  - Else we would have a race condition on `ts[i].ans`
- 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) 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

**Now forget a lot of what we just did 😊**

Several reasons why this is a poor way to sum an array in parallel!

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

```java
int sum(int[] arr, int numThreads){
    // note: shows idea, but has integer-division bug
    int subLen = arr.length / numThreads;
    SumThread[] ts = new SumThread[numThreads];
    for(int i=0; i < numThreads; i++) {
        ts[i] = new SumThread(arr, i*subLen, (i+1)*subLen);
        ts[i].start();
    }
    for(int i=0; i < numThreads; i++) {
        ...   // numThreads == numProcessors is bad
        // if some are needed for other things
        int sum(int[] arr, int numThreads){
            } ...
```
Now forget a lot of what we just did 😊

3. Though unlikely for \texttt{sum}, in general different threads 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 than others
     * 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 doesn’t work

• Suppose we create 1 thread to process every 100 elements

\begin{verbatim}
int sum(int[] arr){
    int numThreads = arr.length / 100;
    SumThread[] ts = new SumThread[numThreads];
    ...
}
\end{verbatim}

• Then combining results will have \( \text{arr.length} / 100 \) additions to do – still linear in size of array

• In the extreme, suppose we create a thread to process every 1 element – then we’re back to where we started even though we said more threads was better

A better idea

This is straightforward to implement using divide-and-conquer

– Parallelism for the recursive calls
**Divide-and-conquer to the rescue!**

```java
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 l, int h) { ... }
    public void run(){
        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 paralleizes 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 $O(n/\text{numProcessors} + \log n)$
- In practice, creating all that inter-thread communication swamps the savings, so:
  - Use a sequential cutoff, typically around 500-1000
    - As in quicksort, eliminates almost all recursion, but here it is even more important
    - Don’t create two recursive threads; create one and do the other “yourself”
  - Cuts the number of threads created by another 2x

**Half the threads**

```java
// wasteful: don’t
SumThread left = ...  
SumThread right = ...
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```

```java
// better: do
SumThread left = ...
SumThread right = ...
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

- If a *language* had built-in support for fork-join parallelism, I would expect this hand-optimization to be unnecessary
- But the *library* we are using expects you to do it yourself
  - And the difference is surprisingly substantial
- Again, no difference in theory
**That library, finally**

- Even with all this care, Java’s threads are too “heavy-weight”
  - Constant factors, especially space overhead
  - Creating 20,000 Java threads just a bad idea 😊

- The ForkJoin Framework is designed to meet the needs of divide-and-conquer fork-join parallelism
  - Will be in Java 7 standard libraries, but available in Java 6 as a downloaded .jar file
  - Section will focus on pragmatics/logistics
  - Similar libraries available for other languages
    - C/C++: Cilk (inventors), Intel’s Thread Building Blocks
    - C#: Task Parallel Library
    - ...
  - How the library works is fascinating, but a bit beyond CSE332

---

**Different terms, same basic idea**

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a ForkJoinPool)
- Don’t subclass Thread Do subclass RecursiveTask<V>
- Don’t override run Do override compute
- Do not use an ans field Do return a V from compute
- Don’t call start Do call fork
- Don’t just call join Do call join which returns answer
- Don’t call run to hand-optimize Do call compute to hand-optimize

---

**Example: final version (missing imports)**

```java
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    protected Integer compute() { // return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for(int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        } else {
            SumArray left = new SumArray(arr,lo,(hi+lo)/2);
            SumArray right= new SumArray(arr,(hi+lo)/2,hi);
            left.fork();
            int rightAns = right.compute();
            int leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}
static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr){
    return fjPool.invoke(new SumArray(arr,0,arr.length));
}
```

---

**Getting good results in practice**

- Sequential threshold
  - Library documentation recommends doing approximately 100-5000 basic operations in each “piece” of your algorithm

- Library needs to “warm up”
  - May see slow results before the Java virtual machine re-optimizes the library internals
  - Put your computations in a loop to see the “long-term benefit”

- Wait until your computer has more processors 😊
  - Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important

- Beware memory-hierarchy issues
  - Won’t focus on this, but often crucial for parallel performance