CSE332: Data Abstractions

Lecture 17: Introduction to Multithreading and Fork-Join Parallelism

Ruth Anderson
Winter 2011

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

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

```
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);
  }
  return res[0]+res[1]+res[2]+res[3];
}
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

Ex: Multiple threads accessing a hash-table, but not getting in each others' 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

```
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)

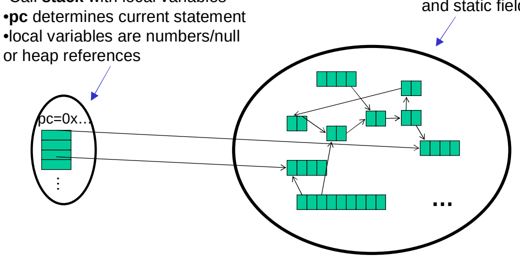
New story:

- 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

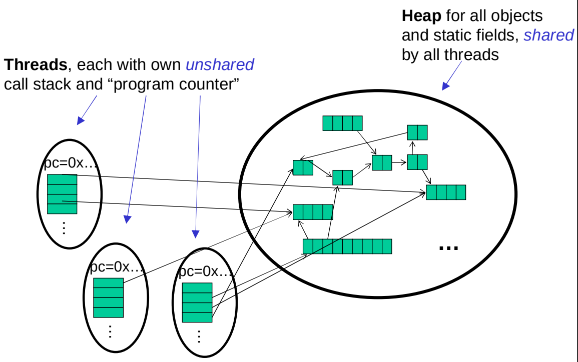
Old Story : one call stack, one pc

•Call **stack** with local variables
•**pc** determines current statement
•local variables are numbers/null or heap references

Heap for all objects and static fields



New Story: Shared memory with Threads



Other models

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

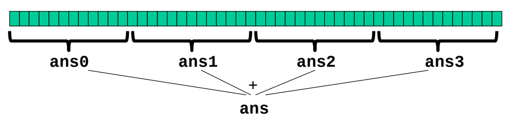
- Message-passing: Each thread has its own collection of objects. Communication is via explicit messages; language has primitives for sending and receiving them.
  - Cooks working in separate kitchens, emailing back and forth
- Dataflow: Programmers write programs in terms of a DAG and a node executes 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 **`C`** 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



- Steps:
  1. Create 4 new thread objects, assigning their portion of the work
  2. Call **`start()`** on each thread object to actually run it
  3. Somehow 'wait' for threads to finish
  4. Add together their 4 answers for the final result

Partial Code for first attempt (with Threads)

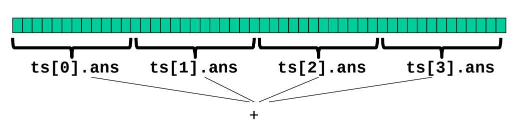
- Assume SumThread's run() simply loops through the given indices and adds the elements

```
int sum(int[] arr){
  int len = arr.length;
  int ans = 0;
  SumThread[] ts = new SumThread[4];
  for(int i=0; i < 4; i++){// 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;
}
```

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



`ans = ts[0].ans + ts[1].ans + ts[2].ans + ts[3].ans`

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

Complete Code (correct in spirit)

```
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) {
    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
      this.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;
   } }
```

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

Problems with our current approach

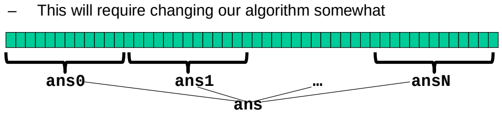
The above method would work, but we can do better for several reasons:

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 **`f`** to every array element, but maybe **`f`** is much slower for some data items than others; say, verifying primes will take much longer for big values than for small values
   - 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')

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

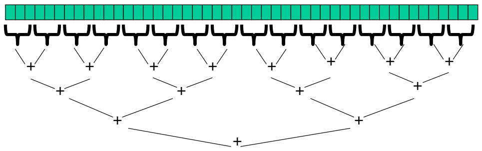
Naïve algorithm doesn't work

- Suppose we create 1 thread to process every 100 elements

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

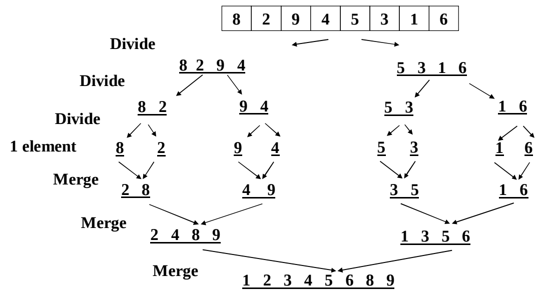
- Then combining results will have: **`numThreads = arr.length / 100`** additions to do – linear in size of array (before we only had 4 pieces $\Theta(1)$ to combine)
- In the extreme, suppose we create one thread per element – If we use a for loop to combine the results, we have N iterations
- In either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

A better idea for combining… look familiar?



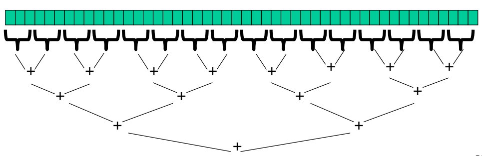
- Start with full problem at root
- Halve and make new thread until size is at some cutoff
- Combine answers in pairs as we return
- This will start small, and 'grow' threads to fit the problem
- This is straightforward to implement using divide-and-conquer

Remember Mergesort?



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 +




```
Thread: sum range [0,10)
   Thread: sum range [0,5)
      Thread: sum range [0,2)
         Thread: sum range [0,1) (return arr[0])
         Thread: sum range [1,2) (return arr[1])
         add results from two helper threads
      Thread: sum range [2,5)
         Thread: sum range [2,3) (return arr[2])
         Thread: sum range [3,5)
            Thread: sum range [3,4) (return arr[3])
            Thread: sum range [4,5) (return arr[4])
            add results from two helper threads
         add results from two helper threads
      add results from two helper threads
   Thread: sum range [5,10)
      Thread: sum range [5,7)
         Thread: sum range [5,6) (return arr[5])
         Thread: sum range [6,7) (return arr[6])
         add results from two helper threads
      Thread: sum range [7,10)
         Thread: sum range [7,8) (return arr[7])
         Thread: sum range [8,10)
            Thread: sum range [8,9) (return arr[8])
            Thread: sum range [9,10) (return arr[9])
            add results from two helper threads
         add results from two helper threads
   add results from two helper threads
```

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

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

Half the threads!

order of last 4 lines
Is critical – why?

```
// wasteful: don't
SumThread left  = …
SumThread right = …

left.start();
right.start();


left.join();
right.join();
ans=left.ans+right.ans;
```

```
// better: do!!
SumThread left  = …
SumThread right = …

left.start();
right.run();


left.join();
// no right.join needed
ans=left.ans+right.ans;
```

*Note: **run** is a normal function call! execution won't continue until we are done with **run***

- 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
    - …
  - Library's implementation is a fascinating but advanced topic

Different terms, same basic idea

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a **`ForkJoinPool`**)

Java Threads	ForkJoin Framework
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 in ForkJoin Framework (missing imports)

```
class SumArray extends RecursiveTask<Integer> {
  int lo; int hi; int[] arr;//fields to know what to do
  SumArray(int[] a, int l, int h) { … }
  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));
}
```


For comparison - Java Threads Version

```
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 { // create 2 threads, each will do ½ the work
      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;
    }
  }
}
class C {
 static int sum(int[] arr){
   SumThread t = new SumThread(arr,0,arr.length);
   t.run(); // only creates one thread
   return t.ans;
 }
}
```

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
  - When evaluating speed, put your computations in a loop to see the "long-term benefit" after these optimizations have occurred
- 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