New Story

Threads, each with its own unshared:
- Call Stack
- Program Counter
- Local Variables (primitives and references to Heap objects)

Heap Containing Objects and Static Fields

```
x = Object
x = new Object
```
Back to Summing an Array

• Goal: Find the sum of an array
• Idea: 4 threads each find the sum of one quarter of the array
• Process:
  • Create 4 thread objects, each given a portion of the work
  • Call start() on each thread object to run it in parallel
  • Wait for threads to finish using join()
  • Add together their 4 answers for the final result
Parallel Sum

• **Base Case:**
  • If the list’s length is smaller than the Sequential Cutoff, find the sum sequentially

• **Divide:**
  • Split the list into two “sublists” of (roughly) equal length, create a thread to sum each sublist.

• **Conquer:**
  • Call `start()` for each thread

• **Combine:**
  • Sum together the answers from each thread

ans=15

ans=14

ans=29
Divide and Conquer with Threads

```java
class SumThread extends java.lang.Thread {
    public void run() { // override
        if (hi - lo < SEQUENTIAL_CUTOFF) // “base case”
            for (int i = lo; i < hi; i++)
                ans += arr[i];
        else {
            SumThread left = new SumThread(arr, lo, (hi + lo) / 2); // divide
            SumThread right = new SumThread(arr, (hi + lo) / 2, hi); // divide
            left.start(); // conquer
            right.start(); // conquer
            left.join(); // don’t move this up a line - why?
            right.join();
            ans = left.ans + right.ans; // combine
        }
    }
}

int sum(int[] arr) { // just make one thread!
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```
ForkJoin Framework

- This strategy is common enough that Java (and C++, and C#, and...) provides a library to do it for you!

<table>
<thead>
<tr>
<th>What you would do in Threads</th>
<th>What to instead in ForkJoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclass Thread</td>
<td>Subclass RecursiveTask&lt;V&gt;</td>
</tr>
<tr>
<td>Override run</td>
<td>Override compute</td>
</tr>
<tr>
<td>Store the answer in a field</td>
<td>Return a V from compute</td>
</tr>
<tr>
<td>Call start</td>
<td>Call fork</td>
</tr>
<tr>
<td><strong>join</strong> synchronizes only</td>
<td><strong>join</strong> synchronizes and returns the answer</td>
</tr>
<tr>
<td>Call run to execute sequentially</td>
<td>Call compute to execute sequentially</td>
</tr>
<tr>
<td>Have a topmost thread and call run</td>
<td>Create a pool and call <strong>invoke</strong></td>
</tr>
</tbody>
</table>
Divide and Conquer with ForkJoin

class SumTask extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    SumTask(int[] a, int l, int h) { ... }
    protected Integer compute(){// return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) { // base case
            int ans = 0; // local var, not a field
            for(int i=lo; i < hi; i++) {
                ans += arr[i];
            }
            return ans;
        } else {
            SumTask left = new SumTask(arr,lo,(hi+lo)/2); // divide
            SumTask right= new SumTask(arr,(hi+lo)/2,hi); // divide
            left.fork(); // fork a thread and calls compute (conquer)
            int rightAns = right.compute(); //call compute directly (conquer)
            int leftAns = left.join(); // get result from left
            return leftAns + rightAns; // combine
        }
    }
}
Divide and Conquer with ForkJoin (continued)

```java
static final ForkJoinPool POOL = new ForkJoinPool();
int sum(int[] arr){
    SumTask task = new SumTask(arr, 0, arr.length)
    return POOL.invoke(task); // invoke returns the value compute returns
}
```
Find Max with ForkJoin

class MaxTask extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    SumTask(int[] a, int l, int h) { ... }
    protected Integer compute() { // return answer
        if(hi – lo < SEQUENTIAL_CUTOFF) { // base case
            int ans = Integer.MIN_VALUE; // local var, not a field
            for(int i=lo; i < hi; i++) {
                ans = Math.max(ans, arr[i]);
            }
            return ans;
        } else {
            MaxTask left = new MaxTask(arr,lo,(hi+lo)/2); // divide
            MaxTask right= new MaxTask(arr,(hi+lo)/2,hi); // divide
            left.fork(); // fork a thread and calls compute (conquer)
            int rightAns = right.compute(); //call compute directly (conquer)
            int leftAns = left.join(); // get result from left
            return Math.max(rightAns, leftAns); // combine
        }
    }
}
Other Problems that can be solved similarly

• **Element Search**
  • Is the value 17 in the array?

• **Counting items with a certain property**
  • How many elements of the array are divisible by 5?

• **Checking if the array is sorted**

• **Find the smallest rectangle that covers all points in the array**

• **Find the first thing that satisfies a property**
  • What is the leftmost item that is divisible by 20?
**Reductions**

- All examples of a category of computation called a reduction
  - We “reduce” all elements in an array to a single item
  - Requires operation done among elements is associative
    - \((x + y) + z = x + (y + z)\)
  - The “single item” can itself be complex
    - E.g. create a histogram of results from an array of trials
Map

• Perform an operation on each item in an array to create a new array of the same size

• Examples:
  • **Vector addition:**
    • \(\text{sum}[i] = \text{arr1}[i] + \text{arr2}[i]\)
  • **Function application:**
    • \(\text{out}[i] = f(\text{arr}[i])\);
Map with ForkJoin

class AddTask extends RecursiveAction {
    int lo; int hi; int[] arr; // fields to know what to do
    AddTask(int[] a, int[] b, int[] sum, int l, int h) { ... }
    protected void compute(){ // return answer
        if(hi – lo < SEQUENTIAL_CUTOFF) { // base case
            for(int i=lo; i < hi; i++) {
                sum[i] = a[i] + b[i];
            }
        } else {
            AddTask left = new AddTask(a, b, sum, lo, (hi+lo)/2); // divide
            AddTask right = new AddTask(a, b, sum, (hi+lo)/2, hi); // divide
            left.fork(); // fork a thread and calls compute (conquer)
            right.compute(); // call compute directly (conquer)
            left.join(); // get result from left
            return; // combine
        }
    }
}
Map with ForkJoin (continued)

```java
static final ForkJoinPool POOL = new ForkJoinPool();
Int[] add(int[] a, int[] b){
    ans = new int[a.length];
    AddTask task = new AddTask(a, b, ans, 0, a.length)
    POOL.invoke(task);
    return ans;
}
```
Maps and Reductions

- "Workhorse" constructs in parallel programming
- Many problems can be written in terms of maps and reductions
- With practice, writing them will become second nature
  - Like how over time for loops and if statements have gotten easier
Parallel Algorithm Analysis

• How to define efficiency
  • Want asymptotic bounds
  • Want to analyze the algorithm without regard to a specific number of processors
Work and Span

• Let $T_P(n)$ be the running time if there are $P$ processors available

• Two key measures of run time:
  • Work: How long it would take 1 processor, so $T_1(n)$
    • Just suppose all forks are done sequentially
    • Cumulative work all processors must complete
    • For array sum: $\Theta(n)$
  • Span: How long it would take an infinite number of processors, so $T_\infty(n)$
    • Theoretical ideal for parallelization
    • Longest “dependence chain” in the algorithm
    • Also called “critical path length” or “computation depth”
    • For array sum: $\Theta(\log n)$
Directed Acyclic Graph (DAG)

- A directed graph that has no cycles
- Often used to depict dependencies
  - E.g. software dependencies, Java inheritance, dependencies among threads!
ForkJoin DAG

- Fork and Join each create a new node
  - Fork branches into two threads
    - Those two threads “depended on” their source thread to be created
  - Join combines to threads
    - The thread doing the combining “depends on” the other threads to finish

\[
\log n \quad \log n \quad \log n
\]
More Vocab

Speed Up:
- How much faster (than one processor) do we get for more processors
  \[ \frac{T_1(n)}{T_P(n)} \]

Perfect linear Speedup
- \[ \frac{T_1}{T_P} = P \]
- Hard to get in practice
- “Holy Grail” or parallelizing

Parallelism
- Maximum possible speedup
  \[ \frac{T_1}{T_\infty} \]
- At some point more processors won’t be more helpful, when that point is depends on the span

Writing parallel algorithms is about increasing span without substantially increasing work
Asymptotically Optimal $T_P$

• We know how to compute $T_1$ and $T_\infty$, but what about $T_P$?
  • $T_P$ cannot be better than $\frac{T_1}{P}$
  • $T_P$ cannot be better than $T_\infty$

• An asymptotically optimal execution would be
  • $T_P(n) \in O\left(\frac{T_1(n)}{P} + T_\infty(n)\right)$
  • $T_1(n)/P$ dominates for small $P$, $T_\infty(n)$ dominates for large $P$

• ForkJoin Frameworks gives an expected time guarantee of asymptotically optimal!
Division of Responsibility

- **Our job as ForkJoin Users:**
  - Pick a good algorithm, write a program
  - When run, program creates a DAG of things to do
  - Make all the nodes a small-ish and approximately equal amount of work

- **ForkJoin Framework Developer’s job:**
  - Assign work to available processors to avoid idling
    - Abstract away scheduling issues for the user
  - Keep constant factors low
  - Give the expected-time optimal guarantee
And now for some bad news...

• In practice it’s common for your program to have:
  • Parts that parallelize well
    • Maps/reduces over arrays and other data structures
  • And parts that don’t parallelize at all
    • Reading a linked list, getting input, or computations where each step needs the results of previous step

• These unparallelized parts can turn out to be a big bottleneck
Amdahl’s Law (mostly bad news)

• Suppose $T_1 = 1$
  • Work for the entire program is 1

• Let $S$ be the proportion of the program that cannot be parallelized
  • $T_1 = S + (1 - S) = 1$

• Suppose we get perfect linear speedup on the parallel portion
  • $T_P = S + \frac{1-S}{p}$

• For the entire program, the speed is:
  • $\frac{T_1}{T_P} = \frac{1}{\frac{S}{S+\frac{1-S}{p}}}$

• And so the parallelism (infinite processors) is:
  • $\frac{T_1}{T_\infty} = \frac{1}{S}$
Ahmdal’s Law Example

• Suppose 2/3 of your program is parallelizable, but 1/3 is not.
  • \( S = \frac{2}{3} \)
  • \( T_1 = \frac{2}{3} + \frac{1}{3} = 1 \)
  • \( T_P = S + \frac{1-S}{P} \)

• So if \( T_1 \) is 100 seconds:
  • \( T_P = 33 + \frac{67}{P} \)
  • \( T_3 = 33 + \frac{67}{3} = 33 + 22 = 55 \)
Conclusion

• Even with many processors the sequential part of your program becomes a bottleneck

• Parallelizable code requires skill and insight from the developer to recognize where parallelism is possible, and how to do it well.