Where are we

Done:
• How to use fork, and join to write a parallel algorithm
• Why using divide-and-conquer with lots of small tasks is best
  – Combines results in parallel
• Some Java and ForkJoin Framework specifics
  – More pragmatics in section and posted notes

Now:
• More examples of simple parallel programs
• Arrays & balanced trees support parallelism, linked lists don’t
• Asymptotic analysis for fork-join parallelism
• Amdahl’s Law

What else looks like this?
• Saw summing an array went from $O(n)$ sequential to $O(\log n)$ parallel (assuming a lot of processors and very large $n$!)
  – An exponential speed-up in theory

• Anything that can use results from two halves and merge them in $O(1)$ time has the same property…

Examples
• Maximum or minimum element
• Is there an element satisfying some property (e.g., is there a 17)?
• Left-most element satisfying some property (e.g., first 17)
  – What should the recursive tasks return?
  – How should we merge the results?
• In project 3: corners of a rectangle containing all points
• Counts, for example, number of strings that start with a vowel
  – This is just summing with a different base case
  – Many problems are!
Reducions

- Computations of this form are called reductions (or reduces?)
- They take a set of data items and produce a single result
- Note: Recursive results don’t have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: Histogram of test results
  - Example on project 3: Kind of like a 2-D histogram
- While many can be parallelized due to nice properties like associativity of addition, some things are inherently sequential
  - How we process arr[i] may depend entirely on the result of processing arr[i-1]

Even easier: Data Parallel (Maps)

- While reductions are a simple pattern of parallel programming, maps are even simpler
  - Operate on set of elements to produce a new set of elements (no combining results)
  - For arrays, this is so trivial some hardware has direct support
- Canonical example: Vector addition

```java
int[] vector_add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    result = new int[arr1.length];
    len = arr.length;
    FORALL(i=0; i < arr.length; i++) {
        result[i] = arr1[i] + arr2[i];
    }
    return result;
}
```

Maps in ForkJoin Framework

```java
class VecAdd extends RecursiveAction {
    int lo; int hi; int[] res; int[] arr1; int[] arr2;
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2) { ... }
    protected void compute() {
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            for (int i=lo; i < hi; i++)
                res[i] = arr1[i] + arr2[i];
        } else {
            int mid = (hi+lo)/2;
            VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);
            VecAdd right = new VecAdd(mid, hi, res, arr1, arr2);
            left.fork();
            right.compute();
        }
    }
    static final ForkJoinPool fjPool = new ForkJoinPool();
    int[] add(int[] arr1, int[] arr2) {
        assert (arr1.length == arr2.length);
        int[] ans = new int[arr1.length];
        fjPool.invoke(new VecAdd(0, arr.length, ans, arr1, arr2);
        return ans;
    }
}
```

Digression on maps and reduces

- You may have heard of Google’s “map/reduce”
  - Or the open-source version Hadoop
- Idea: Perform maps and reduces on data using many machines
  - The system takes care of distributing the data and managing fault tolerance
  - You just write code to map one element and reduce elements to a combined result
- Separates how to do recursive divide-and-conquer from what computation to perform
  - Old idea in higher-order programming (see 341) transferred to large-scale distributed computing
  - Complementary approach to declarative queries (see 344)
**Trees**

- Our basic patterns so far – maps and reduces – work just fine on balanced trees
  - Divide-and-conquer each child rather than array subranges
  - Correct for unbalanced trees, but won’t get much speed-up
- Example: minimum element in an unsorted but balanced binary tree in \( O(\log n) \) time given enough processors
- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or I guess you could approximate it with, e.g., AVL height

**Linked lists**

- Can you parallelize maps or reduces over linked lists?
  - Example: Increment all elements of a linked list
  - Example: Sum all elements of a linked list

**Analyzing algorithms**

- Parallel algorithms still need to be:
  - Correct
  - Efficient
- For our algorithms so far, correctness is “obvious” so we’ll focus on efficiency
  - Still want asymptotic bounds
  - Want to analyze the algorithm without regard to a specific number of processors
  - The key “magic” of the ForkJoin Framework is getting expected run-time performance asymptotically optimal for the available number of processors
    - Lets us just analyze our algorithms given this “guarantee”

**Work and Span**

Let \( T_p \) be the running time if there are \( P \) processors available

Two key measures of run-time for a fork-join computation

- **Work**: How long it would take 1 processor = \( T_1 \)
  - Just “sequentialize” all the recursive forking
- **Span**: How long it would take infinity processors = \( T_\infty \)
  - The longest dependence-chain
  - Example: \( O(\log n) \) for summing an array since > \( n/2 \) processors is no additional help
  - Also called “critical path length” or “computational depth”
The DAG

- A program execution using fork and join can be seen as a DAG
  - I told you graphs were useful! 😊
- Nodes: Pieces of work
- Edges: Source must finish before destination starts
  - A fork "ends a node" and makes two outgoing edges
    - New thread
    - Continuation of current thread
  - A join "ends a node" and makes a node with two incoming edges
    - Node just ended
    - Last node of thread joined on

More interesting DAGs?

- The DAGs are not always this simple
- Example:
  - Suppose combining two results might be expensive enough that we want to parallelize each one
  - Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

Connecting to performance

- Recall: $T_p = \text{running time if there are } P \text{ processors available}$
- Work = $T_1 = \text{sum of run-time of all nodes in the DAG}$
  - That lonely processor has to do all the work
  - Any topological sort is a legal execution
- Span = $T_\infty = \text{sum of run-time of all nodes on the most-expensive path in the DAG}$
  - Note: costs are on the nodes not the edges
  - Our infinite army can do everything that is ready to be done, but still has to wait for earlier results
Definitions

A couple more terms:

• **Speed-up** on \( P \) processors: \( T_1 / T_P \)

• If speed-up is \( P \) as we vary \( P \), we call it **perfect linear speed-up**
  – Perfect linear speed-up means doubling \( P \) halves running time
  – Usually our goal; hard to get in practice

• **Parallelism** is the maximum possible speed-up: \( T_1 / T_\infty \)
  – At some point, adding processors won’t help
  – What that point is depends on the span

Division of responsibility

• Our job as ForkJoin Framework users:
  – Pick a good algorithm
  – Write a program. When run it creates a DAG of things to do
  – Make all the nodes a small-ish and approximately equal amount of work

• The framework-writer’s job (won’t study how to do it):
  – Assign work to available processors to avoid idling
  – Keep constant factors low
  – Give an **expected-time guarantee** (like quicksort) assuming framework-user did his/her job

\[
T_P \leq (T_1 / P) + O(T_\infty)
\]

What that means (mostly good news)

The fork-join framework guarantee

\[
T_P \leq (T_1 / P) + O(T_\infty)
\]

– No implementation of your algorithm can beat \( O(T_\infty) \) by more than a constant factor

– No implementation of your algorithm on \( P \) processors can beat \( (T_1 / P) \) (ignoring memory-hierarchy issues)

– So the framework on average gets within a constant factor of the best you can do, assuming the user did his/her job

So: You can focus on your algorithm, data structures, and cut-offs rather than number of processors and scheduling

• Analyze running time given \( T_1, T_\infty \) and \( P \)

Examples

\[
T_P \leq (T_1 / P) + O(T_\infty)
\]

• In the algorithms seen so far (e.g., sum an array):
  – \( T_1 = O(n) \)
  – \( T_\infty = O(\log n) \)
  – So expect (ignoring overheads): \( T_P \leq O(n/P + \log n) \)

• Suppose instead:
  – \( T_1 = O(n^2) \)
  – \( T_\infty = O(n) \)
  – So expect (ignoring overheads): \( T_P \leq O(n^2/P + n) \)
Amdahl’s Law (mostly bad news)

- So far: talked about a parallel program in terms of work and span
- In practice, it's common that there are parts of your program that parallelize well…
  - Such as maps/reduces over arrays and trees
  …and parts that don’t parallelize at all
  - Such as reading a linked list, getting input, or just doing computations where each needs the previous step
    - “Nine women can’t make a baby in one month”

Why such bad news

\[ \frac{T_1}{T_p} = \frac{1}{S + (1-S)/P} \]

\[ \frac{T_1}{T_\infty} = \frac{1}{S} \]

- Suppose 33% of a program is sequential
  - Then a billion processors won’t give a speedup over 3
- Suppose you miss the good old days (1980-2005) where 12ish years was long enough to get 100x speedup
  - Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
  - For 256 processors to get at least 100x speedup, we need
    \[ 100 \leq \frac{1}{S + (1-S)/256} \]
    Which means \( S \leq 0.0061 \) (i.e., 99.4% perfectly parallelizable)

Plots you gotta see

1. Assume 256 processors
   - x-axis: sequential portion \( S \), ranging from .01 to .25
   - y-axis: speedup \( T_1 / T_p \) (will go down as \( S \) increases)

2. Assume \( S = .01 \) or .1 or .25 (three separate lines)
   - x-axis: number of processors \( P \), ranging from 2 to 32
   - y-axis: speedup \( T_1 / T_p \) (will go up as \( P \) increases)

Too important for me just to show you: Homework problem!

- Chance to use a spreadsheet or other graphing program
- Compare against your intuition
  - A picture is worth 1000 words, especially if you made it
All is not lost

Amdahl’s Law is a bummer!
  – But it doesn’t mean additional processors are worthless

• We can find new parallel algorithms
  – Some things that seem clearly sequential turn out to be parallelizable

• We can change the problem we’re solving or do new things
  – Example: Video games use tons of parallel processors
    • They are not rendering 10-year-old graphics faster
    • They are rendering more beautiful monsters

Moore and Amdahl

• Moore’s “Law” is an observation about the progress of the semiconductor industry
  – Transistor density doubles roughly every 18 months

• Amdahl’s Law is a mathematical theorem
  – Implies diminishing returns of adding more processors

• Both are incredibly important in designing computer systems