Announcements

- Homework 5 – due NOW, at the BEGINNING of lecture
- Homework 6 – due Friday Feb 25th at the BEGINNING of lecture
- Project 3 – the last programming project!
  - Partner Selection - Tues, Feb 22, 11pm
  - Version 1 & 2 - Tues March 1, 2011 11PM - (10% of overall grade)
  - ALL Code - Tues March 8, 2011 11PM - (65% of overall grade):
  - Writeup - Thursday March 10, 2011, 11PM - (25% of overall grade)

Outline

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 (e.g., installation) in separate 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

We looked at summing an array

- 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
  - Not bad; that’s 4 billion versus 32 (without constants, and in theory)

Examples

- Parallelization (for some algorithms)
  - Describe how to compute result at the ‘cut-off’
  - Describe how to merge results
- How would we do the following (assuming data is given as an array)?
  1. Maximum or minimum element
  2. Is there an element satisfying some property (e.g., is there a 17)?
  3. Left-most element satisfying some property (e.g., first 17)
  4. Smallest rectangle encompassing a number of points (proj3)
  5. Counts; for example, number of strings that start with a vowel
  6. Are these elements in sorted order?
Reductions

• This class of computations are called reductions
  – We ‘reduce’ a large array of data to a single item
• Note: Recursive results don’t have to be single numbers or strings. They can be arrays or objects with multiple fields.
  – Example: create a Histogram of test results from a much larger array of actual test results
• 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); generally input and output are of the same length
  – Eg. Multiply each element of an array by 2.

Maps in ForkJoin Framework

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();
      left.join(); //this was missing on orig slide
    }
  }
  static final ForkJoinPool fJPool = new ForkJoinPool();
  int[] add(int[] ar1, int[] ar2){
    assert (ar1.length == ar2.length);
    int[] ans = new int[ar1.length];
    for(int i=0; i < ar1.length; i++) {
      result[i] = arr1[i] + arr2[i];
    }
    return result;
  }
}

Map vs reduce in ForkJoin framework

• In our examples:
  • Reduce:
    – Parallel-sum extended RecursiveTask
    – Result was returned from compute()
  • Map:
    – Class extended was RecursiveAction
    – Nothing returned from compute()
    – In the above code, the ‘answer’ array was passed in as a parameter
  • Doesn’t have to be this way
    – Map can use RecursiveTask to, say, return an array
    – Reduce could use RecursiveAction; depending on what you’re passing back via RecursiveTask, could store it as a class variable and access it via ‘left’ or ‘right’ when done

Digression on maps and reduces

• You may have heard of Google’s ‘map/reduce’
  – Or the open-source version Hadoop
• Idea: Want to run algorithm on enormous amount of data; say, sort a petabyte (10^16 gigabytes) of data
  – 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 CSE 341) transferred to large-scale distributed computing

Works on Trees as well as Arrays

• Our basic patterns so far – maps and reduces – work just fine on balanced trees
  – Divide-and-conquer each child rather than array sub-ranges
  – 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 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
- Once again, data structures matter!
- For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster \( O(\log n) \) vs. \( O(n) \)
  - Trees have the same flexibility as lists compared to arrays (in terms of say inserting an item in the middle of the list)

Analyzing algorithms

- Like all algorithms, parallel algorithms should be:
  - Correct
  - Efficient
- For our algorithms so far, correctness is “obvious” so we’ll focus on efficiency:
  - We 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
  - This 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
- Type/power of processors doesn’t matter: \( T_P \) used asymptotically, and to compare improvement by adding a few processors

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 hypothetical ideal for parallelization
  - This is the longest “dependence chain” in the computation

The DAG

- A program execution using \( \text{fork} \) and \( \text{join} \) can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source node must finish before destination node starts
- A \( \text{fork} \) “ends a node” and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A \( \text{join} \) “ends a node” and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

Our simple examples

- \( \text{fork} \) and \( \text{join} \) are very flexible, but our divide-and-conquer maps and reduces so far use them in a very basic way:
  - A tree on top of an upside-down tree

In this context, the span (\( T_\infty \)) is:
- The longest dependence chain; longest branch in parallel tree
- Example: \( O(\log n) \) for summing an array; we halve the data down to our cut-off, then add back together; \( O(\log n) \) steps, \( O(1) \) time for each
- Also called “critical path length” or “computational depth”
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: $T_p$ = running time if there are $P$ processors available
- Work = $T_1$ = sum of run-time of all nodes in the DAG
  - One processor has to do all the work
  - Any topological sort is a legal execution
- Span = $T_\infty$ = 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

What that means (mostly good news)

The fork-join framework guarantee:

$$T_p = O(T_1 / P + T_\infty)$$

- No implementation of your algorithm can beat $O(T_1 / P)$ by more than a constant factor
- No implementation of your algorithm on $P$ processors can beat $O(T_1 / P)$ (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

$$T_p = O(T_1 / P + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect (ignoring overheads): $T_p = O(nP + \log n)$

- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads): $T_p = O(n^2P + n)$
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 your program has:
  a) parts that parallelize well:
    - Such as maps/reduces over arrays and trees
  b) ...and parts that don't parallelize at all:
    - Such as reading a linked list, getting input, or just doing 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)

Let the work (time to run on 1 processor) be 1 unit time.

Let $S$ be the portion of the execution that can’t be parallelized (i.e. must be run sequentially)

Then:
$$T_1 = S + (1-S) = 1$$

Suppose we get perfect linear speedup on the parallel portion

Then:
$$T_P = S + (1-S)/P$$

So the overall speedup with $P$ processors is (Amdahl's Law):

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

And the parallelism (infinite processors) is:

$$T_1 / T_\infty = 1 / S$$

Amdahl's Law Example

Suppose:
$$T_1 = S + (1-S) = 1$$
(aka total program execution time)
$$T_1 = 1/3 + 2/3 = 1$$

$$T_1 = 33\text{ sec} + 67\text{ sec} = 100\text{ sec}$$

Time on $P$ processors: $T_P = S + (1-S)/P$

So:
$$T_P = 33\text{ sec} + (67\text{ sec})/P$$
$$T_P = 33\text{ sec} + (67\text{ sec})/3 =$$

Why such bad news?

- Suppose 33% of a program is sequential
  - Then a billion processors won’t give a speedup over 3!!!
- No matter how many processors you use, your speedup is bounded by the sequential portion of the program.

The future and Amdahl's Law

Speedup: $T_1 / T_P = 1 / (S + (1-S)/P)$

Max Parallelism: $T_1 / T_\infty = 1 / S$

- 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
  - What portion of the program must be parallelizable to get 100x speedup?

For 256 processors to get at least 100x speedup, we need
$$100 \leq 1 / (S + (1-S)/256)$$
Which means $S \leq 0.0061$ (i.e., 99.4% must be parallelizable)
**Plots you have to 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)

I encourage you to try this out!
- 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 entirely sequential turn out to be parallelizable
  - Eg. How can we parallelize the following?
    - Take an array of numbers, return the ‘running sum’ array:

<table>
<thead>
<tr>
<th>Input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>14</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>6</td>
<td>10</td>
<td>26</td>
<td>36</td>
<td>52</td>
<td>66</td>
<td>68</td>
<td>76</td>
</tr>
</tbody>
</table>

- At a glance, not sure; we’ll explore this shortly

- We can also 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 richer environments and more beautiful (terrible?) 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