CSE373: Data Structures & Algorithms
Lecture 26: Parallel Reductions, Maps, and Algorithm Analysis

Aaron Bauer
Winter 2014
Outline

Done:
• How to write a parallel algorithm with fork and join
• Why using divide-and-conquer with lots of small tasks is best
  – Combines results in parallel
  – (Assuming library can handle “lots of small threads”)

Now:
• More examples of simple parallel programs that fit the “map” or “reduce” patterns
• Teaser: Beyond maps and reductions
• Asymptotic analysis for fork-join parallelism
• Amdahl’s Law
• Final exam and victory lap
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 $n$!)
  – Exponential speed-up in theory ($n / \log n$ grows exponentially)

• 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?

• Corners of a rectangle containing all points (a “bounding box”)

• Counts, for example, number of strings that start with a vowel
  – This is just summing with a different base case
  – Many problems are!
Reductions

• Computations of this form are called reductions (or reduces?)

• Produce single answer from collection via an associative operator
  – Associative: $a + (b+c) = (a+b) + c$
  – Examples: max, count, leftmost, rightmost, sum, product, …
  – Non-examples: median, subtraction, exponentiation

• But some things are inherently sequential
  – How we process arr[i] may depend entirely on the result of processing arr[i-1]
**Even easier: Maps (Data Parallelism)**

- A **map** operates on each element of a collection independently to create a new collection of the same size
  - 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];
    FORALL(i=0; i < arr1.length; i++) {
        result[i] = arr1[i] + arr2[i];
    }
    return result;
}
```
In Java

class VecAdd extends java.lang.Thread {
    int lo; int hi; int[] res; int[] arr1; int[] arr2;
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2) {
        ... }
    protected void run() {
        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.start();
            right.run();
            left.join();
        }
    }
}

int[] add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    (new VecAdd(0, arr.length, ans, arr1, arr2).run());
    return ans;
}
Maps and reductions

Maps and reductions: the “workhorses” of parallel programming

– By far the two most important and common patterns

– Learn to recognize when an algorithm can be written in terms of maps and reductions

– Use maps and reductions to describe (parallel) algorithms

– Programming them becomes “trivial” with a little practice
  • Exactly like sequential for-loops seem second-nature
Beyond maps and reductions

- Some problems are “inherently sequential”
  “Six ovens can’t bake a pie in 10 minutes instead of an hour”

- But not all parallelizable problems are maps and reductions

- If had one more lecture, would show “parallel prefix”, a clever algorithm to parallelize the problem that this sequential code solves

<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>

```java
int[] prefix_sum(int[] input){
    int[] output = new int[input.length];
    output[0] = input[0];
    for(int i=1; i < input.length; i++)
        output[i] = output[i-1]+input[i];
    return output;
}
```
Digression: MapReduce on clusters

• You may have heard of Google’s “map/reduce”
  – Or the open-source version Hadoop

• Idea: Perform maps/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
  – Separating concerns is good software engineering
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
  – Want asymptotic bounds
  – Want to analyze the algorithm without regard to a specific number of processors
  – Here: Identify the “best we can do” if the underlying thread-scheduler does its part
Work and Span

Let $T_p$ be the running time if there are $P$ processors available

Two key measures of run-time:

- **Work**: How long it would take 1 processor = $T_1$
  - Just “sequentialize” the recursive forking

- **Span**: How long it would take infinite processors = $T_∞$
  - The longest dependence-chain
  - Example: $O(\log n)$ for summing an array
    - Notice having $> n/2$ processors is no additional help
Our simple examples

- Picture showing all the “stuff that happens” during a reduction or a map: it’s a (conceptual!) DAG
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 does everything
  - Any topological sort is a legal execution
  - $O(n)$ for maps and reductions

- 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
  - $O(\log n)$ for simple maps and reductions
Speed-up

Parallel algorithms is about decreasing span without increasing work too much

- Speed-up on \( P \) processors: \( \frac{T_1}{T_P} \)
- Parallelism is the maximum possible speed-up: \( \frac{T_1}{T_\infty} \)
  - At some point, adding processors won’t help
  - What that point is depends on the span

- In practice we have \( P \) processors. How well can we do?
  - We cannot do better than \( O(T_\infty) \) (“must obey the span”)
  - We cannot do better than \( O(T_1 / P) \) (“must do all the work”)
  - Not shown: With a “good thread scheduler”, can do this well (within a constant factor of optimal!)
Examples

\[ T_P = O(\max((T_1 / P) , T_\infty)) \]

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

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

- So far: analyze parallel programs in terms of work and span

- In practice, typically have parts of programs that parallelize well…
  - Such as maps/reductions over arrays

  …and parts that don’t parallelize at all

  - Such as reading a linked list, getting input, doing computations where each needs the previous step, etc.
**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

Then:

\[
T_1 = S + (1-S) = 1
\]

Suppose *parallel portion parallelizes perfectly (generous assumption)*

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:

\[
T_1 / T_\infty = 1 / S
\]
Why such bad news

\[ \frac{T_1}{T_P} = \frac{1}{S + \frac{(1-S)}{P}} \quad \text{and} \quad \frac{T_1}{T_\infty} = \frac{1}{S} \]

- Suppose 33% of a program’s execution 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
    \[ 100 \leq \frac{1}{S + \frac{(1-S)}{256}} \]
    Which means \( S \leq 0.0061 \) (i.e., 99.4% perfectly parallelizable)
All is not lost

Amdahl’s Law is a bummer!
  – Unparallelized parts become a bottleneck very quickly
  – But it doesn’t mean additional processors are worthless

• We can find new parallel algorithms
  – Some things that seem sequential are actually parallelizable

• We can change the problem or do new things
  – Example: computer graphics use tons of parallel processors
    • Graphics Processing Units (GPUs) are massively parallel
    • They are not rendering 10-year-old graphics faster
    • They are rendering more detailed/sophisticated images
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
  – Diminishing returns of adding more processors

• Both are incredibly important in designing computer systems
Final Exam

As also indicated on the web page:

• Next **Tuesday**, 2:30-4:20

• Cumulative but topics post-midterm-2 about 1/2 the questions

• See information on course web-page

• Not unlike the midterms in style, structure, etc.

• Tough-but-fair exams are the most equitable approach
  – And/but 110 minutes will make a big difference
Topics since midterm 2

• Preserving the abstraction (copy in, copy out, immutability)
• Sorting
  – Simple comparison sorts (selection, insertion)
  – Fancy comparison sorts (heap, merge, quick)
  – Comparison sort lower bound
    • Proof details not on exam
  – Beyond comparison sorting (bucket, radix)
  – External sorting
• Memory hierarchy (registers – cache – main memory – disk)
• Locality (temporal, spatial)
• Parallelism
  – fork/join parallelism, analysis, Amdahl’s Law
Victory Lap

A victory lap is an extra trip around the track
  – By the exhausted victors
    (that’s us) 😊

Review course goals
  – Slides from Lecture 1
  – What makes CSE373 special
Thank you!

Big thank-you to your TAs
  – Amazingly cohesive “big team”
  – Prompt grading and question-answering
  – Optional TA sessions weren’t optional for them!

Fall 2013 CSE373: Data Structures & Algorithms
Thank you!

And huge thank you to all of you
- Great attitude
- Good class attendance and questions for a large CSE373
- Occasionally laughed at stuff 😊
Now three slides, completely unedited, from Lecture 1
  – Hopefully they make more sense now
  – Hopefully we succeeded
Data Structures

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Trees, Hashing, Dictionaries
- Heaps, Priority Queues
- Sorting
- Disjoint Sets
- Graph Algorithms

*May have time for other brief exposure to topics, maybe parallelism*
What 373 is about

- Deeply understand the basic structures used in all software
  - Understand the data structures and their trade-offs
  - Rigorously analyze the algorithms that use them (math!)
  - Learn how to pick “the right thing for the job”
  - More thorough and rigorous take on topics introduced in CSE143 (plus more new topics)

- Practice design, analysis, and implementation
  - The mixing of “theory” and “engineering” at the core of computer science

- More programming experience (as a way to learn)
Goals

- Be able to make good design choices as a developer, project manager, etc.
  - Reason in terms of the general abstractions that come up in all non-trivial software (and many non-software) systems
- Be able to justify and communicate your design decisions

Aaron’s take:
- Key abstractions used almost every day in just about anything related to computing and software
- It is a vocabulary you are likely to internalize permanently
Last slide

I had a lot of fun and learned a great deal this quarter

You have learned the key ideas for organizing data, a skill that far transcends computer science