Parallel Prefix
CSE 332 Spring 2021

Instructor:   Hannah C. Tang

Teaching Assistants:
Aayushi Modi    Khushi Chaudhari    Patrick Murphy
Aashna Sheth    Kris Wong          Richard Jiang
Frederick Huyan Logan Milandin    Winston Jodjana
Hamsa Shankar   Nachiket Karmarkar
Define **work** and **span**

- How do we calculate work and span?
- What, if any, effect does adding more processors have on work? On span?
Announcements

- P2 due this week
- Keep up with the readings if you have any questions
Lecture Outline

❖ Amdahl’s Law: Is the ☕ half-empty or half-full?

❖ Parallel Prefix-Sum
And Now for the Good / Bad News ...

- In practice, it’s common that a program has:
  a) Parts that **parallelize** well:
      - E.g. maps/reduces over arrays and trees
  b) ... and parts that **don’t parallelize** at all:
      - E.g. reading a linked list
      - E.g. waiting on input
      - E.g. computations where each step needs the results of previous step

- These unparallelizable parts turn out to be a big bottleneck, which brings us to Amdahl’s Law ...
Amdahl’s Law

- Let the work ($T_1$) be 1 unit of time and $S$ be the unparallelizable portion of execution time:
  \[ T_1 = 1 = S + (1-S) \]

- Suppose perfect linear speed-up on the parallelizable portion. Then:
  \[ T_p = S + (1-S)/P \]

- Amdahl’s Law states the speed-up with $P$ processors is:
  \[ T_1 / T_p = 1 / (S + (1-S)/P) \]

- and the parallelism (maximum possible speed-up) is:
  \[ T_1 / T_\infty = 1 / S \]

**Definitions:***

- **Span** = $T_\infty$ = sum of runtime of all nodes in the DAG’s most-expensive path
- **Work** = $T_1$ = sum of runtime of all nodes in the DAG
- **Speed-up** = $T_1 / T_p$
  - Perfect linear speedup when $T_1 / T_p = P$
- **Parallelism** = $T_1 / T_\infty$
Amdahl’s Law Example

❖ Recall: \( T_1 = 1 = S + (1-S) \) and \( T_P = S + (1-S)/P \)

❖ Suppose: \( T_1 = \frac{1}{3} + \frac{2}{3} = 1 \)  \( (eg, \ T_1 = 100s = 33s + 67s) \)

❖ Then: \( T_P = 33 \text{ sec} + \frac{(67 \text{ sec})}{P} \)
  \[ T_3 = 33 \text{ sec} + \frac{(67 \text{ sec})}{3} = \frac{33 + 22}{3} = \frac{55}{3} \]
  \[ T_6 = 33 \text{ sec} + \frac{(67 \text{ sec})}{6} = \frac{33 + 11}{6} = \frac{44}{6} \]
  \[ T_{67} = 33 \text{ sec} + \frac{(67 \text{ sec})}{67} = \frac{33 + 1}{67} = \frac{34}{67} \]

❖ If 33% of a program is sequential, 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
# Implications of Amdahl’s Law

<table>
<thead>
<tr>
<th>Speedup:</th>
<th>( \frac{T_1}{T_P} = \frac{1}{S + \frac{1-S}{P}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Parallelism:</td>
<td>( \frac{T_1}{T_\infty} = \frac{1}{S} )</td>
</tr>
</tbody>
</table>

- In “the good old days” (1980-2005), ~12 years = 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)
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
The Challenge Posed by Amdahl’s Law

- Amdahl’s Law tells us unparallelized parts become a bottleneck very quickly
  - But it doesn’t tell us additional processors are worthless

- ... because we can find new parallel algorithms
  - Some things that seem sequential turn out to be parallelizable
  - Eg: How can we parallelize a ‘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>15</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>67</td>
<td>69</td>
<td>77</td>
</tr>
</tbody>
</table>

- We can also change the problem we’re solving
  - Eg: Video games use tons of parallel processors; they are not rendering 10-year-old graphics faster
Lecture Outline

❖ Amdahl’s Law: Is the 🚨 half-empty or half-full?

❖ Parallel Prefix-Sum
The Prefix-Sum Problem (1 of 2)

- Given `int[] input`, produce `int[] output` where:
  \[
  output[i] = input[0] + input[1] + \ldots + input[i]
  \]

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>15</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>67</td>
<td>69</td>
<td>77</td>
</tr>
</tbody>
</table>

- Problem is “inherently sequential” because each value depends on the values before it
The Prefix-Sum Problem (2 of 2)

- Sequential solution feels like a CSE142 exam problem:

```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;
}
```

- Doesn’t seem parallelizable!
  - Work: O(n), Span: O(n)
  - There’s a different algorithm with Work: O(n), Span: O(log n) 😞
Parallel Prefix-Sum: Overview (1 of 2)

- Local bragging:
  - Algorithm due to R. Ladner and M. Fischer at UW in 1977
  - Richard Ladner joined the UW faculty in 1971 and hasn’t left

- Parallel-prefix sum algorithm has two passes:
  - Each pass is \( O(n) \) work and \( O(\log n) \) span
  - So – as with array summing – parallelism is \( n/\log n \): exponential!
Parallel Prefix-Sum: Overview (2 of 2)

- First pass builds a *binary tree* from the bottom: the “up” pass
- Second pass *processes* the binary tree: the “down” pass

- Sequential algorithm is linear, but this algorithm uses two logarithmic passes
Parallel Prefix-Sum: The “Up” Pass: Overview

- This first pass builds a *binary tree* from the bottom: the “up” pass

- Parallel Prefix-Sum’s binary tree:
  - Internal nodes have a range and sum of \([lo, hi)\)
    - ... and the root has \([0, n+1)\)
  - Left child has range and sum of \([lo, middle)\)
  - Right child has range and sum of \([middle, hi)\)
  - A leaf has range and sum of \([i, i+1)\); the sum is simply input\([i]\)

- Unlike parallel-sum, we actually *create the tree*; we need it for the next pass (the “down” pass)
  - Doesn’t have to be an actual tree; could use an array (eg, binary heap)
Parallel Prefix-Sum: The “Up” Pass: Details

❖ Parent has range and sum of \([lo, hi)\)
  ▪ left has \([lo, \text{middle}), \text{and right has } [\text{middle}, hi)\)

❖ Build sum from the bottom of the tree:
  ▪ A leaf’s sum is just its value: \text{input}[i]

❖ Easy fork-join computation!
  ▪ Save partial sums from parallel-sum algorithm
  ▪ Tree is built from bottom-up, in parallel

❖ Analysis of the up pass:
  ▪ Work: \(O(n)\)
  ▪ Span: \(O(\log n)\)
Parallel Prefix-Sum’s Tree

<table>
<thead>
<tr>
<th>input</th>
<th>6</th>
<th>4</th>
<th>16</th>
<th>10</th>
<th>16</th>
<th>15</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
</table>

- **range: [0, 8)**
  - **sum:** 77
  - fromleft:
    - **range: [0, 4)**
      - **sum:** 36
      - fromleft:
        - **range: [0, 2)**
          - **sum:** 10
          - fromleft:
            - r: [0, 1)
              - s: 6
              - f:
            - r: [1, 2)
              - s: 4
              - f:
          - r: [2, 3)
            - s: 16
            - f:
        - r: [3, 4)
          - s: 10
          - f:
    - **range: [4, 8)**
      - **sum:** 41
      - fromleft:
        - **range: [4, 6)**
          - **sum:** 31
          - fromleft:
            - r: [4, 5)
              - s: 16
              - f:
            - r: [5, 6)
              - s: 15
              - f:
          - r: [6, 7)
            - s: 2
            - f:
        - r: [6, 8)
          - s: 8
          - f:
- **output**
Parallel Prefix-Sum: The “Down” Pass: Overview

- This second pass processes the binary tree: the “down” pass

- All nodes have a range and sum of [lo, hi); now populate fromLeft
  - Invariant: fromLeft is sum of elements left of the node’s range: [0, lo)

\[
\text{output}[lo] = s + s' + \text{input}[lo]
\]
Parallel Prefix-Sum: The “Down” Pass: Details

- Propagate fromLeft down:
  - Root starts with a fromLeft of 0 (why?)
  - Internal node takes its fromLeft value and
    - Passes its left child the same fromLeft
    - Passes its right child its fromLeft plus its left child’s sum
  - At the leaf, must also output[\textit{i}] = fromLeft + input[\textit{i}]

- Also an easy fork-join computation!
  - Traverse the tree built in step 1
  - Don’t produce an explicit result; the leaves will assign to output

- Analysis of down pass: Work: \(O(n)\), Span: \(O(\log n)\)
- Total for algorithm: Work: \(O(n)\), Span: \(O(\log n)\)
Parallel Prefix-Sum’s Example: The “Down” Pass

| Input | 6 | 4 | 16 | 10 | 16 | 15 | 2 | 8 |

Internal nodes:
- Left child: parent’s
- Right child: parent’s + sibling’s sum

Leaves:
- Same as internal node, then output[i] = fromLeft + input[i]

Input: 6 4 16 10 16 15 2 8

Output: 0 10 26 36 52 67 69 77
Sequential Cutoff for Prefix-Sum

- Adding a sequential cut-off isn’t too bad:

  1. Propagating up the sums:
     - Leaf node just holds the sum of a range of values (i.e., sequentially compute sum for that range)
     - The tree itself will be shallower

  2. Propagating down the fromLefts:
     - Have leaf compute prefix sum sequentially over its [lo,hi), then:

```c
output[lo] = fromLeft + input[lo];
for(i=lo+1; i < hi; i++)
    output[i] = output[i-1] + input[i]
```

Generalized Parallel-Prefix-Sum = Parallel-Prefix

- Sum-array was an example of a common pattern

- Prefix-sum is also a pattern that arises in many problems:
  - Minimum, maximum of all elements to the left of i
  - Is there an element to the left of i satisfying some property?
  - Count of elements to the left of i satisfying some property

You now know the “one weird trick”: parallel-prefix!