CSE 332 Autumn 2023
Lecture 23: Ahmdal’s Law, Parallel Prefix

Nathan Brunelle
http://www.cs.uw.edu/332
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: $\Theta(\log n)$
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!
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:
  • $\frac{T_1}{T_P} = \frac{1}{S+\frac{1-S}{P}}$

• 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.
Which Data Structures are “Suitable” for Parallelism?

• For each data structure, can we write a parallel algorithm to some all of its values such that $T_1 > T_\infty$?
  • Array
  • Linked List
  • Tree
Reductions

• “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
Reduction (sum an array)

• **Base Case:**
  - If the list’s length is smaller than the Sequential Cutoff, reduce things sequentially

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

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

• **Combine:**
  - Reduce the answers from each thread

$\text{5 8 2 9 4 1}$

$$\begin{align*}
\text{ans} &= 15 \\
\text{ans} &= 14 \\
\text{ans} &= 29
\end{align*}$$
Map

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

• Examples:
  • Vector addition:
    • sum[i] = arr1[i] + arr2[i]
  • Function application:
    • out[i] = f(arr[i]);
Map (double each value)

- **Base Case:**
  - If the list’s length is smaller than the Sequential Cutoff, convert each thing sequentially

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

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

- **Combine:**
  - No additional work necessary
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

- Today:
  - Filter/Pack to complete the trio!
Pack/Filter

• Given an array of values and a Boolean function, return a new array which contains only elements that were “true

\[ f(x) = x > 9 \]
Prefix Sum

• Given an array, compute a new array where each index $i$ is the sum of all values up to $i$

10 16 4 18 8 2 → 10 26 30 48 56 58

```java
int[] prefixSum(int[] arr)
{
    int[] output = new int[arr.length];
    output[0] = arr[0];
    for (int i = 1; i < arr.length; i++)
        output[i] = output[i-1] + arr[i];
    return output;
}
```
Parallel Prefix Sum

• Algorithm will have two major parallel steps
  • Called a “two pass” parallel algorithm

• First step:
  • Create a tree data structure

• Second Step:
  • Use the tree to fill in the output array
Step 1: Create a Tree, Fill in sum

For this pass we will only fill in sum
In the next pass we will find leftSum
Step 1: Create a Tree, Fill in sum

**Base Case:**
- If the rand is smaller than the Sequential Cutoff, create a node for that range and find the sum sequentially.

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

**Conquer:**
- Call `start()` for each thread to compute the left and right subtrees.

**Combine:**
- Create parent node, connect to children, fill in sum.
class BuildTree extends RecursiveTask<Node> {
    protected Node compute() {
        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 new Node(lo, hi, ans);
        } else {
            BuildTree left = new BuildTree(arr, lo, (hi + lo) / 2);
            BuildTree right = new BuildTree(arr, (hi + lo) / 2, hi);
            left.fork();
            Node rightChild = right.compute();
            Node leftChild = left.join();
            int ans = rightChild.sum + leftChild.sum;
            parent = new Node(lo, hi, ans);
            parent.left = leftChild;
            parent.right = rightChild;
            return parent;
        }
    }
}
After Step 1

All sums filled in per node
In the next pass we will find leftSum

leftSum is the sum of all elements strictly to the left of the current range.
Step 2: fill in \textit{leftSum} and Output

\textit{leftSum} is the sum of all elements strictly to the left of the current range.

To calculate we can use: any node's sum, parent's \textit{leftSum}.
If this is a left child:
leftSum = parent.leftSum
If this is a right child:
leftSum = parent.leftSum + sibling.sum

Input:
[10 16 4 18 8 2 14 9]

Output:

Step 2: fill in leftSum and Output
Step 2: fill in leftSum and Output

If this is a left child:
leftSum = parent.leftSum

If this is a right child:
leftSum = parent.leftSum + sibling.sum

For the leaves:
use leftSum+sum to complete output

Input:

```
10 16 4 18 8 2 14 9
```

Output:

```
0 1 2 3 4 5 6 7
```

Starting with range [0,8), leftSum = 0.

- Range [0,4):
  - LeftSum = 0
  - Sum = 48

- Range [4,8):
  - LeftSum = 48
  - Sum = 33

- Range [0,2):
  - LeftSum = 0
  - Sum = 26

- Range [2,4):
  - LeftSum = 26
  - Sum = 22

- Range [4,6):
  - LeftSum = 54 (26 + 22 + 8)
  - Sum = 10

- Range [6,8):
  - LeftSum = 64 (54 + 10)
  - Sum = 23

- Range [0,1):
  - LeftSum = 0
  - Sum = 10

- Range [1,2):
  - LeftSum = 0
  - Sum = 16

- Range [2,3):
  - LeftSum = 26
  - Sum = 4

- Range [3,4):
  - LeftSum = 30
  - Sum = 18

- Range [4,5):
  - LeftSum = 48
  - Sum = 8

- Range [5,6):
  - LeftSum = 56
  - Sum = 2

- Range [6,7):
  - LeftSum = 58
  - Sum = 14

- Range [7,8):
  - LeftSum = 72
  - Sum = 9
Step 2: fill in leftSum and Output

If this is a left child:
leftSum = parent.leftSum

If this is a right child:
leftSum = parent.leftSum + sibling.sum

For the leaves:
use leftSum+sum to complete output

Input: 10 16 4 18 8 2 14 9
Output: 10 26 30 48 56 58 72 81
class CompleteTree extends RecursiveAction {
    public CompleteTree(Node curr, Node parent, Node sibling, boolean isLeftChild, int[] output, int[] input){...
    protected void compute(){
        if(isLeftChild)
            curr.sumLeft = parent.sumLeft;
        else
            curr.sumLeft = parent.sumLeft + sibling.sum;
    if (curr.leftChild != null && curr.rightChild != null){ // if this isn’t a leaf
            CompleteTree left = new CompleteTree(curr.leftChild, curr, curr.rightChild, true, output, input);
            left.fork();
            CompleteTree right = new CompleteTree(curr.rightChild, curr, curr.leftChild, false, output, input);
            right.compute();
            left.join();
        }
    else{
            output[curr.lo] = curr.sumLeft + input[curr.lo];
            for(int i = curr.lo; i < curr.hi; i++){
                output[i] = output[i-1] + input[i]
            }
        }
    }
}
Whew! Back to Pack/Filter

• Given an array of values and a Boolean function, return a new array which contains only elements that were “true

\[ f(x) = x > 9 \]
Parallel Pack

Input: \[ \begin{array}{cccccccc}
10 & 16 & 4 & 18 & 8 & 2 & 14 & 9 \\
\end{array} \]

Output: \[ \begin{array}{cccc}
10 & 16 & 18 & 14 \\
\end{array} \]

1. Do a map to identify the true elements

\[ \begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
\end{array} \]

2. Do prefix sum on the result of the map to identify the count of true elements seen to the left of each position

\[ \begin{array}{cccccccc}
1 & 2 & 2 & 3 & 3 & 3 & 4 & 4 \\
\end{array} \]

3. Do a map using the previous results fill in the output

\[ \begin{array}{cccc}
10 & 16 & 18 & 14 \\
\end{array} \]
3. Do a map using the result of the prefix sum to fill in the output

<table>
<thead>
<tr>
<th>Input:</th>
<th>10</th>
<th>16</th>
<th>4</th>
<th>18</th>
<th>8</th>
<th>2</th>
<th>14</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Result:</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Prefix Result:</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Because the last value in the prefix result is 4, the length of the output is 4
- Each time there is a 1 in the map result, we want to include that element in the output
- If element $i$ should be included, its position matches prefixResult[$i$]-1

```java
Int[] output = new int[prefixResult[input.length-1]];
FORALL(int i = 0; i < input.length; i++){
    if (mapResult[i] == 1)
        output[prefixResult[i]-1] = input[i];
}