CSE 332 Summer 2024
Lecture 19: Parallel Prefix

Nathan Brunelle

http://www.cs.uw.edu/332
ForkJoin Framework

- This strategy is common enough that Java (and C++, and C#, and...) provides a library to do it for you!

<table>
<thead>
<tr>
<th>What you would do in Threads</th>
<th>What to instead in ForkJoin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclass Thread</td>
<td>Subclass RecursiveTask&lt;V&gt;</td>
</tr>
<tr>
<td>Override run</td>
<td>Override compute</td>
</tr>
<tr>
<td>Store the answer in a field</td>
<td>Return a V from compute</td>
</tr>
<tr>
<td>Call start</td>
<td>Call fork</td>
</tr>
<tr>
<td><strong>join</strong> synchronizes only</td>
<td><strong>join</strong> synchronizes and returns the answer</td>
</tr>
<tr>
<td>Call run to execute sequentially</td>
<td>Call compute to execute sequentially</td>
</tr>
<tr>
<td>Have a topmost thread and call run</td>
<td>Create a pool and call invoke</td>
</tr>
</tbody>
</table>
Divide and Conquer with ForkJoin

class SumTask extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    SumTask(int[] a, int l, int h) { ... }
    protected Integer compute(){// return answer
        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 ans; }
        } else {
            SumTask left = new SumTask(arr,lo,(hi+lo)/2);    // divide
            SumTask right= new SumTask(arr,(hi+lo)/2,hi);     // divide
            left.fork();  // fork a thread and calls compute (conquer)
            int rightAns = right.compute();  //call compute directly (conquer)
            int leftAns = left.join(); // get result from left
            return leftAns + rightAns;    // combine
        }
    }
}
Divide and Conquer with ForkJoin (continued)

```java
static final ForkJoinPool POOL = new ForkJoinPool();

static int parallelSum(int[] arr) {
    SumTask task = new SumTask(arr, 0, arr.length);
    return POOL.invoke(task); // invoke returns the value compute returns
}
```
Find Max with ForkJoin

class MaxTask extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // fields to know what to do
    SumTask(int[] a, int l, int h) { ... }
    protected Integer compute(){// return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) { // base case
            int ans = Integer.MIN_VALUE; // local var, not a field
            for(int i=lo; i < hi; i++) {
                ans = Math.max(ans, arr[i]);
            }
            return ans;
        }
        else {
            MaxTask left = new MaxTask(arr,lo,(hi+lo)/2); // divide
            MaxTask right = new MaxTask(arr,(hi+lo)/2,hi); // divide
            left.fork(); // fork a thread and calls compute (conquer)
            int rightAns = right.compute(); //call compute directly (conquer)
            int leftAns = left.join(); // get result from left
            return Math.max(rightAns, leftAns); // combine
        }
    }
}

}
Which Data Structures are “Suitable” for Parallelism?

• For each data structure, can we write a parallel algorithm to sum all of its values that’s more efficient than a sequential one?
  • Array
  • Linked List
  • Tree
Other Problems that can be solved similarly

• Element Search
  • Is the value 17 in the array?

• Counting items with a certain property
  • How many elements of the array are divisible by 5?

• Checking if the array is sorted

• Find the smallest rectangle that covers all points in the array

• Find the first thing that satisfies a property
  • What is the leftmost item that is divisible by 20?
Reduction/Fold

• All examples of a category of computation called a reduction (or fold)
  • We “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
Map

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

• Examples:
  • Vector addition:
    • \( \text{sum}[i] = \text{arr1}[i] + \text{arr2}[i] \)
  • Function application:
    • \( \text{out}[i] = f(\text{arr}[i]) \)
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
Map with ForkJoin

class AddTask extends RecursiveAction {
    int lo; int hi; int[] arr; // fields to know what to do

    AddTask(int[] a, int[] b, int[] sum, int l, int h) { ... }

    protected void compute(){ // return answer
        if(hi – lo < SEQUENTIAL_CUTOFF) { // base case
            for(int i = lo; i < hi; i++) {
                sum[i] = a[i] + b[i];
            }
        } else {
            AddTask left = new AddTask(a,b,sum,lo,(hi+lo)/2); // divide
            AddTask right = new AddTask(a,b,sum,(hi+lo)/2,hi); // divide
            left.fork(); // fork a thread and calls compute (conquer)
            right.compute(); // call compute directly (conquer)
            left.join(); // get result from left
            return; // combine
        }
    }
}
}
static final ForkJoinPool POOL = new ForkJoinPool();
Int[] add(int[] a, int[] b){
    ans = new int[a.length];
    AddTask task = new AddTask(a, b, ans, 0, a.length)
    POOL.invoke(task);
    return ans;
}
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
Map/Reduction Example

- Multiply together the lengths of all of the odd-length strings in a given array
  - First, do a map to covert the array of strings into an array of their lengths
  - Then do a map on that array so each value maps to 1 if it’s even and itself if it’s odd
  - Then do a reduction to multiply together that final result

- Note: You could do this in a single ForkJoin RecursiveTask, but it’s worthwhile to recognize how to “deconstruct” it since some programming languages designed specifically for parallelism have Map/Reduce built in.
  - Map and Reduce are two from a trio, with Pack/Filter being the third
Pack/Filter

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

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

\[
\begin{array}{cccccc}
10 & 16 & 4 & 18 & 8 & 2 \\
\end{array}
\rightarrow
\begin{array}{cccccc}
10 & 16 & 18 \\
\end{array}
\]
Prefix Sum

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

```
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

Tree Node:

- range: \([lo, hi)\)
- sum:
- leftSum:

The “subproblem” this node represents
  - lower bound is inclusive, upper is exclusive

The sum of all values in the range

The sum of all values to the left of the range
  - i.e. in the range \([0, lo)\)
Step 1: Using D&C
Create a Tree, Fill in sum

For this pass we will only fill in sum
In the next pass we will find leftSum

Input:

Output:
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<PrefixSumNode> {
    protected PrefixSumNode 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 PrefixSumNode(lo, hi, ans);
        } else {
            BuildTree left = new BuildTree(arr, lo, (hi + lo) / 2);
            BuildTree right = new BuildTree(arr, (hi + lo) / 2, hi);
            left.fork();
            PrefixSumNode rightChild = right.compute();
            PrefixSumNode leftChild = left.join();
            int ans = rightChild.sum + leftChild.sum;
            Parent = new PrefixSumNode(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 leftSum and Output

leftSum is the sum of all elements strictly to the left of the current range

We’re going to go root-down, so we can use: any node’s sum, parent’s leftSum
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

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

Output:

Step 2: fill in leftSum

range: [0,8)  
sum: 81  
leftSum:

range: [0,4)  
sum: 48  
leftSum:

range: [0,2)  
sum: 26  
leftSum:

range: [0,1)  
sum: 10  
leftSum:

range: [1,2)  
sum: 16  
leftSum:

range: [2,3)  
sum: 4  
leftSum:

range: [2,4)  
sum: 22  
leftSum:

range: [3,4)  
sum: 18  
leftSum:

range: [3,4)  
sum: 8  
leftSum:

range: [4,5)  
sum: 8  
leftSum:

range: [4,5)  
sum: 2  
leftSum:

range: [4,5)  
sum: 2  
leftSum:

range: [4,6)  
sum: 10  
leftSum:

range: [5,6)  
sum: 2  
leftSum:

range: [5,6)  
sum: 14  
leftSum:

range: [5,6)  
sum: 9  
leftSum:

range: [6,7)  
sum: 14  
leftSum:

range: [6,7)  
sum: 9  
leftSum:

range: [6,8)  
sum: 23  
leftSum:

range: [7,8)  
sum: 9  
leftSum:
Step 2: fill in `leftSum` and Output

If this is a left child:
\[ \text{leftSum} = \text{parent.leftSum} \]

If this is a right child:
\[ \text{leftSum} = \text{parent.leftSum} + \text{siblingsum} \]

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
```

```
range: [0,8)
sum: 81
leftSum: 0
```
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(PrefixSumNode curr, PrefixSumNode parent, PrefixSumNode 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+1; 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

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

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

\[
\begin{array}{cccc}
10 & 16 & 18 & 14 \\
\end{array}
\]
Parallel Pack

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

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

```
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];
}
```
Map/Reduction/Pack Example

- Multiply together the lengths of all of the odd-length strings in a given array
  - First, do a map to covert the array of strings into an array of their lengths
  - Then do a map on that array so each value maps to 1 if it’s even and itself if it’s odd
    - Alternatively, do a pack on the array to remove all even values
  - Then do a reduction to multiply together that final result