CSE 332: Data Structures and Parallelism

Exercise 11 - Spec

Exercise 11 Spec (25sp)

The objectives of this exercise are

- **Implement reduce using forkjoin**
- **Implement map using forkjoin**
- **Implement filter/pack using forkjoin**

Overview

This exercise consists of the following parts:

1. Use the Java ForkJoin framework to write a parallel implementation of the parallel suffix
2. Use the Java ForkJoin framework to write a parallel implementation of a filter operation (which will filter out empty strings from a given array of strings.

Motivating Application: JSON Filter

A **CSV/JSON ETL (Extract-Transform-Load) pipeline** is useful for processing millions of log lines or sensor-data records. As you stream in raw text lines (say, from server logs), many entries may be blank, malformed, or marked as comments. In that case, we must filter out these bad or empty records. Once you know which records survive filtering, you often need to lay them out into a single output buffer (or write them to a file/database) in one contiguous block. We can implement that algorithm efficiently using the ForkJoin framework.

Problem Statements

You will be implementing two files for this assignment (parallel suffix and filter/empty). We have provided an equivalent sequential implementation for both parallel suffix and filter/empty. The behavior of your methods should exactly match the behavior of these sequential versions, but should be in parallel!

You will notice, though, that each method you are instructed to implement has one additional parameter beyond what its sequential version has – an input to define the sequential cutoff. Your implementations should be defined to work for any sequential cutoff given, so long as it is at least 1.

For ParallelSuffix, you should implement two classes: BuildTreeTask and PopulateSuffixSum. BuildTreeTask is a class that implements the summation and builds the whole tree. PopulateSuffixSum fills in the **right** sum for each node and fills the suffixsum array. **You should utilize the Node class we provided in Node.java to implement that.**

For FilterEmpty, we have not provided a class structure, so that will be up to you. We have, though, provided a parallel implementation of suffix sum that matches the algorithm presented in the lecture. We suggest you use that implementation.

Without further ado, let's discuss the expected behavior of each method.

1. `suffixSum`: Given an array of integers, this method should return a new array of integers, which is the suffix sum for the original array. Note that the definition of suffix sum is: $suffixSum[j] = \sum_{i=j}^{n} array[i]$
2. `filterEmpty`: Given an array of Strings, this method should return a new array of strings containing only the non-empty strings from the original. This relative order of the remaining strings should remain the same. The length of the output array should match the number of non-empty strings in the input.

Implementation Guidelines

Your implementations in this assignment must follow these guidelines

- You may not add any import statements beyond those that are already present (except for where expressly permitted in this spec). This course is about learning the mechanics of various data structures so that we know how to use them effectively, choose among them, and modify them as needed. Java has built-in implementations of many data structures that we will discuss, in general you should not use them.
- Do not have any package declarations in the code that you submit. The Gradescope autograder may not be able to run your code if it does.
- Remove all print statements from your code before submitting. These could interfere with the Gradescope autograder (mostly because printing consumes a lot of computing time).
- Your code will be evaluated by the gradescope autograder to assess correctness. It will be evaluated by a human to verify running time. Please write your code to be readable. This means adding comments to your methods and removing unused code (including "commented out" code).

Provided Code

Several java classes have been provided for you in this zip file. Here is a description of each.

- `Sequential`
  - This class contains sequential versions of each method above. The behavior of the methods you write should match these, but should be parallelized using Java's ForkJoin framework.
  - *Do not submit this file*
- `TestClient`
  - This class contains a main method. This main method invokes subroutines which compare your code's behavior with the sequential code's behavior by running both on arrays with random contents. We recommend you try varying the sizes of the arrays and selections of sequential cutoffs to further test your code.
  - *Do not submit this file*
- `Node`
  - You should use this predefined Node class to implement the ParallelSuffix class.
  - *Do not submit this file*
- `ParallelSuffix`
  - Implement the ParallelSuffix class using ForkJoin.
  - *You will submit this file to Gradescope*
- `FilterEmpty`
  - Implement the FilterEmpty class in this class using ForkJoin.
  - *You will submit this file to Gradescope*

Part 1: Suffix Sum

Firstly, you need to implement the suffix sum with forkjoin. We have provided you with two subclasses: PopulateSuffixSum and BuildTreeTask. You will firstly implement the BuildTreeTask to build the tree before formally calculating suffix sum. Then you will populate the suffix sum and filling fromRight property in the nodes (which is the suffix sum from the right child).

In the lecture, we talked about the algorithm parallel prefix sum. Think of what might need to be changed to implement a suffix sum compared with prefix sum.

Part 2: Filter Empty

The second class to implement is filter/pack. Our goal is to filter out all of the empty strings from a given array of strings. At a high level, here is how we can adapt the procedure we discussed in class to this problem:

1. For the given array of strings, create a new array of ints of the same length.
2. Do a map operation on the array of strings to populate the array of ints such that empty strings map to 0 and non-empty strings map to 1.
3. Perform a suffix sum operation to the match result
4. Create an array of Strings whose length matches the **first** value in the suffix sum result
5. Use all three of the original input array, the map array, and the suffix sum array to populate this new array with only the non-empty strings. We expect the order of your result to be the same as the original order.
6. Return that array

Note that we require the output from your algorithm should be in the same order as the input. For example, if the input is given as:

index	0	1	2	3
element	“” (empty string)	“string1”	“” (empty string)	“string2”

Your result should be an array of size 2:

index	0	1
element	“string1”	“string2”

Since you are using suffixSum array instead of the prefixSum array, putting the element in the mapped suffix index would **NOT** give you the correct result.