CSE 332: Data Structures and Parallelism

Section 8: Parallelism and Divide-and-Conquer

0. Multiply and surrender

Consider the following algorithm, which sorts an array in parallel. Write a recurrence for the work and a recurrence of the span of this program in terms of n, where n is the length of the array.

```java
public void parallelSort(int[] arr, int lo, int hi) {
   if (hi - lo > 1) {
      int mid = lo + (hi - lo) / 2

      parallelSort(arr, lo, mid)
      parallelSort(arr, mid, hi)

      // Move the larger of the two sorted regions
      // to the end
      if (arr[mid - 1] > arr[hi - 1]) {
         swap(arr, mid - 1, hi - 1)
      }

      parallelSort(arr, lo, hi - 1)
   }
}
```

1. Sum of sums

Use the `ForkJoin` framework to write a parallelized method that returns the sum of every number contained with the given nested array. For example, if `arr` is `[[0, 1, 2], [3, 4, 5]]`, then the output is 15.

Your code must have $\mathcal{O}(mn)$ work and $\mathcal{O}(\lg(m) + \lg(n))$ span, where m is the length of `arr` and n is length of the largest subarray `arr[i]`.

2. Rotation

Use the `ForkJoin` framework to write a parallelized method `void rotate(int[] arr)` that modifies the given array by rotating each item to the left exactly once (and moving the item at index 0 to the end). For example, rotating `[1, 2, 3, 4]` should result in `[2, 3, 4, 1]`. Find the work and span of your algorithm.

3. Underwater

Suppose we're given an array of integers where each element represents the "height" of a hill viewed from the side. Now, suppose we have water pouring in at some height h from the left. Write a parallelized algorithm using the `ForkJoin` framework that determines if the k-th element is underwater. Your algorithm must have $\mathcal{O}(n)$ work and $\mathcal{O}(\lg(n))$ span, where n is the length of the array.

For example, suppose we have an array `[3, 1, 2, 5, 3, 2, 1, 7, 2]`, $h = 4$, and $k = 6$. Since hill 3 has height 5, the water cannot spill further to the right, so we conclude hill $k = 6$ is NOT underwater.

As a second example, suppose we use the same array and k but set $h = 10$. Then, hill k (and every other hill) *will* be underwater because no hill is taller then 10.

4. Mountains

Given an array a and some index i, a **peak element** a_i is any element where a_i is greater then or equal to its surrounding elements – that is, $a_i \geq a_{i-1}$ and $a_i \geq a_{i+1}$.

For example, the array `[3, 6, 5, 2, 1, 9, 1]` has two peak elements: the 6 and the 9. The array `[1, 1, 1, 1, 1]` has five peak elements: every item is greater then or equal to the surrounding ones.

Implement a parallelized algorithm using the `ForkJoin` framework to find the largest peak element in an array. Find the work and span of your algorithm.

5. Mixing Trees

Suppose we have an AVL tree where each node contains a key-value pair, where the key is an int and the value is a string. Write a parallelized algorithm using the `ForkJoin` framework that returns an array containing all key-value pairs where the key is even. Your algorithm should have $\mathcal{O}(n)$ work and $\mathcal{O}(\lg(n))$ span.

6. Majority

Given an array containing elements of type `E`, write a parallelized algorithm using the `ForkJoin` framework to find the **majority element**, namely an element that appears than $n/2$ times. If no majority element exists, return `null`. Your algorithm should have $\mathcal{O}(n \lg(n))$ work, $\mathcal{O}(n)$ span, and use $\mathcal{O}(1)$ extra memory.

Note: The items in the array do **not** implement `compareTo`. This means you cannot sort the array!

Challenge: Can you find the majority with $\mathcal{O}(n)$ work, $\mathcal{O}(\lg(n))$ span, and $\mathcal{O}(1)$ extra memory?

7. Multiplication

(a) Suppose we have two polynomials represented as two int arrays, where the i-th item represents the i-th coefficient. So, the array `[5, 10, 0, 2, -3]` would represent the polynomial $5 + 10x + 2x^3 - 3x^4$.

Write a parallelized algorithm using the `ForkJoin` framework that returns a new array representing the product of those two polynomials. You may assume the two input arrays both have length n. A naive implementation using nested loops will have $\mathcal{O}(n^2)$ work; your algorithm must be asymptotically better.

Hint: Note that a polynomial A can be written as $A_0 + A_1 x^{n/2}$, where A_0 is the first $n/2$ terms and A_1 is the latter $n/2$ terms. This means that $A \cdot B = (A_0 + A_1 x^{n/2})(B_0 + B_1 x^{n/2})$. With some algebra, we can simplify to obtain:

$$A \cdot B = A_0 B_0 + ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) x^{n/2} + A_1 B_1 x^{n/2}$$

This means that computing the product of A and B requires you to multiply polynomials exactly three times (note, not 5 times – why?). You should exploit this property when implementing your algorithm.

(b) Write recurrences for the work and span of your algorithm, then find a Big-O bound for both.