CSE 332: Data Structures and Parallelism

Section 8: Parallelism and Divide-and-Conquer Solutions

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.

```
1  public void parallelSort(int[] arr, int lo, int hi) {
2     if (hi − lo > 1) {
3        int mid = lo + (hi − lo) / 2
4
5        parallelSort(arr, lo, mid)
6        parallelSort(arr, mid, hi)
7
8        // Move the larger of the two sorted regions
9        // to the end
10       if (arr[mid − 1] > arr[hi − 1]) {
11          swap(arr, mid − 1, hi − 1)
12       }
13
14       parallelSort(arr, lo, hi − 1)
15    }
16 }
```

Solution:

This sorting algorithm makes a recursive call three times, but only two of them (the former two) are actually parallelizable. Because the if statement, where we find the largest element in both sorted regions, requires the first two sorts to complete first.

Consequently, the recurrence for the work is:

$$\text{work}(n) = \begin{cases} 2 \cdot \text{work}(n/2) + \text{work}(n-1) + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

...and the span is:

$$\text{span}(n) = \begin{cases} \text{span}(n/2) + \text{span}(n-1) + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Finding a closed form for these recurrences is beyond the scope of this class, but it should hopefully be clear that this sort is absolutely terrible.

This sorting algorithm is known as “slowsort": see the following articles for more details:

(a) https://en.wikipedia.org/wiki/Slowsort

(b) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.9158&rep=rep1&type=pdf

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]`.

Solution:

```
1  int parallelSumOfSums(int[][] arr, int lo, int hi) {
2     if (hi − lo <= CUTOFF) {
3        int sum = 0
4        for (int i = lo; i < hi; i++) {
5           sum += parallelSum(arr[i], 0, arr[i].length)
6        }
7        return sum
8     }
9     int mid = lo + (hi − lo) / 2;
10    return parallelSumOfSums(arr, lo, mid) + parallelSumOfSums(arr, mid, hi)
11 }
12
13 int parallelSum(int[] arr, int lo, int hi) {
14   if (hi − lo <= CUTOFF) {
15       return sum from lo to hi
16    }
17    int mid = lo + (hi − lo) / 2;
18    return parallelSum(arr, lo, mid) + parallelSum(arr, mid, hi)
19 }
```

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.

Solution:

A naive (and buggy) solution would be to try and rotate using a single pass, copying each item to the left once we hit our cutoff. However, trying to simultaneously read and modify the array will end up overwriting and corrupting it. Instead, we must make two passes: copy to a temp array, then copy back into the input:

```
1  void rotate(int[] arr) {
2     int[] temp = new int[arr.length];
3     parallelCopy(arr, temp, 0, arr.length, −1);
4     parallelCopy(temp, arr, 0, arr.length, 0);
5  }
6
7  void parallelCopy(int[] src, int[] dest, int lo, int hi, int offset) {
8     if (hi − lo <= CUTOFF) {
9        for (int i = lo; i < hi; i++) {
10          dest[(i + offset) % dest.length] = src[i]
11       }
12    }
13    int mid = lo + (hi − lo) / 2
14    parallelCopy(src, dest, lo, mid, offset)
15    parallelCopy(src, dest, mid, hi, offset)
16 }
```

Each pass has a work and span of $\mathcal{O}(n)$ and $\mathcal{O}(\lg(n))$, so the total work and span will be the same.

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.

Solution:

There are two different solutions we can use, both of which fit within the given limits.

Solution 1: Using a reduction

One solution is to do a reduction within the range $[0, k]$ and check to see if any items within that range is greater then or equal to h. If so, then hill k cannot be underwater. This will end up having $\mathcal{O}(n)$ work and $\mathcal{O}(\lg(n))$ span.

Solution 2: Using `scan`

A second solution is to do a `scan` on the input array using $\oplus = \max$ as our binary operator. This will give us an array where each element contains the largest number encountered up to that point. If the item at index k in this new array is smaller then h, we know hill k is underwater.

For example, suppose the initial array was `[3, 1, 2, 5, 3, 2, 1, 7, 2]` and we pick $h = 4$, and $k = 6$. The scan would return `[3, 3, 3, 5, 5, 5, 5, 7, 7]`, we observe `output[k] == 5`, which is greater then h, so conclude k is not underwater.

Since `scan` has $\mathcal{O}(n)$ work and $\mathcal{O}(\lg(n))$ span, and array indexing is a constant time operation, we know the total work and span also fits within the given limits.

Comparison

Whether we use the reduction approach or the scan approach depends on which tradeoffs we want to make.

The scan approach will require $\mathcal{O}(n)$ extra memory, but in exchange will let us *preprocess* the initial array such that all future lookups against that array for arbitrary k will take $\mathcal{O}(1)$ time.

In contrast, the reduction approach requires only $\mathcal{O}(1)$ extra memory but would also require us to to repeat the reduction every time we change k, which is potentially less efficient in the long run.

(Technically, you could also argue that since we're using `ForkJoin`, the Task objects we instantiate will end up forcing us to allocate $\mathcal{O}(n)$ memory in both cases anyways, making the tradeoffs murkier.)

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.

Solution:

The solution looks like the following:

```
1  int parallelFindMaxPeak(int[] arr, int lo, int hi) {
2     if (hi − lo <= CUTOFF) {
3        int best = −inf
4        for (int i = lo; i < hi; i++) {
5           if (arr[i] bigger then arr[i − 1] and arr[i + 1]) {
6              best = max(best, curr)
7           }
8        }
9        return best
10    } else {
11       int mid = lo + (hi − lo) / 2
12
13       return max(
14          parallelFindMaxPeak(arr, lo, mid)
15          parallelFindMaxPeak(arr, mid, hi))
16    }
17
18 }
```

The work and span is:

$$\text{work}(n) = \begin{cases} 2 \cdot \text{work}(n/2) + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases} \qquad \text{span}(n) = \begin{cases} \text{span}(n/2) + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

So we know work $\in \mathcal{O}(n)$ and span $\in \mathcal{O}(\lg(n))$.

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.

Solution:

This algorithm consists of two phases: flattening the AVL tree and generating a temporary array containing every key-value pair, then performing a pack/scan on the array to find all even keys.

(a) Turning the AVL tree into an array

  (a) Recursively traverse the tree and generate a new tree where each node contains the key-value pair and the number of nodes within that subtree. (So, a leaf node would have value 1, a node with two leaf nodes would have value 3...)

  (b) Note the root's value indicates the number of items in the tree. Create a new array of that size.

  (c) Use this new tree as the prefix sum tree. If we pass this tree into the `scan` algorithm from lecture and an initial prescan of 0, we can find which index in the array each pair belongs to.

(b) Extracting all pairs with even keys

  (a) We can do the same steps we did in the quickcheck: make a bitmap, then run `scan` and `pack`.

Notice that each step in this procedure either takes constant work and span, or takes $\mathcal{O}(n)$ work and $\mathcal{O}(\lg(n))$ span. Therefore, the total work is $\mathcal{O}(n)$ and the total span is $\mathcal{O}(\lg(n))$.

Pseudocode for the first phase is included below (although code to handle null branches is omitted for brevity):

```
1  class CountNode {
2     Pair keyValuePair
3     int numNodes
4     CountNode left, right
5     // etc...
6  }
7
8  Pair[] flattenTree(BSTNode node) {
9     CountNode countsNode = makeCounts(node)
10    Pair[] output = new int[countNodes.numNodes]
11    makeOutput(output, countsNode, 0)
12    return output
13 }
14
15 CountNode makeCounts(BSTNode node) {
16    if (node is a leaf) {
17       return new CountNode(node.keyValuePair, 1, null, null)
18    }
19    CountNode left = makeCounts(node.left)
20    CountNode right = makeCounts(node.right)
21    return new CountNode(node.keyValuePair, left.numNodes + right.numNodes + 1, left, right)
22 }
23
24 void makeOutput(int[] output, CountNode node, int prescan) {
25    if (node is a leaf) {
26       output[prescan + node.numNodes − 1] = node.keyValuePair
27    } else {
28       makeOutput(output, node.left, prescan)
29       makeOutput(output, node.right, prescan + node.left.numNodes)
30    }
31 }
```

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?

Solution:

One solution is to use divide-and-conquer: we split the array in half and find the majority element (if one exists) for each half. Then, we count up the number of times each majority element appears in total in both halves (which we can also do in parallel, using divide-and-conquer). The pseudocode:

```
1  E parallelMajority(E[] arr, int lo, int hi) {
2     if (hi − lo == 0) { return null }
3     if (hi − lo == 1) { return arr[lo] }
4
5     int mid = lo + (hi − lo) / 2
6     E left = parallelMajority(arr, lo, mid)
7     E right = parallelMajority(arr, mid, hi)
8
9     int leftCount = parallelCount(arr, left, lo, hi)
10    int rightCount = parallelCount(arr, right, lo, hi)
11
12    if (leftCount > (hi − lo) / 2 + 1) { return left }
13    if (rightCount > (hi − lo) / 2 + 1) { return right }
14    return null
15 }
16
17 int parallelCount(E[] arr, E item, int lo, int hi) {
18    if (hi − lo <= CUTOFF) {
19       return number of elements from lo to hi − 1 equal to item
20    } else {
21       int mid = lo + (hi − lo) / 2
22       return parallelCount(arr, item, lo, mid) + parallelCount(arr, item, mid, hi)
23    }
24 }
```

The parallel count has $\text{work}_c(n) = 2 \cdot \text{work}_c(n/2) + 1$ work and $\text{span}_c(n) = \text{span}_c(n/2) + 1$ in span in the recursive cases – so the work is $\mathcal{O}(n)$ and the span is $\mathcal{O}(\lg(n))$.

Therefore, the recurrences for the work and span of the majority algorithm are:

$$\text{work}_m(n) = \begin{cases} 2 \cdot \text{work}_m(n/2) + n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases} \qquad \text{span}_m(n) = \begin{cases} \text{span}_m(n/2) + \lg(n) & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Using the master theorem, we know the work is $\mathcal{O}(n \lg(n))$. Analyzing the span is somewhat harder. It turns out a tight upper bound for the span is $\mathcal{O}(\lg^2(n))$, but for now, we can take advantage of the fact that the span is upper-bounded by the looser recurrence $\text{span}_m(n) = \text{span}_m(n/2) + n$ which we know is n.

The solution to the more optimal algorithm ($\mathcal{O}(n)$ work, $\mathcal{O}(\lg n)$ span, and $\mathcal{O}(1)$ memory) is left as an exercise to the reader. In short, you will need to parallelize the Boyer-Moore majority voting algorithm. Googling "majority element parallel algorithm" will bring up a few results that discuss this approach.

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.

Solution:

This algorithm is known as the "Karatsuba multiplication" and is used to efficiently multiply large integers and polynomials. You can find more detailed info about this algorithm (and faster variations!) online.

```
1  int[] parallelMultiply(int[] A, int[] B) {
2     if (A.length == 0 or B.length == 0) {
3        return empty array
4     } else if (A.length == 1 and B.length == 1) {
5        return {A[0] * B[0]}
6     } else {
7        int n, mid = A.length, A.length / 2
8        int[] A0, A1 = A.sublist(0, mid), A.sublist(mid, n)
9        int[] B0, B1 = B.sublist(0, mid), B.sublist(mid, n)
10
11       int[] X = parallelMultiply(A0, B0)
12       int[] Y = parallelMultiply(A1, B1)
13       int[] Z = parallelMultiply(add(A0, A1), add(B0, B1))
14
15       int[] out = new int[2 * n]
16       add X to out starting at 0
17       add Z - X - Y to out starting at mid
18       add Y to out starting at n
19
20       return out
21    }
22 }
```

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

Solution:

The recurrence for work and span is:

$$\text{work}(n) = \begin{cases} 3 \cdot \text{work}(n/2) + n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases} \qquad \text{span}(n) = \begin{cases} \text{span}(n/2) + n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

After applying the master theorem, we see the work is $\mathcal{O}(n^{\lg(3)})$ and the span is $\mathcal{O}(n)$.