CSE341T/CSE549T **09/17/2014**

Scan and its Uses

Lecture 8

1 Scan

Today, we start by learning a very useful primitive. First, lets start by thinking about what other primitives we have learned so far? The most important one was REDUCE — given n elements, you can return the sum (or really, apply any associative operator) of all elements in $O(n)$ work and $O(\lg n)$ span.

Today, we will learn a much more powerful primitive. Given a sequence $A[1..n]$, the scan operation returns another sequence $B[0..n]$ such that

$$B[i] = \sum_{j=1}^{i} A[j]$$

Where the sum here can be any associative operation. That is, you can have any operator or function f, such that $f(f(x, y), z) = f(x, f(y, z))$. Since it computes prefix of each index, this operation is sometimes called ***prefix sums***. Note that $B[0]$ always contains the identity of the function you are applying and $B[n]$ contains the answer to REDUCE(A).

One way to do a scan is to do it sequentially with $O(n)$ work and $O(n)$ span. Another way is to do n reduce operations $O(n \lg n)$ work and $O(\lg n)$ span. We want the best of both worlds.

1.1 Contraction

In order to learn about scan, we are first going to learn about another algorithmic technique. What techniques have you learned so far? Divide and Conquer is the most important one. This is another important one, called ***contraction***. This is another common inductive technique in algorithms design. It is inductive in that such an algorithm involves solving a smaller instance of the same problem, much in the same spirit as a divide-and-conquer algorithm. In particular, the contraction technique involves the following steps:

1. Reduce the instance of the problem to a (much) smaller instance (of the same sort)
2. Solve the smaller instance recursively
3. Use the solution to help solve the original instance

The contraction approach is a useful technique in algorithms design. For various reasons, it is more common in parallel algorithm than in sequential algorithms, usually because the contraction and expansion can be done in parallel and the recursion only goes logarithmically deep because the problem size is shrunk by a constant fraction each time.

1.2 Applying Contraction to Scan

We'll demonstrate this technique by applying it to the scan problem. To begin, we have to answer the following question: *How do we make the input instance smaller in a way that the solution on this smaller instance will benefit us in constructing the final solution?* Let's look at an example for motivation.

Suppose we're to run `plus_scan` (i.e. `scan (op +)`) on the sequence $[2, 1, 3, 2, 2, 5, 4, 1]$. What we should get back is

$$[0, 2, 3, 6, 8, 10, 15, 19, 20]$$

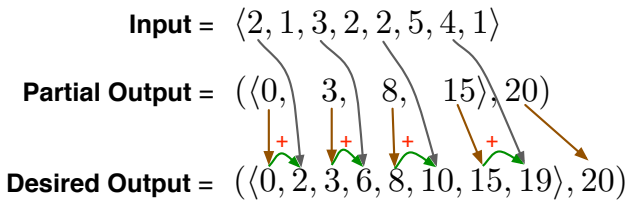
We will call the last element the `final` element.

Thought Experiment I: At some level, this problem seems like it can be solved using the divide-and-conquer approach. Let's try a simple pattern: divide up the input sequence in half, recursively solve each half, and "piece together" the solutions. A moment's thought shows that the two recursive calls are not independent—indeed, the right half depends on the outcome of the left one because it has to know the cumulative sum. So, although the work is $O(n)$, we effectively haven't broken the chain of sequential dependencies. In fact, we can see that any scheme that splits the sequence into left and right parts like this will essentially run into the same problem.

Thought Experiment II: The crux of this problem is the realization that we can easily generate a sequence consisting of every other element of the final output, together with the final sum—and this is enough information to produce the desired final output with ease. Let's say we are able to somehow generate the sequence

$$[0, 3, 8, 15, 20].$$

Then, the diagram below shows how to produce the final output sequence:



But how do we generate the "partial" output—the sequence with every other element of the desired output? The idea is simple: we pairwise add adjacent elements of the input sequence and

recursively run `scan` on it. That is, on input sequence $\langle 2, 1, 3, 2, 2, 5, 4, 1\rangle$, we would be running `scan` on $\langle 3, 5, 7, 5\rangle$, which will generate the desired partial output.

Scan(A, n)

```
1  parallel_for i ← 1 to n/2
2      do C[i] ← A[2i − 1] + A[2i]
3  C′ ← SCAN(C, n/2)
4  parallel_for i ← 0 to n/2 − 1
5      do B[2i] ← C′[i]
6         B[2i] ← C′[i] + A[2i + 1]
7  B[n] ← C[n/2]
8  return B
```

Now, lets analyze it. The work recurrence is $W(n) = W(n/2) + \Theta(n) = \Theta(n)$. The span recurrence is $S(n) = S(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$.

Best Known Bounds for Scan: You can do scan in $O(n)$ work and $O(\lg n)$ span. However, the algorithm is a little complicated and we won't cover it in class. For the purposes of this class, you may assume that you are given that algorithm as a black-box and you can cite this result when analyzing other algorithms. Feel free to think about how you would go about improving the span of scan.

2 Parenthesis Matching

We first look at the parenthesis matching problem, which is defined as follows: You are given a sequence of characters such that each character is either `(` or `)`. You want to return true if the sequence is well formed and false if the sequence is not well-formed. For instance $\langle$ `(`, `(`, `)`, `(`, `)`, `)` $\rangle$is a well formed sequence, while $\langle$ `)`, `(`, `)`, `(`, `)`, `)` $\rangle$ is not.

2.1 Sequence Fold

Lets start with the simplest sequential solution. You can just go through and keep a counter. When you see an open parenthesis, increment the counter. When you see a closed parenthesis, decrement it. If the counter ever goes negative, return false. The counter should be 0 at the end of the sequence.

You can show that this solution has $O(n)$ work and span, where n is the length of the input sequence. *How can we make it more parallel?*

2.2 Divide and Conquer

Lets try the simplest: Divide the sequence into two equal halves. What should the recursive calls return for us to be able to merge?

The first thing that comes to mind might be that the function returns whether the given sequence is well-formed. Clearly, if both s_1 and s_2 are well-formed expressions, s_1 concatenated with s_2 must be a well-formed expression. The problem is that we could have s_1 and s_2 such that neither of which is well-formed but $s_1 s_2$ is well-formed (e.g., "(((" and ")))"). This is not enough information to conclude whether $s_1 s_2$ is well-formed.

We need more information from the recursive calls. We'll crucially rely on the following observations (which can be formally shown by induction):

Observation 1 *If s contains "()" as a substring, then s is a well-formed parenthesis expression **if and only if** s' derived by removing this pair of parenthesis "()" from s is a well-formed expression.*

Applying this reduction repeatedly, we can show that a parenthesis sequence is well-formed if and only if it eventually reduces to an empty string.

Observation 2 *If s does **not** contain "()" as a substring, then s has the form "$)^i($". That is, it is a sequence of close parens followed by a sequence of open parens.*

That is to say, on a given sequence s, we'll keep simplifying s *conceptually* until it contains no substring "()" and return the pair (i, j) as our result. This is relatively easy to do recursively. Consider that if $s = s_1 s_2$, after repeatedly getting rid of "()" in s_1 and separately in s_2, we'll have that s_1 reduces to "$)^i($" and s_2 reduces to "$)^k($" for some i, j, k, ℓ. To completely simplify s, we merge the results. That is, we merge "$)^i($" with "$)^k($". The rules are simple:

- If $j \leq k$ (i.e., more close parens than open parens), we'll get "$)^{i+k-j}($".
- Otherwise $j > k$ (i.e., more open parens than close parens), we'll get "$)^i($".

This directly leads to a divide and conquer algorithm.

Exercise 1 *Write down this divide and conquer algorithm.*

What is the work and span of this algorithm?

$$W(n) = 2W(n/2) + O(1) = O(n)$$

$$S(n) = S(n/2) + O(1) = O(\lg n)$$

2.3 Using Scan

We can use `scan` to solve the parenthesis matching problem even more easily. Remember our original sequential algorithm? If we first map each open parenthesis to 1 and each close parenthesis to -1, then we just need to make sure that the $\sum_{i\leftarrow -1}^{k} s_i$ is never negative for any k. This is exactly what scan can do.

We do a $+$-`scan` on this integer sequence. The elements in the sequence returned by `scan` exactly correspond how many unmatched parenthesis there are in that prefix of the string. Therefore, if the sum of any prefix is ever negative, then we had too many closed parenthesis and the sequence is not well-formed.

For example:

$$\langle (,), (, (,),),) \rangle$$

becomes

$$\langle 1, -1, 1, 1, -1, -1, -1 \rangle$$

and then

$$\langle 0, 1, 0, 1, 2, 1, 0, -1 \rangle$$

and then fails, because the counter went negative at some point indicating an imbalance.

3 MCSS Problem

We looked at the maximum contiguous subsequence sum problem last week; given a sequence S, we wanted to find a contiguous subsequence that had the largest sum.

Lets say that we (for the heck of it) do a sum-scan of the sequence and store the result in array X. What is the meaning of $X[j]$? It is the sum of the subsequence from $S[1]$ to $S[j]$. What if we wanted to calculate the sum of subsequence from i to j (both inclusive) — we can simply calculate $X[j] - X[i-1]$. Can we use this to solve a simpler problem: What if you wanted to find out the MCSS that ended at a particular index j? Can I write down this solution in terms of just X?

Well this is

$$R_j = \max_{i=1}^{j} \sum_{k=i}^{j} S_k = \max_{i=1}^{j} (X[j] - X[i-1]) = X_j - \min_{i=0}^{j-1} X[i]$$

What is the last term? It is just the minimum value of X up to j (exclusive). Now we want to calculate it for all j, so we can use a scan (min) operation. Once we have R_j, we can calculate the minimum R_j using reduce. Lets write down the pseudocode:

```
1  X ← sum-scan(S)
2  Y ← min-scan(X)
3  parallel_for all i
4       do R[i] ← X[i] − Y[i − 1]
5  return max-reduce(R)
```

The work of each of the steps (two scans, a parallel-for loop and reduce) is $O(n)$ and the span is $O(\log n)$. We therefore have a routine that is even better than any of our divide and conquer routines.

4 Remove Duplicates

You are given a sorted array A which contains duplicate elements. How can we create an array B which contains all the elements of A, but without the duplicates?

We first simply create a boolean array C such that $C[i] = 1$ if $A[i] \neq A[i-1]$ and 0 otherwise. We can now do a sum-scan on C — if $D[i] = k$, then the number of distinct elements in A which are smaller than or equal $A[i]$ is k. Therefore, $A[i]$ should go in the kth position in B.

```
1  parallel_for i ← 1 to n
2       do if A[i] ≠ A[i − 1]
3            then C[i] ← 1
4            else  C[i] ← 0
5  D ← sum-scan(C)
6  parallel_for i ← 1 to n
7       do if A[i] ≠ A[i − 1]
8            then B[D[i]] ← A[i]
```

The work and the span of this algorithm are the same as that of the Scan algorithm.