Course: Comtemporary Algorithms T.II/2019-20

Lecture 8: Parallel Algorithms II

29 January 2020

Lecturer: Dr. Kanat Tangwongsan *Scribe: Kriangsak T. & Apivich H.*

1 The Scan Problem

In the scan problem, we will deal with binary operators $\oplus$ which have associative properties. These are operators which will give the same answer regardless of the order in which the operations are performed in, i.e. $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. Example of such operations are the addition and multiplication of numbers, matrix multiplication, string concatenation, or greatest common divisor operations.

For these operators, we can also define the identity element e, such that $x \oplus e = x$ for any x that the operator can be applied on. As an example, for multiplication, its identity element is 1, while for addition, its identity element is 0.

In this section, we will derive a parallel algorithm $\text{SCAN}(\oplus, e, A = [A[0], A[1], \ldots, A[n-1]])$ which will return (S, Σ), where

- $S = [e, e \oplus A[0], e \oplus A[0] \oplus A[1], \ldots, e \oplus A[0], e \oplus A[0] \oplus \cdots \oplus A[n-3] \oplus A[n-2]]$, and
- $\Sigma = e \oplus A[0] \oplus \cdots \oplus A[n-2] \oplus A[n-1]$.

To solve this problem naively, we can just for loop over each elements and add each answer we find to a new array. This would take $O(n)$ work and span (since it has nothing running in parallel whatsoever). We will present a couple of algorithms which improve upon the span of the naive algorithm (since the work cannot be improved upon anyway).

1.1 Divide-and-Conquer

The first idea is to use some sort of a divide-and-conquer algorithm. We can divide A into the left half and the right half, and recursively call on the same function on the two halves, then combine the answers. The problem with this, however, is that the answers that we get from the right half will depend on the answers on the left half, and so cannot be done in parallel.

However, we can get around this by still doing the halves separately, and then combining the answers in a smarter way.

Algorithm 1 Scan algorithm using divide-and-conquer

```
1: function ScanDC(⊕, e, A)
2:     some base case for recursion
3:     mid ← |A|/2
4:     L, R ← A[: mid], A[mid :]
5:     (L_p, t_L), (R_p, t_R) ← ScanDC(⊕, e, L) ∥ ScanDC(⊕, e, R)
6:     parfor i in 0, ..., |R_p| do
7:         R_p[i] ← R_p[i] ⊕ t_L
8:     end parfor
9:     return (L_p + R_p, t_L ⊕ t_R)
```

We can show that for an array of size n, the work done by the algorithm follows the recursive formula

$$W(n) = \underbrace{2W(n/2)}_{\text{Line 5}} + \underbrace{O(n)}_{\text{Loop in Line 6}}$$

which results in $O(n \lg n)$ overall work, while the span follows the recursive formula

$$S(n) = \underbrace{S(n/2)}_{\text{Line 5}} + O(1)$$

which results in $O(\lg n)$ overall span. We can see that while the span has gone down, the work done is now higher than in the naive algorithm, which isn't good. Let's look at another method.

1.2 Contraction

Let's look at a concrete example. Suppose we have an array

$$A = [1, 2, 3, 4, 5, 6, 7, 8]$$

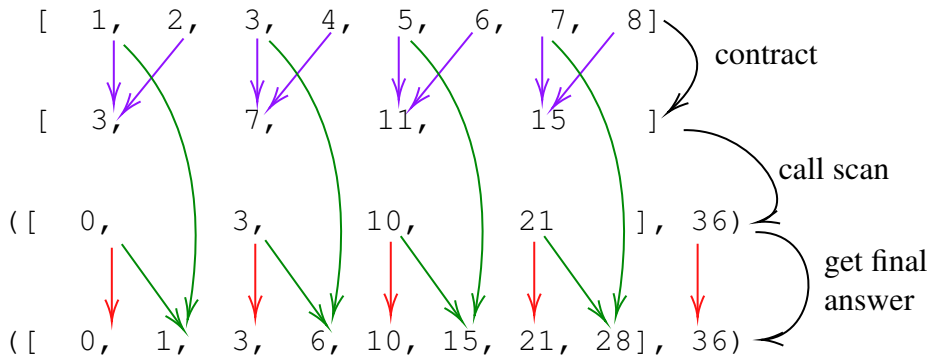
and the $\oplus$ operator is just an addition operator. It would be much easier if we instead have an array

$$A' = [3, 7, 11, 15]$$

simply because it's much shorter. Note that A' is just a reduced version of A, where every two elements in A is added together. We can call Scan on the smaller array and get

$$\text{Scan}(+, 0, A') = ([0, 3, 10, 21], 36)$$

and somehow use this to come up with a final answer. This idea is summarised in the diagram below.



What we have described is the idea of contraction. We can reduce our problem size first, call SCAN recursively, then come up with the final answer from there. The following is how this would be implemented in pseudocode.

Algorithm 2 Scan algorithm using contraction

```
1:  function ScanContract(⊕, e, A)
2:      some base case for recursion
3:      mid ← |A|/2
4:      B ← array of size mid
5:      parfor i in 0, ..., mid do
6:          B[i] ← A[2i] ⊕ A[2i + 1]          ▷ also have to deal with odd numbered cases
7:      end parfor
8:      (P, Σ) = ScanContract(⊕, e, B)
9:      S ← array of size |A|
10:     parfor i in 0, ..., mid do
11:         S[i] ← B[i]
12:         S[i + 1] ← B[i] ⊕ A[2i]           ▷ also have to deal with odd numbered cases
13:     end parfor
14:     return (S, Σ)
```

For the algorithm above, we can see that Lines 5-7 deals with array contraction, Line 8 recursively calls SCANCONTRACT on the contracted array, and Linea 10-13 “expands" the array and forms the final answer.

Again, we can analyse the work and the span of this algorithm. We see that for an array of size n, the work done by the algorithm follows the recursive formula

$$W(n) = \underbrace{W(n/2)}_{\text{recursive call}} + \underbrace{O(n)}_{\text{contract and expand}}$$

which results in $O(n)$ overall work, and the span follows the recursive formula

$$S(n) = \underbrace{S(n/2)}_{\text{recursive call}} + \underbrace{O(1)}_{\text{contract and expand}}$$

which results in $O(\lg n)$ overall span. We see that the work done is now the same as the naive method, but the span has improved.

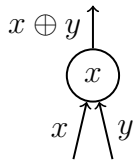
1.3 Using a Tree

Another method we can use to solve this problem is to use trees. This method is the same method presented in the parallel algorithms survey paper by Blelloch and Dhulipala[1].

The main steps of this algorithm are as follows.

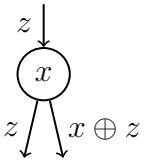
1. Form a binary tree. The tree should be a perfect binary tree, with each of the items of the array be in the leaves of the tree.

2. Scan up. Starting from the bottom, send the values from the leaf nodes up to its parent. At the parent level, you will get two values x and y from your left and right child. Take the value passed up from the left child and store it at the node itself. Then, calculate $x \oplus y$ and send it up to its parent.



   Repeat this until you get to the root level. The value that the root node sends up is just Σ.

3. Scan down. Starting from the root, note the value z sent to it from its parent (if the node is the root, then $z = e$). Send that value to your left child. With the value x which is stored in that node, work out $x \oplus z$ and send the value to its right child.



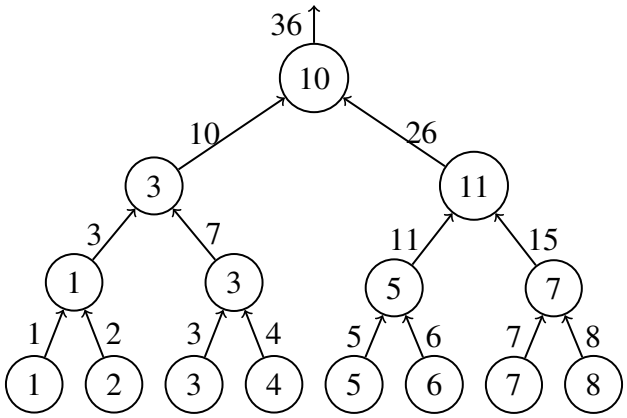
   Repeat until you get to the leaf nodes.

4. Collect the values passed on to the leaf nodes. This is the value in array S in the final answer.

Again, we will illustrate this idea with a concrete example. We will use the same example as before, where we let $A = [1, 2, 3, 4, 5, 6, 7, 8]$ and let $\oplus$ be the addition operator (so $e = 0$).

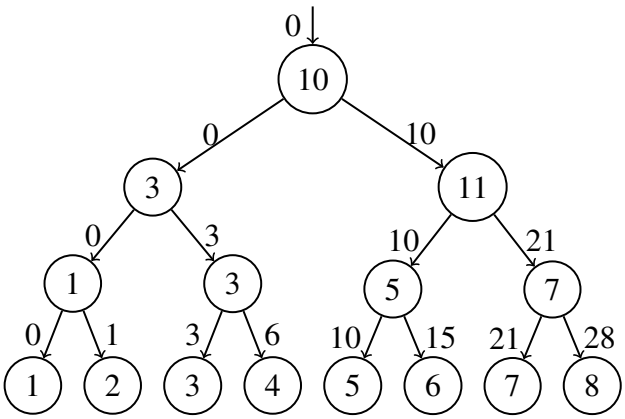
After the scan up procedure, we will have a tree looking as follows.

[1] https://ldhulipala.github.io/notes/parallel.pdf



We can see that the leaves of the tree is just the values in the array A, and the value that the root passes on is 36, and so $\Sigma = 36$.

After the scan down procedure, the tree will look as follows.



We can see that the first value passed to the root is the identity element (which is 0 in this case). If we collect the values that are passed to our leaves, we will get $S = [0, 1, 3, 6, 10, 15, 21, 28]$, which is the array we expect to have at the end of the SCAN process.

The full algorithm will not be presented here, but you can refer to the paper mentioned earlier for it. For an array of size n, the work done in this algorithm will follow the recurrence

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

resulting in $O(n)$ work, while the span for the algorithm follows the recurrence

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

which solves to $O(\lg n)$. Again, this is an improvement from the naive algorithm.

2 Applications of SCAN

There are many problems where the SCAN algorithm can be used in. Some of its uses are discussed in previous lectures (e.g. the FILTER or REDUCE function). We will now discuss more problems which we can use SCAN, FILTER or REDUCE.

2.1 Paren-Matching

Recall Paren-Matching from DS, how do we make it parallel? Here is what we can do. Let's take an example string of parenthesis:

$$(())()$$

we can assign a score to each character: say 1 to an open and -1 to a close parenthesis. we will now have an array:

$$\{1, 1, -1, -1, -1, 1\}$$

From here on, apply PLUSSCAN to the collection, then the result array,A, will be:

$$A = \{\{0, 1, 2, 1, 0, 1\}, 0\}$$

Now we can do MAP($\lambda x : x \geq 0, A$) resulting in:

$$A' = \{T, T, T, T, T, T\}$$

Finally, we can do REDUCE($\wedge, A'$) which will give the final Boolean **T**, meaning the string has the perfect parenthesis matching. In addition to this, the number of left parentheses and right parentheses needs to be the same, which is also checkable from PLUSSCAN.

2.2 Left Copy

Let's say there is a group of friends at the restaurant trying to order some food but not all of them actually know what to order. So, some of them will take what ever the guy on the left does. That is:

$$O = \{a, None, b, None, None, d\}$$

To solve this problem, one can apply SCAN on the array O. We first need to define an operator $*$ such tha

$$x * y = \begin{cases} x & \text{if } y \text{ is } None \\ y & \text{otherwise} \end{cases}$$

we can show that $*$ operator is associative (and that the identity element of $*$ is $None$), and hence SCAN can be used on it by calling

$$\text{SCAN}(*, None, O).$$

2.3 Fibonacci Sequence

Definition 2.1. the Fibonacci numbers, commonly denoted F_n, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F_0 = 0$$

$$F_1 = 1$$

and

$$F_n = F_{n-1} + F_{n-2}$$

for n > 1.

We could also express the Fibbonacci numbers in the form of a series of matrix multiplication by writing

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} F_{n-1} + F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

Notice that F_n is a linear combination of F_{n-1} and F_{n-1}, i.e.

$$F_n = aF_{n-1} + bF_{n-2}$$

when $a, b = 1$. Now, if we want the nth number of Fibonacci sequence, we can do:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdots \underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}}_{A} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

now what we want looks somewhat like:

$$\{A^0, A^1, A^2, ..., \}$$

we can then apply scan as follows:

$$\text{SCAN}(\text{matrix multiply}, I, [A, A, A, ...A])$$

2.4 List Ranking

Let's say we have a linked list and we want to give them some ranks (find the index of an element in Java Array). If this were to be a Java ArrayList, we could definitely call INDEXOF that return the index of element with linear time complexity. How about trying to solve it in parallel? How would one go about parallelizing this problem?

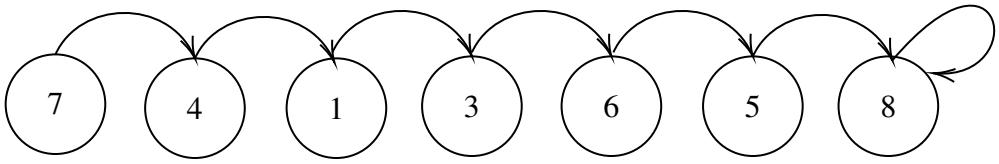
In making it parallel, one can adopt SCAN function employing the technique of contraction. However, by contracting the list naively, there might be the case when a node get contracted twice with its (two) adjacent nodes. To prevent this from happening, we will go back to our old good friend, coin-flipping technique. that is to say, we will flip a coin for every node in the given

LinkedList and will do contraction of two nodes if and only if the flips of the two nodes turn out to be exactly $H \to T$, this will be discussed in more detail when we touch upon TREECONTRACTION in the next lecture. Once we have tossed a coin for every node in our list, we can then proceed with node contractions.

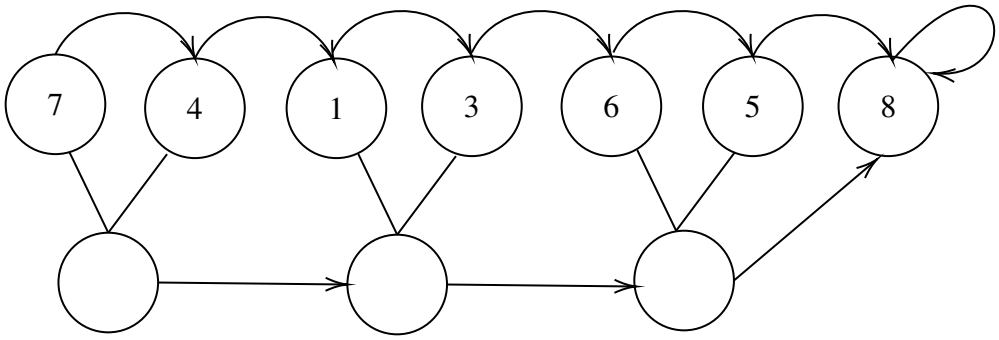
In general, we can represent a linked list in an array as follows:

2	3	7	6	1	8	5	4	8
0	1	2	3	4	5	6	7	8

Now, we can translate the array representation to the actual linked list:



Then contract the list:



2.5 Quick Sort

QUICKSORT is naturally very parallel as we can use the FILTER function when splitting the elements with the pivot which is parallel.

2.6 Merge Sort

Is MERGESORT parallel? Say if we halve an array

$$\{\underbrace{\ldots}_{n/2}\ \underbrace{\ldots}_{n/2}\}$$

then, the span of our MERGESORT will be:

$$S(n) = S(n/2) + S_{\text{merge}}$$

The question is, how fast can we merge two arrays? We first try to create another function KTH(k, A, B) to help us merging.

Claim 2.2. *Given 2 sorted sequence A and B, there is a function,* KTH(k, A, B)*, that returns $l_A, l_B, l_A + l_B = k$ such that $A[: l_A]$ and $B[: l_B]$ are the smallest k elements, of A+B done in $O(|A| + |B|)$ work, $O(\log^2(|A| + |B|))$ span.*

The implementation of this is given in the review paper. With the KTH function, we can now fo merge easily.

Algorithm 3 Parallel Merge

```
1: function MERGE(A, B)
2:     if len(A) = 0 then
3:         return B
4:     if len(B) = 0 then
5:         return A
6:     m ← (len(A) + len(B))/2
7:     la, lb ← KTH(m, A, B)
8:     A', B' = MERGE(A[: la], B[: lb]) || MERGE(A[la :], B[lb :])
9:     return A' + B'
```

The work done for MERGE above is

$$W(n) = W(n/2) + W(n - n/2) + O(\log n)$$

which solves to $O(n)$ where $n = |A| + |B|$, and the span is given by

$$S(n) = S(n/2) + O(\log n)$$

which solves to $O(\log^2 n)$

Now that, the span of MERGE is $O(log^2 n)$, we can go back to MERGESORT span analysis to see that the span of MERGESORT can be expressed to

$$S(n) = S(n/2) + O(\log^2 n)$$

which solves to $O(\log^3 n)$.