Parallel prefix scan

Now we consider a slightly different problem: Given an array $<a_0, a_1, a_2, \ldots, a_{n-1}>$, we want to compute the sum of every possible prefix of the array:

$$<a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + a_1 + a_2 + \ldots + a_{n-1}>$$

3.1. The prefix scan algorithm

This is easy enough to do on a single processor with code that takes $O(n)$ time.

```
total = array[0];
for(i = 1; i < array.length; i++) {
    total += array[i];
    array[i] = total;
}
```

With p processors, we might hope to compute all prefix sums in $O(n / p + \log p)$ time, since we can find the sum in that amount of time. However, there's not a straightforward way to adapt our earlier algorithm to accomplish this. We can begin with an overall outline, though.

1. Each processor finds the sum of its segment.

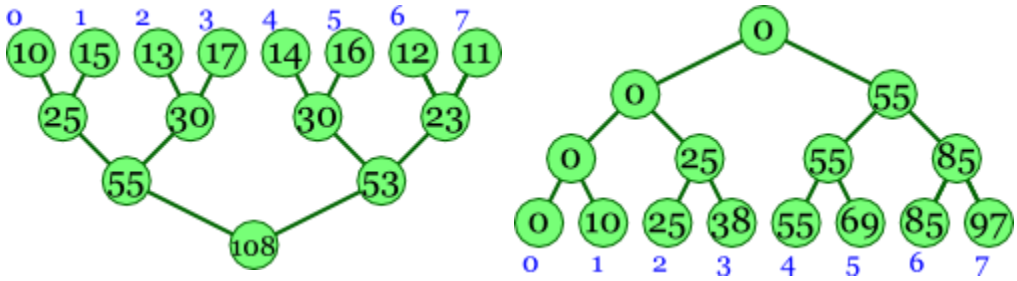
   ```
   total = segment[0];
   for(i = 1; i < segment.length; i++) total += segment[i];
   ```

2. Somehow the processors communicate so that each processor knows the sum of all segments preceding it (excluding the processor's segment). Each processor will now call this sum `total`.
3. Now each processor updates the array elements in its segment to hold the prefix sums.

   ```
   for(i = 0; i < segment.length; i++) {
       total += segment[i];
       segment[i] = total;
   }
   ```

If we divide the array equally among the processors, then the first and last steps each take $O(n / p)$ time. The hard part is step 2, which we haven't specified exactly yet. We know we have each processor imagining a number (the total for its segment), and we want each processor to know the sum of all numbers imagined by the preceding processors. We're hoping we can perform this in $O(\log p)$ time.

In 1990, parallel algorithms researcher Guy Blelloch described just such a technique. He imagined performing the computation in two phases, which we'll call the *collect phase* and the *distribute phase*, diagrammed below. (Blelloch called the phases *up-sweep* and *down-sweep*; these names make sense if you draw the diagrams upside-down from what I have drawn. You then have to imagine time as proceeding up the page.)

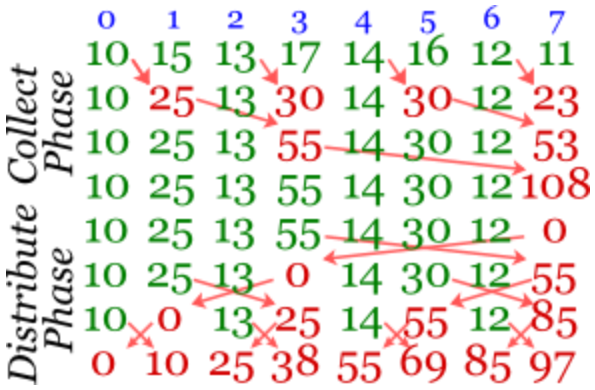


Collect Phase *Distribute Phase*

The collect phase is just like our parallel sum algorithm: Each starts with its segment's subtotal, and we have a binary tree that eventually sums these subtotals together. This takes $\log_2 p$ rounds of communication.

The distribute phase is more difficult. You can see that we start with 0, which that processor sends to its left. But the processor also sends 55 to its right — and where did the 55 come from? Notice that in the collect phase's tree 55 is the final node's left child. Similarly, the processor receiving the 55 in the first distribution round sends that to its left, and to the right it sends 85; it arrives at 85 by adding its left child from the collect phase, 30, onto the 55. At the same time, the processor that received 0 in the first distribution round sends 0 to its left child and 25 to its right; the 25 comes from adding its left child of 25 from the collect phase to the 0 it now holds.

While the tree may make sense, seeing exactly how to implement the distribute phase is a bit difficult at first. But the following diagram is quite helpful.



The first four rows represent the collect phase, just like the parallel sum from Section 2.1, except now the numbers are sent to the right, not the left. After the collect phase is finished, the final processor holds the overall total — but then it promptly replaces 108 with 0 before beginning the distribute phase. In the first round of the distribute phase, processors 3 and 7 perform something of a swap: Each processor sends the number it ended up with from the collect phase, with processor 7 adding what it receives (55) to what it previously held (0) to arrive at 55; processor 3 simply updates what it remembers to its received value, 0. In the next round, we perform the same sort of modified swap between processors 1 and 3 and between processors 5 and 7; in each case, the first processor in the pair simply accepts its received value, while the second processor adds its received value to what it held previously. By the last round, when all processors participate in the swap, each processor ends up with the sum of the numbers preceding it at the beginning.

You may well object that this is fine as long as the number of processors is a power of 2, but it doesn't address other numbers of processors. Indeed, it doesn't work unless p is a power of 2. If it isn't, then one simple solution is simply to round p down to the highest power of 2. There has to be a power of 2 between $p / 2$ and p, so we'll still end up using at least half of the available processors. In big-O bounds, we won't lose anything: Our algorithm takes $O(n / p + \log p)$ time when p is a power of 2; if we halve p(which would be the case where we lose the most processors), we end up at $O\left(n / (p / 2) + \log (p / 2)\right) = O(2 n / p + \log p - \log 2) = O(n / p + \log p)$ just as before. It's a bit frustrating to simply ignore up to half of our processors, but most often it wouldn't be worth complicating our program, increasing the likelihood of errors and costing programmer time for the sake of a constant factor of 2.

And that brings us to actually writing some code implementing our algorithm.

```
// Note: Number of processors must be power of 2 for this to work correctly.

// find sum of this processor's segment
total = segment[0];
for(i = 1; i < segment.length; i++) total += segment[i];

// perform collect phase
for(k = 1; k < procs; k *= 2) {
    if((pid & k) == 0) {
        send(pid + k, total);
        break;
    } else {
        total = receive(pid - k) + total;
    }
}

// perform distribute phase
if(pid == procs - 1) total = 0;  // reset last processor's subtotal to 0
if(k >= procs) k /= 2;
while(k > 0) {
    if((pid & k) == 0) {
        send(pid + k, total);
        total = receive(pid + k);
    } else {
        int t = receive(pid - k);
        send(pid - k, total);
        total = t + total;
    }
    k /= 2;
}

// update array to have the prefix sums
for(i = 0; i < segment.length; i++) {
    total += segment[i];
    segment[i] = total;
}
```

3.2. Filtering an array

Why would we ever want to find the sum for all prefixes of an array? This problem isn't quite as intuitive as wanting just the overall sum. But the algorithm still proves useful for several applications. One interesting application is filtering an array so that all elements satisfying a particular property are at its beginning. Below is code accomplishing this on a non-parallel machine, based on a pre-existing method named `shouldKeep` for determining whether a particular number satisfies the desired property.

```
int numKept = 0;
for(int i = 0; i < arrayLength; i++) {
    if(shouldKeep(array[i])) {
        array[numKept] = array[i];
        numKept++;
```

```
    }
}
```

One example where we want to do this is during the pivot step of Quicksort: After selecting a pivot element, we want to move all elements that are less than this selected pivot to the beginning of the array (and all elements greater than the pivot to the end — but that is basically the same problem).

To accomplish our filtering, we first create another array where each entry is 1 if `shouldKeep` says to keep the corresponding element of the original array and 0 otherwise. Then we compute all prefix sums of this array. For each element that `shouldKeep` says to keep, the corresponding prefix sum indicates where it should be placed into the new array; so finally we can just place each kept number into the indicated index (minus 1).

For example, suppose we want to keep only the numbers ending in 3 or 7 in the array 2, 3, 5, 7, 11, 13, 17, 19. We compute the following values.

input:	2,	3,	5,	7,	11,	13,	17,	19
keep:	0,	1,	0,	1,	0,	1,	1,	0
prefix:	0,	1,	1,	2,	2,	3,	4,	4
result:	3,	7,	13,	17				

The *keep* array is the result of testing whether each number ends in either 3 or 7. The *prefix* array holds the sum of each prefix of this array. And to compute *result*, we take each entry $input_i$ where `shouldKeep`($input_i$) is *true*, and we copy it into index $prefix_i - 1$.

3.3. Adding big integers

Just as we generalized our parallel sum algorithm from Section 2.1 to parallel scan in Section 2.2, we can also generalize our parallel prefix sum algorithm to work with an arbitrary operator $\otimes$. For this to work, our $\otimes$ operator must be associative — but it must also have an identity element: That is, there must be some value a for which $a \otimes x = x$ regardless of x. This identity is necessary for resetting the last processor's value as the distribute phase begins. In our work in Section 3.1, we reset the last processor's total to 0, since we were using addition for $\otimes$, and since 0 is the identity for addition.

We'll see an example of using a special definition for $\otimes$ by examining how to add two very large integers. Each big-integer is represented using an array $<a_0, a_1, \ldots, a_{n-2}, a_{n-1}>$ with each array entry representing one digit: a_0 is the

big-integer's 1's digit, a_1 is its 10's digit, and so forth. We'll represent our second big-integer to add as $<b_0, b_1, \ldots, b_{n-2}, b_{n-1}>$ in the same format. We want to compute the sum of these two big-integers.

On a single-processor computer, we could accomplish this using the following code fragment.

```
carry = 0;
for(i = 0; i < n; i++) {
    sum[i] = (a[i] + b[i] + carry) % 10;
    carry  = (a[i] + b[i] + carry) / 10;
}
```

Notice how this program uses the `carry` variable to hold a value that will be used in the next iteration. It's this `carry` variable that makes this code difficult to translate into an efficient program on multiple processors.

But here's what we can do: First, notice that our primary problem is determining whether each column has a carry for the next column. For determining this, we'll create a new array c, where each value is either C (for **c**arry), M (for **m**aybe), or N (for **n**ever). To determine entry c_i of this array, we'll compute $a_i + b_i$ and assign c_i to be C if the sum is 10 or more (since in this case this column will certainly carry 1 into the following column), M if the sum is 9 (since this column may carry into the next column, if there is a carry from the previous column), and N if it is below 9 (since there is no possibility that this column will carry).

Now we'll do a parallel prefix scan on this c array with our $\otimes$ operator defined as in the below table.

x	y	$x \otimes y$
N	N	N
N	M	N
N	C	C
M	N	N
M	M	M
M	C	C
C	N	N
C	M	C
C	C	C

More compactly, $x \otimes y$ is y if y is either C or N, but it is x if y is M. We can confirm that this $\otimes$ operator is associative by simply listing all 27 possible combinations for three variables x, y, and z and confirming that for each of them, $(x \otimes y) \otimes z$ matches $x \otimes (y \otimes z)$.

x	y	z	$(x \otimes y) \otimes z$	$x \otimes (y \otimes z)$
N	N	N	N	N
N	N	M	N	N
N	N	C	C	C
N	M	N	N	N
N	M	M	N	N
N	M	C	C	C
N	C	N	N	N
N	C	M	C	C
N	C	C	C	C

x	y	z	$(x \otimes y) \otimes z$	$x \otimes (y \otimes z)$
M	N	N	N	N
M	N	M	N	N
M	N	C	C	C
M	M	N	N	N
M	M	M	M	M
M	M	C	C	C
M	C	N	N	N
M	C	M	C	C
M	C	C	C	C

x	y	z	$(x \otimes y) \otimes z$	$x \otimes (y \otimes z)$
C	N	N	N	N
C	N	M	N	N
C	N	C	C	C
C	M	N	N	N
C	M	M	C	C
C	M	C	C	C
C	C	N	N	N
C	C	M	C	C
C	C	C	C	C

Moreover, we can see this operator has an identity because M $\otimes$ x is x regardless of x's value. Since our $\otimes$ operator is associative and has an identity, we can legimately use it for our prefix scan algorithm.

Let's see an example of how this might work. Suppose we want to add 13579 and 68123. These two five-digit numbers are represented with the arrays <9,7,5,3,1> and <3,2,1,8,6>. The steps we undertake to compute the sum are summarized in the table below.

***a*:**	< 9, 7, 5, 3, 1>
***b*:**	< 3, 2, 1, 8, 6>
***c*:**	< C, M, N, C, N>
prefix scan:	< C, C, N, C, N>
carries:	< 0, 1, 1, 0, 1>
sum:	< 2, 0, 7, 1, 8>

We first compute the array c of C/M/N values based on summing corresponding columns from a and b; each column here is computed independently of the ohers, so c can be determined completely in parallel. We end up with <C,M,N,C,N>. Then we apply the parallel prefix scan algorithm on c using our $\otimes$ operator, arriving at <C,C,N,C,N>. Each entry c_i of this prefix scan indicates whether that column would carry into the *following* column. Thus, the carry into column i, which we term $carry_i$, will be 1 if

entry c_{i-1} exists and is C, and otherwise $carry_i$ will be 1. In our example, the *carry* array would be <0,1,1,0,1>. Our final step is to add corresponding columns of *a*, *b*, and *carry*, retaining only the 1's digit of each sum. In our example, we end up with <2,0,7,1,8>, corresponding to the result 81702.

Source: http://www.toves.org/books/distalg/index.html#3