The Parallel Prefix Sum Problem

This is a problem that comes up fairly often in parallel computing. It's interesting because it is very easy to solve sequentially, but pretty tricky to solve (efficiently) in parallel.

Suppose we have an array **x** = $x_0, x_1, \ldots, x_{n-1}$ of length n, where n is a power of 2.

The *prefix sum* of **x** is an array **y** in which each element y_i is the sum of all the elements in **x** starting from x_0 up to (and including) x_i. In other words:
y = $x_0,\ x_0+x_1,\ x_0+x_1+x_2,\ x_0+x_1+x_2+x_3,\ \ldots,\ x_0+x_1+\ldots+x_{n-1}$

An example might make more sense:

If **x** =

1	2	3	4	5	6	7	8

Then **y** =

1	3	6	10	15	21	28	36

The *prefix sum problem* is just the problem of calculating **y** from **x**.

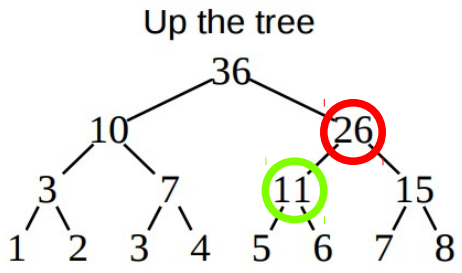
If we were writing a sequential program, we could solve the prefix sum problem something like this:

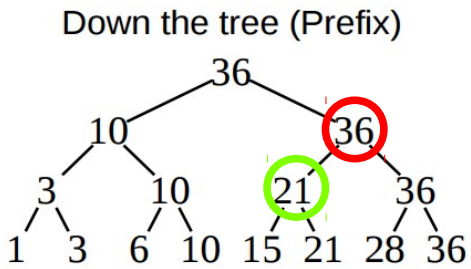
```
y[0] = x[0];
for (int i = 1; i < n; i++)
{
  y[i] = y[i – 1] + x[i];
}
```

The trouble with parallelizing this approach is that in order to calculate each new y[i], we need to know y[i – 1]. So we we can't calculate all the elements of **y** at the same time.

One solution is to use a "recursive" approach that sums up pairs of elements. This approach only uses n/2 processes. Here's how it works:

1. First, add up consecutive pairs of **x**. In the "up the tree" diagram below, **x** is shown at bottom level of the tree. Adding consecutive pairs gives the partial sums in the row above it (3, 7, 11, 15). Recursively repeat this process (i.e. sum up the partial sums), moving up the tree until you're left with only one value (36).





2. Now, we make one more pass, starting at the top and working our way back down. The "down the tree" diagram above illustrates this process. As we move down the tree, we follow three rules:

Rule A: If you're the root, your value remains unchanged.
Rule B: If you're a right child, you take on the value of your parent.
Rule C: If you're a left child, your value becomes the sum of your current value (the one you had in the "up the tree" diagram") and the value of your parent's left sibling (if it exists).

For example, look at the node circled in red in the diagrams. This node is the right child of the root. Therefore we follow rule B, and it goes from its current value (26) to it's parent's value (36).

Now consider the node in green. This node is a left child. Therefore we follow rule C. We take the current value of the node (11) and add it to the value of its parent's left sibling (10). This gives 21.

It's important that the downward pass starts at the root and works its way to the leaves in a breadth-first fashion.

When you finish the downward pass, the leaf nodes of the tree contain the prefix sum values (values of **y**).

Details / Hints:

- Start by assuming that n = 8, (so p = 8/2 = 4). Assume that process 0 has the **x** array at the start (You can hard code it – I'd recommend using the numbers in the diagrams above so you can compare your output with the diagrams when debugging). Process 0 should send two elements to each of the other processes (don't forget to keep two for process 0 itself).
- I'd recommend avoiding fancy data structures like trees and just sticking with arrays. You can either create a new array for each level (see point below for how to calculate the number of levels).
- Since the algorithm looks like a binary tree, it's possible to calculate the total number of levels the processes will need to go through. If we have n elements, then there will be $\log_2 n$ levels. This means that you don't actually need to use recursion (in fact, I'd recommend against it, since recursion + parallelism = madness :) Instead, have each process run a loop that iterates $\log_2 n$ times (once for each level of the tree). You'll need to have one loop like this for the upward pass, and then one more loop for the downward pass.
- In the upward pass, each process will need to be able to figure out which elements it must add at each level of the tree. Then, it may need communicate with another process to receive one of these values. Figuring this out is probably the trickiest part of the upward pass. Let me know if you get stuck and I can try to help.
- In the downward pass, you'll need to do the same. This is also tricky, but should be relatively similar to what you did in the point above.

- At the end, have everyone send the final prefix sum values to the process 0, and print it out.

This problem was adapted from the MIT lecture notes posted here:
http://courses.csail.mit.edu/18.337/2006/book/Lecture_03-Parallel_Prefix.pdf
This document uses fancy language and may or may not be helpful to look at.