Parallel scan

Summing the elements of an n-element array takes $O(n)$ time on a single processor. Thus, we'd hope to find an algorithm for a p-processor system that takes $O(n / p)$ time. In this section, we'll work on developing a good algorithm for this problem, then we'll see that this algorithm can be generalized to apply to many other problems where, if we were to write a program to solve it on a single-processor system, the program would consist basically of a single loop stepping through an array.

2.1. Adding array entries

So how does our program of Section 1.2 do? Well, the first loop to add the numbers in the segment takes each processor $O(n / p)$ time, as we would like. But then processor 0 must perform its loop to receive the subtotals from each of the $p - 1$ other processors; this loop takes $O(p)$ time. So the total time taken is $O(n / p + p)$ — a bit more than the $O(n / p)$ time we hoped for.

(In distributed systems, the cost of communication can be quite large relative to computation, and so it sometimes pays to analyze communication time without considering the time for computation. The first loop to add the numbers in the segment takes no communication, but processor 0 must wait for$p - 1$ messages, so our algorithm here takes $O(p)$ time for communication. In this introduction, though, we'll analyze the overall computation time.)

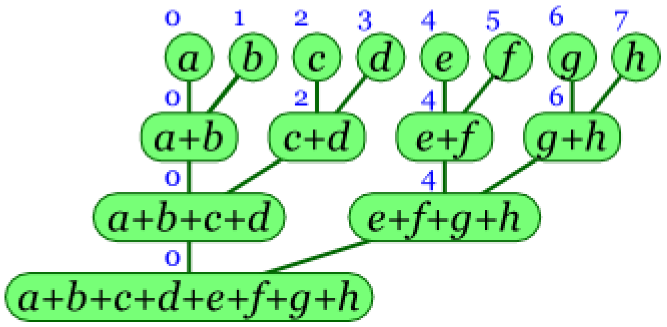
Is the extra $+ p$ in this time bound something worth worrying about? For a small system where p is rather small, it's not a big deal. But if p is something like $n^{0.8}$, then it's pretty notable: We'd be hoping for something that takes $O(n / n^{0.8}) = O(n^{0.2})$ time, but we end up with an algorithm that actually takes$O(n^{0.8})$ time.

Here's an interesting alternative implementation that avoids the second loop.

```
total = segment[0];
for(i = 1; i < segment.length; i++) total += segment[i];
if(pid < procs - 1) total += receive(pid + 1);
if(pid > 0) send(pid - 1, total);
```

Because there's no loop, you might be inclined to think that this takes $O(n / p)$ time. But we need to remember that `receive` is a blocking call, which can involve waiting. As a result, we need to think through how the communication works. In this fragment, all processors except the last attempt to`receive` a message from the following processor. But only processor $p - 1$ skips over the `receive` and sends a message to its predecessor, $p - 2$. Processor $p - 2$ receives that message, adds it into its total, and then sends that to processor $p - 3$. Thus our totals cascade down until they reach processor 0, which does not attempt to send its total anywhere. The depth of this cascade is $p - 1$, so in fact this fragment takes just as much time as before, $O(n / p + p)$.

Can we do any better? Well, yes, we can: After all, in our examples so far, we've had only one addition of subtotals at a time. A simple improvement is to divide the processors into pairs, and one processor of each pair adds its partner's subtotal into its own. Since all these $p / 2$ additions happen simultaneously, this takes $O(1)$ time — and we're left with half as many subtotals to sum together as before. We repeat this process of pairing off the remaining processors, each time halving how many subtotals remain to add together. The following diagram illustrates this. (The blue numbers indicate processor IDs.)



Thus, in the first round, processor 1 sends its total to processor 0, processor 3 to processor 2, and so on. The $p / 2$ receiving processors all do their work simultaneously. In the next round, processor 2 sends its total to processor 0, processor 6 to processor 4, and so on, leaving us with $p / 4$ sums, which again can all be computed simultaneously. After only $\lceil \log_2 p \rceil$ rounds, processor 0 will have the sum of all elements.

The code to accomplish this is below. The first part is identical; the second looks a bit tricky, but it's just bookkeeping to get the messages passed as in the above diagram.

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

for(int k = 1; k < procs; k *= 2) {
    if((pid & k) != 0) {
        send(pid - k, total);
        break;
    } else if(pid + k < p) {
        total += receive(pid + k);
    }
}
```

This takes $O(n / p + \log p)$ time, which is as good a bound as we're going to get.

(A practical detail that's beyond the scope of this work is how to design a cluster so that it can indeed handle sending $p / 2$ messages all at once. A poorly designed network between the processors, such as one built entirely of Ethernet hubs, could only pass one message at a time. If the cluster operated this way, then the time taken would still be $O(n / p + p)$.)

2.2. Generalizing to parallel scan

This algorithm we've seen to find the sum of an array easily generalizes to other problems too. If we want to multiply all the items of the array, we can perform the same algorithm as above, substituting `*=` in place of `+=`. Or if we want to find the smallest element in the array, we can use the same code again, except now we substitute an invocation of `Math.min` in place of addition.

To speak more generically about when we can use the above algorithm, suppose we have a binary operator $\otimes$ (which may stand for addition, multiplication, finding the minimum of two values, or something else). And suppose we want to compute

$$a_0 \otimes a_1 \otimes \ldots \otimes a_{n-2} \otimes a_{n-1} .$$

We can substitute $\otimes$ for addition in our above parallel-sum algorithm *as long as $\otimes$ is associative*. That is, we need it to be the case that regardless of the values of x, y, and z, $(x \otimes y) \otimes z = x \otimes (y \otimes z)$. When we're using our parallel sum algorithm with a generic associative operator $\otimes$, we call it a **parallel scan**.

The parallel scan algorithm has many applications. We already saw some applications: adding the elements of an array, multiplying them, or finding their minimum element. Another application is determining whether all values in an array of Booleans are *true*. In this case, we can use AND ($\wedge$) as our $\otimes$ operation. Similarly, if we want to determine whether any of the Boolean values in an array are*true*, then we can use OR ($\vee$). In the remainder of this section, we'll see two more complex applications: counting how many 1's are at the front of an array and evaluating a polynomial.

2.3. Counting initial 1's

The parallel scan algorithm has some less obvious applications. Suppose we have an array of 0's and 1's, and we want to determine how many 1's begin the array. If our array were <1, 1, 1, 0, 1, 1, 0, 1>, we would want to determine that the array starts with three 1's.

You probably won't be able to imagine on your own some associative operator that we might use here. However, there is a common trick that we can apply, and with some practice you'll learn how to use this trick.

Our trick is to replace each element in the array with a pair. In this case, we'll change element a_i to become the pair (a_i, a_i) in our new array. We'll perform our scan on this new array of pairs using the ⊗ operator defined as follows:

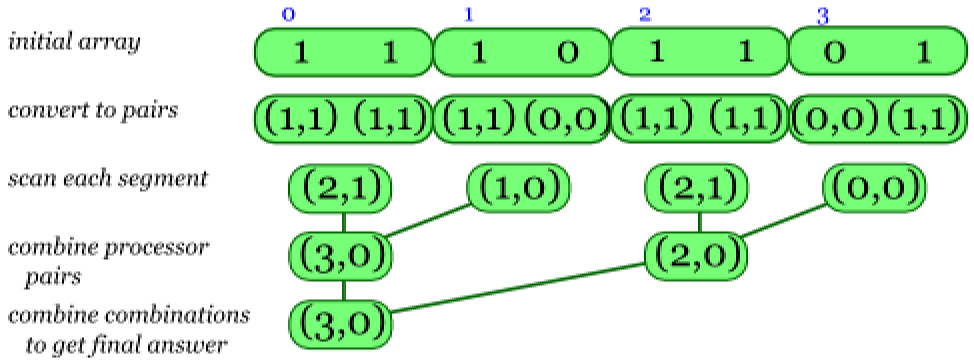
$$(x, p) \otimes (y, q) = (x + p\,y, p\,q)$$

To make sense of this, consider each pair to represent a segment of the array, with the first number saying how many 1's start the segment, while the second number is 1 or 0 depending on whether the segment consists entirely of 1's. When we combine two adjacent segments represented by the pairs (x, p) and (y, q), we first want to know the number of initial 1's in the combined segment. If p is 1, then the first segment consists entirely of x 1's, and so the total number of initial 1's in the combination includes these x 1's plus the y initial 1's of the second segment. But if p is 0, then the initial 1's in the combined segment lie entirely within the beginning of the first segment, where there are x 1's. The expression $x + p\,y$ combines these two facts into one arithmetic expression. For the second part of the resulting pair, we observe that the combined segment consists of all 1's only if both segments consist of all 1's, and $p\,q$ will compute 1 only if both p and q are both 1.

In order to know that our parallel scan algorithm works correctly, we must verify that our ⊗ operator is associative. We try both ways of associating $(x, p) \otimes (y, q) \otimes (z, r)$ and see that they match.

$$((x, p) \otimes (y, q)) \otimes (z, r) = (x + p\,y, p\,q) \otimes (z, r) = (x + p\,y + p\,q\,z, p\,q\,r)$$
$$\begin{aligned}(x, p) \otimes ((y, q) \otimes (z, r)) &= (x, p) \otimes (y + q\,z, q\,r)\\ &= (x + p\,(y + q\,z), p\,q\,r) = (x + p\,y + p\,q\,z, p\,q\,r)\end{aligned}$$

The following diagram illustrates how this would work for the array <1, 1, 1, 0, 1, 1, 0, 1> on four processors.



(A completely different approach to this problem — still using the parallel scan algorithm we've studied — is to change each 0 in the array to be its index within the array, while we change each 1 to be the array's length. Then we could use parallel scan to find the minimum element among these elements.)

2.4. Evaluating a polynomial

Here's another application that's not immediately obvious. Suppose we're given an array *a* of coefficients and a number *x*, and we want to compute the value of

$$a_0 \cdot x^{n-1} + a_1 \cdot x^{n-2} + \ldots + a_{n-2} \cdot x + a_{n-1}.$$

Again, the solution isn't easy; we end up having to change each array element into a pair. In this case, each element a_i will become the pair (a_i, x). We then define our operator ⊗ as follows.

$$(p, y) \otimes (q, z) = (p\,z + q, y\,z)$$

Where does this come from? It's a little difficult to understand at first, but each such pair is meant to summarize the essential knowledge needed for a segment of the array. This segment itself represents a polynomial. The first number in the pair is the value of the segment's polynomial evaluated for *x*, while the second is x^n, where *n* is the length of the represented segment.

But before we imagine using our parallel scan algorithm, we need first to confirm that the operator is indeed associative.

$$\begin{aligned} ((a, x) \otimes (b, y)) \otimes (c, z) &= (a\,y + b, x\,y) \otimes (c, z) \\ &= ((a\,y + b)\,z + c, x\,y\,z) = (a\,y\,z + b\,z + c, x\,y\,z) \\ (a, x) \otimes ((b, y) \otimes (c, z)) &= (a, x) \otimes (b\,z + c, y\,z) = (a\,y\,z + b\,z + c, x\,y\,z) \end{aligned}$$

Both associations yield the same result, so the operator is associative. Let's look at an example to see it at work. Suppose we want to evaluate the polynomial $x^3 + x^2 + 1$ when x is 2. In this case, the coefficients of the polynomial can be represented using the array <1, 1, 0, 1>. The first step of our algorithm is to convert this into an array of pairs.

$$(1, 2), (1, 2), (0, 2), (1, 2)$$

We can then repeatedly apply our ⊗ operator to arrive at our result.

$$\begin{aligned}&(1, 2) \otimes (1, 2) \otimes (0, 2) \otimes (1, 2)\\ &\quad = (1 \cdot 2 + 1, 2 \cdot 2) \otimes (0, 2) \otimes (1, 2) = (3, 4) \otimes (0, 2) \otimes (1, 2)\\ &\quad = (3 \cdot 2 + 0, 4 \cdot 2) \otimes (1, 2) = (6, 8) \otimes (1, 2)\\ &\quad = (6 \cdot 2 + 1, 8 \cdot 2) = (13, 16)\end{aligned}$$

So we end up with (13, 16), which has the value we wanted to compute — 13 — as its first element: $2^3 + 2^2 + 1 = 13$.

In our computation above, we proceeded in left-to-right order as would be done on a single processor. In fact, though, our parallel scan algorithm would combine the first two elements and the second two elements in parallel:

$$(1, 2) \otimes (1, 2) = (1 \cdot 2 + 1, 2 \cdot 2) = (3, 4)$$
$$(0, 2) \otimes (1, 2) = (0 \cdot 2 + 1, 2 \cdot 2) = (1, 4)$$

And then it would combine these two results to arrive at $(3 \cdot 4 + 1, 4 \cdot 4) = (13, 16)$.

Source: http://www.toves.org/books/distalg/index.html#2