Lecture 3

Parallel Prefix

3.1 Parallel Prefix

An important primitive for (data) parallel computing is the *scan operation*, also called *prefix sum* which takes an associated binary operator $\oplus$ and an ordered set $[a_1, \ldots, a_n]$ of n elements and returns the ordered set

$$[a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_n)].$$

For example,

$$\texttt{plus\_scan}([1, 2, 3, 4, 5, 6, 7, 8]) = [1, 3, 6, 10, 15, 21, 28, 36].$$

Notice that computing the scan of an n-element array requires $n - 1$ serial operations.

Suppose we have n processors, each with one element of the array. If we are interested only in the last element b_n, which is the total sum, then it is easy to see how to compute it efficiently in parallel: we can just break the array recursively into two halves, and add the sums of the two halves, recursively. Associated with the computation is a complete binary tree, each internal node containing the sum of its descendent leaves. With n processors, this algorithm takes $O(\log n)$ steps. If we have only $p < n$ processors, we can break the array into p subarrays, each with roughly $\lceil n/p \rceil$ elements. In the first step, each processor adds its own elements. The problem is then reduced to one with p elements. So we can perform the $\log p$ time algorithm. The total time is clearly $O(n/p + \log p)$ and communication only occur in the second step. With an architecture like hypercube and fat tree, we can embed the complete binary tree so that the communication is performed directly by communication links.

Now we discuss a parallel method of finding *all* elements $[b_1, \ldots, b_n] = \oplus\texttt{\_scan}[a_1, \ldots, a_n]$ also in $O(\log n)$ time, assuming we have n processors each with one element of the array. The following is a Parallel Prefix algorithm to compute the scan of an array.

Function $\texttt{scan}([a_i])$:

1. Compute pairwise sums, communicating with the adjacent processor
   $c_i := a_{i-1} \oplus a_i$ (if i even)

2. Compute the even entries of the output by recursing on the size $\frac{n}{2}$ array of pairwise sums
   $b_i := \texttt{scan}([c_i])$ (if i even)

3. Fill in the odd entries of the output with a pairwise sum
   $b_i := b_{i-1} \oplus a_i$ (if i odd)

4. Return $[b_i]$.

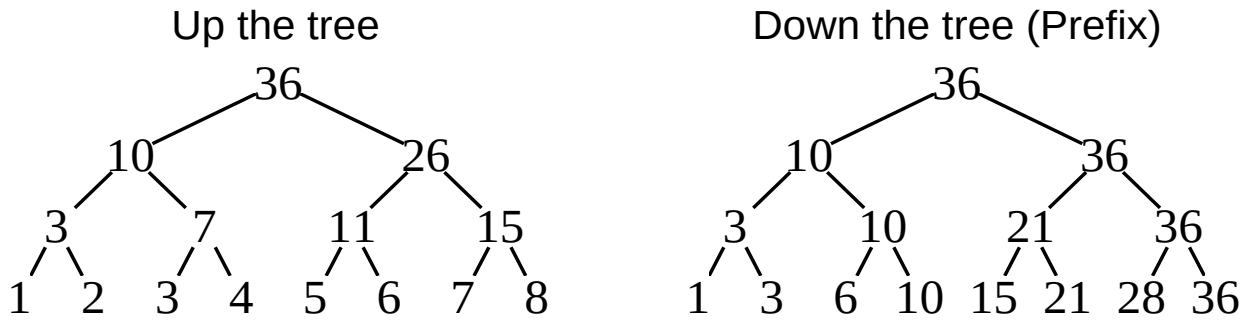

Figure 3.1: Action of the Parallel Prefix algorithm.

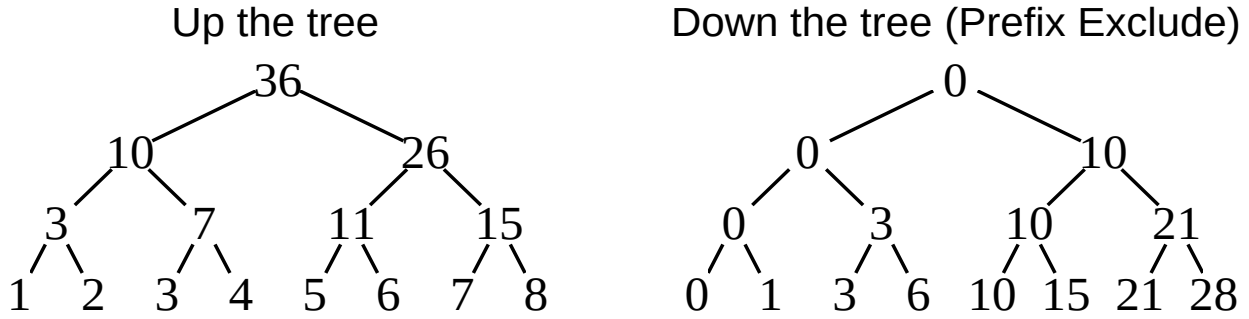

Figure 3.2: The Parallel Prefix Exclude Algorithm.

An example using the vector $[1, 2, 3, 4, 5, 6, 7, 8]$ is shown in Figure 3.1. Going up the tree, we simply compute the pairwise sums. Going down the tree, we use the updates according to points 2 and 3 above. For even position, we use the value of the parent node (b_i). For odd positions, we add the value of the node left of the parent node (b_{i-1}) to the current value (a_i).

We can create variants of the algorithm by modifying the update formulas 2 and 3. For example, the *excluded prefix sum*

$$[0, a_1, (a_1 \oplus a_2), \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$$

can be computed using the rule:

$$b_i := \texttt{excl\_scan}([c_i]) \qquad \text{(if } i \text{ odd)}, \tag{3.1}$$

$$b_i := b_{i-1} \oplus a_{i-1} \qquad \text{(if } i \text{ even)}. \tag{3.2}$$

Figure 3.2 illustrates this algorithm using the same input vector as before.

The total number of $\oplus$ operations performed by the Parallel Prefix algorithm is (ignoring a constant term of ± 1):

$$\begin{aligned} T_n &= \overbrace{\frac{n}{2}}^{\text{I}} + \overbrace{T_{n/2}}^{\text{II}} + \overbrace{\frac{n}{2}}^{\text{III}} \\ &= n + T_{n/2} \\ &= 2n \end{aligned}$$

If there is a processor for each array element, then the number of parallel operations is:

$$\begin{aligned} T_n &= \overbrace{1}^{\text{I}} + \overbrace{T_{n/2}}^{\text{II}} + \overbrace{1}^{\text{III}} \\ &= 2 + T_{n/2} \\ &= 2 \lg n \end{aligned}$$

3.2 The "Myth" of $\lg n$

In practice, we usually do not have a processor for each array element. Instead, there will likely be many more array elements than processors. For example, if we have 32 processors and an array of 32000 numbers, then each processor should store a contiguous section of 1000 array elements. Suppose we have n elements and p processors, and define $k = n/p$. Then the procedure to compute the scan is:

1. At each processor i, compute a local scan serially, for n/p consecutive elements, giving result $[d_1^i, d_2^i, \ldots, d_k^i]$. Notice that this step vectorizes over processors.
2. Use the parallel prefix algorithm to compute
$$\texttt{scan}([d_k^1, d_k^2, \ldots, d_k^p]) = [b_1, b_2, \ldots, b_p]$$
3. At each processor $i > 1$, add b_{i-1} to all elements d_j^i.

The time taken for the will be

$$T = 2 \cdot \left(\begin{array}{c} \text{time to add and store} \\ n/p \text{ numbers serially} \end{array} \right) + 2 \cdot (\log p) \cdot \left(\begin{array}{c} \text{Communication time} \\ \text{up and down a tree,} \\ \text{and a few adds} \end{array} \right)$$

In the limiting case of $p \ll n$, the $\lg p$ message passes are an insignificant portion of the computational time, and the speedup is due solely to the availability of a number of processes each doing the prefix operation serially.

3.3 Applications of Parallel Prefix

3.3.1 Segmented Scan

We can extend the parallel scan algorithm to perform segmented scan. In *segmented scan* the original sequence is used along with an additional sequence of booleans. These booleans are used to identify the start of a new segment. Segmented scan is simply prefix scan with the additional condition the the sum starts over at the beginning of a new segment. Thus the following inputs would produce the following result when applying segmented plus scan on the array A and boolean array C.

$$\begin{aligned} A &= [1\ 2\ 3\ 4 \quad 5\ 6 \quad 7 \quad 8\ 9 \quad 10] \\ C &= [1\ 0\ 0\ 0 \quad 1\ 0 \quad 1 \quad 1\ 0 \quad 1\ ] \\ \texttt{plus\_scan}(A, C) &= [\underline{1\ 3\ 6\ 10}\ \underline{5\ 11}\ \underline{7}\ \underline{8\ 17}\ \underline{10}\ ] \end{aligned}$$

We now show how to reduce segmented scan to simple scan. We define an operator, $\bigoplus_2$, whose operand is a pair $\begin{pmatrix} x \\ y \end{pmatrix}$. We denote this operand as an element of the 2-element representation of A and C, where x and y are corresponding elements from the vectors A and C. The operands of the example above are given as:

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \end{pmatrix} \begin{pmatrix} 3 \\ 0 \end{pmatrix} \begin{pmatrix} 4 \\ 0 \end{pmatrix} \begin{pmatrix} 5 \\ 1 \end{pmatrix} \begin{pmatrix} 6 \\ 0 \end{pmatrix} \begin{pmatrix} 7 \\ 1 \end{pmatrix} \begin{pmatrix} 8 \\ 1 \end{pmatrix} \begin{pmatrix} 9 \\ 0 \end{pmatrix} \begin{pmatrix} 10 \\ 1 \end{pmatrix}$$

The operator $(\bigoplus_2)$ is defined as follows:

$\bigoplus_2$	$\begin{pmatrix} y \\ 0 \end{pmatrix}$	$\begin{pmatrix} y \\ 1 \end{pmatrix}$
$\begin{pmatrix} x \\ 0 \end{pmatrix}$	$\begin{pmatrix} x \oplus y \\ 0 \end{pmatrix}$	$\begin{pmatrix} y \\ 1 \end{pmatrix}$
$\begin{pmatrix} x \\ 1 \end{pmatrix}$	$\begin{pmatrix} x \oplus y \\ 1 \end{pmatrix}$	$\begin{pmatrix} y \\ 1 \end{pmatrix}$

As an exercise, we can show that the binary operator $\bigoplus_2$ defined above is associative and exhibits the segmenting behavior we want: for each vector A and each boolean vector C, let AC be the 2-element representation of A and C. For each binary associative operator $\oplus$, the result of $\bigoplus_2$ `_scan`(AC) gives a 2-element vector whose first row is equal to the vector computed by segmented $\oplus$`_scan`(A, C). Therefore, we can apply the parallel scan algorithm to compute the segmented scan.

Notice that the method of assigning each segment to a separate processor may results in load imbalance.

3.3.2 Csanky's Matrix Inversion

The Csanky matrix inversion algorithm is representative of a number of the problems that exist in applying theoretical parallelization schemes to practical problems. The goal here is to create a matrix inversion routine that can be extended to a parallel implementation. A typical serial implementation would require the solution of $O(n^2)$ linear equations, and the problem at first looks unparallelizable. The obvious solution, then, is to search for a parallel prefix type algorithm.

Csanky's algorithm can be described as follows — the Cayley-Hamilton lemma states that for a given matrix x:

$$p(x) = \det(xI - A) = x^n + c_1 x^{n-1} + \ldots + c_n$$

where $c_n = \det(A)$, then

$$p(A) = 0 = A^n + c_1 A^{n-1} + \ldots + c_n$$

Multiplying each side by A^{-1} and rearranging yields:

$$A^{-1} = (A^{n-1} + c_1 A^{n-2} + \ldots + c_{n-1})/(-1/c_n)$$

The c_i in this equation can be calculated by Leverier's lemma, which relate the c_i to $s^k = \mathrm{tr}(A^k)$. The Csanky algorithm then, is to calculate the A^i by parallel prefix, compute the trace of each A^i, calculate the c_i from Leverier's lemma, and use these to generate A^{-1}.

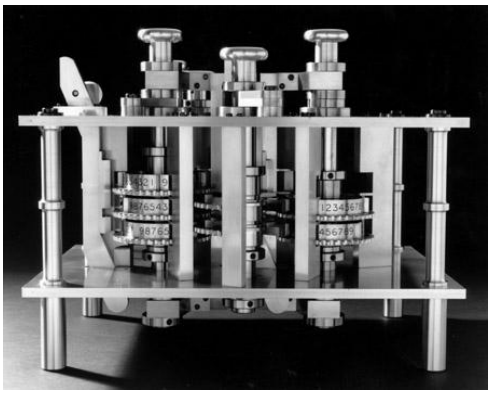

Figure 3.3: Babbage's Difference Engine, reconstructed by the Science Museum of London

While the Csanky algorithm is useful in theory, it suffers a number of practical shortcomings. The most glaring problem is the repeated multiplication of the A matix. Unless the coefficients of A are very close to 1, the terms of A^n are likely to increase towards infinity or decay to zero quite rapidly, making their storage as floating point values very difficult. Therefore, the algorithm is inherently unstable.

3.3.3 Babbage and Carry Look-Ahead Addition

Charles Babbage is considered by many to be the founder of modern computing. In the 1820s he pioneered the idea of mechanical computing with his design of a "Difference Engine," the purpose of which was to create highly accurate engineering tables.

A central concern in mechanical addition procedures is the idea of "carrying," for example, the overflow caused by adding two digits in decimal notation whose sum is greater than or equal to 10. Carrying, as is taught to elementary school children everywhere, is inherently serial, as two numbers are added left to right.

However, the carrying problem can be treated in a parallel fashion by use of parallel prefix. More specifically, consider:

	c_3	c_2	c_1	c_0		Carry
		a_3	a_2	a_1	a_0	First Integer
+		b_3	b_2	b_1	b_0	Second Integer
	s_4	s_3	s_2	s_1	s_0	Sum

By algebraic manipulation, one can create a transformation matrix for computing c_i from c_{i-1}:

$$\begin{pmatrix} c_i \\ 1 \end{pmatrix} = \begin{pmatrix} a_i + b_i & a_i b_i \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{i-1} \\ 1 \end{pmatrix}$$

Thus, carry look-ahead can be performed by parallel prefix. Each c_i is computed by parallel prefix, and then the s_i are calculated in parallel.

3.4 Parallel Prefix in MPI

The MPI version of "parallel prefix" is performed by `MPI_Scan`. From Using MPI by Gropp, Lusk, and Skjellum (MIT Press, 1999):

> [`MPI_Scan`] is much like `MPI_Allreduce` in that the values are formed by combining values contributed by each process and that each process receives a result. The difference is that the result returned by the process with rank r is the result of operating on the input elements on processes with rank $0, 1, \ldots, r$.

Essentially, `MPI_Scan` operates locally on a vector and passes a result to each processor. If the defined operation of `MPI_Scan` is `MPI_Sum`, the result passed to each process is the partial sum including the numbers on the current process.

`MPI_Scan`, upon further investigation, is not a true parallel prefix algorithm. It appears that the partial sum from each process is passed to the process in a serial manner. That is, the message passing portion of `MPI_Scan` does not scale as $\lg p$, but rather as simply p. However, as discussed in the Section 3.2, the message passing time cost is so small in large systems, that it can be neglected.