Massachusetts Institute of Technology — Lecture 3
6.892: Theory of Parallel Systems — February 11, 1997
Professor Charles E. Leiserson — Scribe: Bin Song

Lecture 3: Scans, Improved Radix Sort, and Batcher's Bitonic Sort

1 Implementation of scans (parallel prefix)

This section covers the implementation of parallel *scan* operations.

Review of Scan. Recall that *scan* is a *parallel prefix operation*. Given the input sequence $x_1, x_2, \ldots, x_n$ and an associative binary operator $\otimes$ (for the sake of convenience, we call it a product), the output of *scan* is:

$$\begin{aligned} y_1 &= x_1 \,, \\ y_2 &= x_1 \otimes x_2 \,, \\ &\ldots \\ y_n &= x_1 \otimes x_2 \otimes \ldots \otimes x_n \,. \end{aligned}$$

Or, defined recursively,

$$y_k = \begin{cases} x_k & \text{if } k = 1 \\ y_{k-1} \otimes x_k & \text{if } k > 1 \end{cases}$$

Scans can be easily performed in serial with $\Theta(n)$ work. A naive parallel algorithm for scans is the following: compute each y_k separately, performing the products with a $\log n$-depth binary tree. This scheme has critical path $\Theta(\lg n)$, but needs $\Theta(n^2)$ work.

Parallel Implementation of Scan. Below, we describe a parallel implementation that requires only $\Theta(n)$ work. It uses an n-leaf binary tree. There is one processor on each node of the tree. The input is stored in the leaf processors, from left to right.

Figure 1 illustrates the operation of the algorithm. For convenience, we do not indicate the associative binary operator $\otimes$. The algorithm works in two phases. In the first phase, the computation goes up the tree, and in the second phase values are propagated down.

Phase 1 Each leaf passes its value to the parent. Each non-leaf node computes the product of the values received from the children, and sends the result to its parent (if the parent exists). Non-leaf nodes store the value received from the left child into the right link. We can imagine that the identity value is stored on left links.

Phase 2 The root sends down the identity value to both children. While going downwards, the value is mutiplied by whatever is already on the links. The process is repeated recursively, as illustrated in the bottom half of Figure 1.

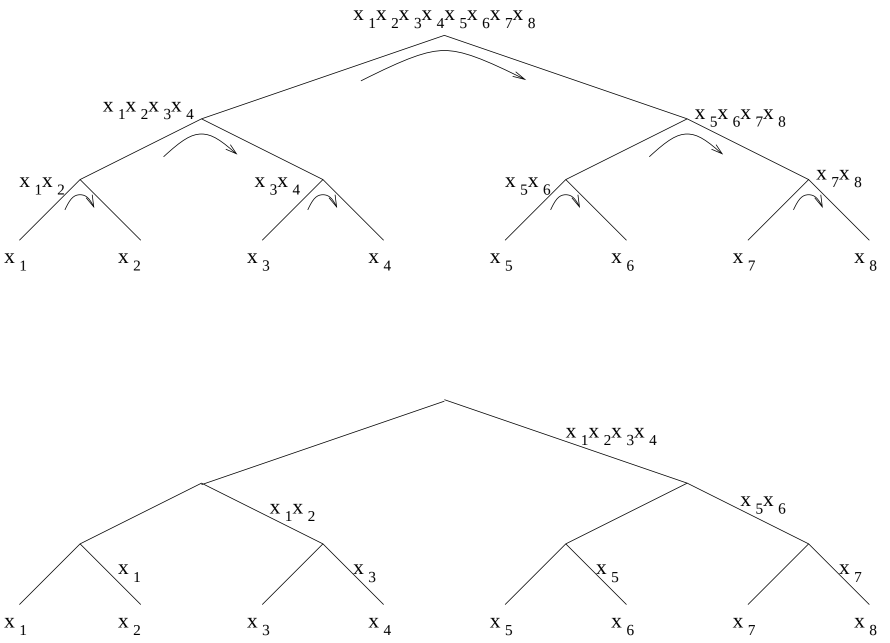

Figure 1: Implementation of Scan

For a balanced binary tree: total work = $\Theta(n)$, since each node only does constant number of calculations and communications. The critical path is $\Theta(\lg n)$.

Note: The algorithm works on any binary tree, not necessarily balanced. In this case, the critical path is $2 \times$ height.

2 Scan vs. Route

In this section we shall discuss the relative performance of the scan and route operations. Although in real life scans are much faster then routing operations, most models in the literature charge $\Theta(1)$ time for routes, and $\Theta(\lg n)$ time for scans.

For both the scan and the route problems, let n be the size of the input, and p be the number of processors. The time for routing n inputs with p processors in a bounded-degree network has a lower bound of:

$$\Omega\left(\frac{n}{p}\lg p\right) \approx R\left(\frac{n}{p}\right) .$$

This follows from the fact that the diameter of the network is at least $\Omega(\lg p)$. For any given machine, there is some constant R that characterizes the cost of routing.

On the other hand, a scan on a binary tree takes

$$\Theta\left(\frac{n}{p} + \lg p\right) \approx A\left(\frac{n}{p}\right) + S\ ,$$

where $\frac{n}{p}$ represents the computation cost, and $\lg p$ represents the communication cost. We group together all the computation costs in the constant A and all the communication costs in the constant S. In practice, $A \ll R$, and scan is much faster.

3 Radix Sort

In this section we analyze the running time of radix sort in terms of the cost of routing and scan.

Suppose that we want to sort n k-bits integers using radix sort. If we sort r bits on each pass, then we have to do b/r passes. In each pass we need to do 2^r scans in order to rank each of the combinations or r bits. Then, we need to do a permutation that orders the n elements according to their positions. The total running time is therefore

$$T \approx \frac{b}{r}\left(2^r\left(A\left(\frac{n}{p}\right) + S\right) + R\left(\frac{n}{p}\right)\right)\ .$$

T is minimized when

$$\begin{aligned} r &= \lg\left(\frac{Rn/p}{An/p + S}\right) - \lg\left(r\ln 2 - 1\right) \\ &\approx \lg\frac{R}{A} - \lg\left(r\ln 2 - 1\right) \\ &\approx 4 \text{ or } 5, \text{ in real life} \end{aligned}$$

4 Improved Radix Sort (Cole & Vishkin)

Idea. Recall that the scan algorithm does some work locally (with work An/p) and some communication (with work S). In the sorting algorithm from the previous section, we do 2^r scans. However, by being smart, there is a way to do the local work only once, while still performing 2^r communications. The idea is to keep a histogram of keys locally in the processors, so that the scan algorithm does not need to traverse all the keys every time.

We shall illustrate the algorithm with the following example. Suppose we have $p = 5$ processors, and $n = 30$. We choose $r = 2$. The keys of the elements in the processors are:

	proc 1	proc 2	proc 3	proc 4	proc 5
$\frac{n}{p}$	3	1	3	1	1
	0	0	1	1	0
	0	0	2	2	3
	1	0	0	2	2
	2	3	0	2	2
	0	0	1	2	1

Each processor has n/p elements in its local memory. It builds a histogram at the beginning of each pass. In our example, the histogram is:

	entry	proc 1	proc 2	proc 3	proc 4	proc 5
2^r	0:	3	4	2	0	1
	1:	1	1	2	2	2
	2:	1	0	1	4	2
	3:	1	1	1	0	1

The entry i of processor j represents the number of i's in the memory of processor j.

Then the processors perform $2^r = 4$ exclusive plus-scans on the histogram. After the scans the histogram looks like:

	entry	proc 1	proc 2	proc 3	proc 4	proc 5
2^r	0:	0	3	7	9	9
	1:	10	11	12	14	16
	2:	18	19	19	20	24
	3:	26	27	28	29	29

Then according to the final histogram, we can update the ranks of the elements with a single pass over the data, done locally to each processor. In the example, we get:

	proc 1	proc 2	proc 3	proc 4	proc 5
$\frac{n}{p}$	3 (26)	1 (11)	3 (28)	1 (14)	1 (16)
	0 (0)	0 (3)	1 (12)	1 (15)	0 (9)
	0 (1)	0 (4)	2 (19)	2 (20)	3 (29)
	1 (10)	0 (5)	0 (7)	2 (21)	2 (24)
	2 (18)	3 (27)	0 (8)	2 (22)	2 (25)
	0 (12)	0 (6)	1 (13)	2 (23)	1 (17)

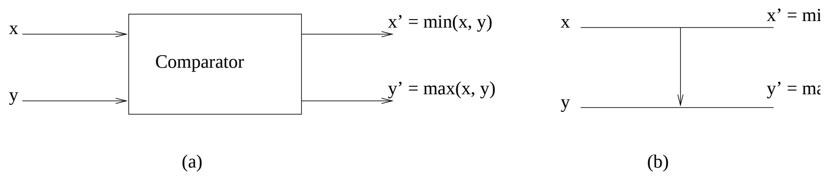

Figure 2: (a) A comparator. (b) Notation for a comparator

At the end of the pass, we do a permutation according to the ranks. The algorithm runs in time

$$T \approx \frac{b}{r}\left(2^r S + A\left(\frac{n}{p}\right) + R\left(\frac{n}{p}\right)\right) .$$

(There are hidden constants we do not care about).

We can choose r in order to minimize T, as follows:

$$\begin{aligned} r &= \lg\left(\frac{(R+A)\,n/p}{S}\right) - \lg\left(r\ln 2 - 1\right) \\ &\approx \lg\left(\frac{n}{p}\right) + \lg\left(\frac{R}{S}\right) - \lg\lg\left(\frac{n}{p}\right) . \end{aligned}$$

Thus, as n/p gets large, it pays to increase the number of bits in the radix, unlike the simple algorithm.

5 Batcher's Bitonic Sort

Comparison Network. A *comparison network* consists of only comparators and wires. A comparator (see Figure 2(a)) is a device with two inputs x and y and two output x' and y' such that

$$x' = \min(x, y) ,$$

$$y' = \max(x, y) .$$

We use the notation in Figure 2(b) to represent a comparator.

5.1 Sorting Network.

A *sorting network* is a comparison network for which the output sequence is monotonically increasing, for all input sequences.

The work for a sorting network = number of comparators; the critical path = longest path from left to right.

5.2 The zero-one principle.

One important result is the so-called *zero-one principle.* The principle says that in order to prove that a sorting network works correctly, it suffices to prove that it works correctly on sequences of 0's and 1's. This leads to dramatic simplifications in the design of such networks.

Theorem 1 (Zero-one Principle) *If a comparison network with n inputs sorts all 2^n sequences of 0's and 1's, then it sorts all sequences of arbitrary numbers.*

Proof. Let f be a monotonically increasing function: $x \leq y \Rightarrow f(x) \leq f(y)$. Then,

$$\begin{aligned} \min\left(f(x), f(y)\right) &= f\left(\min(x, y)\right) \\ \max\left(f(x), f(y)\right) &= f\left(\max(x, y)\right) \ . \end{aligned}$$

Thus, if the network transforms $\langle a_1, \ldots, a_n \rangle$ into $\langle b_1, \ldots, b_n \rangle$, then it transforms the sequence $\langle f(a_1), \ldots, f(a_n) \rangle$ into $\langle f(b_1), \ldots, f(b_n) \rangle$.

Now, suppose that the theorem is not true, that is, suppose that the network sorts all 0-1 sequences, but there exists $\langle a_1, \ldots, a_n \rangle$ with $a_i < a_j$, and a_i comes after a_j in the output. Since f can be an arbitrary monotonic function, we can choose it as follows:

$$f(x) = \begin{cases} 0 & \text{if } x \leq a_i \\ 1 & \text{if } x > a_i \end{cases}$$

So, $f(a_i) = 0$, $f(a_j) = 1$. Therefore, $f(a_i)$ and $f(a_j)$ will be out of order in the output, meaning that the network fails to sort $\langle f(a_1), \ldots, f(a_n) \rangle$. This contradiction proves the theorem. □

5.3 Sort Bitonic Sequences.

A *bitonic sequence* either

1. monotonically increases and then monotonically decreases,
2. or, it is a cyclic rotation of the first case.

A 0-1 bitonic sequence either has the form $1^*0^*1^*$ or the form $0^*1^*0^*$.

A bitonic sorting network is comprised of several building blocks. The building block for the bitonic sorter is the so-called *half-cleaner.* A *half-cleaner* is a comparison network of depth 1 in which the input of line i is compared with the input of line $i + n/2$. Figure 3(a) is an example of a half-cleaner.

Claim: For a half-cleaner, if the input is a 0-1 bitonic sequence, then both the top and bottom halves of the output are bitonic, and at least one half is clean. That is, either everything at the top is all 0's, or everything at the bottom is all 1's. Any element of the top half is smaller than any element of the bottom half.

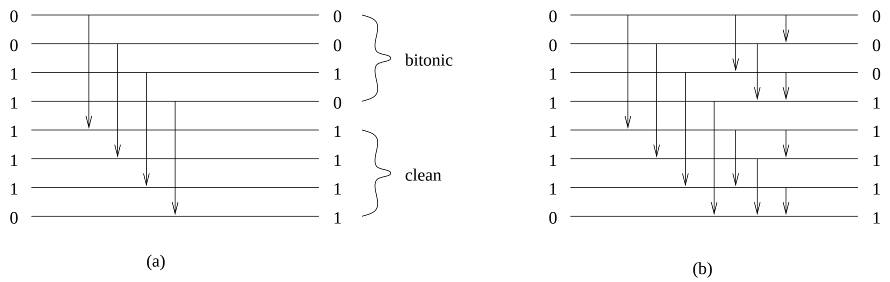

Figure 3: (a) Half-cleaner. (b) A sorter for bitonic sequence built from half-cleaners.

The outline of the proof is shown in Figure 4. In short, there are two cases: either there are more 0's or more 1's. The figure shows what happens in both cases.

To sort a bitonic sequence, we can use a half-cleaner to partition the input into two parts, and then recursively sort the two halves. Figure 3 (b) gives an example that sorts 8 inputs.

Critical path: $C(n) \leq C(n/2)+1 = O(\lg n)$. Work: $W(n) = 2W(n/2)+n/2 = \Theta(n \lg n)$.

5.4 Batcher's Bitonic Sort.

We can use the bitonic sorters to build a complete sorting network. We use the the bitonic sorters as *mergers* for a merge-sort algorithm.

To merge-sort a 0-1 sequence of n elements, we first recursively sort the $n/2$ elements in parallel, and then merge the two sorted sequences. Merging two sorted sequences is the same as sorting the concatenation of the first sequence and the reverse of the second sequence. Notice that the concatenation is a bitonic sequence, so we can use the bitonic sorters. Figure 5 represents the sorting network.

The critical path is $C(n) = C(n/2) + \lg n = O(\lg^2 n)$. The work is $W(n) = 2W(n/2) + \Theta(n \lg n) = \Theta(n \lg^2 n)$.

Based on 0-1 principle, we know that this sorting network sorts any set of numbers.

5.5 Further Study.

Batcher's Bitonic Sort has critical path $O(\lg^2 n)$. Another network called the AKS network has critical path $O(\lg n)$. Interested students can refer to the papers [1, 2].

References

[1] M. Ajtai, J. Komolòs, and E. Szemerédi. An $O(n \log n)$ sorting network. *Proc. 15th Ann. ACM Symp. on Theory of Computing*, pages 1–9, 1983.

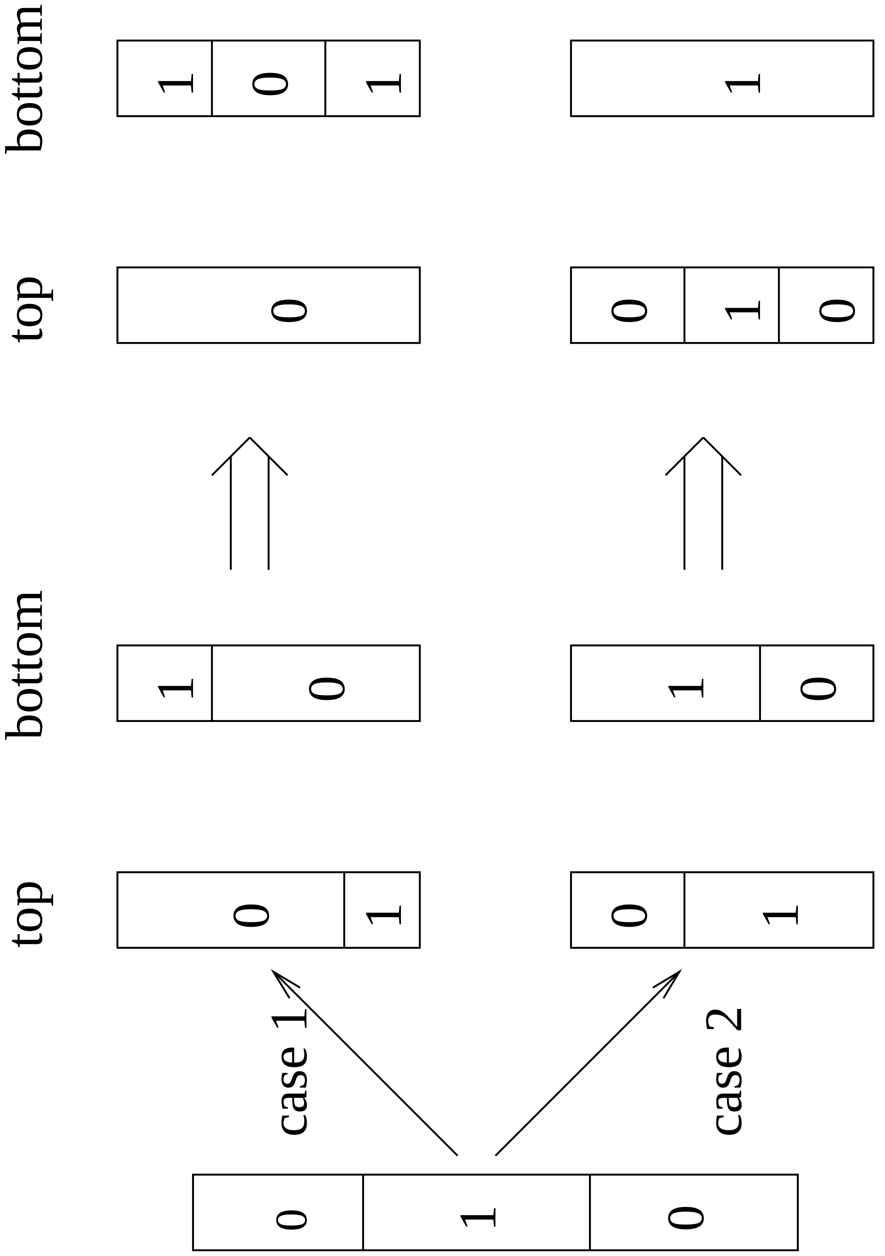

Figure 4: Sketch of why half-cleaner works.

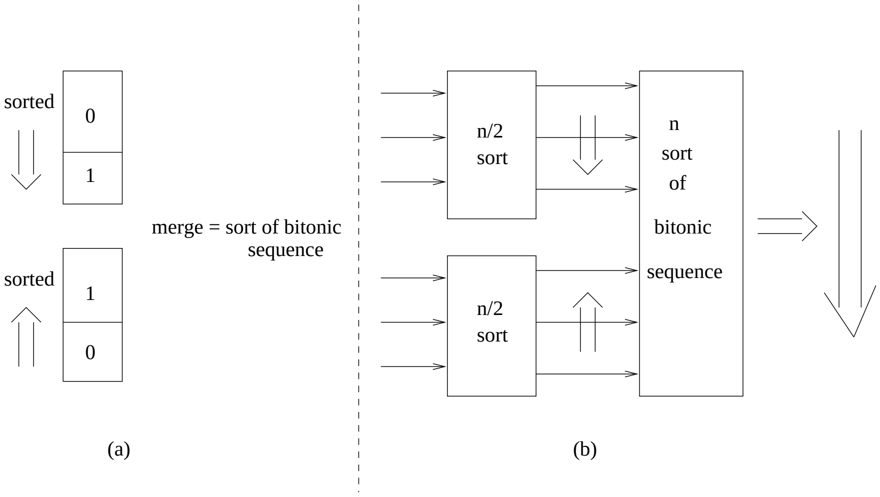

Figure 5: Batcher's Bitonic Sort.

[2] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. Research report 89, University of Warwick, January 1987.