Massachusetts Institute of Technology | Handout 14
6.892: Theory of Parallel Systems | February 27, 1997
Professor Charles E. Leiserson

Problem Set 1 Solutions

Problem 1-1. Segmented prefix

Like an ordinary prefix computation, a **segmented prefix computation** is defined in terms of a binary associative operator $\oplus$. It takes an input sequence $x = \langle x_1, x_2, \ldots, x_n \rangle$ whose elements are drawn from a domain S, and a **segment** sequence $b = \langle b_1, b_2, \ldots, b_n \rangle$ whose elements are drawn from the domain $\{0, 1\}$. It produces an output sequence $y = \langle y_1, y_2, \ldots, y_n \rangle$ over the domain S. The bits of b determine a partitioning of x and y into segments; a new segment begins wherever $b_i = 1$, and the current one continues if $b_i = 0$. The segmented prefix computation performs an independent prefix computation within each segment of x to produce the corresponding segment of y.

Here is an example that illustrates segmented prefix for the case where the associative operator is ordinary addition:

$$\begin{array}{rcrrrrrrrrrrrr} b & = & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ x & = & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ y & = & 1 & 3 & 6 & 4 & 9 & 6 & 7 & 15 & 24 & 34 & 45 & 57 \end{array}$$

Show how to implement any segmented prefix computation on an n-element list in $O(\lg n)$ time on a binary tree.

Solution Our solution will reduce the segmented prefix to an ordinary prefix over a different semi-group.

Consider the domain $S' = \{0, 1\} \times S$ with the binary operator $\widehat{\oplus}$ defined as follows:

$$(a, z)\widehat{\oplus}(a', z') = \begin{cases} (a, z \oplus z') & \text{if } a' = 0 \\ (1, z') & \text{if } a' = 1 \end{cases} .$$

*For those people who proposed this solution: you **must** state that the new operator is associative. We are not really interested in the proof, but the algorithm for parallel-prefix works only for associative operators.*

Claim: $\widehat{\oplus}$ is associative. We must prove that

$$((a_1, z_1)\widehat{\oplus}(a_2, z_2))\widehat{\oplus}(a_3, z_3) = (a_1, z_1)\widehat{\oplus}((a_2, z_2)\widehat{\oplus}(a_3, z_3)) .$$

There are three cases:

1. If $a_3 = 1$, then

$$\begin{aligned}((a_1, z_1)\widehat{\oplus}(a_2, z_2))\widehat{\oplus}(a_3, z_3) &= (1, z_3) \\ &= (a_1, z_1)\widehat{\oplus}(a_3, z_3) \\ &= (a_1, z_1)\widehat{\oplus}((a_2, z_2)\widehat{\oplus}(a_3, z_3)) \ .\end{aligned}$$

2. If $a_3 = 0$ and $a_2 = 1$, it follows that

$$\begin{aligned}((a_1, z_1)\widehat{\oplus}(a_2, z_2))\widehat{\oplus}(a_3, z_3) &= (1, z_2)\widehat{\oplus}(0, z_3) \\ &= (1, z_2 \oplus z_3) = (a_1, z_1)\widehat{\oplus}(1, z_2 \oplus z_3) \\ &= (a_1, z_1)\widehat{\oplus}((1, z_2))\widehat{\oplus}(0, z_3)) \\ &= (a_1, z_1)\widehat{\oplus}((a_2, z_2)\widehat{\oplus}(a_3, z_3)) \ .\end{aligned}$$

3. If $a_3 = 0$ and $a_2 = 0$, it follows that

$$\begin{aligned}((a_1, z_1)\widehat{\oplus}(a_2, z_2))\widehat{\oplus}(a_3, z_3) &= (a_1, z_1 \oplus z_2 \oplus z_3) \\ &= (a_1, z_1)\widehat{\oplus}((a_2, z_2)\widehat{\oplus}(a_3, z_3)) \ .\end{aligned}$$

Therefore, the claim is proven.

Based on the above claim, we can compute a segmented prefix as follows. We start with leaf node i of the binary tree containing the i-th element x_i of the input sequence and the i-th element b_i of the segment sequence. Every node computes the pair (b_i, x_i) (in time $O(1)$). Then the tree computes the prefix of the pairs (b_i, x_i) with respect to the operator $\widehat{\oplus}$, using the standard $O(\lg n)$ algorithm. Finally, each node outputs the second element c_i of the pair (a_i, c_i) produced by the prefix. The algorithm runs in $O(\lg n)$, as required.

Correctness: we are left to prove that the algorithm actually computes the segmented prefix of the original input. Thus amounts to prove that the output satisfies the defining recurrence

$$c_i = \begin{cases} x_i & \text{if } b_i = 1 \\ c_{i-1} \oplus x_i & \text{if } b_i = 0 \end{cases} \ .$$

This can easily be done by induction on the length n of the sequence.

Problem 1-2. Brent's theorem

Brent's theorem states that for any dag with work T_1 and critical-path length T_∞, a greedy scheduler can execute the dag on P processors in time at most

$$T_P \le \frac{T_1}{P} + T_\infty \ .$$

Prove the tighter bound

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \ .$$

Solution We give two solutions. The first is a straightforward generalization of the proof given in class. The second is a clever trick that Michael Bender came up with.

1. Let S_i be the size of the vectors on the i-th step. That is,

$$T_1 = \sum_{i=1}^{T_\infty} S_i$$

where T_∞ is the number of data-parallel steps. We schedule the processors using a greedy strategy. That is, we divide the vector data evenly among the processors. Let t_i be the time to execute the i-th step. Then

$$t_i = \left\lceil \frac{S_i}{P} \right\rceil .$$

Therefore, the total running time T_P is

$$\begin{aligned} T_P &= \sum_{i=1}^{T_\infty} \left\lceil \frac{S_i}{P} \right\rceil \\ &\le \sum_{i=1}^{T_\infty} \left(\frac{S_i}{P} + \frac{P-1}{P} \right) \\ &= \frac{T_1}{P} + \frac{P-1}{P} \cdot T_\infty , \end{aligned}$$

which is equivalent to the stated proposition. The trick lies in the observation that the S_i's are integers and thus there is a better bound to the ceiling function.

2. The solution we present here may be different than what you might have thought of. The approach we give here is a different way to see Brent's theorem. Michael Bender first proposed this proof; the beauty of his method is that it requires almost no math.

The setup is somewhat different than what we have seen in class. Instead of considering a data-parallel computation, we view the computation as a **directed acyclic graph** (dag), in which the vertices represent tasks (units of work), and the edges represent dependencies among the tasks.

Consider a dag with work T_1 and critical-path length T_∞, along with Brent's greedy scheduler. If at any step a processor is idle, however, we do not say that it is idle. Rather, we imagine that the processor be executing some *shadow chain* of threads. We can also say, without loss of generality, that processor 1 is busy all the time (there is at least one ready thread per step — just allocate it to processor 1). There are therefore $P-1$ shadow chains. Moreover, we know that every chain is at most T_∞ threads long, since when a processor executes a shadow thread the depth of the dag is reduced by at least 1. The total work (real plus shadow) performed by the machine is thus at most $T_1 + (P-1)T_\infty$. Since all processors are now busy all the time, it follows that the execution time is at most

$$T_P \le \frac{1}{P}(T_1 + (P-1)T_\infty) ,$$

which is equivalent to the statement of the problem.

Problem 1-3. Routing on butterflies

A **packing** problem consists of routing any collection of $M \leq N$ packets contained (one per processor) in level $\lg N$ of an N-input butterfly into the first M processors in level 0 of the butterfly so that the relative order of the packets is unchanged. For example, the routing problem $1 \rightarrow 0$, $3 \rightarrow 1$, $4 \rightarrow 2$, $7 \rightarrow 3$ is a packing problem (we mean that a packet goes from processor 1 in level $\lg N$ to processor 0 in level 0, and so on.)

(a) Give an algorithm for packing that requires at most $O(\lg N)$ steps on an N-input butterfly. (*Hint:* Before routing, do a prefix operation.)

A **monotone** routing problem is one for which the relative order of packets is unchanged. For example, the routing problem $1 \rightarrow 2$, $4 \rightarrow 3$, $5 \rightarrow 4$, $7 \rightarrow 6$ is monotone. The routing problem $2 \rightarrow 6$, $3 \rightarrow 5$, $6 \rightarrow 7$ is *not* monotone, because the relative order of the packets originating at nodes 2 and 3 is reversed by the routing.

(b) Give an $O(\lg N)$-time algorithm for monotone routing on an N-input butterfly.

(c) Give an $O(\lg^2 N)$-time algorithm for routing an arbitrary permutation on an N-input butterfly.

Solution

(a) *This problem was tricky. The packing algorithm works only if the butterfly is traversed from the minor to the major cycles, but not vice versa.*

The solution we give will work in two phases. In the first phase we compute the destination node of every packet. In the second phase we route the packets to their destination.

Finding the destination of a packet is easy. Let p_i be 1 if processor i has a packet, and 0 otherwise. Then, if processor i has a packet, the destination node of the packet is $d_i = \sum_{j=0}^{i} p_i$. In other words, the sequence of the d_i's is just the $+$-scan of the sequence of the p_i's. Recalling that a butterfly network embeds a binary tree (indeed, it embeds many), the destinations can be computed in time $O(\lg n)$ using the algorithm explained in class.

Once the destinations nodes have been computed, it remains only to route the packets to their destinations. Amazingly enough, the greedy algorithm explained in class will work fine with no collisions (recall that the greedy algorithm corrects one address bit at the time). At first glance, it seems surprising that the greedy routing of a packing problem is collision-free. Indeed, this result is only true if we route the packets from level $\lg N$ (minor cycles) to level 0 (major cycles). In fact, packing is one of the rare examples where the order of traversals of the levels of the butterfly really matters.

The proof that no two packets enter the same node (and hence there are no collisions) is by induction on the number of levels. The base case is obvious. We shall now prove that no two packets enter the same node at level $\lg N - 1$ of the butterfly (whence the proof follows by induction). In order for two packets to enter the same node at level $\lg N - 1$, they must come from two consecutive nodes at level $\lg N$. This means that the two packets are going to two consecutive destination nodes, and thus one is destined to an odd row, and the other goes to an even row. Recall now that the greedy algorithm adjusts one bit at the time. In our case, the greedy algorithm first adjusts the low-order bit (since we are going from minor cycles to major cycles). This means that one of the two packets will be routed to an odd row at level $\lg N - 1$, and the other will be routed to an even row. Hence, they cannot collide at this level.

(b) A **spreading** problem is the reverse of packing: there are M packets at level 0 of the butterfly, and they need to go to destinations at level $\lg N$, preserving the relative order of the packets. The greedy algorithm will route any spreading problem in time $O(\lg N)$, as for packing. Indeed, the path packets follow in the spreading is the reverse of the one they follow in packing problem, hence there are no collisions for spreading, either.

Monotone routing can then be done as follows: start with the packets in the minor cycles. Perform a packing (ending with packets in the major cycles). Spread the packets back to their destination. The total running time is $O(\lg N)$, since this is the time required by both packing and spreading.

(c) Our solution works by divide and conquer, using packing as a subroutine. The idea is the following. First, pack together all the packets that are going a destination in the upper half of the butterfly (that is, whose most-significant destination bit is 0). Then, route these packets down to the same row at level $\lg N$. Do the same thing to the packets whose most-significant destination bit is 1. At this point, all the packets destined to the upper half are in the upper half, and all the packets whose destination is in the lower half are in the lower half. We then apply the algorithm recursively to both halves, in parallel.

The algorithm fixes one bit at the time, and requires time $O(\lg N)$ per bit. Hence, the total running time is $O(\lg^2 N)$.

Problem 1-4. Sorting on a matrix

Do Exercise 28.5-6 on page 650 of CLR.

Solution Somewhat surprisingly, the output will appear in snakelike order after $O(\lg N)$ phases, where $N = m^2$ is the total number of elements in the array. **Snakelike order** means that, if one reads the first row left to right, then the second row right to left, then the third row left to right, and so on, the elements will be sorted[1]. This algorithm is known as *Shearsort*.

The proof that Shearsort completes the sorting in $O(\lg N)$ can be done by applying the 0–1 Sorting Lemma. Since the algorithm is oblivious algorithm, we only need to check that it correctly sorts any set of 0s and 1s.

> *Many people were confused on this point. Let me recall some things you already know, and try to clear the confusion.*
>
> *A* **comparison-based** *algorithm can never look at the data. All it can do is to compare two items and get back an answer (smaller, equal, greater). Many well-known algorithms (like quicksort) are comparison-based. The $\Omega(n \log n)$ lower bound for sorting holds for comparison-based algorithms. However, being comparison-based is* not *sufficient for the 0–1 principle to hold.*
>
> *The 0–1 principle works for* **comparator networks**. *The important property of networks is that they always do the same comparisons, in the same order, no matter what the input is. Algorithms that enjoy this property are called* **oblivious**, *and they can actually be implemented as networks, even if we describe them in words or in a programming language.*
>
> *An example of non-oblivious comparison-based algorithm is, again, quicksort: the sequence of comparisons depends on the result of previous comparisons, and there is no way to implement it in a network whose wires are fixed.*
>
> *The algorithm in the problem can be implemented as a network, if desired: the row and column sort can be done in an oblivious fashion, and the algorithm makes no choices (there is no `if` statement). This is why the 0–1 principle can be applied.*

Let's call **clean** those rows that consist only of zeros or only of ones, and **dirty** those rows that contain both zeros and ones. The idea of the proof is to show that each phase of the algorithm cleans at least half of the dirty rows. The bound on the number of phases then follows immediately.

To see why it is true that that each phase of the algorithm cleans at least half of the dirty rows, divide all the rows of the matrix into three regions: an upper region of all-0 rows, a lower region of all-1 rows, and a middle region containing the rest of the rows. The following

[1] Erudite hackers and philologists call this order "boustrophedon", from the Greek word $\beta o \upsilon \sigma \tau \rho o \varphi \eta \delta \acute{o} \nu$, *turning like oxen in ploughing*. This is the early Greek manner of writing, which went from left to right, and right to left, alternatively. For example, Solon's laws were written in this way.

matrix is an example:

$$\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}$$

In the example, the upper region consists of the first row, and the lower region of the bottom three rows.

Next, group the dirty rows into pairs, and examine the effect of sorting the rows. There are three possible outcomes of each pair, depending on whether the pair contains more 0s or 1s or an equal number of both. The three possibilities are shown below:

$$\begin{array}{cccccccccc}
0 & \cdot & \cdot & \cdot & \cdot & 0 & 1 & \cdot & \cdot & 1 \\
1 & \cdot & \cdot & 1 & 0 & \cdot & \cdot & \cdot & \cdot & 0
\end{array}
\qquad
\begin{array}{cccccccccc}
0 & \cdot & \cdot & 0 & 1 & \cdot & \cdot & \cdot & \cdot & 1 \\
1 & \cdot & \cdot & \cdot & \cdot & 1 & 0 & \cdot & \cdot & 0
\end{array}
\qquad
\begin{array}{cccccccccc}
0 & \cdot & \cdot & \cdot & 0 & 1 & \cdot & \cdot & \cdot & 1 \\
1 & \cdot & \cdot & \cdot & 1 & 0 & \cdot & \cdot & \cdot & 0
\end{array}$$

There are also three possibilities corresponding to the reverse of those just described, depending on whether the first row of the pair is sorted in increasing or decreasing order.

After sorting the rows, we sort the columns. Since any sorting algorithm for the columns will end up with the same result, we can choose a sorting algorithm that makes our proof easier.

> *It is not sufficient to say "we have more zeros than one and therefore after sorting the columns we will create a new row of zeros". The problem is that the zeros in the pair could be scattered across different rows.*
>
> *There are many ways to prove the abovementioned fact. Here we do the following: assume that the column sort algorithm does something we like. Since any sorting algorithm will produce the same result, the assumption is without loss of generality.*

Thus, we will assume that the first step of the sorting algorithm consists in comparing items in the same column of paired rows, exchanging items if necessary. Once this is done, there are again three possibilities, depending on the number of 0s and 1s in the pair. These possibilities are illustrated below:

$$\begin{array}{cccccccccc}
0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\
1 & \cdot & \cdot & 1 & 0 & \cdot & 0 & 1 & \cdot & 1
\end{array}
\qquad
\begin{array}{cccccccccc}
0 & \cdot & \cdot & 0 & 1 & \cdot & 1 & 0 & \cdot & 0 \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{array}
\qquad
\begin{array}{cccccccccc}
0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{array}$$

As the next step in sorting the columns, let's move the all-0 rows to the upper region and all the all-1 rows to the lower region. Then, we finish sorting the columns in an arbitrary way.

Since at least one row of each pair is moved out of the middle region at each phase, we conclude that after $O(\lg N)$ phases all numbers are sorted, except for one row. One final phase suffices to sort this row.