Fixed Connection Networks

Algorithms on

Linear Arrays

and

Binary Trees

Introduction

- Linear Arrays
- Notions of global control, local control, pipelining, systolic computation.
- Bit and word computation model and relationships
- A linear array is a network where each processor is connected with bidirectional links to two of its neighbors (left and right). The outermost processors may have just one connection each and may serve for input/output purposes as well. Each processor has local program control and local storage. It is the simplest fixed connection network (no changes during time of the connections, underlying structure remains the same).
- Local program control is **simple** (few instructions, usually implementable in hardware). Local storage is small (few words and registers required). Input/output occurs at fixed and specified ports.
- At each step of a computation each processor

  (1) receives input from neighbors
  (2) inspects local storage
  (3) performs local computation
  (4) sends/generates output for its neighbors
  (5) updates local storage

- INCLUDE FIGURE OF A LINEAR ARRAY HERE
- A global clock is available and time is partitioned into **steps** (which are large enough to complete (1)-(5)) so that all processors operate **synchronously**. This form of computation is called **systolic computation**. (data pulses through the network the same way blood pulses through the blood vessels/heart).
- An array as the one described above is also called a **systolic array**.

First Algorithm on arrays: Sorting

Input: n keys $x_0, \ldots, x_{n-1}$ and an n processor linear array.

Output: $x_{j_0} < x_{j_1} < \ldots < x_{j_{n-1}}$. Both input and output come in/out of port 0.

Assumptions: Key values are words storable in local storage (word model). Comparison-and-exchange sorting algorithm (compare whole words).

Round 1 Processing / Sorting round.
Each processor:

(a) Accepts left input.

(b) Compares received input with stored value (default:MAXINT).

(c) Outputs larger one to the right processor (if first step, do not send MAXINT).

(d) Stores smaller value locally.

End of Round 1

Example goes here.

Time for Round 1 is $2n - 1$ steps.

Claim. i-th smallest key is stored in processor $i - 1$, $1 \leq i \leq n$.

Proof of Correctness. Observe operation of processor 0. It examines n keys in turn, holds the smallest key and output the remaining ones to its right processor. Processor 1 works similarly, ie. accepts $n - 1$ keys holds the smallest and output the remaining $n - 2$ ones to its right neighbor. By an inductive argument, i-th processor keeps i-th smallest key and passes $n - i$ largest keys to the right. Largest value reaches processor $n - 1$ at time step $2n - 1$ (n steps in a processor, $n - 1$ steps in transition).

Round 2. Output keys in sorted order.

How? Local vs Global control (need to know when to start Round 2). Use Counters??

Solution 1: As soon as Round 1 completes enter/issue a pass left 'mode'.

Solution 2: Each processor knows its index i and n. It counts input keys. As soon as counter + position = n, it starts passing to the left inputs it receives from the right.

Solution 3: Rightmost processor is UNIQUE. As soon as it receives data it starts passing left. Then the remaining processors are doing the same thing as well (as soon as they receive something from the right).

Solution 4: Each processor starts passing left as soon as no more input is received from the left.

Question: Which of the four alternatives/solutions is efficient and implementable (given restrictions on local control)?

S1: Interior processor does not know when round 1 has been completed (pass left requires more than local control storage).

S2: Counters required (extra hardware, n must be known in advance).

S3: Not efficient. Round 1 requires $2n-1$ steps, Round 2 requires additional $2n-1$ for a total of $4n-2$. One can save one step as the last step of round 1 instead of storing value locally sends it left immediately (i.e. $4n-3$ steps).

S4: More efficient: $3n-1$ steps. Entries are out on alternate steps (eg processor 0 starts sending left on $n+1$ time step).

Solution 5: Use "end of input marker"?

Review

$f(n) = O(g(n))$: there exists n_0 and constant $c > 0$ such that for all $n \geq n_0$, $f(n) \leq cg(n)$.

$f(n) = \Omega(g(n))$: there exists n_0 and constant $c > 0$ such that for all $n \geq n_0$, $f(n) \geq cg(n)$.

$f(n) = \Theta(g(n))$: there exists n_0 and constants $c, d > 0$ such that for all $n \geq n_0$, $dg(n) \leq f(n) \leq cg(n)$. Equivalently, if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

What if we want to sort N keys on P processors using a linear array where $N > P$?

Then we need to simulate N/P processors on a single one. This is ok if granularity of processors is sufficiently large so that a single processor has large enough memory (local storage) to accommodate the memory requirements of N/P processors.

Review

Fine-grain: processors with few registers

Coarse-grain: processor allow large memories

Conclusion: It is *easy* to convert an N processor algorithm into a P processor one $N >> P$.

Reverse is not easy i.e. there is no easy way to convert an 1-processor (sequential algorithm) into an N-processor one.

No General Way To Parallelize A Sequential Algorithm

Comparing k-bit words

Bit Model: Each processor performs operations on bits (ie bit processors not word processors are available). As opposed to the bit model, in the word model a comparison of a whole word takes a single step along the receipt/transmission of the result. In cases where word size is large and not fixed, the bit model is preferable as the word model is unrealistic.

Problem 1: Comparison of two k-bit words $a_1 \ldots a_k$ and $b_1 \ldots b_k$.

Linear Array. Algorithm A. A k-processor linear array is utilized. Bits a_i and b_i are stored and compared in the i-th processor (whose index is $i-1$). On the ith processor the result of the comparison regarding $a_1 \ldots a_{i-1}$ and $b_1 \ldots b_{i-1}$ is combined with the result of b_i and a_i so that the result related to $a_1 \ldots a_i$ and $b_1 \ldots b_i$ be forwarded to processor i holding a_{i+1}, b_{i+1}.

Example here

$T = k$ and $P = k$. The result of the comparison becomes available to all processors in additional $k-1$ steps for a $T = 2k - 1$ and $P = k$ overall.

By using this network for comparison, the linear array for sorting words becomes a two dimensional array of dimension $k \times n$, where k is word size and n the number of keys. Then time for sorting becomes $T = O(kn)$ and $P = O(kn)$.

Complete Binary Tree on k leaves. Algorithm B. Instead of using a linear array for comparison we can use a binary tree (cf PRAM algorithm). Each of the k leaves (i leaf) contains a_i, b_i and a comparison of the two bits is performed at that leaf. The outcome of the comparison is $L, R, =$. Then, the result is propagated up (right) towards the root. The rules of propagation are as follows. If result of left subtree is $=$ the result of the tight subtree is propagated. Otherwise the result of the left subtree is propagated (left subtree== most significant bits of word).

Example here

Time to compare is $T = \lg k + 1$ and $P = O(k)$. In additional $\lg k$ steps the root broadcasts the results to the leaves. Total time is $T = 2 \lg k + 1$ and $P = 2k - 1$.

Between Algs A and B, Algorithm B is faster althought the size of network for A is smaller by a factor of 2.

Put Both Examples for A and B here.

Sorting on the Bit Model

We sort on the bit model by utilizing the tree network described before. We start with the word model sorting network. We replace each word processor by a tree of k leaves (to utilize network of Algorithm B). We connect the leaves in a lineary array-like fashion as before.

Example here

This way one step of the word model requires in the bit model $2\lg k - 1$ steps.
The original word model network had: $P = n$ and $T = 2n - 1$ steps for Round 1 ONLY.
The bit model requires $P = n(2k-1)$ and $T = (2n-1)(2\lg k - 1)$ for Round 1.
The time complexity of Round 2 remains the same.

Had we used the lineary array for comparison (network of Algorithm A) we would have needed $P = nk$ and $T = O(nk)$.

Can we do better?? YES. Use network of Algorithm A and pipelining

Lower bounds for sorting on arrays

- **Low input output bandwidth.** On a $k \times n$ array only k processors receive input. In order to read all kn bits we need time $\Omega(kn/k) = \Omega(n)$ (lower bound for sorting).
- **Large diameter of the network**. A $k \times n$ array has diameter $n+k-2$. How can we prove a lower bound based on diameter arguments? If two processors are separated by distance d then just to allow them to communicate requires time d. If we can prove that for some input such a communication is required $\Omega(d)$ is established if $d = D$ (D diameter).

How can we show such a result for sorting? Take input $x_1, \ldots x_n$, where $x_1 = a0\ldots01$ and $x_2, \ldots, x_n = 01\ldots0$. The lsb of the largest number is 1 if and only if $a = 1$ which means contents of processor $(1,1)$ need to be communicated to processor (k,n) ie in time $D = n+k-2$ and a corresponding lower bound for sorting is thus established.

- **Small bisection width** (find bottleneck). A $k \times n$ array has bisection width $\min k, n$. If we can establish that data on the left of the bisection must be communicated to the right part through the bisection width we can establish a lower bound of $\Omega((n/2 \cdot k)/bw)$. For example if $nk/2$ bits are communicated and the bisection width is k a lower bound $\Omega(n)$ can be shown.

One can prove lower bounds using any combination of these techniques. One needs, however, to be very cautious with the type of results one establishes as it is illustrated in the following section.

A counterexample using counting

We are going to use counting (bypassing the lower bound assumptions) a better upper bound can be proved. In order to solve the problem we introduce some simpler ones.

Problem 1: Counting 0 and 1's in the word model.

As in the PRAM, a cbt on k leaves can be used to find the parallel sum of k bits in the word model in $\lg k$ steps. In $\lg n$ steps the sum arrives in the root. In additional $\lg n$ steps the sum is disseminated to the leaves (NOTE: an internal processor is allowed to communicate the sum to its two children in a SINGLE step).

Problem 2: Counting 0 and 1's in the bit model.

Sum n bits in $2 \lg n$ bit steps using bit processors. Each node of the cbt becomes a bit processor. Each such bit processor has two input lines (one per child) and one output line (towards its father). The operations performed by each bit processors are outlined below. Let x and y be the inputs from its two children and z a local register initialized to 0. Then a bit processor sends to its father lsb(x+y+z) and stores into z msb(x+y+z).

Bit processor structure here

It is straightforward that if in input lines x and y are presented two binary numbers respectively, lsb first, the output of the bit processor is the sum of these two number lsb first.

Claim: The root of the binary tree whose nodes consist of bit processors as described above add n one-bit numbers.

Proof of Claim. The proof is by induction on the level of the tree. The base case is proved by the description of the bit processor. Let a $n/2$-leaf subtree add $n/2$ one-bit numbers. The root processor is then a binary number adder and by the observation before the statement of the claim the result follows for the n-leaf binary tree.

The total time of the algorithm is $2 \lg n$ steps as the lsb bit of the sum appears in $\lg n$ steps after the start and the msb appears at most $\lg n + 1$ steps later (if sum is n). We assume that attached to the root processor is a linear array of size $\lg n$ to hold the bits of the sum.

Problem 3: Sort n bits on a binary tree on the bit model.

We first peform the counting step and store in the linear array attached to the root the at most $\lg n + 1$ bits of the sum ($\lg n$ bits in the linear array, msb stored in the root processor's register). Then an operation is initiated that slides the sum from the root towards the leaves, msb first. The operations performed are as follows.

- Strip current msb off the sum sequence.
- If it is 1 send remaining bits to the left and a "make all 1's" to the right.
- If it is 0 send remaining bits to the right and a "make all 0's" to the left.
- If a processor receives "all 1's" sends "all 1's" to both its children.
- If a processor receives "all 0's" sends "all 0's" to both its children.
- If a leaf received "all 1's" sets its output register to 1 otherwise to 0. The value of the output register is the outcome of sorting.

There is one case that needs to be handled separately. If there are n 1's, $\lg n + 1$ bits are required to represent the sum, where the msb, which is 1, will be stored in a register of the root bit processor. As soon as this is detected, the root processor sends "all 1's" to both its children and the algorithm above is used for the remainder.

Total time is $T = 4 \lg n + O(1)$.

NOTE. During this algorithm $O(\lg n)$ bits are needed to be moved from one part of the tree to the other as opposed to $\Omega(n)$ of the lower bound. We circumvent the lower bound by using counting.

Odd-Even transposition Sort

Odd-even Transposition sort is sometimes referred to as **bubble-sort**.

Input: n keys $x_0, \ldots, x_{n-1}$ and an n processor linear array. Key x_i resides in processor i.

Output: $x_{j_0} < x_{j_1} < \ldots < x_{j_{n-1}}$. Key x_{j_i} resides is processor i.

Description of the Algorithm

At (odd) steps $i = 1, 3, 5, \ldots$

Processor i compares its key to that in processor $i+1$. If it larger, keys are exchanged so that the minimum is stored in processor i.

At (even) steps $i = 2, 4, 6, \ldots$

Processor i compares its key to that in processor $i+1$. If it larger, keys are exchanged so that the minimum is stored in processor i.

A proof of the correctness of odd-even transposition sort would utilize the following lemma. The lemma below is applicable to all comparison-exchange sorting algorithms that are oblivious (cells compared are not dependent on the results of other comparison-exchange operations).

0-1 Sorting Lemma. If an oblivious comparison-exchange sorting algorithm sorts all input sets of 0's and 1's then it sorts all input sets with arbitrary values.

Outline of proof of correctness of odd-even transposition sort Algorithm runs in n steps on an n processor linear array. It suffices to show that algorithm sorts any sequence of 0's and 1's. Let it be such a sequence containing exactly k 1's.

Proof utilizes the following observation.

Observation. The i-th rightmost one will move right at step $i+1$ and will keep on moving until step n or until it reaches its destination (cell $n-i+1$). As it moves right $n-i$ steps, it does reach cell $n-i+1$ by step n.

- Add two n-bit binary numbers in $2\lg n + 1$ steps using an n-leaf c.b.t.

Sequential algorithm requires n steps.

	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
a	0	1	0	1	1	1	0	0	1	0	0	1	0	0	1	0	
b	0	1	1	0	1	0	0	0	0	1	0	1	1	1	0	0	
	s	g	p	p	g	p	s	s	p	p	s	g	p	p	p	s	s
c																	
$a+b$																	

Table 1: Binary addition example

$(a+b)_i = a_i \oplus b_i \oplus c_{i-1}$, where $\oplus = XOR$.

- s: stops a carry bit $(0+0)$
- g: generates a carry bit $(1+1)$
- p: propagates a carry bit ($0+1$ or $1+0$).

Problem: In order to compute the k-th bit the $k-1$-st carry needs to be computed as well. There exists a non-trivial non-obvious parallel solution.

We shall try for each bit position to find the carry bit required for addition so that all bit positions can be added in parallel. We shall show that carry computation takes $\Theta(\lg n)$ time on a binary tree with a computation that will be later generalized known as parallel prefix (ppf).

Observation. The i-th carry bit is one if the leftmost non-p to the right of the i-th bit is a g.

Question. How can we find i-th carry bit?

The previous observation takes the following algorithmic form.

Scan for $j = i, \ldots 0$
if p ignore else
if g carry=1 exit; else carry=0 exit;

Such a computation requires $O(n)$ time for $j = n$ (n-th bit).

Let the i-th bit position symbol (p, s, g) be denoted by x_i.
Then
$c_0 = x_0 = s$
$c_1 = x_0 \otimes x_1$
$c_2 = x_0 \otimes x_1 \otimes x_2$
$c_{16} = x_0 \otimes \ldots \otimes x_{16}$.
where

$\otimes$	s	p	g
s	s	s	g
p	s	p	g
g	s	g	g

Algorithm for parallel addition

Step 1. Compute symbol ($\{s, p, g\}$) for i bit in parallel for all i.

Step 2. Perform a parallel prefix computation on the n symbols plus 0-th symbol s in parallel where operator is defined as in previous table.

Step 3. Combine (exclusive OR) the carry bit from bit position $i - 1$ (interpret g as an 1 and an s as a 0) with the exclusive OR of bits in position i to find the i-th bit of the sum.

Steps 1 and 3 require constant time. Step 2, on a complete binary tree on n leaves would require $2 \lg n$ steps.
$T = 1 + 1 + 2 \lg n$. $P = 2n - 1 = O(n)$.

Example for ppf here

Phase 1 Each processor forwards its value to its father. The father combines the two received values and forwards the results to its own father. After $\lg n + 1$ steps each processor of the tree knows the combined result of all leaves of the subtree beneath it.

Phase 2 As each nonleaf receives inputs from below it passes value of left son to its right son (no global control is required, ie each processor begins phase 2 when it finishes its own phase 1).

Each nonleaf passes incoming values from above to both its sons.

In this phase each leaf cell concatenates/combines (using associative operator) an incoming value from above with its previously stored value to form its new value.

Claim. Any n leaf binary tree (not necessarily complete) of depth D performs parallel prefix in $2D + 1$ steps.

Proof of Correctness. By induction. $D = 1, n = 2$ case is trivial. Let us assume that algorithm works as claimed for an $n/2$-leaf binary tree.

Segmented Parallel Prefix

A **Segmented Parallel Prefix Computation** consists of a sequence of prefix operations that use the same operator but on dijoint sets of data.

Example $\{2,3\}$ $\{1,7,2\}$ $\{1,3,6\}$.

Create 2 3 |1 7 2|1 3 6.

And result is: 2 5|1 8 10|1 4 10.

Algorithm

1. Insert a barrier | between independent prefix problems.

2. Increase data length by the number of barriers inserted. Alternatively, double input symbols. A symbol is either an original symbol or an original symbol preceded by a barrier.

3. Redefine $\otimes$ as $\bar{\otimes}$ so that the new operator acts on barriers as well.

$\otimes$	b
a	$(a \otimes b)$

$\bar{\otimes}$	b	$\|b$
a	$(a \otimes b)$	b
$\|a$	$\|(a \otimes b)$	b