Fixed Connection Networks

Algorithms on

Linear, 2d and 3d Arrays,

and

Binary Trees

PART II

0-1 Sorting 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.

Proof. (by contradiction). Let the sorting algorithm fail on some input data $x_1, \ldots, x_n$. Let the sorted sequence on this set of input data be

$$x_{\pi(1)} < \ldots < x_{\pi(n)}$$

and the output of the algorithm be

$$x_{\sigma(1)}, \ldots, x_{\sigma(n)}$$

Let k *be the smallest value* such that $x_{\sigma(k)} \neq x_{\pi(k)}$. That is $x_{\sigma(i)} = x_{\pi(i)}$ for all i such that $1 \leq i < k$ and $x_{\sigma(k)} > x_{\pi(k)}$. Therefore there exists an $r > k$ such that $x_{\sigma(r)} = x_{\pi(k)}$.

We then define the following sequence of 0-1 inputs.

$$x_i^* = \begin{cases} 0 & \textit{if} \quad x_i \leq x_{\pi(k)} \\ 1 & \textit{if} \quad x_i > x_{\pi(k)} \end{cases}$$

We then examine the action of the algorithm on x_i^*, $1 \leq i \leq n$. If $x_i \geq x_j$ then it is also $x_i^* \geq x_j^*$ as well. The algorithm performs the same comparison-and-exchange operations on x_i and on x_i^* input. Therefore the output of the algorithm will be

$$x^*_{\sigma(1)}, \ldots, x^*_{\sigma(k-1)}, x^*_{\sigma(k)}, \ldots, x^*_{\sigma(r)}, x^*_{\sigma(n)} = 0 \ldots 01 \ldots 0 \ldots$$

which is a contradiction to the assumption that the algorithm works correctly on 0 and 1 inputs.

Sorting revisited: 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 is 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 is 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 0-1 Sorting Lemma.

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 that such a sequence contains 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.

Shear sort

A sorting algorithm on a 2d array is presented that works in $O(\sqrt{n}(\lg n + 1))$ steps.

The implicit assumption is that keys reside in processor registers, one key per processor. The algorithm works as follows. It requires $\lg \sqrt{n}$ phases. For row or column sorting it utilizes the odd-even transposition sort algorithm on linear arrays.

Shear_Sort

In phases $1, 3, 5, \ldots, 2\lg\sqrt{n} + 1$ do
(1) sort rows alternately in increasing (minimum left) and decreasing order (minimum right).

In phases $2, 4, 6, \ldots, 2\lg\sqrt{n}$ do
(2) sort columns.

Theorem. Algorithm ShearSort works as claimed.

Proof. The proof utilizes the 0-1 sorting lemma and an inductive argument.
Claim 2. At least half of the rows are sorted after one application of row/column sort.

Let "dirty" be those rows which are not "all 1's" or "all 0's" rows. Initially, there are $\sqrt{n}$ dirty rows. If number of dirty rows is 1 the sequence is sorted (if necessary apply row sort on the single dirty column to orient it properly).

For the dirty rows we claim the inductive hypothesis.

(a) After applying a single round of 1-2 we have from top to bottom an all 0 area, dirty rows and an all 1 area (this is feasible by way of step 2 columnsort).

(b) Dirty rows are then grouped into consecutive pairs and examine the effect of sorting the rows (3 sets of cases, 3 extra ones if top row on an odd address instead of an even one).

Shear sort continued

0	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1
1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0
more 0's								more 1's								equal number of 0's and 1's							

After columnsort we get

0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1 dirty, 1 all 0's row								1 dirty, 1 all 1's row								no dirty rows							

Therefore at least half of the rows are all 0's and all 1's and at most half are dirty rows. After $\lg \sqrt{n}$ steps of 1-2 all rows are sorted except for one row which is sorted by row-sort (odd-even transposition sort) in the appropriate direction.

Sorting optimally (asymptotically)

There is an $3\sqrt{n}+o(\sqrt{n})$ step algorithm which is more complicated and consists of 8 phases some of which utilize the previous algorithm. We shall show instead a simpler algorithm whose running time is $O(\sqrt{n})$ and the hidden constant is around 6 but is easier to understand. Note that one can prove an interesting lower bound of $3\sqrt{n} - \Theta(n^{1/4})$ which matches the upper bound up to low order terms.

Optimal Algorithm

A. Split the array into four equal size quadrants.

1. Sort each qudrant recursively in snake-like order.
2. Sort rows in alternate directions (min-left, min-right).
3. Sort columns (min-top).
4. Do $4\sqrt{n}$ steps of snake-order bubble sort.

Running time. Let the time to sort keys of a $2^i \times 2^i$ qudrant be $T(2^i)$. Then $T(2^i) = T(2^{i-1}) + 2^i + 2^i + 4 \cdot 2^i = T(2^{i-1}) + 6 \cdot 2^i = O(2^i) = O(\sqrt{n})$.

Proof by example.

Example

After phase 1 most of the (whole) rows (except 4 boundary ones) are half 0's half 1's rows. After step 3 all such pairs of such rows have been sorted. What is left are possibly 4 dirty rows. Step 4 handles this case.

Note that in general, the 0-1 sorting lemma does not imply that the all 0's portion of the matrix would necessarily contain keys in sorted order. It only guarantees that no two keys are more than $4\sqrt{n}$ from their position in the sorted sequence.

Addition of n k-bit numbers: CLA and CSA

CLA Addition of two k-bit numbers requires $O(\lg k)$ time on a $O(k)$-processor complete binary tree with k leaves. The algorithm utilizes parallel prefix computation on a suitable associative operator and is also known in the literature as **carry-lookahead addition** (CLA).

CSA We show how to add n k-bit numbers by an algorithm known as **carry-save** addition (CSA).

A non-optimal algorithm A for CSA. *CSA-A*

One way to perform CSA addition is to use CLA addition. The algorithm works in $\lg n$ stages. In the first stage, the n numbers form $n/2$ pairs. The two k-bit numbers of each pair are added to produce a $(k+1)$-bit sum. The $n/2$ results then form $n/4$ pairs and this processor is repeated until a single number is generated.

Observation 1. Algorithm CSA-A requires $\lg n$ stages.

Observation 2. In stage $i \leq \lg n$ of Algorithm CSA-A addition of two numbers of $k+i-1$ bits each is performed.

Observation 3. Addition of two numbers of $k+i-1$ bits each on a binary tree of $O(k+\lg n)$ leaves requires parallel time $O(\lg (k+\lg n))$.

As a consequence of the three observations the following is concluded.

Claim 1. Algorithm CSA-A requires time $T = O(\lg n(\lg (k+\lg n))) = O(\lg n \lg k + \lg\lg n \lg n)$. Processor requirements are $P = O(nk)$.

Carry-save addition - The details

An optimal algorithm B for CSA. *CSA-B*

Idea. In order to add 3 k-bit numbers it suffices to add 2 $(k+1)$-bit numbers.

The sum of three k-bit numbers is written in terms of two numbers $(k+1)$-bit numbers in such a way that is illustrated below.

		08	07	06	05	04	03	02	01	
a		1	0	1	1	0	0	0	1	
b		0	1	1	0	1	1	0	1	
c		1	0	1	0	0	1	0	0	
d		0	1	1	1	1	0	0	0	$a \oplus b \oplus c$
e	1	0	1	0	0	1	0	1	0	shifted carry bits

The sum $a+b+c$ is equal to the sum of $d+e$ where $d = a \oplus b \oplus c$ and e is the sequence of carry bits shifted left one bit position. This is because the sum of three bits can be expressed as a two-bit binary number ie $a_1 + b_1 + c_1 = e_1 \cdot 2 + d_1$.

The algorithm works as follows:

(1) The n k-bit numbers are split into $n/3$ groups of three numbers each. Each triplet is reduced to the sum of two $(k+1)$-bit numbers.

(2) As a result, $2n/3$ $(k+1)$-bit numbers are derived. They are groupped in $2n/3^2$ triplets and step (1) is repeated to derive $2^2 n/3^2$ $(k+2)$-bit numbers.

(F) This way, starting from n k-bit numbers, finally, 2 (at most) $(k + \lg n)$-bit numbers are obtained in $\lg_{3/2} n + 1$ parallel steps.

At the final step, the 2 numbers are added using the CLA algorithm in time $O(\lg(k + \lg n) = O(\lg k + \lg\lg n)$.

Each one of the stages requires $O(kn)$ processors to perform the transformation in constant time for a total of $O(\lg n)$ time for all stages. The final step requires $O(\lg k + \lg\lg n)$ processors.

Therefore the total running time and processor requirements of the algorithm are $P = O(nk)$ and $T = O(\lg k + lgn)$.

Multiplication of 2 n-bit numbers

06	05	04	03	02	01	-
			b_3	b_2	b_1	
			a_3	a_2	a_1	
			b_3	b_2	b_1	$\times a_1$
		b_3	b_2	b_1		$\times a_2$
	b_3	b_2	b_1			$\times a_3$

From the table above, in order to multiply 2 n-bit numbers it suffices to add n at most $2n$-bit partial sums using the standard elementary-school algorithm. By using algorithm CSA-B this would require $T = O(\lg n)$ time on $P = O(n^2)$ processors.

Question CAN WE MULTIPLY 2 NUMBERS USING FEWER PROCESSORS?

Matrix-vector and Matrix multiplication

Given an $n \times n$ matrix $A = (a_{ij})$ and a vector $\vec{x} = (x_j)$, matrix vector product Ax is defined to be the a vector $\vec{y} = (y_i)$ such that $y_i = \sum_{j=1}^{n} a_{ij}x_j$, for $1 \leq i \leq n$. Sequentially, $2n^2 - n$ operations are required. An n-cell linear array requires $2n - 1$ steps. The input is fed through cell 0 (x_1 first) and row i is fed into cell $i - 1$ (element of first column first) with a delay of $i - 1$ steps.

The requirements of the algorithm are $P = n$ and $T = 2n - 1$.

Given $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ the product $C = A \times B$,where $C = (c_{ij})$ is defined as follows: $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. Matrix multiplication consists of n matrix-vector products and it requires $2n - 1 + n - 1 = 3n - 2$ steps on an $n \times n$ 2d array. Matrix A is input as before. For matrix B its i-th column is input to the i linear array with a delay $i - 1$ (first row element first (cell (j, i) computes $a_{ik}b_{kj}$ at step $i + j + k - 2$).

The efficiency of the algorithms can be improved by a constant factor if data are in place in the beginning of the algorithm and wrap-around edges are added, or alternatively, pipelining is used to solve multiple problems.

Example: Matrix-vector multiplication.

Example: Matrix multiplication.