PART II

Binary Trees

and

Linear, 2d and 3d Arrays,

Algorithms on

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

\[ (\gamma)^{\\overline{x}} x < (\gamma)^{\overline{x}} x < (\gamma)^{\overline{x}} x < \ldots \]

Therefore the output of the algorithm performs the same comparison-exchange operations on \( x \) and on \( \overline{x} \). Then it is also as well. The algorithm then examine the action of the algorithm on \( \overline{x} \), \( x \leq \overline{x} \leq 1 \).

\[ \begin{cases} (\gamma)^{\overline{x}} x < \overline{x} \leq x \leq 1 \quad (\gamma)^{\overline{x}} x < \overline{x} \leq x \leq 1 \quad \text{for all } \gamma \text{ such that } (\gamma)^{\overline{x}} x = (\gamma)^{\overline{x}} x \end{cases} \]

\[ \begin{cases} (\gamma)^{\overline{x}} x = \overline{x} \leq x \leq 1 \quad (\gamma)^{\overline{x}} x = \overline{x} \leq x \leq 1 \quad (\gamma)^{\overline{x}} x = \overline{x} \leq x \leq 1 \end{cases} \]

Therefore the output of the algorithm be

\[ (\gamma)^{\overline{x}} x > \cdots > (\gamma)^{\overline{x}} x \]

Proof. (By contradiction.) The sorting algorithm fails on some input data \( x, x', \ldots \). Let the sorted sequence on this set of input data be

0-1 Sorting Lemma 0-1 sorting algorithm sorts all input sets of 0's and 1's, then

The lemma below is applicable to all comparison-exchange sorting algorithms that are oblivious (cells compared are

Linear Arrays
By step 1, processor i can move to the right, so that it's left is larger. It's also possible for processor 1 to move to the right, so that it's left is larger. The number of steps is 2, 4, 6, ... for processor 1. For processors 2, 3, 4, this is the case for all odd steps. For processors 2, 3, 4, this is the case for all even steps. A sequence of steps is 1, 3, 5, ..., 2n-1, n = 1, 2, 3, ..., where n is the number of processors. This sequence is the same as the sequence of steps for bubble sort. A linear array is sometimes referred to as a bubble-sort.
extra one! If row on an odd address is used instead of an even one.

(b) Dirty rows are then grouped into consecutive pairs and examine the effect of sorting the rows (3 sets of cases, 3 cases).

is feasible by way of step 7 column sort.

After applying a single round of 1-7 we have from top to bottom an all O area, dirty rows and an all 1 area (this

For dirty rows we claim the following hypotheses:

dirty rows is the sequence is sorted (if necessary apply row sort on the single dirty column to obtain it properly).

Let „dirty” be those rows which are not all 1’s or all 0’s. Initially, there are √n dirty rows. If number of

Claim 2. All least half of the rows are sorted after one application of row/column sort.

Proof. The proof utilizes the 0-1 sorting lemma and an inductive argument.

Theorem. Algorithm ShearSort works as claimed.

2 sorts columns

In phases 2, 4, 6, . . . , 2|\sqrt{n}| do

1 sorts alternately in increasing (minimum left) and decreasing order (minimum right).

In phases 1, 3, 5, . . . , 2|\sqrt{n}| + 1 do

Shear Sort

Linear Arrays. If requires |\sqrt{n}| phases. For row of column sort it utilizes the odd-even transposition sort algorithm on

The implicit assumption is that keys reside in processor registers, one key per processor. The algorithm works as

A sorting algorithm on 2d array is presented that works in O(|\sqrt{n}| + 1) steps.
Rows are sorted except for one row which is sorted by row-sort (odd-even transposition sort) in the appropriate direction.

Therefore at least half of the rows are all 0's and all 1's and at most half are dirty rows. After \( \log^2 n \) steps of 1-2 all

<table>
<thead>
<tr>
<th>Dirty rows</th>
<th>1 dirty, 1 all 1's row</th>
<th>1 dirty, all 0's row</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1</td>
<td>1 1 1 1 1 1 1</td>
<td>1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0</td>
<td>0 0 0 1 1 0 0</td>
<td>0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

After column sort we get

<table>
<thead>
<tr>
<th>Equal number of 0's and 1's</th>
<th>More 1's</th>
<th>More 0's</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 1 1</td>
<td>0 0 0 1 1 1</td>
<td>1 1 1 0 0 0</td>
</tr>
<tr>
<td>1 1 1 1 0 0 0</td>
<td>1 1 1 1 1 0</td>
<td>1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

Shear sort continued
sequence.

containing keys in sorted order. If only guarantees that no two keys are more than $\sqrt{n}$ from their position in the sorted

Note that in general, the 0-1 sorting lemma does not imply that the $0's$ position of the middle would necessarily

of such rows have been sorted. What is left are possibly 4 dirty rows. Step 4 handles this case:

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 parts

Example

Proof by example:

\[ (\mathbb{W}/\mathbb{O}) O = (\mathbb{O}/\mathbb{Z}) O = (\mathbb{Z}/\mathbb{L}) L = (\mathbb{L}/\mathbb{Z}) L. \]

Running time. Let the time to sort keys of a $\mathbb{Z}/\mathbb{L}$ quadrant be $T$. Then $O = (\mathbb{W}/\mathbb{O}) O$.

• Do $4\sqrt{n}$ steps of snake-order bubble sort.
• Sort columns (min-top).
• Sort rows in alternate directions (min-left, min-right).
• Sort each quadrant recursively in snake-order.
• Split the array into four equal size quadrants.

Algorithm

matches the upper bound up to low order terms.

is around 6 but is easier to understand. Note that one can prove an interesting lower bound of $3\sqrt{n} - \Theta(1/n)$ which

previous algorithm. We shall show instead a simpler algorithm whose running time is $O(\sqrt{n}/\mathbb{W})$ and the hidden constant

There is an $\Theta(n^2)$ step algorithm which is more complicated and consists of 5 phases some of which utilize the

Sorting optimally (asymptotically)

2D Arrays
Claim. Algorithm CSA-VA requires time $O(n)$.

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

Observation 1. Algorithm CSA-VA requires time $O(n)$.

Observation 2. A non-optimal algorithm $A$ for CSA addition is to use CLA addition, the algorithm works in $\alpha n$ stages. In the first stage, the

Observation 3. Addition of two numbers of $\alpha + \beta$ bits each on a binary tree of $O(\log n)$ leaves requires parallel

Then form $n/2$ parts and this process is repeated until a single number is generated.

The $2\alpha$-bit numbers of each part are added to produce a $\alpha + 1$-bit sum. The $n/2$ results

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

CLA addition of two $n$-bit numbers requires $O(n\log n)$ time on a $O(n \cdot \log n)$-processor complete binary tree with $\alpha$
Therefore the total running time and processor requirements of the algorithm are 0, and 0)O = 0.

Each of the steps requires O processors to perform the transformation in constant time, for a total of

\( (u + \gamma \beta) O = (u + \gamma \beta) O \)

At the end of the 2 steps, the 2 numbers are added using the CTA algorithm in linear time.

\( (u + \gamma \beta) O \)

Parallel steps.

\( \frac{u}{h} \) \( \frac{u}{h} \)

The algorithm works as follows:

<table>
<thead>
<tr>
<th>Sum of Carry Bits</th>
<th>( a \oplus q \oplus c )</th>
<th>( a \oplus q \oplus c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The sum of 3-bit numbers is written in terms of two numbers \( (\gamma + 1) \)-bit numbers. The idea is to add \( 3 \) \( \gamma \)-bit numbers to achieve this. The details of combinations of Networks for Carry- Save addition are:

\( (u + \gamma \beta) \)

\( (u + \gamma \beta) \)

\( (u + \gamma \beta) \)
Question: Can we multiply 2 numbers using fewer processors?

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 CSV-B this would require \( T = O(n^\log_2(7)) \) time on one processor. \( T = O(n) \) time on one processor. By using the algorithm CSV-B this would require \( T = O(n^2) \) time on one processor.

<table>
<thead>
<tr>
<th>( a \times )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a \times b )</td>
<td>( d \times e )</td>
<td>( c \times a )</td>
<td>( b \times c )</td>
<td></td>
</tr>
<tr>
<td>( d \times e )</td>
<td>( c \times a )</td>
<td>( b \times c )</td>
<td>( a \times b )</td>
<td></td>
</tr>
<tr>
<td>( e \times b )</td>
<td>( a \times d )</td>
<td>( c \times a )</td>
<td>( b \times c )</td>
<td></td>
</tr>
<tr>
<td>( c \times a )</td>
<td>( b \times c )</td>
<td>( a \times b )</td>
<td>( d \times e )</td>
<td></td>
</tr>
<tr>
<td>( b \times c )</td>
<td>( a \times b )</td>
<td>( d \times e )</td>
<td>( c \times a )</td>
<td></td>
</tr>
<tr>
<td>( a \times b )</td>
<td>( d \times e )</td>
<td>( c \times a )</td>
<td>( b \times c )</td>
<td></td>
</tr>
</tbody>
</table>

Multiplication of two n-bit numbers by binary trees.
Example: Matrix multiplication.

Matrix-vector multiplication.

Given an $n \times n$ matrix $A$ and a vector $x$, the product $Ax = C$ is defined as follows:

\[ (Ax)_i = \sum_{j=1}^{n} A_{ij}x_j \]

The requirements of the algorithm are

1. $A$ is an $n \times n$ matrix
2. $x$ is a vector of length $n$
3. $C$ is the vector where the product is computed

The efficiency of the algorithm can be improved by a constant factor if data are in place in the beginning of the array.

The first row element first (cell 1) computes the sum $a + b$ and step 1.

Example: Matrix-vector multiplication.

Matrix multiplication consists of $n$ matrix-vector products and is defined as follows:

Given $n \times n$ matrices $A$ and $B$, the product $C = AB$ is defined as follows:

\[ (C)_i = \sum_{j=1}^{n} A_{ij}B_{jk} \]

The efficiency of the algorithm can be improved by a constant factor if data are in place in the beginning of the array.

The first row element first (cell 1) computes the sum $a + b$ and step 1.

Example: Matrix multiplication.