A CHARACTERIZATION OF A PARALLEL ROW-COLUMN SORTING TECHNIQUE FOR RECTANGULAR ARRAYS

Isaac D. Scherson and Sandeep Sen,
Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106.

Index Terms

parallel sorting, complexity, upper bound, lower bound, area-time tradeoffs, parallel architecture, orthogonal memory, fixed-connection network, very large scale integration.

Abstract

In an $m \times n$ rectangular array, row-column sorting technique is shown to yield a snake-like ordered sequence. By characterizing the data movement under successive row and column sorts, we prove that the procedure converges in $O(\log_2 m)$ iterations. A direct application of this technique is an efficient bubble-sort algorithm, suitable for VLSI implementation, with near optimal area- time^2 performance. The inherent parallelism of row-column sort is discussed in light of a novel orthogonal access memory architecture.

I. Introduction

The problem of sorting numbers arranged on a two dimensional array has been studied by Nasssimi & Sahni[2], Thompson & Kung[1], Kumar & Hirschberg[10] and more recently by Leighton[3] and Lang et al.[8]. Each data item has to be routed to a distinct position of the array in a sorted sequence predetermined by some indexing scheme. Three different indexing schemes have been considered by Thompson & Kung[1] : row major, shuffled row-major and snake like row major (see fig 1). More recently Leighton[3] has used a column-major scheme which may be considered the same as a transposed row-major form. These sorting algorithms in rectangular arrays were based on two-dimensional adaptations of very powerful sorting schemes like bitonic sort([9]) in [1] and [2], and odd-even merge sort([9]), in [10] and they have been shown to perform optimally. However these algorithms involve some complex operations like data shuffles (in [1], [2], [8], [10]) and transposition of a rectangular matrix(in [3]). A seemingly obvious way of performing sorting in a real two-dimensional sense would be to sort rows and columns which unfortunately has been found to be ineffective when implemented in a straight forward manner. In a very recent paper Leighton[3] observes that '.. if the matrix were square, we would essentially just be sorting rows and columns which is well known to leave entries arbitrarily away from their correct sorted position." Obviously this was in reference to sorting rows and columns in directions imposed by the row-major indexing scheme. Paradoxically, with a row-major snake like indexing it is possible to obtain a sorted sequence by sorting rows and columns. We shall demonstrate that such an iterative procedure converges in a finite number of steps. It is easy to see that sorting rows in a row-major or a snake-like pattern has the same complexity since they differ only in the direction of sorting.

In other words, by sorting adjacent rows in opposite directions (one in ascending order from left to right and the other in descending order) and sorting the columns in ascending order from top to bottom, the elements tend to move closer to their final sorted positions. In the end we shall have a sorted sequence in snake-like row-major form, from which we can obtain a row-major form by simply inverting the alternate rows, an operation, that does not affect the asymptotic performance of the overall procedure.

This simple algorithm, which we call **row-column** sort will be formally introduced in the next section. Section III will provide the complexity analysis of the algorithm which is not as simple as the algorithm. In section IV we discuss a method for optimizing bubble-sort in VLSI using row-column sort. In section V we introduce a multiprocessor architecture suitable for implementing this algorithm and compare its performance with some relevant existing architectures.

II. Row-column sort

Let Q = [$q_{i,j}$] be an $m \times n$ matrix onto which we have mapped a linear integer sequence S. Sorting the sequence S is then equivalent to sorting the elements of Q in some predetermined indexing scheme. We suggest an iterative algorithm in which every iteration consists of two basic operations :

(1) Column-sort - Sort independently, in an ascending order from top to bottom all column vectors of Q. After this step $q_{i,j} \leq q_{i+1,j}$ for all j = 1,..,n .

(2) Row-sort - Sort independently all row vectors of Q such that adjacent rows are sorted in opposite directions (alternate rows in the same direction). In a normal snake-like row-major indexing scheme, sort the first row from left to right. At the end of this step, $q_{i,j} \leq q_{i,j+1}$ for all i = 1,3,5,..2p+1, and $q_{i,j} \geq q_{i,j+1}$ for all i = 2,4,6,..2p.

The row-column sort algorithm is defined as a repetitive application of steps 1 and 2 until one of the following terminating conditions is satisfied :

(a) all the columns are sorted i.e. no element has moved in the present column-sort **after a row sort** , or

(b) no element has moved in the present row-sort **after a column sort.**

A step-by-step application of row-column sort is showed in Fig. 2.

In the remaining of this section we shall prove that the row-column sort algorithm converges to a snake-like sorted sequence in a finite number of iterations . Furthermore, we shall show that the terminating conditions (a) and (b) above are necessary and sufficient. The actual complexity analysis is left for the next section because it involves a complicated but interesting development which deserves a separate discussion.

Theorem 1 The row-column sort algorithm terminates successfully after a finite number of iterations.

proof Consider the n smallest elements in Q and assume they are randomly distributed over the rows and columns of the array. The first column sort will move these elements to the first row if initially they all happen to be in different columns. The following row-sort will order them properly in the first row , where they will remain regardless of further row or column sorts. However, if all n smallest elements happen to be in the same column (which is only possible if $m \geq n$), the first column sort will order them in that same column. By virtue of the alternating sorting direction on the rows , a row-sort will have the effect of moving the elements on odd rows to the leftmost column of the array and the remaining half to the rightmost column. Clearly, at the beginning of the second iteration, a column sort will move the n smallest elements to the upper half of Q. The second row-sort will then pair the elements on the two left-most columns for odd rows, and on the two right-most columns for the even rows. Following the same reasoning, it is not difficult to see that the column sort of the p+1 iteration will move the n smallest elements of Q to the band defined from rows 1.. $\frac{n}{2^p}$ (recall we have assumed $m \geq n$). Without loss of generality we can assume n to be a power of 2 and conclude that the n smallest elements of Q will move to their sorted positions in at most log[1]n iterations.

It follows from the above discussion that for $m < n$ bringing the n smallest elements to their final sorted position will take atmost log m iterations.

It is evident that once the first row is in place it will remain there throughout the remaining iterations . Therefore, we face now a problem of sorting a reduced array Q_{-1} of dimension $m-1 \times n$. Each time a row is in its place, we reduce the problem to a smaller array on which the smallest elements are brought to the 'first' row in $\lceil \log(m-k) \rceil$ iterations where k is the number of previously discarded 'first' rows. The total number of iterations to sort the whole array Q thus becomes

$$\sum_{k=0}^{k=m-1} \lceil \log (m-k) \rceil$$

which is bounded from above by m logm and is a finite number. Q.E.D.

In proving theorem 1 we have assumed that after the 'first' row of array Q_{-k} is in place all the remaining elements are randomly distributed in the reduced array Q_{-k-1} . This is not the case as we will show in the next section which gives us a much better bound than what the theorem suggests. Nevertheless, we should allow the algorithm to terminate if the current permutation has achieved the desired snake-like row-major sorted sequence. This is the purpose of the terminating conditions (a) and (b) defined previously. It is obvious that in a snake-like sorted array, both rows and columns are sorted in directions imposed by this indexing scheme. Conversely, if the rows and columns are sorted, the array

1. Throughout this paper log will be assumed to be to the base 2 unless otherwise mentioned.

is sorted. Therefore, conditions (a) and (b) above are equivalent and are necessary and sufficient terminating conditions.

The reader may note that in snake-like row-major indexing scheme, an array is sorted if all the rows are sorted (in the required directions) and columns 1 and n are sorted from top to bottom. In fact, our proposed algorithm will also converge if the column-sort operation is restricted to the edge-columns only but we shall see in the next example that it does not pay in the overall number of iterations. To illustrate the row-column sort algorithm ,strengthen our observations, and motivate the reader to follow the complexity analysis in the next section, let us study two simple cases.

Example 1 : row-column sort in a $2 \times n$ array.

Consider only the first two rows of the array (fig 3). The arrows indicate the direction of sorting. Consider two elements q_{1j} and $q_{2,j}$ in these rows (they are in the same column). If $q_{1,j} \leq q_{2,j}$ then all elements to the left of $q_{1,j}$ are less than all the elements to the left of $q_{2,j}$. Clearly, all elements to the left of $q_{1,j}$ are less than $q_{1,j}$ from the direction of sorting and hence are less than $q_{2,j}$. Therefore, they are less than all the elements to the left of $q_{2,j}$ which are greater than $q_{2,j}$ from the direction of sorting. Consequently, if $q_{1,j}$ and $q_{2,j}$ are the elements in the last column all the elements of the first row are less than those of the second row and thus the sequence is correctly sorted in a snake-like row-major form. It only takes 1 iteration to converge. On the other hand if $q_{1,j} \geq q_{2,j}$,then all elements to the right of $q_{1,j}$ are greater than all the elements to the right of $q_{2,j}$. Further if $q_{1,j}$ and $q_{2,j}$ is the first such column-inverted pair (all pairs of elements to their left are in correct order) all the elements will be in their proper sorted rows after we swap all pairs of elements to their right. In context of m = 2 (two rows), a column sort is simply a compare exchange operation. With another stage of row-sort we will get a correctly sorted sequence. This means we need at most two iterations to converge. If we sort the edges alone we will take as many iterations more as there are elements to be swapped. Intuitively one can see that only two elements can change rows in each iteration and in the worst case where all elements have to get into the other row we will need at least $\frac{n}{2}$ iterations.

Example 2 : row-column sort in an $m \times 2$ array.

Let us consider now the special case of a $m \times 2$ array. We will designate the 2m elements as 'Light' or 'Heavy' depending on whether they belong to the upper $\frac{m}{2}$ rows or the lower $\frac{m}{2}$ rows in the final sorted array. After a column sort let us assume we have k 'Heavy' elements in one column and m-k 'Heavy' elements in the other column. The number of 'Light' elements are m-k and k respectively. The next row-sort will redistribute the elements such that we will have exactly $\frac{m}{2}$ 'Light' and 'Heavy' elements in each column(see fig 4). So after the next column sort we have the 'Light' and 'Heavy' elements in the proper halves. This argument can be applied recursively to show that we take O(log m) iterations to obtain a correctly sorted sequence. It is not easy to extend this simple analysis as we go on adding more rows or columns (to make up an $m \times n$ array) since it becomes increasingly difficult to keep track of the relationship between any two arbitrary elements of the array. In the next section we will show that the algorithm converges in O($\log_2 m$) iterations in the general case. For the sake of simplicity we will assume m and n to be a powers of 2. This won't harm the asymptotic performance since it will only affect the performance by a constant factor. Moreover for the convenience of the arguments we will consider all the elements to be distinct which will not affect the validity of the proof.(Knuth[6],pp 196 due W.G. Bouricius 1954)

III. Analysis of the algorithm

We shall prove that the row-column sort converges in O(log m) iterations, for an $m \times n$ array, by induction on the number of iterations it takes for an element to go within a specified distance of its final sorted row. It will be seen that the row distance for all elements decreases by a factor of 1/2 every 2 iterations. Once all elements are in their final sorted row (distance is zero), the procedure terminates with a row-sort. This brings all the elements to their final sorted positions in both rows and columns. The outline of the proof is as follows

(1) We shall first show that O(1) iterations (actually 2) leaves any element, and hence all elements, within $\frac{m}{2}$ rows of its final sorted row. Henceforth we will refer to the final sorted row of an element as its 'destination row'. This will form the basis for induction.

(2) We shall assume that an arbitrary element (and hence all elements) will be within $\frac{m}{2^p}$ of its destination row after O(p) iterations (actually 2p iterations) and prove that the same element will be within $\frac{m}{2^{p+1}}$ of its destination row in at most two more iterations. This will establish that the algorithm converges in O(log m) *iterations*[2]. Recall from the previous assumption that m is a power of 2, which does not affect the asymptotic performance.

Before we proceed with the actual proof let us define some terms formally to avoid ambiguity.

Definitions :

1 Each 'iteration' consists of (a) column-sort, followed by,

(b) row-sort.

2. Note that the number of iterations should not be confused with the actual number of steps of the whole process which is [m*(time for row-sort) + n*(time for column-sort)] * logm

2 An element is said to be within k rows of its 'destination' row r if the element is in a row $\bar{r}$ in the range r ± (k-1) .

3 An element 'E' in a column will be called even(odd) if it is in an even(odd) numbered row, the top row being assigned number 1.

We shall make use of a very interesting result, published by Gale & Karp[4], and stated again here for the sake of completeness.

Theorem 2: In a rectangular array, a row-sort preserves a column-sort when all rows are sorted in the same direction and all the columns are sorted from top to bottom.

An important implication for our analysis is stated below as Lemma 2.1.

Lemma 2.1:For the row-column sort algorithm, the elements in alternate rows in a column remain sorted after a row sort has been done following a column sort. In other words, the row sort does not destroy column sorting in alternate row positions.

proof : From theorem 2 we know that a row sort (here we mean all the rows being sorted in the same direction) does not destroy the column sort. Since all the even (odd) rows are sorted in the same direction independently of the other rows, we may consider two separate arrays of odd and even rows each of dimension $\frac{m}{2} \times n$. Therefore, the ordering of the elements in the even (odd) positions in any column is not tampered.

Our analysis of the algorithm is based on counting elements smaller(greater) than any given element in the array during various stages of the procedure. The following theorem provides important results which will be frequently used in the rest of this section.

Theorem 3 Let E be an element in column vector c after 'i' iterations where $i \geq 1$, and let n_e and n_o be the number of elements in even and odd rows in column c, respectively less than (greater than) 'E'. Let $N_{p,k}$ represent the minimum number of elements less than or equal to E that can be accounted for in the whole array. The indices p and k can assume values e(even) or o(odd) according to the following convention :

p = e implies $n_e \geq n_o$ else if p = e then $n_o \geq n_e$ and k = e denotes that E is in an even row of column c else if k =o, then E is in an odd row. Then the following hold for each of the four cases corresponding to the possible combinations of values of p and k : the cases :

$$N_{ee} = (2n_o - 1)(max\left\{c, n-c+1\right\})+2(n_e - n_o + 3/2)(n-c+1)$$

$$N_{oe} = (2n_e + 2)(max\left\{c, n-c+1\right\})+2(n_o - n_e - 3/2)(c)$$

$$N_{oo} = (2n_e)(max\left\{c, n-c+1\right\})+2(n_o - n_e + 1/2)(c)$$

$$N_{eo} = (2n_o + 1)(max\left\{c, n-c+1\right\})+2(n_e - n_o - 1/2)(n-c+1)$$

proof As it will be time and space consuming to go through all the cases individually, we shall concentrate on one of them(N_{ee}) knowing that the rest can be proved similarly. Fig 5 shows the situation after a column and row sort. The arrows indicate the direction of row-sort and E is in an even row. Since we have assumed that there are only n_o elements in the odd rows less than or equal to E, from Lemma 2.1 these must be in the first n_o odd rows. Similarly all the n_e ($\geq n_o$) elements must occupy the first n_e even rows which implies that E must be in 2 n_e + 2 row. Two cases arise according to max{c, n-c+1}.

Case : c ≤ n-c+1.

Consider the $2n_o^{th}$ row after a row sort (recall that an iteration ends with a row sort) and let E_e be an element in this row and column c. E_e is less than E from lemma 2.1 and so are n-c+1 elements to the right of E_e , from the direction of sorting (Fig 5). Now consider the situation just before the row sort. Since a row-sort permutes elements within rows, the n-c+1 elements less than E are already in row $2n_o$. As a column sort precedes a row sort,all columns are sorted, and there are at least n-c+1 elements less than 'E' in row $2n_o$, there must be at least n-c+1 elements less than E in the all the rows from 1.. $2n_o-1$. In the row where E is present, there are atleast n-c+1 elements less E and so the same number of elements less than E exist in all the the rows from $2n_o$ to $2n_e$ + 2. Adding these numbers we can account for ($2n_o$ -1)(n-c+1) + 2($n_e - n_o$ + 3/2)(n-c+1) elements less than E in the whole array.

Case : c ≥ n-c+1.

Now consider the $2n_o-1^{st}$ row. From the direction of sorting (odd row) there are at least c elements less than E. Following the same argument as above there will be at least c elements less than E in each of the rows above. Also we have n-c+1 elements less than E in the n_e + 2 even row and by an analogous argument we have n-c+1 elements less than E in each of the $2n_e$ .. n_o +2 rows. Adding the two components we have N_{ee} = ($2\,n_o$ -1)(c) + 2($n_e - n_o$ + 1)(n-c+1) Q.E.D.

Corollary 3.1 : The above results also hold for any subarray $k \times n$ of the entire array for $k \leq m$.

Theorem 3 is a representative of the many cases that may arise from the various positions of an arbitrary element in the array. To simplify our analysis and prevent unnecessary rigor, we shall work out representative cases. Nevertheless, we stress that the proof is complete, and the reader may satisfy himself by working out each individual case in a similar fashion. Moreover, we will only show the proof for E moving within $r + \frac{m}{2^p}$ rows in O(p) iterations ; the arguments for E moving within $r - \frac{m}{2^p}$ rows apply from symmetry by numbering the rows and columns in opposite directions, and keeping track of elements greater than E.

Corollary 3.1 : The above results also applies to any subarray $k \times n$ of the entire array for $k \leq m$.

PROOF OF THE INDUCTION BASIS

Consider an element 'E' in column c whose destination row is r such that there is no element below E in column c with the same destination row. Assume that E is in even row. We are considering the situation after one iteration. This element will not move within $\frac{m}{2}$ rows of row r in the next column sort if, and only if, there are r + $\frac{m}{2}$ −1 or more elements in column c less than E. We will prove by contradiction that this cannot be true. Assume that there are m_o and m_e elements in the odd and even rows less than E such that

$$m_o + m_e \geq r + \frac{m}{2} - 1 \tag{1a}$$

Without loss of generality assume $m_o \leq m_e$. Clearly $r \leq m_o$ and $m_e \leq \frac{m}{2} - 1$ since the maximum number of even (odd) elements in a column is $\frac{m}{2}$.

From Theorem 3, the number of elements less than E in the array is atleast

$$N = (2m_o - 1)(max\left\{c, n - c + 1\right\}) + 2(m_e - m_o + 3/2)(n - c + 1)$$

from case N_{ee}. Two cases arise according to max{ c, n-c+1}.

Case : If c $\geq$ n-c+1 then $c \geq \frac{n+1}{2}$ which implies $c = \frac{n+1}{2} + \varepsilon$ where

$0 \leq \varepsilon \leq \frac{n-1}{2}$ as $c \leq n$ (2a)

N can now be written as

$$N = (2m_o - 1)(c) + 2(m_e - m_o + 3/2)(n - c + 1) \tag{3a}$$

From the assumption $m_e \geq m_o$, the value of the second term is at least 3(n - c + 1).

Substituting the value of c from 2a

$$N \geq 2m_o\left[\frac{n+1}{2} + \varepsilon\right] - \frac{n+1}{2} - \varepsilon + 3\left[\frac{n+1}{2} - \varepsilon\right] \tag{4a}$$

$$=> \quad N \geq m_o n + m_o + 2m_o\varepsilon - \frac{n+1}{2} - \varepsilon + 3\frac{n+1}{2} - 3\varepsilon \tag{5a}$$

$$=> \quad N \geq m_o n + m_o + (2m_o\varepsilon - 2\varepsilon) + 2\frac{n+1}{2} - 2\varepsilon$$

As $m_o \geq 1$ we can rewrite the inequality by dropping the term $(2m_o\varepsilon - 2\,\varepsilon)$

$$=> \quad N \geq m_o n + m_o + 2\frac{n+1}{2} - 2(\frac{n-1}{2})$$

The maximum value of ε from 4a was substituted to get the lower bound of N and by straightforward manipulation we obtain

$$N \geq m_0 n + m_0 + 2$$

This is clearly greater than rn since $r \geq 1$ and $m_0 \geq r$. For the case $n-c+1 \geq c$ we can account for more elements. Since there can be atmost rn elements less than E in the whole array, we have arrived at a contradiction by assuming that there are r + $\frac{m}{2}$ (or more) elements in column c less than E.

Proceeding in a similar manner it can be shown that E won't move above $r-\frac{m}{2}$ rows by enumerating the elements greater than E in column c . E will move up further than this only if the number of elements in column c greater than E exceeds $3\frac{m}{2}-r$. We can also argue, by numbering the rows from below that all the expressions remain the same. As E was assumed to be the extreme element in an arbitrary column c with destination row r obviously all other elements with destination row r will also move within the $r \pm \frac{m}{2}$ range. Since this holds for an arbitrary row r it occurs simultaneously for all rows in the array. This completes the proof of the induction basis. Q.E.D.

INDUCTIVE STEP

For this part we will show that if an arbitrary element E is within $\frac{m}{2^p}$ rows of its destination row after O(p) iterations, it will be within $\frac{m}{2^{p+1}}$ after 2 more iterations. Stated otherwise, we will show that the algorithm converges in 2(log m) iterations.

Theorem 4 :After O(p) iterations all the elements currently in rows 1 .. (r- $\frac{m}{2^p}$)

are less than E and all elements currently in rows r + $\frac{m}{2^p}$ are greater than E.

proof This follows directly from the induction hypothesis. Since all the elements are assumed to be within $\frac{m}{2^p}$ of their destination rows after O(p) iterations, all the elements in rows 1 .. (r - $\frac{m}{2^p}$) have their destination rows above r and thus these elements are less than E. A similar argument follows for the elements greater than E. Q.E.D.

The method used to prove the inductive step will be similar to the proof of induction basis : counting elements in column c which will prevent E from moving within the specified distance of $\frac{m}{2^{p+1}}$ rows of its destination row r in two more iterations. We shall need to introduce some more properties.

Definition 4 The columns in the ranges 1.. $\frac{n}{3}$ and $\frac{2}{3}n..n$ will be referred to as 'side-bands'.

The columns in the range $\frac{n}{3}$ .. $\frac{2}{3}n$ will be called the 'center-band'.

Theorem 5: In the next column-sort, following the O(p) iterations, all elements whose destination row is r, and currently present in the side bands, will move within the required range of $r \pm \frac{m}{2^{p+1}}$.

proof From Theorem 4 we know that all elements in the in the rows outside the range $r \pm \frac{m}{2^p}$ are less than or greater than E and thus will not interfere with E moving within $\frac{m}{2^{p+1}}$ rows of its destination row in the subsequent iterations. E will be prevented from moving within the required range if there are more than $\frac{m}{2^p} + \frac{m}{2^{p+1}} - 1$ elements in the column, within this region (see fig 6), less than or greater than E, which is assumed to be in odd row.

Let q_o and q_e denote the number of odd and even elements in this region less than E in column c. If r is an even row it is clear that

$$q_o\ , q_e \leq \frac{m}{2^p} - 1 \qquad \text{(1b)}$$

Assume that E will not move above $r + \frac{m}{2^{p+1}}$ in the next column sort which is possible if, and only if,

$$q_o + q_e \geq \frac{m}{2^p} + \frac{m}{2^{p+1}} - 1 \qquad \text{(2b)}$$

Without loss of generality assume $q_e \leq q_o$. (3b)

From 1b and 2b $q_o , q_e \geq \frac{m}{2^{p+1}}$ (4b)

Also from 2b and 3b

$$q_o + q_e \geq \frac{m}{2^p} + \frac{m}{2^{p+1}}$$

$$=> \quad q_o \geq \frac{1}{2}\left[\frac{m}{2^p} + \frac{m}{2^{p+1}}\right] \qquad \text{(5b)}$$

Applying corollary 3.1, the number of elements less than E in this region is (corresponding to N_{oo}) given by

$$N=(2q_e)(max\{c,n-c+1\})+2(q_o-q_e+1/2)(c)$$

Note that the lower bound for N will be given by the condition

$$q_o+q_e = \frac{m}{2^p}+\frac{m}{2^{p+1}}-1 \qquad (6b)$$

To ensure minimality, assume that n-c+1 ≥ c. Substituting the value of q_e from 6b we have

$$N \geq 2(n-c+1)(\frac{m}{2^p}+\frac{m}{2^{p+1}}-q_o-1) + 2c\,(2q_o + 1-\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]+1/2)$$

$$=> \quad N = 2(n-c+1)\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]-2(q_o+1)(n-c+1)-2c\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]+4q_oc+3c$$

From 4b

$$q_o \geq \frac{1}{2}\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]$$

Substituting

$$q_o = \frac{1}{2}\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]+\varepsilon$$

where $0 \leq \varepsilon \leq \frac{m}{2^{p+2}}-1$ from 1b

$$N \geq -c\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]+n\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]-2n-2n\varepsilon+5c+\varepsilon(6c-2)+\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]-2$$

$$=> \quad N \geq (n-c+1)\left[\frac{m}{2^p}+\frac{m}{2^{p+1}}\right]-2n-2n\varepsilon+5c-2\varepsilon-1$$

Following from the condition (n -c + 1) $\geq \frac{2}{3}n$ i.e. $c \leq \frac{1}{3}n$ and substituting $c = \frac{1}{3}n + 1 - \bar{c}$ we obtain

$$N = \frac{mn}{2^p} + \bar{c}\left[\frac{m}{2^p} + \frac{m}{2^{p+1}}\right] - 2n - 2n\varepsilon + 6\varepsilon\left(\frac{n}{3} + 1 - \bar{c}\right) + 5\left(\frac{n}{3} + 1 - \bar{c}\right) - 2\varepsilon - 2$$

$$\Rightarrow \quad \frac{mn}{2^p} + \bar{c}\left[\frac{m}{2^p} + \frac{m}{2^{p+1}} - 6\varepsilon - 5\right] + 4\varepsilon + 3 - \frac{n}{3}$$

Since $\varepsilon \leq \frac{m}{2^{p+2}} - 1$ for all values of ε, we get

$$N \geq \frac{nm}{2^p} + \frac{n}{3} - c - \frac{n}{3} + 4\varepsilon + 4 \tag{7b}$$

by substituting back the value of $\bar{c}$

From Theorem 4 we know that in rows 1 .. (r - $\frac{m}{2^p}$) there are $n \times \left[r - \frac{m}{2^p}\right]$ elements less than E. Also note that in the $r - \frac{m}{2^p} + 1$ row, all the elements have a destination row r or less than r, though we have accounted for only (n-c+1) elements (which we know are less than E). By adding the remaining amount to N, the number of elements with destination row 1,..,r ,and currently present in rows $r \pm \frac{m}{2^p}$, is $\frac{mn}{2^p} + 4\varepsilon + 3$, which is greater than $\frac{mn}{2^p}$. We have thus accounted for more than nr elements in the array with destination rows 1..r , which evidently is impossible. Therefore there cannot be more than $\frac{m}{2^p} + \frac{m}{2^{p+1}} - 1$ elements in column c which will prevent E from moving within the required region.

We arrive at a similar conclusion by considering $c \geq n-c+1$. Q.E.D.

Lemma 5.1

After one iteration, following the O(p) iterations, all the elements present in the center band below the range $r + \frac{m}{2^p}$ have a destination row greater than r.

proof Consider the situation after a column-sort (following the O(p) iterations). From theorem 5 the side bands do not contain any element of row r below this range. Hence all the elements in these bands and below this range are greater than the elements of row r. With the next row sort the element 'E', if present in the center band and below row $r + \frac{m}{2^{p+1}}$, will definitely move into one of the side-bands (depending on the direction of sorting). After the row-sort, all the elements that occupy the center band below row $r + \frac{m}{2^{p+1}}$ are greater than all the elements of row r because they came from the side bands after row sorting. Q.E.D.

With the next column sort two cases arise

Case 1 : Any element present in the side bands with destination row r will move up within the required range (from theorem 5).

Case 2 : Any element that moved up within the required range during the last column sort and presently in the center band, owing to the last row-sort will remain in this range since all elements in the center band below $r + \frac{m}{2^{p+1}}$ are greater than E (following Lemma 5.1).

Following a similar procedure, we can show that the element E does not move above $r - \frac{m}{2^{p+1}}$ row after two more iterations following the O(p) iterations.

This completes the proof of the inductive step. Q.E.D.

At the end of section II we mentioned the difficulty in keeping track of the data movement in an $m \times n$ array under successive row-column sort iterations. We were successful in showing that the elements move within $\frac{m}{2^p}$ rows of their destination row in O(p) iterations. Hence the O(log m) number of iterations for the whole procedure to converge.

To show that this bound is tight consider the following initial data distribution : All the elements belonging to the first row being distributed one in each row in a $m \times n$ array. This is similar to the case considered under theorem 1 in the previous section and will obviously take Ω (logn) iterations to converge.

For the remaining discussion we will assume the array to be square i.e. $n \times n$. This will help to achieve a balance between the two steps of each iteration.

IV. Optimizing bubble-sort for VLSI implementation

In spite of the terrible performance of a normal single processor bubble sort, efforts have directed towards obtaining efficient VLSI implementations ([5], [6]) because of the inherent simplicity of the algorithm. For this purpose a parallel version of bubble sort viz. odd-even transposition sort([6]) has been adopted. By using crossing sequence techniques, several researchers have shown that the optimal AT^2 bound for sorting n elements is O(n^2) in word model and O($n^2 \log^2 n$) in bit model ([3],[5],[6]). The normal N/2 processor bubble sort where each processor performs one compare-exchange operation during each of the N iterations behaves horribly (O(n^3)) with respect to the AT^2 measure. This remains unchanged even by using completely pipelined bit-parallel comparison exchange modules to sort more than one problem instance. The pipelined scheme consists of O (n^2) comparators which reduces the effective area by a concurrency factor(n) - the time remaining unchanged ([5]).

Let us analyze the performance of our row-column sort algorithm by using bubble sort to execute the basic sorting step of each iteration. Fig 7c shows the implementation of the row-column sort using a pipelined scheme where each of the n rows(columns) are pipelined through this sorting network. The 'Transpose/ Detranspose ' network aligns the array properly for the next column (row) sort. Following Leighton's[3] argument, the Transpose/Detranspose network needs n non-unit length wires and hence occupies $O(n^2)$ area where the transposition is performed in n parallel stages by hardwiring the rows to the corresponding columns (and vice versa- see fig 7b). The bubble-sort network consists of n^2 comparators and thus the total area of the network is O($n^2\log n$). We will need 2n word steps to sort all the n rows (columns). Each comparator is capable of performing a compare-exchange operation of two O(log n) bit numbers in O(1) time. As observed previously the Transpose/Detranspose network also needs O(n) time for each iteration. Since we need log n iterations the AT^2 performance for this scheme will be

$$O(n^2 logn)\times O((n\log n)^2) = O(n^4\log^3 n)$$

This is only O($\log^3 n$) away from the lower bound. A similar result can be obtained for the bit model by using bit-serial compare exchange modules. Each compare-exchange module can perform a compare exchange operation every O(log n) time units and can fit into an O(1) by O(log n) unit rectangle. Thus the bubble-sort circuit occupies an area of O($n^2\log n$) units. The Transpose/Detranspose circuit consists of n non-unit length wires which occupy an area of O (n^2) units. Each of these wires routs O(n logn) bits of data and thus takes O(n logn) units of time to complete the operation. The total time is O($n\log^2 n$) and so the AT^2 measure for this scheme is O($n^4\log^5 n$). This is a factor of $\log^3 n$ away from the optimal which is O($n^4\log^2 n$) for O(n^2) O(log n) bit numbers ([3]).

As noted by Thompson[5], this network needs very little in the way of control as no complicated operations are involved and may be more attractive than its AT^2 performance indicates (being a couple of log n factors away from the optimal). There is hardly any need to overemphasize that this scheme of sorting which exploits the powerful property of the algorithm has made bubble-sort comparable to some more sophisticated VLSI sorting networks as far as area-*time*^{2} trade-off is concerned.

V. An orthogonal access multiprocessing architecture for row-column sort

It has been customary to associate two-dimensional sorting algorithms with mesh connected or systolic processor arrays. Current VLSI technology has strongly influenced efforts towards better and faster sorting in such architectures. However regardless of the simplicity of the basic processing element, a mesh-connected processor array is far more complex than standard VLSI dynamic memory devices. While the latter have reached 256K bits per package, only a 72 PE array is currently available from NCR. It is therefore still of theoretical interest, and probably impractical to sort large sequences in VLSI PE arrays. Other multiprocessor architectures may prove to be more viable solutions to the problem by taking advantage of high-density storage devices while utilizing fewer, faster, and more powerful processors. Consider the row-column sort algorithm - it exhibits an inherent parallelism in the basic sorting step that can be executed simultaneously for all rows(columns) of the array. The approach requires a storage scheme that will allow a number of processors to access the two-dimensional array both by rows and columns. Data skewing and scrambling together with appropriate interconnection networks have been suggested for this purpose (ILLIAC IV and STARAN among others). In memory organisations, where simultaneous access to more than one row or column is not possible, the row column sort algorithm would execute by sequentially sorting rows and columns. The row and column sequences would be sorted by an array of processors using any of the well known parallel schemes for sorting linear sequences. Table 1 shows the performance of row-column sort using some well known parallel schemes for performing the basic n element sorting step.

Table 1

Complexity figures with different sorting algorithms	
Bitonic sort (n proc.)	$n\log^3 n$
Bubble sort (n proc.)	$n^2\log n$
mergesort(log n proc)	$n^2\log n$

Sorting all rows(columns) simultaneously using O(nlogn) algorithms yield a time complexity of O(n $\log^2$ n) which is at least a factor of log n better than the approaches mentioned above. Fig (8) shows an architecture where 'p' processors are connected to p^2 memory banks which contain the n^2 elements. Each element $q_{i,j}$ is mapped into a memory bank $M_{s,t}$ such that s = i mod p and t = j mod p.

Further, each of the memory banks $M_{i,j}$ is dynamically switched between P_i and P_j during the row-sort and column-sort stage. This means that processor P_n has access to memory banks $M_{n,k}$ during the row-sort stage and $M_{k,n}$ during column sort (k = 1 .. p). So during a row(column) sort stage each of the processors has to sort $\frac{n}{p}$ rows(columns) of the $n \times n$ array. Also note that in any of these two configurations there is no contention for memory and the sorting can proceed independently. This feature led to the architecture's name - Orthogonal access memory.

To sort an individual row or column, the processors may use any of the optimal single processor sorting algorithms like the heapsort or the mergesort Thus we need O(n logn) time for sorting each row or column and $n \times \frac{nlogn}{p}$ for sorting all the rows with p processors. Recall from section III, that except for the first iteration in the column sort stage, we only need to merge two sequences of $\frac{n}{2}$ elements in each column (from Lemma 2.1). Thus each iteration will take O($\frac{n^2logn+n^2}{p}$) steps. Since the algorithm converges in O(log n) iterations the total time needed is O($\frac{n^2\log^2 n+n^2logn}{p}$) units, or O($\frac{n^2\log^2 n}{p}$) units. For a single processor implementation,(p =1) we need O($n^2\log^2 n$) time units which means we are only a factor of log n away from the theoretical lower bound. (For n^2 elements the optimal number of steps is O($n^2\log n^2$). The speed-up obtained in this implementation is directly proportional to the number of processors, the maximum being n by using 'n' processors to sort all the rows or columns concurrently.

VI. Summary

The feasibility of sorting a rectangular array of elements by sorting rows and columns was demonstrated. The algorithm was shown to execute in atmost O(log m) iterations and in section III we traced the data movement during each iteration. Recall that we showed that all elements moved within O($\frac{m}{2^p}$) rows of its final destination row in O(p) iterations. Also taking advantage of this phenomenon it is possible to optimize a simple algorithm like bubble sort. Sorting a row(column) is actually sorting $\sqrt{n}$ elements in a n element sequence which may be expensive. However this basic operation may be optimized by using a sorting network as was demonstrated in section IV. The complexity of this algorithm is more appropriately expressed as O ($\sqrt{n} \times k \times \log n$), for an n element sequence, where k is the time for sorting $\sqrt{n}$ elements, This is O($\sqrt{n} \log n$) for a single processor sort and was used as a basis for the multiprocessor implementation. For the sorting network, allowing pipelining, the complexity turns out to be O($(\sqrt{n} + k) log n$) which at the best will give us a time of O($\sqrt{n} \log n$).

The multiprocessor architecture which can implement the algorithm elegantly because of its orthogonal access to the memory banks was originally conceived as a high performance graphics system which can draw very fast vectors of any orientation ([11]). This is one of the first general purpose applications suggested and it might be interesting to find other applications, which may be efficiently implemented taking advantage of the orthogonal access capability.

Another interesting problem is mapping a two dimensional odd-even transposition sort on this row-major snake-like indexing scheme. Each iteration will consist of performing compare-exchange on elements ($x_{2i-1,j}$, $x_{2i,j}$) on all rows independently followed by the same procedure on all the columns ($x_{i,2j-1}$, $x_{i,2j}$) and then repeating the same with the elements ($x_{2i,j}, x_{2i+1,j}$) in all rows and elements ($x_{i,2j}, x_{i,2j+1}$) in all columns. It is not difficult to see that such an algorithm will result in a sorted sequence. If the upper bound for the number of iterations is $\sqrt{n}$, then an optimal AT^2 implementation can be found using a mesh of processors. The present algorithm can also be easily mapped to a mesh of processors. A column(row) of the mesh of processors can perform the basic column(row) sort by using nearest neighbour communication for odd-even transposition sort. So it will take O($\sqrt{n} log n$) compare-exchange steps which is a factor of log n from optimal performance in such architectures([1]), nevertheless, no complex data routing steps are needed, which makes this scheme attractive.

References

[1] C.D. Thompson & H.T. Kung , "Sorting on a Mesh-Connected Parallel Computer," Communications of the ACM, vol 20, number 4, April 1977.

[2] D. Nassimi & S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Transactions on Computers, vol c-27, no 1, Jan 1979.

[3] T. Leighton , "Tight Bounds on the Complexity of Parallel Sorting," IEEE Transactions on Computers, vol c-34, no. 4, April 1985.

[4] D. Gale & R.M. Karp ," A Phenomenon in the Theory of Sorting," Journal of Computer and System Sciences, no. 6 , 1972 ,pp 103 - 115.

[5] C.D. Thompson, "The VLSI Complexity of Sorting ," IEEE Transaction on Computers, vol c-32, no. 12, Dec 1983.

[6] D.E. Knuth, "The Art of Computer Programming," vol 3. Addison-Wesley 1973.

[7] J.D. Ullman, "The Computational Aspects of VLSI ," Computer Science Press 1984.

[8] Lang Hans-Werner et. al. "Systolic sorting on a Mesh Connected Network," IEEE Transactions on Computers, vol c-34 no. 7 July 1985.

[9] K. Batcher, "Sorting Networks and their applications," in *Proc.* AFIPS Spring Joint Comput. Conf, 1968 vol 32.

[10] M. Kumar & D.S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," IEEE Transaction on Computers, vol c-32, March 1983.

[11] I.D. Scherson, " A Parallel Processing Architecture for Image Generation and Processing, " Preliminary Report, Dept. of Electrical and Computer Engineering, U.C.S.B., E.C.E. Report No. 84-20, Aug 1984.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

fig 1a : Row-major indexing

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

Fig 1b : Row-major snake-like indexing

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

Fig 1c : Shuffled row-major indexing

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

fig 2a :In this distribution, the row and columns are sorted in the direction imposed by row-major scheme but as a whole the array is not ordered. This example shows why simply sorting rows and columns in a straightforward fashion does not work.

1	5	9	13
14	10	6	2
3	7	11	15
16	12	8	4

Step-1b : After row-sort of the first iteration.

1	5	6	2
3	7	8	4
14	10	9	13
16	12	11	15

Step 2a

1	2	5	6
8	7	4	3
9	10	13	14
16	15	12	11

Step 2b

1	2	4	3
8	7	5	6
9	10	12	11
16	15	13	14

Step 3a

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

Step 3b : The array is sorted according to snake-like row major form.

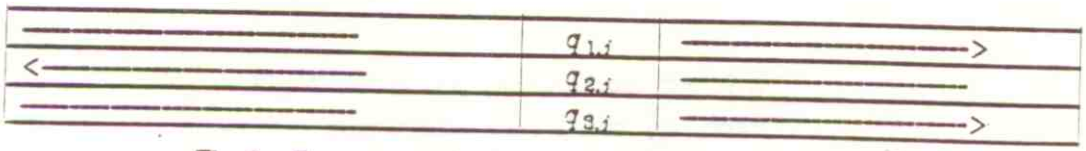

Fig 3 : The arrows indicate the sorting direction.

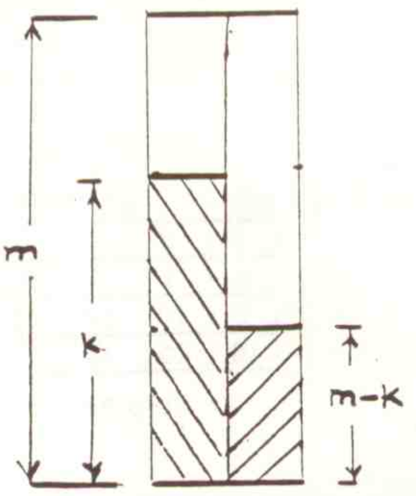

Fig 4 : The shaded area shows the 'Heavy' elements after a column sort. As is evident from the figure, there are three distinct bands, one in which rows have 'Light' elements in both the columns, the ones with 'Heavy' and the ones which are heterogeneous. It is in this region that the row-sort will redistribute the 'Heavy' and 'Light' elements.

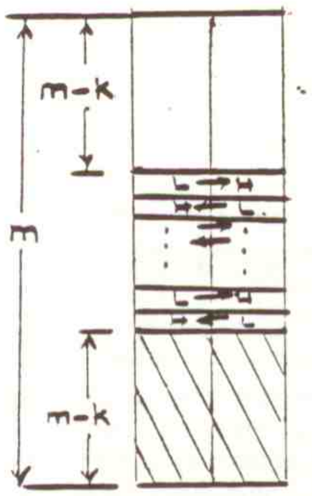

Fig 4b : After the row sort it is clear that the 'Light' and 'Heavy' elements are distributed equally in within each column so that the next column-sort packs them into the proper halves.

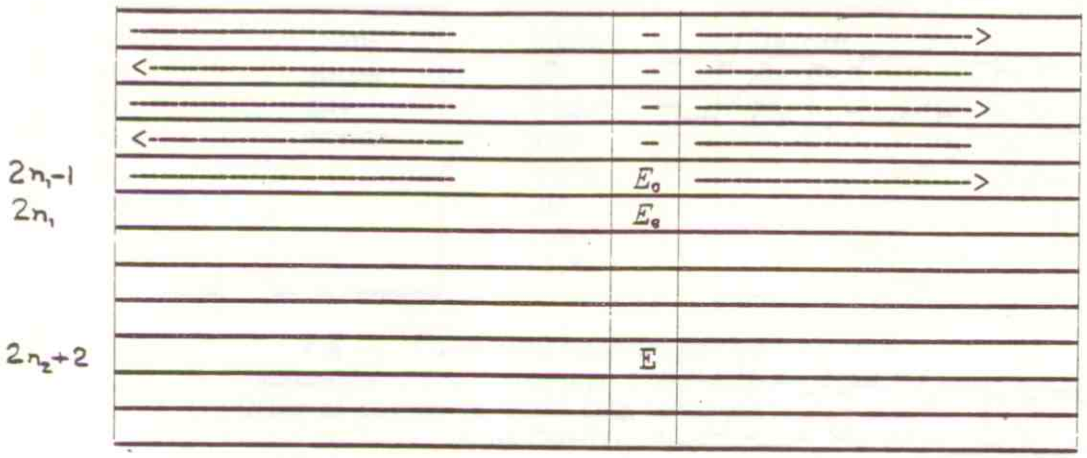

Fig 5 : Element E is in column c from left and column (n - c + 1) from right.

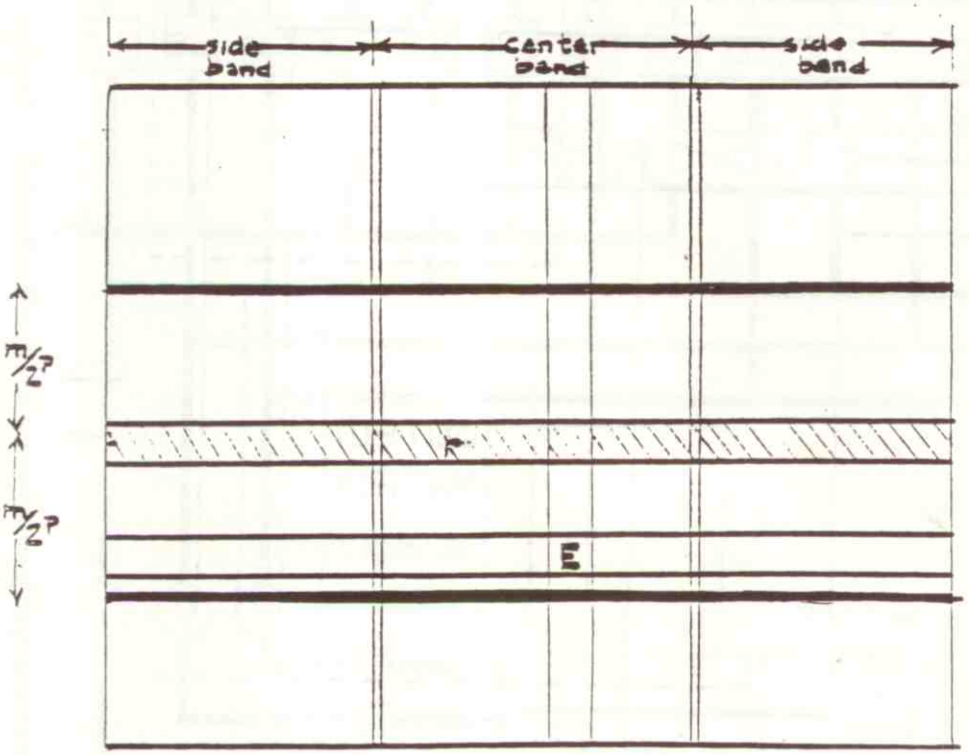

Fig 6: 'r' is the destination row of E
After O(p) iterations E is within $\frac{m}{2^p}$ rows of r.

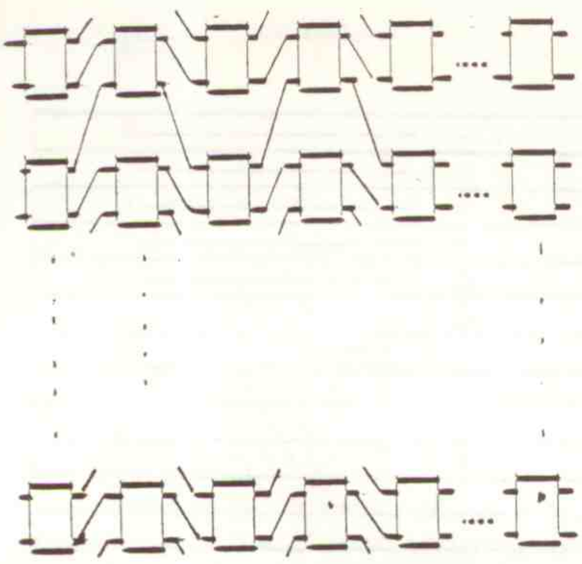

Fig 7a : N^2 bubble-sort network consisting of comparators each of which can perform a compare exchange operation of two O(log n) bit numbers in O(log n) time.

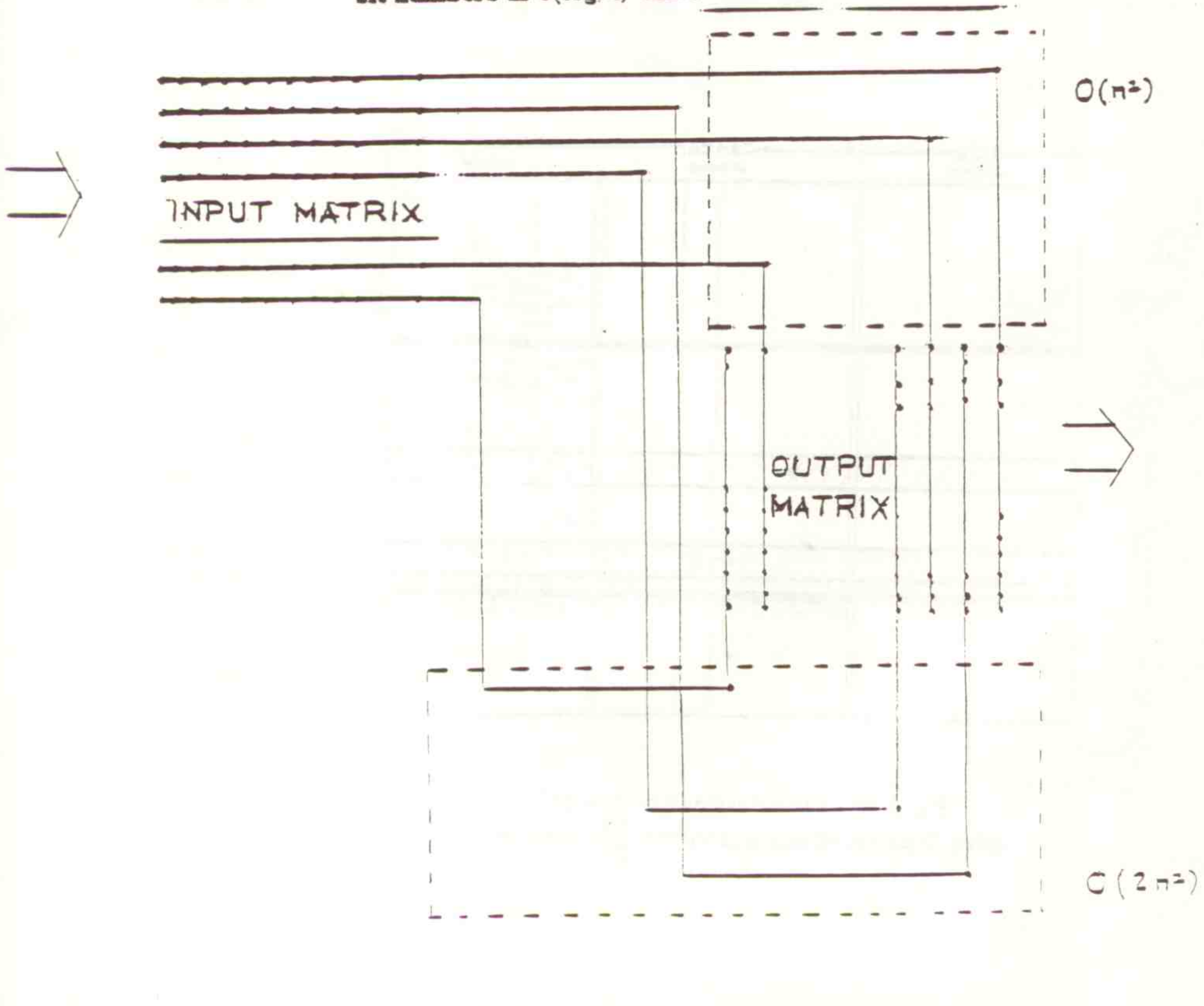

Fig 7b Transpose / Detranspose network consisting of n wires.

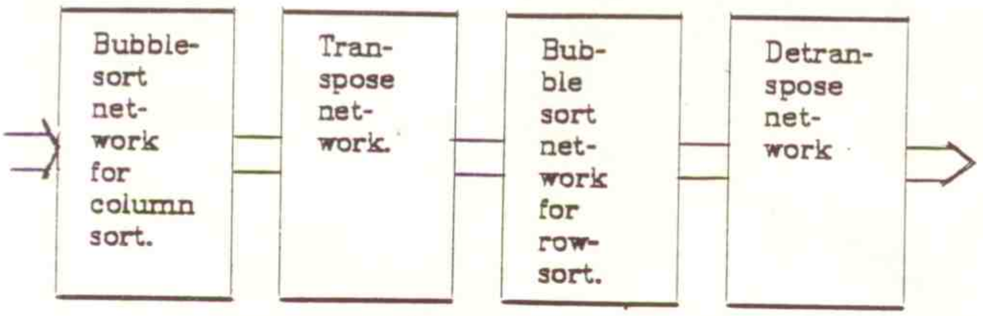
Bubble-sort net-work for column sort.
Tran-spose net-work.
Bub-ble sort net-work for row-sort.
Detran-spose net-work

Fig 7c : Pipelining scheme for sorting n^2 elements using n element bubble-sort network. O(log n) passes through this network will produce a sorted sequence.

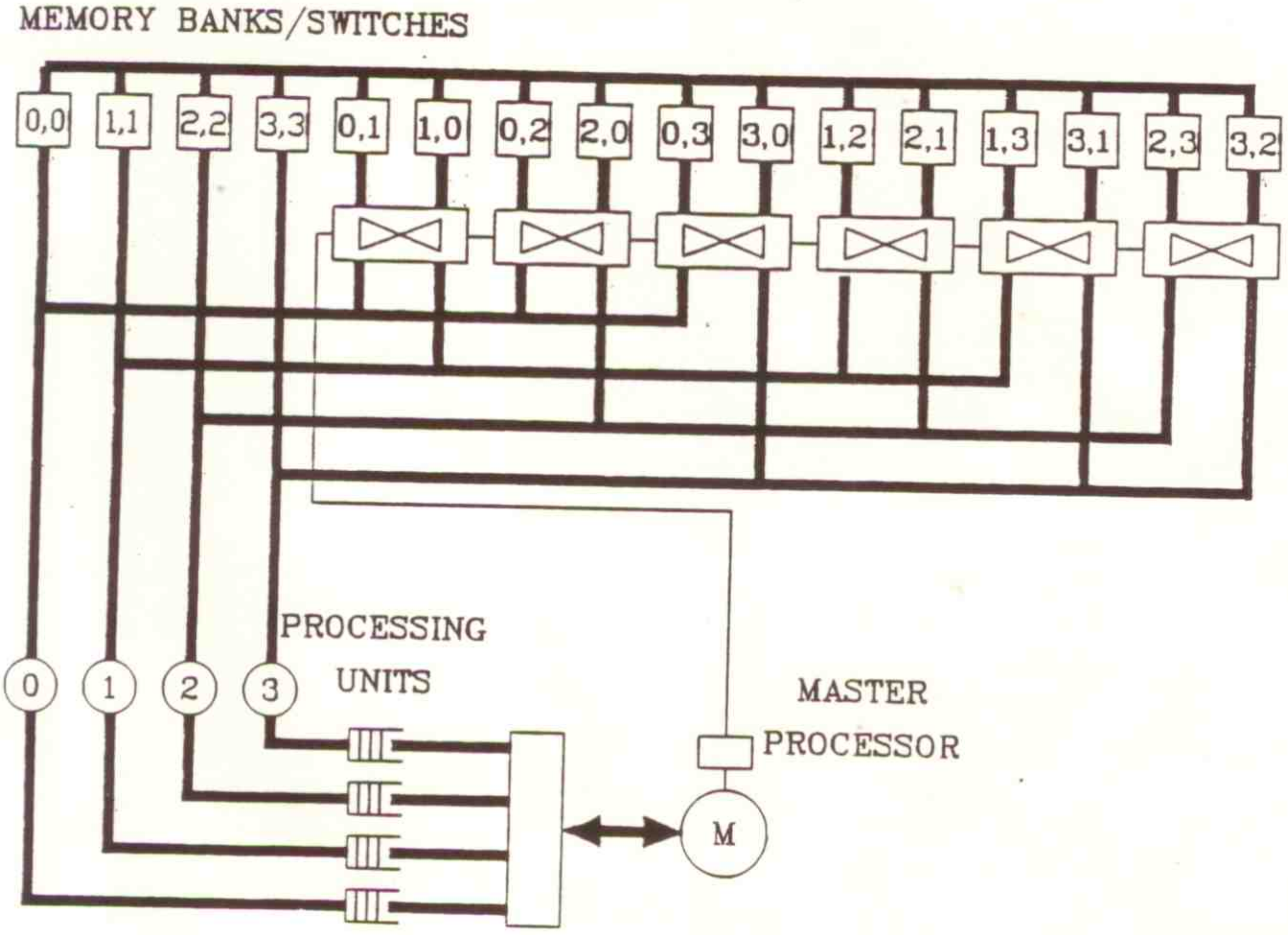

Fig 8 : shows a 4 processor, 16 memory banks system

with the proposed architecture.