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 ≥ 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