SORTING IN CONSTANT NUMBER OF ROW AND COLUMN PHASES ON A MESH

John M. Marberg
Eli Gafni

September 1986
CSD-860043

Sorting in Constant Number of Row and Column Phases on a Mesh [†]

John M. Marberg
Eli Gafni

Computer Science Department
University of California
Los Angeles, CA 90024

Technical Report CSD-860043
September 1986

Abstract

An algorithm for sorting on a mesh by alternately transforming the rows and columns is presented. The algorithm runs in constant number of row- and column-transformation phases (16 phases), an improvement over the previous best upper bound of $O(\log\log m)$ phases, m being the number of rows in the mesh. A corresponding lower bound of five phases is also shown.

[†] Work supported by an NSF PYI Award and by an IBM Faculty Development Award.

1. Introduction

A two-dimensional *mesh* consists of $m \cdot n$ processors organized in a rectangular grid of m rows and n columns. Each pair of adjacent processors along the same row or column are connected by a bidirectional communication link. Computation proceeds in synchronous parallel steps, during each of which a processor may transfer a value to a neighbor or perform some elementary local operation. Each processor has a constant amount of local memory. The regularity of the interconnection pattern and the modularity of the layout make the mesh an ideal model for VLSI applications. It is therefore that considerable attention has been given to the study of algorithms for this model.

In this paper we consider the problem of sorting on a mesh. It is defined as follows: a collection of elements is distributed in the mesh, one element per processor; viewing the input as an $m \times n$ matrix, sort the matrix into row-major order. Two commonly used row-major orders are: (row, column)-lexicographic, and snake-like [Thom77]. We develop a sorting algorithm that works by alternately transforming the rows and columns of the mesh. This approach is particularly attractive for VLSI, since it lends itself easily to systolic implementation.

Recent results by Bilardi and Preparata [Bila85] and by Leighton [Leig85] have established the VLSI optimality of the AKS sorting network [Ajta83]. Yet, the enormous constants involved in the AKS network render it largely impractical. This further motivates the effort to find simple sorting methods for regular interconnection structures such as the mesh.

The asymptotic lower bound for sorting on a mesh is $\Omega(n+m)$ parallel steps, established by Thompson and Kung [Thom77]. Numerous researchers have developed optimal algorithms, including Thompson and Kung [Thom77], Nassimi and Sahni [Nass79], Kumar and Hirschberg [Kuma83], Lang, Schimmler, Schmeck and Schroder [Lang85], and Schnorr and Shamir [Schn86]. All the above-mentioned algorithms proceed by dividing the mesh into submeshes, working on the submeshes recursively in parallel, then combining the results in some fashion. Recently, Scherson, Sen and Shamir [Sche86] have discovered an algorithm that is based on an entirely different approach. It proceeds by alternately sorting the rows and columns of the mesh, as follows.

Algorithm *SHEARSORT* [Sche86]

Repeat $\lceil \log m \rceil + 1$ times steps 1 and 2 below.

1. Sort all even-numbered rows to the right and all odd-numbered rows to the left.
2. Sort all the columns downwards.

This novel algorithm is considerably simpler than previous solutions to the sorting problem. The only computation required is sorting along a line, which can be done using the straightforward odd/even transposition method [Knut73]. Consequently, the algorithm entails relatively little control logic for the routing of data. Moreover, the row/column approach lends itself naturally to systolic implementation. Notice, however, that *SHEARSORT* is not optimal; consisting of $\log m$ sorting phases, it requires a total of $O((m+n)\log m)$ steps. Schnorr and Shamir [Schn86] have subsequently developed an improved version of the algorithm, called *REVSORT*, which runs in $\log\log m$ sorting phases and a total of $O((m+n)\log\log m)$ steps.

Let us define a *row-phase* of a mesh algorithm as a maximal sequence of steps in which all the data movement is along the rows. A *column-phase* is defined similarly. Thus, a row-phase (column-phase) results in some arbitrary transformation of each row (column). An algorithm that operates in alternating row- and column-phases is called a *row/column* algorithm. An open question posed in [Schn86] is whether there exists a row/column sorting algorithm with fewer than $O(\log\log m)$ phases (which is the number of phases in *REVSORT*). In this paper we answer this question affirmatively. Specifically, we develop an algorithm that runs in a constant number of phases (16 phases), requiring a total of $O(m+n)$ steps, which is optimal. We use only two types of transformations during a phase: sorting and rotation. Since rotation can be emulated by sorting, the algorithm can be implemented entirely by means of basic compare/exchange steps. We also prove that five phases is a lower bound for row/column sorting. Our results demonstrate the viability of the row/column approach, providing motivation to investigate its applicability to other problems.

The remainder of the paper is organized as follows. The sorting algorithm is presented in Section 2. Its proof of correctness and complexity analysis are given in Section 3. In section 4 we show the lower bound. Concluding remarks are given in Section 5.

2. The Sorting Algorithm

We denote the positions in the mesh by the (row, column) pairs (i, j), where $0 \le i \le m-1$ and $0 \le j \le n-1$. For simplicity, we assume $m=2^s$ and $n=2^{2t}$, where $s \ge t$ (the results can be generalized for other values of m and n).

A *slice* is a submesh of size $n^{1/2} \times n$, consisting of all rows i such that $kn^{1/2} \le i \le (k+1)n^{1/2}-1$ for some given $k \ge 0$. A *block* is a submesh of size $n^{1/2} \times n^{1/2}$, consisting of all positions (i, j) such that $kn^{1/2} \le i \le (k+1)n^{1/2}-1$ and $ln^{1/2} \le j \le (l+1)n^{1/2}-1$ for some given $k \ge 0$ and $l \ge 0$. Figure 1 illustrates these definitions.

It is convenient to analyze the sorting algorithm via the 0-1 principle [Knut73]. This principle states that if a sorting algorithm works correctly on an arbitrary input of 0's and 1's, then it works correctly on any input. Given a mesh with 0-1 input, we say that a given region (e.g., row, slice, block) is *dirty* if it contains both 0's and 1's; otherwise it is *clean*. The correctness of a sorting algorithm can be shown by identifying the regions of the mesh that become dirty or clean at any given point. Clearly, upon termination, there should remain at most one dirty row.

We begin by specifying several macros (or procedures), which serve as building-blocks for the main algorithm. Each macro comprises a sequence of phases that accomplish a specific transformation on the mesh.

Macro *BALANCE*

Applied to a (sub)mesh of size $v \times w$, this macro "balances" the distribution of 0's and 1's among the columns, thereby reducing the number of dirty rows to at most $\min\{v, w\}$. This is accomplished by "dealing" the sorted elements of each column in round-robin fashion to all the columns. The macro consists of the following three steps.

(a) Sort all the columns downwards.

(b) Rotate each row i $(i \bmod w)$ positions to the right.

(c) Sort all the columns downwards.

Macro *UNBLOCK*

This macro distributes the elements of each block among all the columns (i.e., it ''unblocks'' the blocks), thereby effectively generating a clean row from every clean block, and a (possibly) dirty row from every dirty block. A similar idea is used in the *REVSORT* algorithm [Schn86], except that the goal there is to ''un-rectangle'' arbitrary rectangles of area n, not just square blocks. The macro consists of the following two steps.

(a) Rotate each row i $(i \bmod n^{1/2})$ positions to the right.

(b) Sort all the columns downwards.

Macro *SHEAR*

This macro emulates one iteration of the *SHEARSORT* algorithm [Sche86]. Assuming that all dirty rows are in one contiguous region, each pair of adjacent dirty rows are reorganized into one clean row and one (possibly) dirty row, thereby cutting the number of dirty rows by half. This is effectively accomplished by sorting alternate rows in opposite directions (i.e., ''shearing'' the rows). The macro consists of the following two steps.

(a) Sort all even-numbered rows to the right and all odd-numbered rows to the left.

(b) Sort all the columns downwards.

We now specify the main algorithm, called *ROTATESORT*, using the above macros. A complete proof of correctness and complexity analysis are given in Section 3.

Algorithm *ROTATESORT*

1. *BALANCE* the mesh.
   ⟨ now there are at most n dirty rows ⟩
2. Sort all the rows to the right.
   ⟨ now there are at most $2n^{1/2}$ dirty blocks ⟩
3. *UNBLOCK* the mesh.
   ⟨ now there are at most $2n^{1/2}$ dirty rows ⟩
4. *BALANCE* each slice as if it were an $n \times n^{1/2}$ mesh lying on its side.
   ⟨ now there are at most 6 dirty blocks ⟩
5. *UNBLOCK* the mesh.
   ⟨ now there are at most 6 dirty rows ⟩
6. Repeat macro *SHEAR* three times.
   ⟨ now there is at most one dirty row ⟩
7. Sort all the rows to the right to obtain lexicographic order, or alternating rows in opposite directions to obtain snake-like order.

3. Correctness and Complexity

To prove the correctness of *ROTATESORT* using the 0-1 principle, we need to show that upon termination:

(a) there exists at most one dirty row;
(b) the dirty row is sorted in accordance with the choice of row-major order;
(c) the dirty row separates the clean rows of 0's from the clean rows of 1's.

We first prove the correctness of the three macros. Once the macros are validated, it is straightforward to prove the main algorithm.

Lemma 1. After applying macro *BALANCE* to a mesh of size $v \times w$, there remain at most $\min\{v, w\}$ dirty rows.

Proof. The claim is trivial when $v \le w$. We now discuss the case $v > w$. Step (b) of *BALANCE* has the effect of "dealing" the elements of each column in round-robin fashion among all the columns. Since the columns were previously sorted in step (a), the difference between the number of 0's that any two columns receive from a given column is at most one. Thus, the total difference in the number of 0's between any two columns after step (b) is at most w. Sorting the columns in step (c) thus yields at most w dirty rows. Moreover, all the resulting dirty rows are in one contiguous region that separates the clean rows of 0's from the clean rows of 1's (see Figure 2). ■

Lemma 2. After applying macro *UNBLOCK* to a mesh with k dirty blocks, there remain at most k dirty rows.

Proof. Step (a) of *UNBLOCK* moves each of the n elements of a block into a different column. Since there are k dirty blocks, the difference in the number of 0's between any two columns after step (a) is at most k. The Lemma follows since the columns are subsequently sorted in step (b). As in Lemma 1, the dirty rows are in one contiguous region that separates the clean rows of 0's from the clean rows of 1's. ■

Lemma 3. After applying macro *SHEAR* to a mesh with k dirty rows, there remain at most $\lceil \frac{k}{2} \rceil$ dirty rows.

Proof. The proof is based on [Sche86]. We assume that upon invocation of *SHEAR* all the dirty rows are in one contiguous region. Consider a pair of adjacent dirty rows. After step (a), the 0's are packed in opposite ends in the two rows (see Figure 3). Let z_1 and z_2 denote the number of 0's in the rows. If $z_1+z_2 \geq n$, then there is at least one 0 in each column in the pair; otherwise, there is at least one 1 in each column. In either case, the difference between the number of 0's in any two columns in the pair is at most one. Since there are $\lceil \frac{k}{2} \rceil$ disjoint pairs of adjacent dirty rows (when k is odd, one pair has only one dirty row), the total difference in the number of 0's between any two columns in the mesh at the end of step (a) is at most $\lceil \frac{k}{2} \rceil$. The Lemma follows since the columns are subsequently sorted in step (b). As in Lemmas 1 and 2, the resulting dirty rows are in one contiguous region. ■

Theorem 1. Statements 1-7 below describe the configuration of the mesh after steps 1-7, respectively, of algorithm *ROTATESORT*.

1. After step 1 there are at most n dirty rows.
2. After step 2 there are at most $2n^{1/2}$ dirty blocks.
3. After step 3 there are at most $2n^{1/2}$ dirty rows.
4. After step 4 there are at most 6 dirty blocks.
5. After step 5 there are at most 6 dirty rows.
6. After step 6 there is at most one dirty row.
7. After step 7 the mesh is sorted.

Proof.

1. The result follows immediately from Lemma 1.
2. The proof is based on [Schn86]. Notice that at the end of step 1 the columns are sorted. Sorting the rows in step 2 thus results in the following configuration (see Figure 4): all the 0's are in the upper-left corner of the mesh, whereas the 1's are in the lower-right. The number of dirty rows remains unchanged, i.e., at most n. However, the boundary between the 0's and 1's is now monotonic (in the sense that it does not cross the same row or column twice), and therefore traverses at most $2n^{1/2}$ blocks. Clearly, the blocks traversed by the boundary are dirty, whereas the remaining blocks are clean.
3. The result follows immediately from claim 2 of this Theorem and from Lemma 2.
4. The $2n^{1/2}$ dirty rows that remain at the beginning of step 4 are in one contiguous region (Lemma 2). Hence, there are at most three dirty slices in the mesh. We now view each slice as a mesh of size $n \times n^{1/2}$ lying on its side, and apply *BALANCE* to it. By Lemma 1, this reduces the number of dirty "rows" in each inverted slice to at most $n^{1/2}$. Moreover, the dirty "rows" in each slice are in one contiguous region (see Figure 5). Hence, at the end of step 4 there are at most two dirty blocks in each dirty slice, for a total of six dirty blocks in the entire mesh.
5. The result follows immediately from claim 4 of this Theorem and from Lemma 2.
6. By Lemma 3, each application of macro *SHEAR* reduces the number of dirty rows in the mesh by half. Thus, after $\lceil \log_2 6 \rceil = 3$ applications, there remains at most one dirty row.

Moreover, the dirty row separates the clean rows of 0's from the clean rows of 1's. Hence, termination requirements (a) and (b) are satisfied after step 6.

7. Step 7 sorts each row in accordance with the choice of row-major order, thereby satisfying termination requirement (c). This completes the correctness proof. ■

We now discuss the complexity of *ROTATESORT*. It is easy to see that the algorithm runs in constant number of phases. Each application of *BALANCE* takes three phases, whereas *UNBLOCK* and *SHEAR* each take two phases. Summing over the entire algorithm, one would get a total of 18 phases. Yet, a closer look at the sequence of steps reveals that steps 2 and 3(a) both operate on rows, and hence, by definition, constitute a single phase. The same observation holds for steps 4(c) and 5(a). The correct number of phases is therefore 16.

The algorithm uses only two types of transformations during a phase: sorting and rotation. Since rotation can be emulated by sorting, each row-phase can be implemented in $O(n)$ basic compare/exchange steps, using odd/even transposition sort [Knut73]. Similarly, a column-phase can be implemented in $O(m)$ compare/exchange steps. Thus, the total number of basic steps in the entire algorithm is $O(m+n)$, which is optimal.

It can be seen from Lemma 1 that when $m \leq n$, the application of *BALANCE* in step 1 is redundant and can be replaced with a single phase that sorts all the columns downwards. Consequently, for $m \leq n$, and in particular for square meshes, we obtain an algorithm with 14 phases and $O(n)$ basic steps.

4. A Lower Bound

An obvious question to be posed at this point is whether *ROTATESORT* is optimal in the number of phases. In an effort to answer this question, we show that five phases is a lower bound. However, we suspect that neither this lower bound, nor the upper bound of 16 phases are tight.

The lower bound argument proceeds by constructing for any given algorithm an input that requires five phases. We assume that the algorithm is comparison-based, i.e., that the values of the elements are used only in comparisons with each other. The bound is valid regardless of the sorting order being used (i.e., not just for row-major order).

We use the following notation. Two positions (i_1, j_1) and (i_2, j_2) in the mesh are *relatively independent* if $i_1 \neq i_2$ and $j_1 \neq j_2$. Position (i, j) is said to have *row-rank* r at a given point in the algorithm, if at that point the element located in (i, j) is the r-th smallest element in row i. The term *column-rank* is defined similarly. The row-rank and column-rank of position (i, j) at the end of phase t, $t \geq 1$, are denoted $RR_t(i, j)$ and $CR_t(i, j)$, respectively.

We assume without loss of generality that all elements are distinct. The i-th smallest element, $1 \leq i \leq mn$, is denoted e_i.

The property of row/column sorting upon which the proof of the lower bound hinges is the following.

Lemma 4. Let e_a and e_b be two elements that belong in the same column (in the same row, respectively) in the sorted order. If at the end of a given column-phase t (row-phase t) both e_a and e_b are located in the same row (same column), then the algorithm requires at least $t+3$ phases.

Proof. We only prove the case where e_a and e_b belong in the same column in the sorted order. The other case can be shown by reversing the roles of row and column.

In order to move e_a and e_b from the same row to the same column after phase t, we need at least one more column-phase followed by a row-phase. Since phase $t+1$ is a row-phase, at least 3 additional phases are required. ■

To obtain the lower bound of five phases, we construct an input such that the premise of Lemma 4 is satisfied at the end of phase 2 (clearly, two phases are always necessary).

Theorem 2. Any row/column sorting algorithm on a mesh of size $m \times n$, where $m \geq 8$, $n \geq 8$, requires at least five phases.

Proof. We distinguish between an algorithm that begins with a row-phase, and one that begins with a column-phase. We discuss only the former case. By symmetry, the other case can be shown by reversing the roles of row and column.

Given that the first phase is a row-phase, we show how to construct, for any given sorting algorithm, an input such that two elements which belong in the same column in the sorted order are located in the same row at the end of the phase 2.

Let l be some fixed integer, $l \le mn - \max\{m, n\}$, called the *partition index*. We partition the input into three different classes with respect to l: elements e_i such that $i<l$ are called *very small*; elements e_i such that $l \le i \le l + \max\{m, n\}$ are called *intermediate*; the remaining elements, i.e., elements e_i such that $i>l+\max\{m, n\}$ are called *very large*. Regardless of the actual value of l, one can always identify three intermediate elements: e_{p_1} , e_{p_2} and e_{p_3}, such that e_{p_1} and e_{p_2} belong in the same column in the sorted order, whereas e_{p_3} belongs in a different column. This is because the number of intermediate elements ($\max\{m, n\} + 1$) is larger than the number of rows or columns. We call each of e_{p_1} , e_{p_2} and e_{p_3} a *pivot*.

Consider an input I such that at the end of phase 1 the position of each pivot is independent from the position of any other intermediate element (including the other two pivots). In other words, no pivot "sees" any of the other intermediate elements during phases 1 and 2. Now, let us form another input by permuting the pivots in I among themselves in some arbitrary fashion, leaving all the remaining positions unchanged. It can be seen that at the end of phase 2 each element other than the pivots will occupy the exact same position under any such permutation, whereas the pivots will be permuted. This is because the three pivots have the same relative order with respect to any very small or very large element, thus rendering a permutation of the pivots transparent to the algorithm during the first two phases.

We now prove that the premise of Lemma 4 is satisfied at the end of phase 2 by some of the $3!=6$ possible pivot permutations of input I. Let S denote the set of elements comprising the three pivots plus the $m-2$ non-pivot elements that in the sorted order belong in the same column as e_{p_1} and e_{p_2}. The positions occupied by the elements in S at the end of phase 2 are fixed up to permutation among the pivot positions. Moreover, since there are $m+1$ elements in S but only m rows, two of the elements in S must be in the same row at all times, particularly at the end of phase 2. In other words, there exist two fixed positions, both in the same row, that contain elements from S at the end of phase 2 under all six permutations. Now, e_{p_3} can occupy one of the two positions in at most four permutations. This leaves two permutations in which both positions are occupied by elements from $S-\{e_{p_3}\}$. Since these elements belong in the same column in the sorted order, the premise of Lemma 4 is satisfied.

To complete the proof of the theorem, we need to show how to construct input I. Let us assume, for now, that the partition index l is a parameter. Later, we will show how to determine its exact value.

First, we arbitrarily fix the order among the elements in each row of the input. This enables us to uniquely determine, for a given algorithm, the row-rank of every position at the end of phase 1. Notice that we are still left with complete freedom to decide which element goes in which row. As will be seen below, knowing all the RR_1 values suffices in order to construct the proper input.

We construct the input row by row, first taking care of the rows that contain pivots. To this end, we need to guarantee that at the end of phase 1 the pivots will be in independent positions. Let (r_1, c_1), (r_2, c_2), and (r_3, c_3) be three pairwise independent positions. To each row r_j, $1 \le j \le 3$, we allocate the following elements:

(a) e_{p_j};

(b) $RR_1(r_j, c_j)-1$ very small elements;

(c) $n-RR_1(r_j, c_j)$ very large elements.

It can be seen that e_{p_j} is the element of rank $RR_1(r_j, c_j)$ in its row. Hence, it will be in position (r_j, c_j) at the end of phase 1.

We now construct the rows that contain the remaining intermediate elements. We must make sure that no intermediate element will be in column c_1, c_2 or c_3 at the end of phase 1. Let us consider an arbitrary empty row k. We assume without loss of generality that $RR_1(k, c_1) \le RR_1(k, c_2) \le RR_1(k, c_3)$ (if not, simply "rename" the columns). Since the number of elements in a row is n, at least one of the following four inequalities must always hold.

(a) $RR_1(k, c_1)-1 \ge \frac{n-3}{4}$

(b) $RR_1(k, c_2)-RR_1(k, c_1)-1 \ge \frac{n-3}{4}$

(c) $RR_1(k, c_3)-RR_1(k, c_2)-1 \ge \frac{n-3}{4}$

(d) $n-RR_1(k, c_3) \ge \frac{n-3}{4}$

Thus, it is always possible to allocate $\frac{n-3}{4}$ intermediate elements to row k. For example, assume inequality (b) holds. First, we allocate to row k $RR_1(k, c_1)$ very small elements and $n-RR_1(k, c_2)+1$ very large elements. This leaves room for at least $\frac{n-3}{4}$ additional elements. Any intermediate element we now allocate to row k must, by definition, have a rank (among the elements in the row) which is larger than $RR_1(k, c_1)$ and smaller than $RR_1(k, c_2)$. This guarantees that at the end of phase 1 such an element will not be in any of the columns containing a pivot. Inequalities (a), (c) and (d) are handled in a similar way. Thus, all the $\max\{m, n\}-2$ intermediate elements that are not pivots can be accommodated in $\lceil \frac{4(\max\{m, n\}-2)}{n-3} \rceil$ rows.

This takes care of all the rows that contain intermediate elements. The remaining rows, if any, can be constructed arbitrarily from the remaining elements.

The construction imposes the following constraints on the size of the mesh. We need three rows for the pivots, and as many as $\lceil \frac{4(\max\{m, n\}-2)}{n-3} \rceil$ rows for the remaining intermediate elements. Thus, the inequality $m \geq \frac{4(\max\{m, n\}-2)}{n-3} + 3$ must hold.[1] It can be seen that this occurs when $m \geq 8$, $n \geq 8$.

It remains to fix the value of l, the partition index. Let u and v be the total number of very small and very large elements, respectively, allocated to rows containing intermediate elements. l can be set to any integer in the range $[u+1, mn-\max\{m, n\}-v]$. ■

5. Conclusion

We have presented an optimal algorithm for sorting on a mesh by means of alternating row and column transformations. Our results demonstrate that the row/column approach is viable, providing motivation to investigate its applicability to other problems.

An interesting open question is to determine the VLSI complexity of *ROTATESORT*. Another issue that remains open is to bridge the gap between the upper bound of 16 phases and the lower bound of five phases. We suspect that neither bound is tight. Specifically, the large degree of freedom in the construction in Theorem 2 suggests that there is room for improvement of the lower bound. Also, it might be possible to reduce the upper bound by using more powerful transformations.

[1] The $\lceil\,\rceil$ operator is not needed in the inequality since m is always an integer.

References

[Ajta83] Ajtai, M., J. Komolos, and E. Szemeredi, ''An $O(N\log N)$ Sorting Network,'' in *Proceedings 15th ACM Symp. on Theory of Computing*, 1983, pp. 1-9.

[Bila85] Bilardi, G. and F.P. Preparata, ''The VLSI Optimality of the AKS Sorting Network,'' *Information Processing Letters* **20**, 2 (Feb. 1985), pp. 55-59.

[Knut73] Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison Wesley, Reading, MA, 1973.

[Kuma83] Kumar, M. and D.S. Hirschberg, ''An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application in Parallel Sorting Schemes,'' *IEEE Trans. on Computers* **C-32**, 3 (March 1983), pp. 254-264.

[Lang85] Lang, H.W., M. Schimmler, H. Schmeck, and H. Schroder, ''Systolic Sorting on a Mesh-Connected Network,'' *IEEE Trans. on Computers* **C-34**, 7 (July 1985), pp. 652-658.

[Leig85] Leighton, T., ''Tight Bounds on the Complexity of Parallel Sorting,'' *IEEE Trans. on Computers* **C-34**, 4 (April 1985), pp. 344-354.

[Nass79] Nassimi, D. and S. Sahni, ''Bitonic Sort on a Mesh-Connected Parallel Computer,'' *IEEE Trans. on Computers* **C-28**, 1 (Jan. 1979), pp. 2-7.

[Sche86] Scherson, I.D., S. Sen, and A. Shamir, ''Shear Sort: A True Two-Dimensional Sorting Technique for VLSI Networks,'' in *Proceedings 1986 Int. Conf. on Parallel Processing*, pp. 903-908.

[Schn86] Schnorr, C.P. and A. Shamir, ''An Optimal Sorting Algorithm for Mesh Connected Computers,'' in *Proceedings 18th ACM Symp. on Theory of Computing*, 1986, pp. 255-261.

[Thom77] Thompson, C.D. and H.T. Kung, ''Sorting on a Mesh-Connected Parallel Computer,'' *CACM* **20**, 4 (April 1977), pp. 263-271.

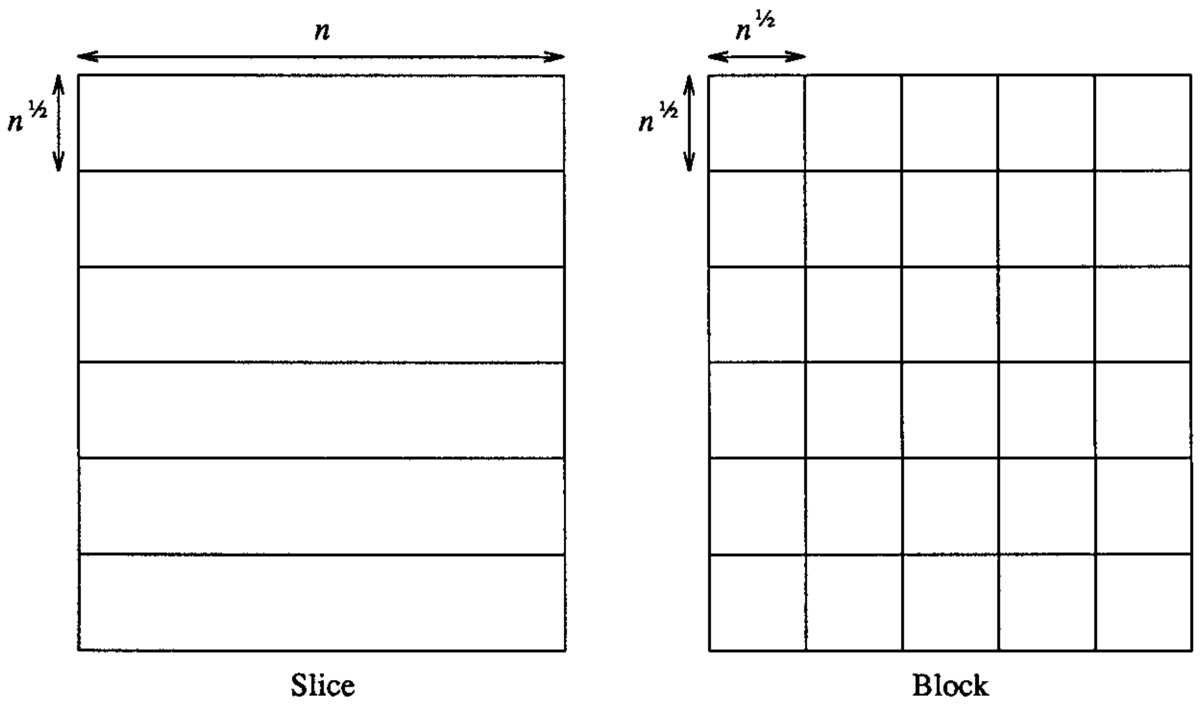

Figure 1. Definition of slice and block

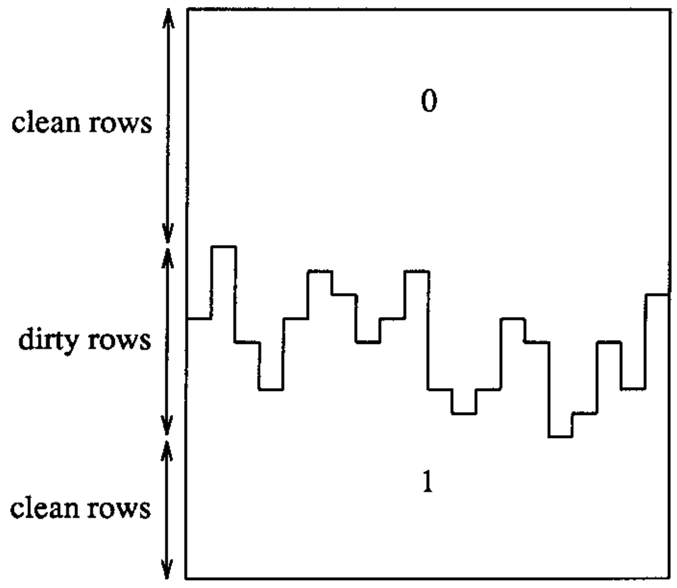

Figure 2. Clean and dirty regions after sorting the columns

0	0	0	0	0	0	0	1	1	1
1	1	1	1	1	0	0	0	0	0

$z_1+z_2 \geq n$

0	0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	0	0

$z_1+z_2 < n$

Figure 3. Two adjacent dirty rows sorted in opposite directions

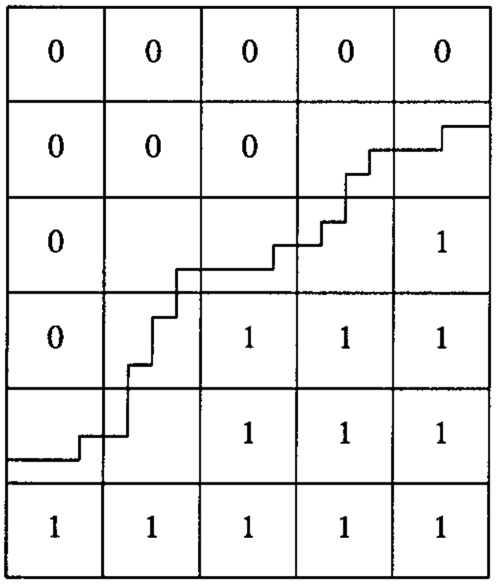

Figure 4. The mesh after step 2

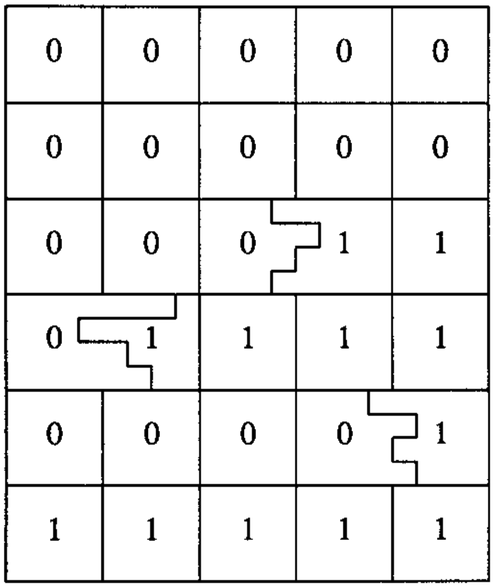

Figure 5. The mesh after step 4