Classifying Matrices Separating Rows and Columns

Alan A. Bertossi, Stephan Olariu, M. Cristina Pinotti, and Si-Qing Zheng

Abstract—The classification problem transforms a set of N numbers in such a way that none of the first $\frac{N}{2}$ numbers exceeds any of the last $\frac{N}{2}$ numbers. A comparator network that solves the classification problem on a set of r numbers is commonly called an r-*classifier*. This paper shows how the well-known Leighton's Columnsort algorithm can be modified to solve the classification problem of $N = rs$ numbers, with $1 \leq s \leq r$, using an r-classifier instead of an r-sorting network. Overall, the r-classifier is used $O(s)$ times, namely, the same number of times that Columnsort applies an r-sorter. A hardware implementation is proposed that runs in optimal $O(s + \log r)$ time and uses an $O(r \log r(s + \log r))$ work. The implementation shows that, when $N = r \log r$, there is a classifier network solving the classification problem on N numbers in the same $O(\log r)$ time and using the same $O(r \log r)$ comparators as an r-classifier, thus saving a $\log r$ factor in the number of comparators over an $(r \log r)$-classifier.

Index Terms—Comparator network, classifier, classification problem, hardware algorithm.

1 Introduction

The current technology has made it possible to implement algorithm-structured devices as building blocks for high-performance computing systems. Thus, general-purpose computer systems could be endowed with a special-purpose parallel sorting device, invoked whenever its services are needed. The design of such a sorting device can be based on *sorting networks*, namely, networks of comparators that sort their input numbers into order.

A relevant problem closely related to sorting is the *classification* problem, where one asks for classifying a set of numbers into halves, in such a way that each number in one class is at least as large as all of those in the other class [5]. Solving such a problem is a frequent computation that occurs in database monitoring to compute order statistics and approximated sorting [12], in parallel scheduling to schedule the tasks with the minimum or maximum priorities [18], and in breadth-first searching algorithms, like the M algorithm and the bidirectional algorithm, used in the decoding of convolutional codes [4]. These applications motivate the study of *classifier networks*, that is, networks of comparators that solve the classification problem.

There is a wide literature on the design and analysis of sorting networks [1], [3], [9], [10], [14], [15]. Clearly, any *r-sorter* (i.e., a sorting network that sorts r input numbers) is also an *r-classifier* (i.e., a classifier network that solves the classification on the same r input numbers). However, classifiers do not have to do as much as sorters. Therefore, there is also a rich literature on the design and analysis of classifiers [4], [8], [11], [13], [17], which yields to simpler and more efficient networks than sorters. In fact, it is well-known that effective r-sorters are still based on Batcher's networks [3] and require $O(\log^2 r)$ time, while the existence of r-sorters taking $O(\log r)$ time is only of theoretical interest, due to the enormous constant hidden in the big-oh notation of the AKS network [1]. In contrast, there exist r-classifiers taking $O(\log r)$ time, where the constant hidden in the big-oh notation is very small [8].

Leighton [10] devised a simple and effective sorting algorithm, called *Columnsort*, to sort rs numbers arranged into an $r \times s$ matrix A. Such an algorithm consists of eight passes on A and repeatedly applies an r-sorter to the columns of A. The original motivation for Leighton's Columnsort algorithm was indeed that the AKS network, by itself, provides a means to sort r items in $O(\log r)$ time using $O(r \log r)$ comparators, but this implies that a total of $O(r \log^2 r)$ work (i.e., time $\times$ comparators) is used, which is inefficient by a factor of $\log r$. Observing that the AKS network can be pipelined, Columnsort shows how to optimally sort $N = r \log r$ numbers, arranged into an $r \times \log r$ matrix, in $O(\log r)$ time and $O(r \log^2 r)$ work. Leighton's solution obtains an optimal work using a sorter of smaller size than that of the input. Applying a similar reasoning to the classification problem, one may use an r-classifier to classify $N \geq r$ numbers. In this case, $\Omega(N/r + \log r)$ time is needed since no more than r numbers can be processed simultaneously and $\Omega(\log r)$ is a lower bound on the network depth [19] and $\Omega(r \log r)$ comparators are required, as proven by Alekseyev [2].

Based on the considerations above, the following natural questions arise: "Does Columnsort solve the classification problem if classifiers replace sorters?" and, if the answer is negative: "How should one modify Columnsort in order to efficiently solve the classification problem using classifiers

- A.A. Bertossi is with the Department of Computer Science, Mura Anteo Zamboni, 7, University of Bologna, Italy. E-mail: bertossi@cs.unibo.it.
- S. Olariu is with the Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162. E-mail: olariu@cs.olu.edu.
- M.C. Pinotti is with the Department of Computer Science and Mathematics, University of Perugia, Via Vanvitelli, 1, 06123 Perugia, Italy. E-mail: pinotti@unipg.it.
- S.-Q. Zheng is with the Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688. E-mail: sizheng@utdallas.edu.

Manuscript received 29 May 2003; revised 23 Sept. 2003; accepted 8 Oct. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0083-0503.

instead of sorters?" This would imply that the classification problem can be solved using a simpler and smaller (in its constant factors) network than the AKS sorting network and, thus, raising a new question: "Given the ability to use an r-classifier for solving the classification problem of r numbers in $O(\log r)$ depth using $O(r \log r)$ comparators, is it possible to derive a circuit that can classify $N = r \log r$ numbers using the same asymptotic depth and number of comparators as an r-classifier?" In the affirmative case, there would be a circuitry for N numbers which has size smaller by a $\log r$ factor than that of an N-classifier network. In this paper, answers to the above questions are given.

The rest of the paper is organized as follows: Section 2 gives a negative answer to the first question, while Section 3 describes the *Row-Column-Classification* algorithm. This algorithm takes a logarithmic number of passes to solve the classification problem on a matrix A of size $r \times s$, with $r = 2^k$ and any $s = 2^h$, where $1 \leq h \leq k$, using an r-classifier (when $h = k$ such an algorithm works also on a square matrix). Although the number of passes is $O(\log s)$ for Row-Column-Classification and $O(1)$ for Columnsort, both such algorithms apply their comparator network, namely, an r-classifier or an r-sorter, respectively, the same number of times, that is, $O(s)$.

Section 4 describes another algorithm, called *Three-Pass-Classification*, which solves in three passes the classification problem on a matrix A of size $r \times s$, with $1 \leq s \leq \sqrt{\frac{r}{2}}$. This algorithm shows that, replacing the sorter with the classifier in the Columnsort algorithm, the classification problem can be solved, still maintaining a constant number of passes as long as s is bounded by $O(\sqrt{r})$. Therefore, both Three-Pass-Classification and Row-Column-Classification provide an answer to the second question, "How should one efficiently modify Columnsort using classifiers instead of sorters?" Moreover, they are based on the simpler r-classifier by Jimbo and Maruoka [8], which takes $O(\log r)$ time, whereas to obtain the same time performance, Columnsort has to employ the ineffective r-sorter by Ajtai et al. [1].

Finally, Section 5 presents a hardware algorithm, based on both the Three-Pass-Classification and Row-Column-Classification algorithms, which solves the classification problem on $N = rs$ numbers, with $1 \leq s \leq r$. Such an algorithm achieves an optimal $O(s + \log r)$ time and uses $O(r \log r)$ comparators. Overall, $O(rs \log r + r \log^2 r)$ work is done which, when $s = \Omega(\log r)$, is better by a factor of $\log r$ than the $O(N \log^2 N) = O(rs \log^2 r)$ work used by an N-classifier. In particular, when the number s of columns of A is $O(\log r)$, the hardware algorithm gives an affirmative answer to our third question, showing that it is possible to build a classifier network that can find the median of $O(r \log r)$ numbers in the same $O(\log r)$ time and using the same $O(r \log r)$ number of comparators as an r-classifier.

2 Columnsort with Classifiers

As a preliminary, let the behavior of the original Leighton's Columnsort algorithm be briefly recalled. This algorithm sorts in column-major order a matrix $A[0 \ldots r-1, 0 \ldots s-1]$, with $r \geq 2(s-1)^2$ and $r \equiv 0 \bmod s$ [10] (recently, such constraints have been slightly relaxed in [6], [7], but this does not affect our counterexample). Columnsort consists of eight passes: The odd passes are sorting passes, while the even passes are data movement passes. During passes 1, 3, 5, and 7, each column of A is locally sorted by means of an r-sorter. During pass 2, the numbers of A are taken in column-major order and put back in A in row-major order, while, during pass 4, the numbers of A are taken in row-major order and put back in column-major order. In passes 6 and 8, the numbers are shifted forward or backward, respectively, by $\lfloor \frac{r}{2} \rfloor$ positions. Overall, the r-sorter is applied $O(s)$ times. Note that data movement passes are used merely for the purpose of sorting only the columns, but one could properly group consecutive rows with r numbers per group and then apply the r-sorter to sort each group of rows.

Now, consider how Columnsort acts when used to solve the classification problem still applying an r-sorter. After pass 4 of Columnsort, every number is within $(s-1)^2$ of its correct sorted position [10]. Since $r \geq 2(s-1)^2$, the numbers of A are already separated, except perhaps either those in the central column, if s is odd, or those in the lowest half and in the highest half of the two central columns, if s is even. More formally, in the third sorting pass, one only needs either to sort column $A[*, \frac{s-1}{2}]$, if s is odd, or to sort $A[\frac{r}{2} \ldots r-1, \frac{s}{2}-1] \cup A[0 \ldots \frac{r}{2}-1, \frac{s}{2}]$, if s is even (hereafter, $A[*, j]$ and $A[i, *]$ denote column j and row i of A, respectively, while $A[i \ldots h, j \ldots k]$ denotes the submatrix of A given by the specified rows and columns). Thus, overall, five passes instead of eight are enough for Columnsort to solve the classification problem using an r-sorter. Hence, only three sorting passes, instead of four, are needed.

Consider then, what happens if one tries to solve the classification problem still using Columnsort, but substituting the r-sorter with an r-classifier. In such a case, the odd passes become separation passes. In the following, a counterexample is exhibited where Columnsort fails because the median number remains in its original position after pass 5 and in the wrong half after pass 8.

Consider the particular matrix A, shown in Fig. 1, built as follows: Let $r = 54$, $s = 6$, and let the input numbers be all the integers between 1 and $rs = 324$. Consider the sequence of the above numbers sorted from 1 to 324. Remove from the sequence the median number 162 and its successor 163 and place 163 between 53 and 54 and 162 between 217 and 218. Then, store the modified sequence in A in column-major order (see Fig. 1), and apply the Columnsort algorithm using the 54-classifier, instead of the 54-sorter. The 54-classifier merely separates the 27 smallest numbers from the 27 largest numbers, but no specific order within each half can be assumed. In particular, applying the 54-classifier in passes 1, 3, and 5, each column of A may remain unchanged. If this is the case, the matrix A after pass 5 is the same as the input matrix (see Figs. 1 and 2). In particular, 162 and 163 remain in their original positions and, thus, Columnsort fails. Even if the computation proceeds with passes 6, 7, and 8 (see Figs. 3, 4, and 5), the separation at pass 7 must move the numbers 162 and 163, but it might rearrange them as depicted in Fig. 4. However, even in this case, Columnsort does not solve the classification problem since the numbers 162 and 163 remain, respectively, in the second half and in the first half of A (see Fig. 5).

0	1	54	108	164	**162**	271
1	2	55	109	165	218	272
2	3	56	110	166	219	273
	⋮	⋮	⋮	⋮	⋮	⋮
51	52	105	159	215	268	322
52	53	106	160	216	269	323
53	**163**	107	161	217	270	324

Fig. 1. The matrix A in input, and after pass 1 (separation), 4 (row-to-column transpostion), and 5 (separation).

Such a counterexample can be generalized to a matrix A with arbitrary size $r \times s$ in such a way that the median number can be hidden virtually in any position of the wrong half of A. Thus, the answer to the first question, "Does Columnsort work if classifiers replace sorters?," is negative.

3 Row-Column-Classification Algorithm

The goal of this section is to solve the classification problem on a set of $N = rs$ numbers, with $1 \leq s \leq r$, stored in a matrix $A[0 \dots r-1, 0 \dots s-1]$, using an optimal number of applications of an r-classifier.

0	1	2	3	4	5	6
1	7	8	9	10	11	12
	⋮	⋮	⋮	⋮	⋮	⋮
8	49	50	51	52	53	**163**
9	54	55	56	57	58	59
	⋮	⋮	⋮	⋮	⋮	⋮
26	156	157	158	159	160	161
27	164	165	166	167	168	169
	⋮	⋮	⋮	⋮	⋮	⋮
35	212	213	214	215	216	217
36	**162**	218	219	220	221	222
	⋮	⋮	⋮	⋮	⋮	⋮
52	313	314	315	316	317	318
53	319	320	321	322	323	324

Fig. 2. The matrix A after pass 2 (column-to-row transposition) and 3 (separation).

0	$-\infty$	28	81	135	191	244	298
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
25	$-\infty$	53	106	160	216	269	323
26	$-\infty$	**163**	107	161	217	270	324
27	1	54	108	164	**162**	271	$+\infty$
28	2	55	109	165	218	272	$+\infty$
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
53	27	80	134	190	243	297	$+\infty$

Fig. 3. A after pass 6 (shift forward), where the empty entries introduced by the shift are filled by $\pm\infty$.

For the sake of simplicity, in this section, both r and s are assumed to be powers of two, that is, $r = 2^k$ and $s = 2^h$, for any $1 \leq h \leq k$.

The building block of the algorithm is a 4-*Partition* procedure which receives as input $2r$ numbers, grouped as C_0, C_1, C_2, C_3, each of size $r/2 = 2^{k-1}$, and partitions the numbers such that all the numbers in C_j are not larger than those in C_{j+1}, with $0 \leq j \leq 3$.

The *Row-Column-Classification* algorithm, illustrated in Fig. 6, consists of $h = \log s$ recursions that reduce the number of columns from s to 2. At each recursion, Row-Column-Classification halves the number c of columns of the previous recursion and perceives A as composed by four submatrices A_1, A_2, A_3, A_4, each of size $r/2 \times c/2$, as depicted in Fig. 7. In particular, the ith recursion works on a matrix A of size $r \times c$, where $c = s/2^i$ and $0 \leq i \leq \log s - 1$ (clearly, when $i = 0$, $c = s$).

Each recursion executes two passes: a *row-pass*, followed by a *column-pass*, executed by the *Row-Column-Pass* procedure. The row-pass examines A row-by-row, applying $c/2$ times the 4-Partition procedure to groups of $2r/c$ rows. The column-pass examines A column-by-column, applying $c/2$ times the 4-Partition procedure to pairs of columns. At the end of a

0	$-\infty$	28	81	135	191	244	298
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
25	$-\infty$	53	106	160	216	269	323
26	$-\infty$	54	107	161	**162**	270	324
27	1	**163**	108	164	217	271	$+\infty$
28	2	55	109	165	218	272	$+\infty$
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
53	27	80	134	190	243	297	$+\infty$

Fig. 4. A after pass 7 (separation).

0	1	**163**	108	164	217	271
1	2	55	109	165	218	272
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
26	27	80	134	190	243	297
27	28	81	135	191	244	298
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
52	53	106	160	216	269	323
53	54	107	161	**162**	270	324

Fig. 5. A after pass 8 (shift backward).

recursion, the numbers belonging to A_1 are followed by all the numbers in $A_3 \cup A_4$, while those belonging to A_4 are preceded by all the numbers in $A_1 \cup A_2$. Therefore, the numbers in A_1 belong to the $rc/2$ smallest numbers of A, while those in A_4 belong to the $rc/2$ largest numbers of A. The algorithm is then recursively applied to the halved submatrix of A consisting of A_2 and A_3. Note that the algorithm works for any $s = 2^h$, with $1 \le h \le \log r$ and, hence, even when A is a square $r \times r$ matrix.

In order to formally describe the above algorithm, the details of how to perform the 4-Partition procedure using the r-classifier are given first.

The 4-Partition procedure, shown in Fig. 8, invokes 6 times the r-classifier. In each invocation, the r-classifier (C, C') receives r numbers, grouped as C and C', each of size $r/2$, and rearranges them so that every number in C is followed by all the numbers in C'. The 4-Partition procedure executes a computation similar to an *odd-even sort* on four items, where each comparison is replaced with a call to the r-classifier. However, the advantage of this procedure versus odd-even sort is that calls $2i$ and $2i+1$, with $0 \le i \le 2$, could be performed simultaneously, for a total of three parallel phases, thus saving one parallel phase over odd-even sort.

Lemma 1. *The 4-Partition procedure partitions the $2r$ input numbers into four groups, C_0, C_1, C_2, C_3, each of size $r/2 = 2^{k-1}$, such that all the numbers in C_j are not larger than those in C_{j+1}, with $0 \le j \le 3$.*

Proof. Let a and b be the medians of $C_0 \cup C_1$ and $C_2 \cup C_3$, respectively. After invoking the r-classifier on $C_0 \cup C_1$ and on $C_2 \cup C_3$, a and b belong to C_0 and C_2, respectively. If $a \le b$, C_0 is followed by all the numbers in $C_1 \cup C_3$ and C_3 is preceded by all the numbers in $C_0 \cup C_2$. Therefore, C_1 and C_2 need to be separated. Instead, if $b < a$, C_2 is followed by all the numbers in $C_1 \cup C_3$, while C_1 is preceded by all the numbers in $C_0 \cup C_2$ and, thus, C_0 and C_3 need to be separated. After invoking r-classifier (C_1, C_2) and r-classifier (C_0, C_3), the classification problem on the input set has been solved, that is, all the numbers in $C_0 \cup C_1$ precede those in $C_2 \cup C_3$. Therefore, to obtain the 4 partition, it is enough to invoke again the r-classifier on $C_0 \cup C_1$ and on $C_2 \cup C_3$. □

At each recursion, the 4-Partition procedure is repeatedly called by the *Row-Column-Pass* procedure, as illustrated in Fig. 9. The row-pass consists of $c/2$ iterations: During the ith iteration, the numbers belonging to r/c consecutive rows of $A_1, \dots, A_4$ are rearranged into four groups in such a way that the numbers in $A[i\frac{r}{c}\dots(i+1)\frac{r}{c}-1, 0\dots\frac{c}{2}-1]$ are not larger than those in $A[\frac{r}{2}+i\frac{r}{c}\dots(i+1)\frac{r}{c}-1, 0\dots\frac{c}{2}-1]$, which are followed by $A[i\frac{r}{c}\dots(i+1)\frac{r}{c}-1, \frac{c}{2}\dots c-1]$ which, in their turn, are not larger than $A[\frac{r}{2}+i\frac{r}{c}\dots(i+1)\frac{r}{c}-1, \frac{c}{2}\dots c-1]$).

Similarly, the column-pass consists of $c/2$ iterations: During the ith iteration, with $0 \le i \le c/2-1$, the $2r$ numbers stored in the two columns $A[*, i]$ and $A[*, i+c/2]$ are rearranged in such a way that all the numbers of $A[i\frac{r}{c}\dots(i+1)\frac{r}{c}-1, 0\dots\frac{c}{2}-1]$ are not larger than those in $A[\frac{r}{2}+i\frac{r}{c}\dots(i+1)\frac{r}{c}-1, 0\dots\frac{c}{2}-1]$, which are followed by those in $A[0\dots\frac{r}{2}-1, i+\frac{c}{2}]$ which, in their turn, are not larger than $A[\frac{r}{2}\dots r-1, i+\frac{c}{2}]$.

The Row-Column-Pass procedure requires $O(c)$ applications of the r-classifier network since there are c iterations and each iteration invokes the 4-Partition procedure. It is worth noting that different iterations work on groups of disjoint rows or columns and, therefore, as will be discussed in Section 5, all the $c/2$ iterations of the row-pass or column-pass could be performed in pipeline.

3.1 Correctness

The correctness of the *Row-Column-Classification* algorithm comes from the following lemma.

Lemma 2. *At the end of the Row-Column-Pass procedure,*

- *no number in A_1 is larger than any number in $A_3 \cup A_4$, and*
- *no number in A_4 is smaller than any number in $A_1 \cup A_2$.*

```
algorithm Row-Column-Classification ($A$, $r$, $s$)
begin
/* let $A$ be perceived as the 4 matrices $A_1, A_2, A_3, A_4$ shown in Figure 7 */
  if $s = 2$ then
    4-Partition ($A_1, A_2, A_3, A_4$)
  else
    Row-Column-Pass ($A$, $r$, $s$);
    Row-Column-Classification ($A_2 \cup A_3$, $r$, $\frac{s}{2}$)
  endif
end
```

Fig. 6. The Row-Column-Classification algorithm.

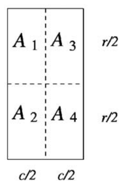

Fig. 7. The submatrices A_1, A_2, A_3, A_4 in which A is decomposed.

Proof. Only the first claim is proven as the proof of the second one follows from a mirror argument.

By contradiction, let $u \in A_1$ be strictly larger than one number $v \in A_3 \cup A_4$. Assume u and v belong, respectively, to columns $A[*, i]$ and $A[*, j]$, where $0 \leq i \leq c/2 - 1$ and $c/2 \leq j \leq c - 1$, and $j \neq i + c/2$.

Since the columns were separated in four parts, there are at least $\frac{3}{2}r + 1$ numbers not smaller than u in columns $A[*, i] \cup A[*, i + c/2]$ and at least $r + 1$ numbers not larger than v in columns $A[*, j - c/2] \cup A[*, j]$. Let U and V denote these sets of numbers, respectively. Observe that all the numbers that, at the end of the row-pass, belonged to the pair of columns i and $i + c/2$ remain in the same pair of columns at the end of the column-pass. Moreover, at the end of the row-pass, for a fixed $q \in [0, c/2 - 1]$, the number stored in $A_1[p, i]$ is not larger than all the $\frac{3}{2}r$ numbers in $A_2[\frac{r}{2} + p, *]$, $A_3[p, *]$, and $A_4[\frac{r}{2} + p, *]$, where $q\frac{r}{c} \leq p \leq (q+1)\frac{r}{c} - 1$. Generalizing to the other submatrices, at the end of the row-pass, the number stored in $A_t[p, i]$ is not larger than the $\frac{(4-t)}{2}r$ numbers in A_{t+1} $[((t-1) \bmod 2)\ \frac{r}{2} + p, *], \ldots, A_4[\frac{r}{2} + p, *]$, where $1 \leq t \leq 4$ and $q\frac{r}{c} \leq p \leq (q+1)\frac{r}{c} - 1$.

Consider now the elements U in the pair of columns i and $i + c/2$ of A. By the previous observations, each element in U implies that there are some elements not smaller than u on columns $j - c/2$ and j and, therefore, forbids some positions for the elements V in such columns. The number of forbidden positions for V is minimized when U contains all the elements in $A_4[*, i + c/2]$, $A_3[*, i + c/2]$, and $A_2[*, i]$. Hence, altogether, U forbids at least r positions for the elements V in columns $j - c/2$ and j. Since there are $2r$ positions available, out of which r are forbidden, only r positions are available for the elements in V.

This shows that $u \geq v$ is impossible. In conclusion, every element in A_1 is followed by all the elements in A_3 and A_4. □

Consider now how many applications $C(s)$ of the r-classifier are required. Since the Row-Column-Pass procedure requires as many r-classifier applications as the number of columns in A, which halves at each recursion, the relation holds

```
procedure 4-Partition ($C_0$, $C_1$, $C_2$, $C_3$)
begin
    0. $r$-classifier ($C_0$, $C_1$);
    1. $r$-classifier ($C_2$, $C_3$);
    2. $r$-classifier ($C_1$, $C_2$);
    3. $r$-classifier ($C_0$, $C_3$);
    4. $r$-classifier ($C_0$, $C_1$);
    5. $r$-classifier ($C_2$, $C_3$);
end
```

Fig. 8. The 4-Partition procedure.

$$C(s) = C\left(\frac{s}{2}\right) + O(s),$$

whose solution is $C(s) = O(s)$.

Although the number of passes is $O(\log s)$ for Row-Column-Classification and $O(1)$ for Columnsort, both such algorithms apply their comparator network, namely, an r-classifier or an r-sorter, respectively, the same number of times, that is, $O(s)$.

4 Three-Pass-Classification Algorithm

The goal of this section is to show how the Row-Column-Classification algorithm can be modified to solve the classification problem in three passes when the number of columns is $O(\sqrt{r})$.

Again, consider the classification problem on a set of $N = rs$ numbers, stored in a rectangular matrix $A[0 \ldots r-1, 0 \ldots s-1]$, with r rows and s columns, where it is assumed that $1 \leq s \leq \sqrt{\frac{r}{2}}$ and $\frac{r}{2s}$ is an integer. The algorithm presented in this section, called *Three-Pass-Classification algorithm*, perceives A composed by four submatrices A_1, A_2, A_3, A_4. With respect to the Row-Column-Classification algorithm, however, such matrices are arranged in a different way, as depicted in Fig. 10. The submatrix A_1 consists of the uppermost $\frac{r}{2} - s$ rows of A, $A[0, *], A[1, *], \ldots, A[\frac{r}{2} - s - 1, *]$, A_2 of the s rows $A[\frac{r}{2} - s, *], \ldots, A[\frac{r}{2} - 1, *]$, A_3 of the s rows $A[\frac{r}{2}, *], \ldots, A[\frac{r}{2} + s - 1, *]$ and, finally, A_4 by the lowermost $\frac{r}{2} - s$ rows $A[\frac{r}{2} + s, *], \ldots, A[r-1, *]$.

The building blocks of the algorithm are the 2-*Partition* and 4-*Skewed-Partition* procedures. The former procedure receives as input $2s^2$ numbers and separates them in two

```
procedure Row-Column-Pass ($A, r, c$)
begin
/* Row-Pass */
    for $i = 0$ to $\frac{c}{2} - 1$ do
        4-Partition( $A[i\frac{r}{c} \ldots (i+1)\frac{r}{c} - 1, 0 \ldots \frac{c}{2} - 1]$, $A[\frac{r}{2} + i\frac{r}{c} \ldots (i+1)\frac{r}{c} - 1, 0 \ldots \frac{c}{2} - 1]$,
                     $A[i\frac{r}{c} \ldots (i+1)\frac{r}{c} - 1, \frac{c}{2} \ldots c - 1]$,$A[\frac{r}{2} + i\frac{r}{c} \ldots (i+1)\frac{r}{c} - 1, \frac{c}{2} \ldots c - 1]$);
    endfor
/* Column-Pass */
    for $i = 0$ to $\frac{c}{2} - 1$ do
        4-Partition($A[0 \ldots \frac{r}{2} - 1, i]$, $A[\frac{r}{2} \ldots r - 1, i]$, $A[0 \ldots \frac{r}{2} - 1, i + \frac{c}{2}]$, $A[\frac{r}{2} \ldots r - 1, i + \frac{c}{2}]$);
    endfor
end
```

Fig. 9. The Row-Column-Pass procedure.

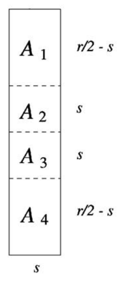

Fig. 10. The submatrices A_1, A_2, A_3, A_4 in which A is decomposed.

groups in such a way that the s^2 numbers of the first group are smaller than or equal to the s^2 numbers of the second group. In contrast, the latter procedure receives as input r numbers, grouped as C_0, C_1, C_2, C_3, where C_0 and C_3 have size $\frac{r}{2} - s$, while C_1 and C_2 have size s, and separates them in such a way that all the numbers in C_j are not larger than those in C_{j+1}, with $0 \leq j \leq 2$.

As shown in Fig. 11, the Three-Pass-Classification algorithm works in three passes: a *row-pass*, which examines A row-by-row, a *column-pass*, which explores A by columns, and a final pass on $A_2 \cup A_3$.

The row-pass consists of s iterations: During the ith iteration, with $0 \leq i \leq s-1$, the r numbers stored in the submatrix $G_i = A[i\frac{r}{s} \ldots (i+1)\frac{r}{s} - 1, *]$ are rearranged into four submatrices $G_{i,0}, \ldots, G_{i,3}$ of $\frac{r}{2s} - 1, 1, 1$, and $\frac{r}{2s} - 1$ rows, respectively, in such a way that all the numbers in $G_{i,j}$ are no larger than all the numbers in $G_{i,j+1}$, with $0 \leq j \leq 2$. Specifically, as depicted in Fig. 12, $G_{i,0}$ consists of the first $\frac{r}{2s} - 1$ rows of G_i, that is, $G_{i,0} = A[i\frac{r}{s} \ldots i\frac{r}{s} + \frac{r}{2s} - 2, *]$, $G_{i,1} = A[i\frac{r}{s} + \frac{r}{2s} - 1, *]$, $G_{i,2} = A[i\frac{r}{s} + \frac{r}{2s}, *]$ and, finally, $G_{i,3}$ consists of the last $\frac{r}{2s} - 1$ rows of G_i, namely, $G_{i,3} = A[i\frac{r}{s} + \frac{r}{2s} + 1 \ldots (i+1)\frac{r}{s} - 1, *]$. This partitioning operation is performed invoking the 4-*Skewed-Partition* procedure on G_i.

The column-pass consists also of s iterations: During the ith iteration, the r numbers of column $A[*, i]$ are rearranged, again by invoking the procedure 4-Skewed-Partition, into four groups of size $\frac{r}{2} - s, s, s$, and $\frac{r}{2} - s$, respectively, in such a way that all the numbers in column i of the submatrix A_j, that is, $A_j[*, i]$, are followed by those in column i of A_{j+1}, namely, $A_{j+1}[*, i]$, for $1 \leq j \leq 3$.

The final pass considers the $2s^2$ numbers stored in A_2 and A_3, which are rearranged, by the 2-*Partition* procedure, in such a way that all the s^2 numbers in A_2 are followed by the s^2 numbers in A_3.

In order to show how to implement the Three-Pass-Classification algorithm using an r-classifier, note first that the 2-Partition procedure can be simply implemented by a single r-classifier application. Indeed, after executing the row-pass and the column-pass, only $2s^2 \leq r$ numbers remain to be separated in A_2 and A_3 (if $2s^2 < r$, it is sufficient to fill the r-classifier with any additional $\frac{r}{2} - s^2$ numbers taken from A_1 and any additional $\frac{r}{2} - s^2$ numbers taken from A_4).

In regard to the 4-Skewed-Partition procedure, observe that an r-classifier alone can only partition r numbers into two halves, each of size $\frac{r}{2}$, such that all the numbers of the first half are not larger than those of the second half. Therefore, to accomplish the final goal of the procedure, one needs to extract the s largest numbers of the first half, and the s smallest numbers of the second half. This can be achieved using a classifier device a bit more complex than a simple r-classifier, as will be shown in the next section, which, however, has the same asymptotic depth and size as the simple r-classifier.

In conclusion, it is worth noting that the Three-Pass-Classification algorithm applies the r-classifier exactly $2s+1$ times, namely, the same number of times that Columnsort applies the r-sorter.

4.1 Correctness

The correctness of the Three-Pass-Classification algorithm results from the following lemma.

Lemma 3. *At the end of the second pass of the Three-Pass-Classification algorithm,*

- *no number in A_1 is larger than any number in $A_3 \cup A_4$, and*
- *no number in A_4 is smaller than any number in $A_1 \cup A_2$.*

Proof. The proof is similar to that of Lemma 2. As before, only the first claim is proven.

By contradiction, let $u \in A_1$ be strictly larger than one number $v \in A_3 \cup A_4$. Assume u and v belong, respectively, to columns $A[*, i]$ and $A[*, j]$, where $0 \leq i \neq j \leq s-1$. Since the columns were separated in four parts, there are at least $\frac{r}{2} + s + 1$ numbers not smaller than u in column $A[*, i]$

```
algorithm Three-Pass-Classification $(A, r, s)$
begin
/* let $A$ be perceived as the 4 matrices $A_1, A_2, A_3, A_4$ shown in Figure 10 */
/* Row-Pass */
    for $i = 0$ to $s-1$ do
        4-Skewed-Partition$(G_{i,0}, G_{i,1}, G_{i,2}, G_{i,3})$;
    endfor
/* Column-Pass */
    for $i = 0$ to $s-1$ do
        4-Skewed-Partition$(A_1[*, i], A_2[*, i], A_3[*, i], A_4[*, i])$;
    endfor
/* Final-Pass */
    2-Partition$(A_2, A_3)$;
end
```

Fig. 11. The Three-Pass-Classification algorithm.

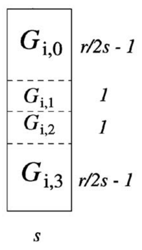

Fig. 12. The submatrices $G_{i,0}, G_{i,1}, G_{i,2}, G_{i,3}$ in which G_i is partitioned.

and at least $\frac{r}{2}+1$ numbers not larger than v in column $A[*, j]$. Let U and V denote these sets of numbers, respectively. Observe that all the elements, that at the end of the row-pass belonged to column i, remain in the same column at the end of the column-pass. Let us denote with $F_{k,t}$, where $0 \leq k \leq s-1$ and $0 \leq t \leq 3$, the numbers at the end of the row-pass, which were stored in column i of $G_{k,t}$.

Hence, every number e that belongs to $F_{k,2}$ is followed by $\frac{r}{2s}-1$ rows of G_k whose numbers are not smaller than e at the end of the row-pass, as well as at the end of the column-pass. Similarly, every number e that belongs to $F_{k,1}$ is followed by $\frac{r}{2s}$ rows of G_k and, therefore, by one more row of G_k with respect to $F_{k,2}$. Finally, every e belonging to $F_{k,0}$ is followed by $\frac{r}{2s}+1$ rows of G_k and, thus, by one more row of G_k with respect to $F_{k,1}$.

Consider now the set U of numbers in column i. By the previous observation, each number in U forces on the other columns of A, and especially on column j, some numbers not smaller than u and, therefore, it forbids some positions for the numbers V in column j. The amount of forbidden positions for V is minimized when U contains all the numbers in

$$\bigcup_{\substack{0 \leq k \leq s-1 \\ 1 \leq t \leq 3}} F_{k,t}.$$

Altogether, U forbids at least $\frac{r}{2}$ positions for the numbers V in column j. Since there are r positions available, out of which $\frac{r}{2}$ are forbidden, only $\frac{r}{2}$ positions are available for the numbers in V.

This shows that $u \geq v$ is impossible. In conclusion, every number in A_1 is followed by all the numbers in A_3 and A_4 and, therefore, A_1 cannot contain the median of A.□

In regard to the time complexity, note that different iterations of the row-pass and of the column-pass work on groups of disjoint rows and columns, respectively. Therefore, all the s iterations of the same pass can be performed in pipeline. Exploiting such a property, a classifier network for $s = \log r$ will be devised in the next section that achieves an optimal $O(\log r)$ time using an $O(r \log^2 r)$ work.

5 Hardware Implementation

In this section, a hardware algorithm is presented for solving the classification problem on $N = rs$ numbers, where $1 \leq s \leq r$ (for the sake of simplicity, it is assumed that r, $\log r$, and s are powers of two). The hardware algorithm behaves recursively as Row-Column-Classification, while the number of columns remains larger than $\log r$, but it acts as Three-Pass-Classification as soon as the number of columns becomes $\log r$.

First, an architectural framework is exhibited which consists of r memory modules and a slightly modified r-classifier network, which also includes some simple networks for performing maximum and minimum computations. Then, pipeline schemes are presented for the proposed algorithms which read/write a row or a column of A from/to the memory modules in constant time and achieve optimal time performance.

5.1 Architecture

Fig. 13 depicts the architecture for $r = 8$. The basic architectural features of the design include:

1. A *data memory* organized into r independent memory modules $M_0, M_1, \ldots, M_{r-1}$. Each M_i is randomly addressed by an address register AR_i, associated with an adder.
2. A set of r data registers, $R_0, R_1, \ldots, R_{r-1}$. In constant time, the content of the r data registers can be loaded in parallel into the r address registers, or can be stored in parallel into the r memory modules.

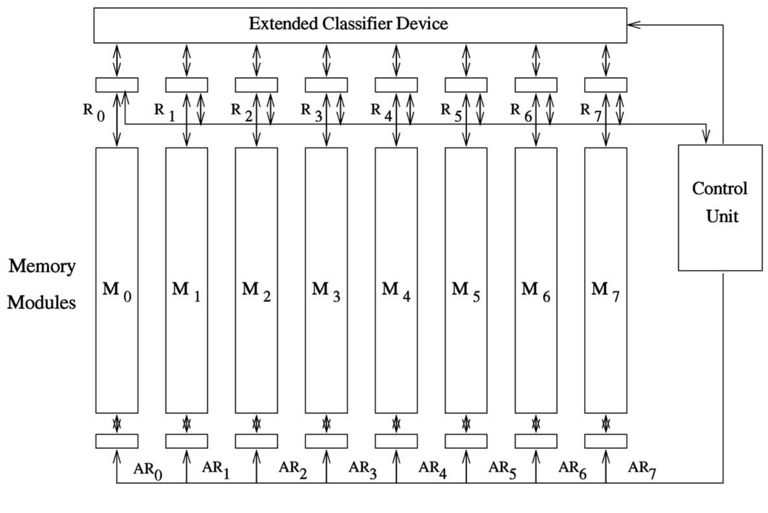

Fig. 13. The proposed architecture with $r = 8$.

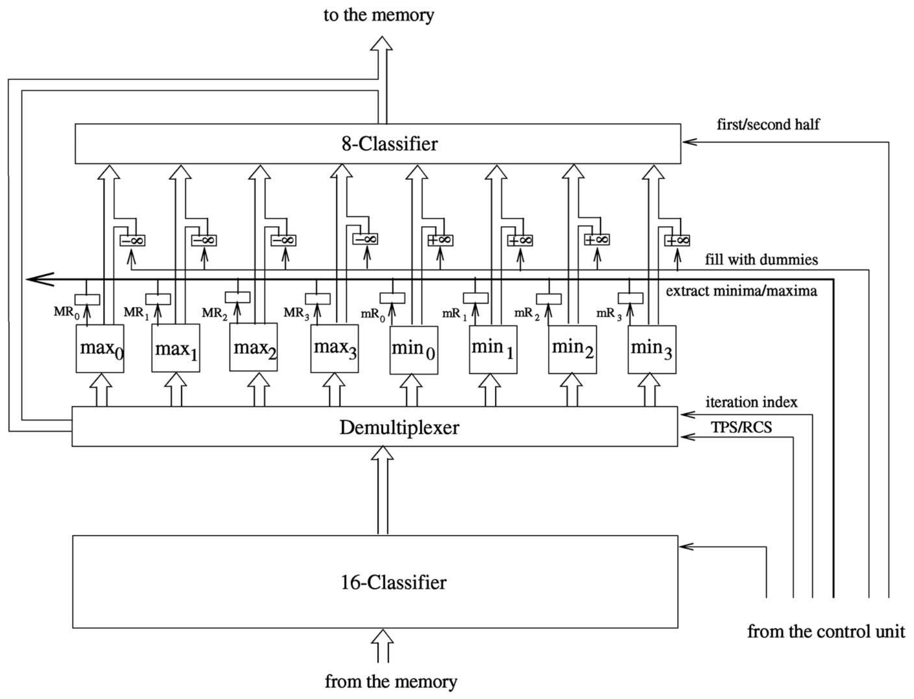

Fig. 14. The structure of the extended classifier for $r = 16$ and $\log r = 4$.

3. An extended classifier device consisting of:
    a. an *r-classifier* network of I/O size r and depth $O(\log r)$;
    b. $\log r$ networks $max_0, \dots, max_{\log r-1}$; each max_i has $O(r)$ comparators and $O(\log r)$ depth, is capable of performing a maximum computation, and is equipped with a register MR_i, which can store $\log r$ numbers;
    c. $\log r$ networks $min_0, \dots, min_{\log r-1}$; each min_i is analogous to max_i, but it is capable of performing a minimum computation and its register is denoted by mR_i;
    d. a *demultiplexer* to route the outcome of the r-classifier either to a suitable max_i/min_i or to the memory;
    e. an $(r - 2\log r)$-$classifier$ network of I/O size $r - 2\log r$ and depth $O(\log r)$.

    The structure of the extended classifier device is illustrated in Fig. 14 for $r = 16$ and $\log r = 4$.
4. A *control unit* (CU, for short) capable of performing simple arithmetic and logic operations and of generating control signals. The CU can broadcast an address to all memory modules and to the data registers and can read an element from any data registers. These operations are assumed to take constant time.

To achieve high performance for the hardware implementation, the r-classifier must be filled at each instant with a new set of r numbers. This can be accomplished only if *conflict-free* access is guaranteed to the memory modules, namely, only when all the r elements of the same row or column of A can be simultaneously read from or written to the r memory modules in constant time. Hereafter, it is assumed that A is stored in such a way that each column of A forms a *memory line*, namely, it is kept in r memory locations having the same address in all the modules. Precisely, the generic element $A[i,j]$ of column j is stored in position j of module M_i. However, in this way, each row is stored in the same memory module. Therefore, the elements of the same row cannot be retrieved conflict-free, but must be accessed one by one, requiring a time linear in the row length. To overcome this drawback, the hardware implementations of the proposed algorithms replace access to rows with access to *diagonals*. This does not hurt Lemmas 2 and 3 whose proofs are based on a counting argument consisting of how many numbers of a row intersect a column. Since such a quantity remains the same when replacing rows with diagonals, the correctness of the hardware implementation of the row-passes is guaranteed.

5.2 Implementation of Three-Pass-Classification

Consider the Three-Pass-Classification algorithm when the number s of columns is exactly $\log r$. The building block of the algorithm is the 4-Skewed-Partition procedure, which works on r numbers, corresponding either to a subset of $\frac{r}{\log r}$ consecutive rows or to a single column of A, depending whether a row-pass or a column-pass is performed.

During the ith iteration of the row-pass, the r-classifier is filled up with the $\frac{r}{\log r}$ rows of $G_i = A[i\frac{r}{\log r} \dots (i+1)\frac{r}{\log r} - 1, *]$ in order to separate them into the four submatrices $G_{i,0}, \dots, G_{i,3}$ of $\frac{r}{2\log r} - 1, 1, 1$, and $\frac{r}{2\log r} - 1$ rows, respectively. Since, as said before, the implementation accesses conflict-free diagonals instead of rows, the generic element $A[i\frac{r}{\log r} + h, k]$, belonging to the submatrix G_i, is retrieved from and stored back by the CU in position k of the memory module $M_{h\log r+(k+i) \bmod \log r}$, where $0 \le h \le \frac{r}{\log r} - 1$ and $0 \le k \le \log r - 1$. Then, in this way, during the ith iteration of the row-pass, each classifier call can access r locations conflict-free, one for each memory module.

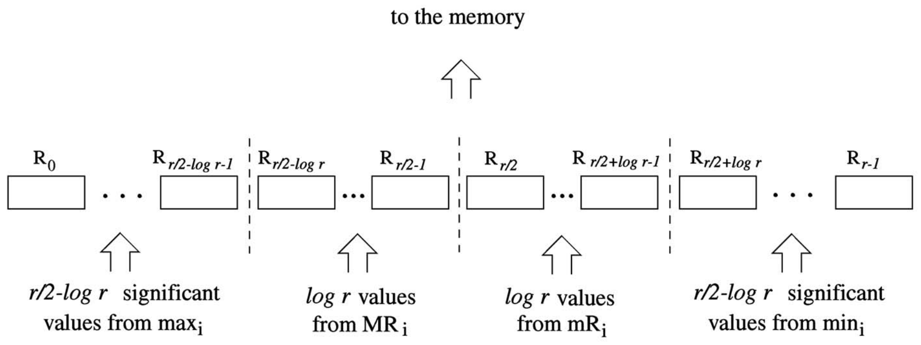

Fig. 15. The loading of the data registers at the end of the $i\mathrm{th}$ iteration of a row-pass or column-pass.

During the $i\mathrm{th}$ iteration of the column-pass, the four groups accessed by the 4-Skewed-Partition procedure correspond to a single column, which is stored in memory line i, whose elements can be retrieved and stored back by the CU without memory conflicts.

The $\log r$ iterations of the row-pass or column-pass are performed in pipeline, starting the $i\mathrm{th}$ iteration at time instant i. Let the r-classifier network work in $C_{\mathrm{JM}} \log r$ time, where C_{JM} is the constant required by the classifier described in [8]. Hence, the r-classifier ends to handle the $i\mathrm{th}$ iteration at time $i + C_{\mathrm{JM}} \log r$, with $0 \leq i \leq \log r - 1$.

During the generic $i\mathrm{th}$ iteration of the row-pass or column-pass, the smallest $r/2$ numbers, output by the r-classifier, are given in input to the max_i network, while the largest $r/2$ numbers are given to the min_i network. Then, for $\log r$ times, max_i (respectively, min_i) extracts in pipeline the maximum (respectively, minimum), stores it in its associated MR_i (respectively, mR_i) register, and replaces the extracted value with a dummy $-\infty$ (respectively, $+\infty$) value. In particular, max_i (respectively, min_i) extracts the first maximum (respectively, minimum) at time $i + C_{\mathrm{JM}} \log r + \log r$, and it extracts the $\log r\mathrm{th}$ maximum (respectively, minimum) of the same iteration at time $i + C_{\mathrm{JM}} \log r + 2\log r - 1$. Subsequently, the CU fills the $(r - 2\log r)$-classifier twice with the content of the minimum/maximum network, in order to clean the significant $\frac{r}{2} - \log r$ values from the $\log r$ dummy values. At instant $i + C_{\mathrm{JM}} \log r + 2\log r$, the CU fills the $(r - 2\log r)$-classifier network with the $\frac{r}{2}$ numbers still stored in max_i along with additional $\frac{r}{2} - 2\log r$ dummy $-\infty$ values. At the next instant, the CU fills the classifier with the other numbers stored in min_i along with additional $\frac{r}{2} - 2\log r$ dummy $+\infty$ values. Hence, every two instants of time, the $(r - 2\log r)$-classifier network is filled with the content of a different pair of maximum and minimum comparator networks, which correspond to different iterations of the same row-pass or column-pass. Once the $(r - 2\log r)$-classifier has separated the content of max_i, the CU moves the $\frac{r}{2} - \log r$ largest numbers (i.e., the significant values of max_i) to the data registers $R_0, \dots, R_{\frac{r}{2} - \log r - 1}$. At the next instant, the content of min_i has been separated and the CU moves the $\frac{r}{2} - \log r$ smallest numbers (i.e., the significant values of min_i) to $R_{\frac{r}{2} + \log r}, \dots, R_{r-1}$. Moreover, the CU also moves the $\log r$ numbers already stored in MR_i to $R_{\frac{r}{2} - \log r}, \dots, R_{\frac{r}{2} - 1}$ and those stored in mR_i to $R_{\frac{r}{2}}, \dots, R_{\frac{r}{2} + \log r - 1}$. The loading of the data registers is shown in Fig. 15. Hence, the $i\mathrm{th}$ iteration is concluded at time $i + C_{\mathrm{JM}} \log r + 2\log r + C_{\mathrm{JM}} \log(r - 2\log r) + 2$, storing back conflict-free the content of the data registers into the memory.

Observe that the $(r - 2\log r)$-classifier works in pipeline on all the $\log r$ iterations of the same row-pass or column-pass producing the output of the same iteration in two subsequent instants. Therefore, since the $(r - 2\log r)$-classifier works in $C_{\mathrm{JM}} \log(r - 2\log r)$ time, overall $(C_{\mathrm{JM}} + 3) \log r + C_{\mathrm{JM}} \log(r - 2\log r) + 1$ time is taken to accomplish an entire row-pass or column-pass.

The final pass of the Three-Pass-Classification algorithm is implemented filling in $\log r$ time the r-classifier network with the $2\log^2 r$ numbers of $A_2 \cup A_3$ along with any additional $\frac{r}{2} - \log^2 r$ numbers taken from A_1 and any additional $\frac{r}{2} - \log^2 r$ numbers taken from A_4. A single application of the r-classifier accomplishes the separation required by the final pass. Thus, the final pass takes time $(C_{\mathrm{JM}} + 1) \log r$.

Note that, to compute the actual median number of the entire matrix A, it is enough to extract the maximum from the smallest $\frac{r}{2}$ elements output by the final pass. This can be accomplished in $\log r$ time by using any maximum comparator network max_i.

Overall, since $C_{\mathrm{JM}} \log(r - 2\log r) < C_{\mathrm{JM}} \log r$, the classifier network based on the Three-Pass-Classification algorithm takes

$$2((C_{\mathrm{JM}} + 3)\log r + C_{\mathrm{JM}} \log(r - 2\log r) + 1) \\ + (C_{\mathrm{JM}} + 1)\log r < (5C_{\mathrm{JM}} + 7)\log r$$

time to solve the problem on $N = r\log r$ numbers. Such a time is optimal in order of magnitude due to the $\Omega(\log r)$ time lower bound given in [19]. Furthermore, the proposed algorithm is competitive versus Columnsort, even considering the constants hidden in the big-oh notation. Let $C_{\mathrm{AKS}} \log r$ be the time required by the AKS sorter. Columnsort involves three sorting passes: the first two pipelined over $s = \log r$ columns and the third one on a single column (see Section 2), for a total of $(2C_{\mathrm{AKS}} + 3)\log r$ time. Therefore, the Three-Pass-Classification algorithm is better than Columnsort to solve the classification problem on $r\log r$ items whenever

$$5C_{\mathrm{JM}} + 7 < 2C_{\mathrm{AKS}} + 3.$$

Noting that $C_{\mathrm{JM}} \approx 1.89 < 2$, as proven in [8], Columnsort could be better than the Three-Pass-Classification algorithm only when C_{AKS} would become smaller than $\frac{5}{2}C_{\mathrm{JM}} + 2 < 7$.

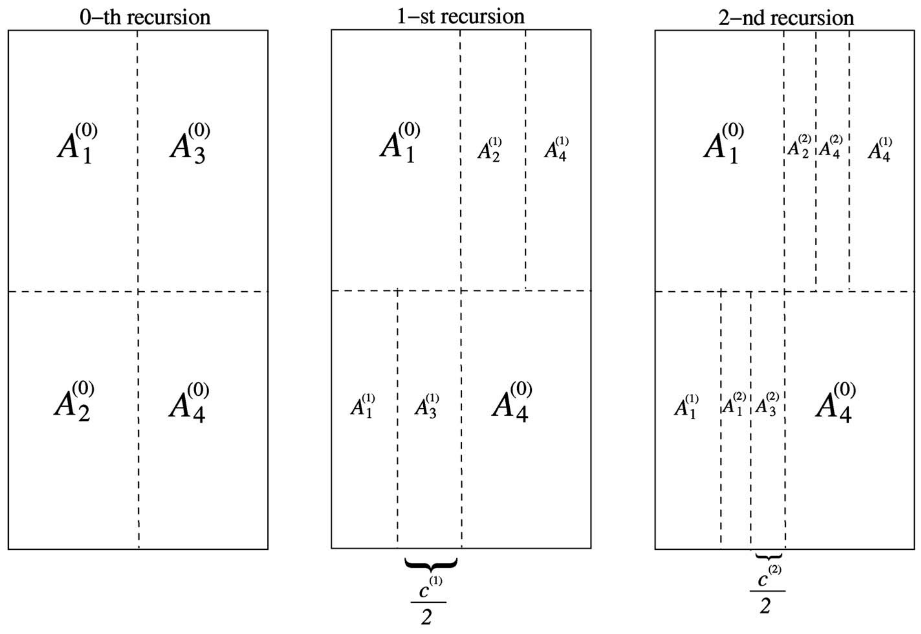

Fig. 16. In place partitioning of A at the $j\text{th}$ recursion, with $j = 0, 1, 2$.

This seems very unlikely because the current value of C_{AKS} is on the order of thousands [16].

To evaluate the work, observe that each of the $2 \log r$ maximum and minimum networks is implemented by a tree of $O(r)$ comparators, for a total of $O(r \log r)$ comparators. Since both the r-classifier and the $(r - 2\log r)$-classifier employ $O(r \log r)$ comparators, a total work of $O(r \log^2 r)$ is used.

Note that, if $s < \log r$, it is sufficient to add $\frac{r}{2}(\log r - s)$ dummy $-\infty$ values along with $\frac{r}{2}(\log r - s)$ dummy $+\infty$ values and, then, solve the so extended $r \times \log r$ problem, with no loss in performance.

5.3 Implementation of Row-Column-Classification

The recursion on the submatrix $A_2 \cup A_3$ in Fig. 6 can be realized as follows: The first call is an anomalous recursion, denoted as 0th recursion, which partitions A into $A_1^{(0)}, A_2^{(0)}, A_3^{(0)}$, and $A_4^{(0)}$ as described in Section 3. In the subsequent recursions, the hardware implementation works in place on the submatrices A_2 and A_3 obtained from the previous recursion. In the $j\text{th}$ recursion, the number of columns of each matrix $A_1^{(j)}, A_2^{(j)}, A_3^{(j)}, A_4^{(j)}$ is $c^{(j)} = \frac{s}{2^j}$. The partitioning of A is illustrated in Fig. 16.

The building block of the Row-Column-Pass procedure is the 4-Partition procedure of Fig. 8, which works on four groups C_0, C_1, C_2, C_3. The memory access, which replaces rows by diagonals, guarantees to conflict-free access r locations at a time, one for each memory module.

For the classifier network to operate at full capacity, the $j\text{th}$ recursion must take $O(c^{(j)})$ time. An efficient implementation of the Row-Column-Pass procedure can be provided which exploits, by means of an *interleaved pipelining*, the parallelism inherent in its $c^{(j)}$ iterations. Consider the generic $j\text{th}$ recursion and focus on the row-pass. The $\frac{c^{(j)}}{2}$ calls to the 4-Partition procedure are performed as follows: The computation starts with classifier call 0 performed in simple pipeline fashion on all the data given by Row-Column-Pass for the iterations $0, 1, \cdots, c^{(j)}/2$. Clearly, this is possible because the classifier call 0 is applied every time to a different set of data. Then, in a perfectly similar fashion, simple pipeline is used to carry out every classifier call 1 of 4-Partition on all the data given by Row-Column-Pass for the iterations $0, 1, \cdots, c^{(j)}/2$. The same approach is followed for the remaining classifier calls of 4-Partition. Moreover, the same interleaved pipelining strategy is used with the six classifier calls of 4-Partition within the column-pass.

At the $(j+1)\text{th}$ recursion, the number of columns and, hence, also the number of iterations, halves. Since the r-classifier works in $C_{\text{JM}} \log r$ time, where $C_{\text{JM}} < 2$, as soon as the number of iterations becomes $\log r$, the interleaved pipelining cannot be applied anymore without slowing the computation. The computation then proceeds as in the Three-Pass-Classification implementation described in the previous section where the number of columns is $\log r$.

To evaluate the time complexity, observe that in the $j\text{th}$ recursion Row-Column-Pass invokes $c^{(j)} = s/2^j$ times 4-Partition which, in turn, calls the classifier six times. Since a new classifier call starts executing at each subsequent instant, the overall time required by the interleaved pipeline is

$$\sum_{j=0}^{\ell} \frac{6s}{2^j} + C_{\text{JM}} \log r = O(s),$$

where $\ell = \log \frac{s}{\log r}$.

In addition, the final computation performed according to the Three-Pass-Classification implementation requires $O(\log r)$ time. Therefore, an optimal $O(s + \log r)$ time is required to solve the problem on $N = rs$ numbers, with $s \leq r$. Since an extended classifier network of depth $O(\log r)$ and $O(r \log r)$ comparators is employed, a total work of $O(rs \log r + r \log^2 r)$ is used.

6 Conclusions

This paper has shown how the well-known Leighton's Columnsort algorithm can be modified so as to solve the classification problem using classifier networks instead of sorting networks. In particular, two classification algorithms have been presented. Both algorithms apply the classifier no more times than Columnsort applies the sorter, but use a simpler and faster network.

The first algorithm takes a logarithmic number of passes and can be implemented using just an r-classifier. The second algorithm takes three passes and uses a slightly modified r-classifier which, however, has the same depth and the same number of comparators (in order of magnitude) as the simple r-classifier. In particular, such an algorithm shows that it is possible to build a classifier network that can solve the classification problem of $O(r \log r)$ numbers using optimal $O(\log r)$ time and $O(r \log r)$ comparators as an r-classifier. Furthermore, in such a case, the proposed algorithm beats Columnsort whenever the constant involved in the AKS network is greater than 7. Finally, these two algorithms can be combined together, leading to a hardware algorithm which solves the classification problem of $N = rs$ numbers, for $1 \leq s \leq r$, in optimal $O(\log r + s)$ time and using $O(r \log r(\log r + s))$ work.

It is worth noting that the extended classifier network has replaced a simple r-classifier only to implement the 4-Skewed-Partition procedure. However, given an (n, m)-classifier which classifies its n input numbers into the m smallest numbers and the $n - m$ largest ones, such a procedure could be easily implemented by connecting in cascade the output of an r-classifier with an $(r/2, r/2 - \log r)$-classifier and an $(r/2, \log r)$-classifier. The extended classifier has been introduced in the architecture because $\log r$ and $r/2 - \log r$ are not $\Omega(r)$. Indeed, according to [8], an (n, m)-classifier can be obtained maintaining the same time and work performances as an n-classifier only when $m = \Omega(n)$.

However, several questions still remain open. The $\Omega(N \log N)$ lower bound on the number of comparators given in [2] holds only for networks with I/O size N. On the other side, $\Omega(N)$ is a lower bound on the work for any algorithm using comparisons [5]. Hence, any hardware algorithm that uses an r-classifier has a trivial $\Omega(N + r \log r)$ lower bound on the work. Therefore, a challenge for the future is either to design a hardware algorithm that matches such a trivial lower bound or to prove a higher lower bound on the work. Moreover, one could generalize the methods presented here for solving the K-Classification problem for an arbitrary K, where it is asked to classify a set of N numbers so as to separate the K smallest numbers and the $N - K$ largest ones.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in $c \log n$ Parallel Steps," *Combinatorica,* vol. 3, pp. 1-19, 1983.

[2] V.E. Alekseyev, "Sorting Algorithms with Minimum Memory," *Kibernetica,* vol. 5, pp. 99-103, 1969.

[3] K.E. Batcher, "Sorting Networks and Their Applications," *Proc. AFIPS Conf.,* pp. 307-314, 1968.

[4] J. Belzile, Y. Savaria, D. Haccoun, and M. Chalifoux, "Bounds on the Performance of Partial Selection Networks," *IEEE Trans. Comm.,* vol. 43, pp. 1800-1809, 1995.

[5] M. Blum, R. W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan, "Time Bounds for Selection," *J. Computer and System Sciences,* vol. 7, pp. 448-461, 1973.

[6] G. Chaudhry and T.H. Cormen, "Stupid Columnsort Tricks," unpublished report, 2003.

[7] G. Chaudhry, T.H. Cormen, and L.F. Wisniewski, "Columnsort Lives! An Efficient Out-of-Core Sorting Program," *Proc. Symp. Parallel Algorithms and Architectures,* 2003.

[8] S. Jimbo and A. Maruoka, "A Method of Constructing Selection Networks with $O(\log n)$ Depth," *SIAM J. Computing,* vol. 25, pp. 709-739, 1996.

[9] M. Kutylowski, K. Lorys, B. Oesterdiekhoff, and R. Wanka, "Periodification Scheme: Constructing Sorting Networks with Constant Period," *J. ACM,* vol. 47, pp. 944-967, 2000.

[10] F.T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Computers,* vol. 34, pp. 344-354, 1985.

[11] F.T. Leighton, Y. Ma, and T. Suel, "On Probabilistic Networks for Selection, Merging and Sorting," *Theory of Computing Systems,* pp. 559-582, 1997.

[12] G.S. Manku, S. Rajagopalan, and B.G. Lindsay, "Approximate Medians and Other Quantiles in One Pass and with Limited Memory," *Proc. ACM Sigmod Int'l Conf. Data Management,* 1998.

[13] S. Olariu, M.C. Pinotti, and S.-Q. Zheng, "An Optimal Hardware-Algorithm for Selection Using a Fixed-Size Parallel Classifier Device," *Proc. Sixth Int'l Conf. High Performance Computing,* pp. 284-288, 1999.

[14] S. Olariu, M.C. Pinotti, and S.-Q. Zheng, "How to Sort N Items Using a Network of Fixed I/O," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, pp. 487-499, 1999.

[15] S. Olariu, M.C. Pinotti, and S.-Q. Zheng, "An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device," *IEEE Trans. Computers,* vol. 49, pp. 1310-1324, 2000.

[16] M.S. Paterson, "Improved Sorting Networks with $O(\log n)$ Depth," *Algorithmica,* vol. 5, pp. 75-92, 1990.

[17] N. Pippenger, "Selection Networks," *SIAM J. Computing,* vol. 20, pp. 878-887, 1991.

[18] B.W. Wah and K.L. Chen, "A Partitioning Approach to the Design of Selection Networks," *IEEE Trans. Computers,* vol. 33, pp. 261-268, 1984.

[19] A.C. Yao, "Bounds on Selection Networks," *SIAM J. Computing,* vol. 9, pp. 566-582, 1980.

Alan A. Bertossi received the Laurea degree summa cum laude in computer science from the University of Pisa (Italy) in 1979. Afterward, he worked as a system programmer and designer. From 1983 to 1994, he was with the University of Pisa as a research associate first, and later as an associate professor. From 1995 to 2001, he was with the University of Trento (Italy) as a full professor. Since 2002, he has been with the Department of Computer Science at the University of Bologna (Italy) as a professor of computer science. His main research interests are the computational aspects of high-performance, parallel, VLSI, distributed, fault-tolerant, and real-time systems. He has published about 40 refereed papers in international journals, as well as several other papers in international conferences, workshops, and encyclopedias. He has authored a book (on design and analysis of algorithms, in Italian) and has served as guest coeditor for special issues of *Algorithmica*, *Discrete Applied Mathematics*, and *Mobile Networks and Applications*. He is a member of the editorial board of *Information Processing Letters*. His biography is included in the editions of *Who's Who in the World* in 1999, and *Who's Who in Science and Engineering* in 2000. Since 1999, he has been a scientific collaborator at the Institute of Information Sciences and Technologies of the Italian National Research Council (ISTI-CNR, Pisa, Italy). From 2001-2003, he was the national coordinator of an Italian research project on algorithms for wireless networks.

Stephan Olariu received the MSc and PhD degrees in computer science from McGill University, Montreal, Canada in 1983 and 1986, respectively. In 1986, he joined the Computer Science Department at Old Dominion University, where he is now a full professor. He has published extensively in various journals, book chapters, and conference proceedings. His research interests include wireless networks and mobile computing, parallel and distributed systems, performance evaluation, and medical image processing. Professor Olariu serves on the editorial board of several archival journals, including *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Parallel and Distributed Computing*, *International Journal of Foundations of Computer Science*, *Journal of Supercomputing*, *International Journal of Computer Mathematics*, *VLSI Design*, and *Parallel Algorithms and Applications*.

M. Cristina Pinotti received the doctorate degree cum laude in computer science from the University of Pisa, Italy, in 1986. Since November 2003, she has been an associate professor in the Department of Computer Science and Mathematics at the University of Perugia, Italy. From 2000-2003, she was an associate professor in the Department of Computer Science and Telecommunications at the University of Trento, Italy. Prior to 2000, she was a researcher with the Consiglio Nazionale delle Ricerche at the Istituto di Elaborazione dell'Informazione, Pisa. She was a visiting researcher in the Department of Computer Science, University of North Texas, Denton, and in the Department of Computer Science, Old Dominion University, Norfolk, Virginia. Her research interests include mobile and wireless networks, parallel data structures and multiprocessor interconnection networks, design and analysis of parallel algorithms, I/O external memories, VLSI layouts, computer arithmetic, and residue number systems. Each year she serves on the program committee of several IEEE and ACM workshops and conferences.

Si-Qing Zheng received the PhD degree in computer science from the University of California, Santa Barbara, in 1987. After having been on the faculty of Louisiana State University for 11 years since 1987, he joined the University of Texas at Dallas, where he is currently a professor of computer science. Dr. Zheng's research interests include algorithms, computer architectures, networks, parallel and distributed processing, telecommunication, and VLSI design. He has published extensively in these areas. He served as the program committee chairman of numerous international conferences and the editor of several professional journals.