An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device

Stephan Olariu, *Member*, *IEEE*, M. Cristina Pinotti, *Member*, *IEEE Computer Society*, and Si Qing Zheng, *Senior Member*, *IEEE*

Abstract—We present a hardware-algorithm for sorting N elements using either a *p*-sorter or a sorting network of fixed I/O size p while strictly enforcing conflict-free memory accesses. To the best of our knowledge, this is the first realistic design that achieves optimal time performance, running in $\Theta(\frac{N \log N}{p \log p})$ time for all ranges of N. Our result completely resolves the problem of designing an implementable, time-optimal algorithm for sorting N elements using a *p*-sorter. More importantly, however, our result shows that, in order to achieve optimal time performance, all that is needed is a sorting network of depth $O(\log^2 p)$ such as, for example, Batcher's classic bitonic sorting network.

Index Terms—Special-purpose architectures, hardware-algorithms, sorting networks, columnsort, VLSI.

1 Introduction

Recent advances in VLSI have made it possible to implement algorithm-structured chips as building blocks for high-performance computing systems. Since sorting is one of the most fundamental computing problems, it makes sense to endow general-purpose computer systems with a special-purpose parallel sorting device, invoked whenever its services are needed.

In this article, we address the problem of sorting N elements using a sorting device of I/O size p, where N is arbitrary and p is fixed. The sorting device used is either a *p*-sorter or a sorting network of fixed I/O size p. We assume that the input, as well as the partial results, reside in several constant-port memory modules. In addition to achieving time-optimality, it is crucial that we sort without memory access conflicts. In real-life applications, the number N of elements to be sorted is much larger than the fixed size p that a sorting device can accommodate. In such a situation, the sorting device must be used repeatedly in order to sort the input. The following natural question arises: "How should one schedule memory accesses and the calls to the sorting device in order to achieve the best possible sorting performance?" Clearly, if this question does not find an appropriate answer, the power of the sorting device will not be fully utilized.

A *p*-sorter is a sorting device capable of sorting p elements in constant time. Computing models for a *p*-sorter do exist. For example, it is known that p elements can be sorted in $O(1)$ time on a $p \times p$ reconfigurable mesh [3], [7], [8]. Beigel and Gill [2] showed that the task of sorting N elements, $N \geq p$, requires $\Omega(\frac{N \log N}{p \log p})$ calls to a *p*-sorter and presented an algorithm that achieves this bound. However, their algorithm assumes that the p inputs to the *p*-sorter can be fetched in unit time, irrespective of their location in memory. Since, in general, the address patterns of the operands of *p*-sorter operations are irregular, it appears that the algorithm of [2] cannot realistically achieve the time complexity of $\Theta(\frac{N \log N}{p \log p})$, unless one can solve in constant time the addressing problem inherent in accessing the p inputs to the *p*-sorter and in scattering the output back into memory. In spite of this, the result of [2] poses an interesting open problem, namely that of designing an implementable $\Theta(\frac{N \log N}{p \log p})$ time sorting algorithm that uses a *p*-sorter.

Consider an algorithm A that sorts N elements using a *p*-sorter in $O(f(N,p))$ time. It is not clear that algorithm A also sorts N elements using a sorting network $\mathcal{T}$ of I/O size p in $O(f(N,p))$ time. The main reason is that the task of sorting p elements using the network $\mathcal{T}$ requires $O(D(\mathcal{T}))$ time, i.e., time proportional to the depth $D(\mathcal{T})$, which is the maximum number of nodes on a path from an input to an output, of the network. Thus, if each *p*-sorter operation is replaced naively by an individual application of $\mathcal{T}$, the time required for sorting becomes $O(D(\mathcal{T}) \cdot f(N,p))$. To eliminate this $O(D(\mathcal{T}))$ slowdown factor, the network must be used in a pipelined fashion. In turn, pipelining requires that sufficient parallelism in the *p*-sorter operations be identified and exploited. Recently, Olariu et al. [9] introduced a simple but restrictive design—the row merge model—and showed that, in this model, N elements can be sorted in $\Theta(\frac{N}{p \log N})$ time using either a

- *S. Olariu is with the Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162. E-mail: olariu@cs.odu.edu.*
- *M.C. Pinotti is with the Dipartimento di Matematica, Università degli Studi di Trento, Via Sommarive 14, Povo (Trento), Italy. E-mail: pinotti@science.unitn.it.*
- *S.Q. Zheng is with the Department of Computer Science, Box 830688, MS EC31, University of Texas at Dallas, Richardson, TX 75083-0688. E-mail: sizheng@utdallas.edu.*

Manuscript received 4 June 1997; accepted 7 Aug. 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 105170.

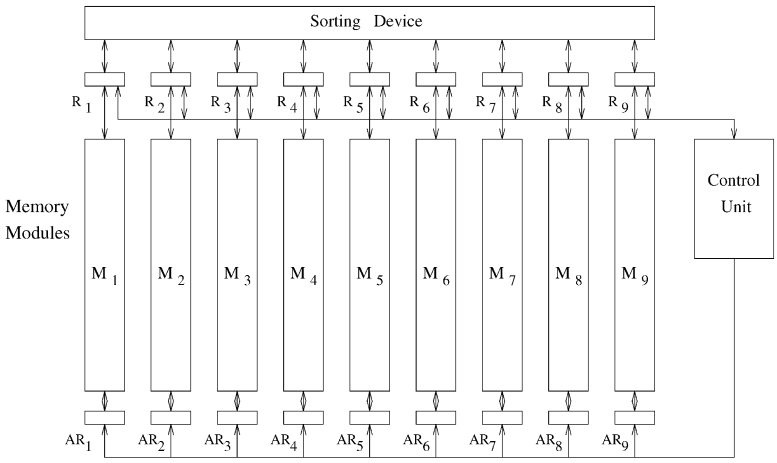

Fig. 1. The proposed architecture for $p = 9$.

p-sorter or a sorting network of I/O size p. We would like to memtion that Zhang and Zheng ([10]) proposed a powerful VLSI architecture which can use a p-sorter or a sorting network of I/O size p to sort N elements in $O(\frac{N}{p \log p})$ time. The lower bound of $\Omega(\frac{N \log N}{p \log p})$ does not apply to this architecture since its memory modules are implemented as linear systolic arrays in which each memory word is equipped with a comparator and is actually a small processing element.

To achieve better sorting performance, a new algorithm-structured architecture must be designed. This involves devising a sorting algorithm suitable for hardware implementation and, at the same time, an architecture on which the algorithm can be executed directly. Such an algorithm-architecture combination is commonly referred to as a *hardware-algorithm*. The major contribution of this article is to present the first realistic hardware-algorithm design for sorting an arbitrary number of input elements using a fixed-size sorting device in optimal time, while strictly enforcing conflict-free memory accesses. We introduce a parallel sorting architecture specially designed for implementing a carefully designed algorithm. The components of this architecture include a parallel sorting device, a set of random-access memory modules, a set of conventional registers, and a control unit. This architecture is very simple and feasible for VLSI realization.

We show that, in our architectural model, N elements can be sorted in $\Theta(\frac{N \log N}{p \log p})$ time using either a p-sorter or a sorting network of fixed I/O size p and depth $O(\log^2 p)$. In conjunction with the theoretical work of [2], our result completely resolves the problem of designing an implementable, time optimal, algorithm for sorting N elements using a p-sorter. More importantly, however, our result shows that, in order to achieve optimal sorting performance, a p-sorter is not really necessary: All that is needed is a sorting network of depth $O(\log^2 p)$ such as, for example, Batcher's classic bitonic sorting network. As we see it, this is exceedingly important since any known implementation of a p-sorter requires powerful processing elements, whereas Batcher's bitonic sort network uses simple comparators.

2 Architectural Assumptions

In this section, we describe the architectural framework within which we specify our optimal sorting algorithm using a fixed-size sorting device. We consider that a sequential sorting algorithm is adequate for the case $N < p^2$. Consequently, from now on, we assume that

$$N \geq p^2. \tag{1}$$

This assumption implies that, just for addressing purposes, we need at least $2 \log p$ bits.[1] For the reader's convenience, Fig. 1 depicts our design for $p = 9$. To keep the figure simple, control signal lines are not shown. The basic architectural assumptions of our sorting model include:

1. A *data memory* organized into p independent, constant-port, memory modules $M_1, M_2, \ldots, M_p$. Each word is assumed to have a length of w bits, with $w \geq 2 \log p$. We assume that the N input elements are distributed evenly, but arbitrarily, among the p memory modules. The words having the same address in all memory modules are referred to as a *memory row*. Each memory module M_i is randomly addressed by an *address register* AR_i, associated with an adder. Register AR_i can be loaded with a word read from memory module M_i or by a row address broadcast from the CU (see below).
2. A set of *data registers*, R_i, $(1 \leq i \leq p)$, each capable of storing a $(w + 1.5 \log p)$-bit word. We refer to the word stored in register R_i as a *composed word* since it consists of three fields:
   - an *element field* of w bits for storing an element,
   - a *long auxiliary field* of $\log p$ bits, and
   - a *short auxiliary field* of $0.5 \log p$ bits.

   These fields are arranged such that the element field is to the left of the long auxiliary field, which is to the left of the short auxiliary field. Each field of register R_i can be loaded independently from memory module M_i, from the ith output of the sorting

1. In the remainder of this article, all logarithms are assumed to be base 2.

device, or by a broadcast from the CU. The output of register R_i is connected to the ith input of the sorting device, to the CU, and to memory module M_i.

We assume that:

- In constant time, the p elements in the data registers can be loaded into the address registers or can be stored into the p modules addressed by the address registers.
- The bits of any field of register R_i, $(1 \leq i \leq p)$, can be set/reset to all 0s in constant time.
- All the fields of data register R_i, $(1 \leq i \leq p)$, can be compared with a particular value and each of the individual fields can be set to a special value depending on the outcome of the comparison. Moreover, this parallel *compare-and-set* operation takes constant time.

3. A sorting device of fixed I/O size p, in the form of a p-sorter or of a sorting network of depth $O(\log^2 p)$. We assume that the sorting device provides data paths of width $w + 1.5 \log p$ bits from its input to its output. The sorting device can be used to sort composed words on any combination of their element or auxiliary fields. In case a sorting network is used as the sorting device, it is assumed that the sorting network can operate in pipelined fashion.
4. A *control unit* (CU, for short), consisting of a *control processor* capable of performing simple arithmetic and logic operations and of a *control memory* used to store the control program as well as the control data. The CU generates control signals for the sorting device, for the registers, and for memory accesses. The CU can broadcast an address or an element to all memory modules and/or to the data registers, and can read an element from any data register. We assume that these operations take constant time.

Described above are minimum hardware requirements for our architectural model. In case a sorting network is used as the sorting device, one can use a "half-pipelining" scheme: The input to the network is provided in groups of D rows. The next group is supplied only after the output of the previous group is obtained. D is the depth of the sorting network. For the sorting network to operate at full capacity, one may add an additional set of address (resp. data) registers. One set of address (resp. data) registers is used for read operations, while the order set is used for write operations; both operations are performed concurrently.

Let us now estimate the VLSI area that our design uses for hardware other than data memory, the sorting device, and the CU under the word model, i.e., assuming that the word length w is a constant. We exclude the area taken by the CU: This is because, in a high-performance computer system, one of the processors can be assigned the task of controlling the parallel sorting subsystem. Clearly, the extra area is only that used for the address and the data registers and this amounts to $O(p)$—which does not exceed the VLSI area of any implementation of a p-sorter or of a sorting network of I/O size p.

We do not include the VLSI area for running the data memory address bus, which has a width of $\log \frac{N}{p}$ bits, and the control signal lines to data memory and to the sorting device since they are needed for any architecture involving a data memory and a sorting device. It should be pointed out that, for any architecture that has p memory modules involving a total of $N \geq p^2$ words, the control circuitry itself requires at least $\Omega(\max\{\log p, \log \frac{N}{p}\}) = \Omega(\log \frac{N}{p})$ VLSI area. Since the operations performed by the control processor are simple, we can assume that it takes constant area. The length of the control memory words is at least $\log \frac{N}{p}$, which is the length of data memory addresses. As will become apparent, our algorithms require $O(\frac{N}{p})$ rows of data memory and, consequently, the control memory words have length $O(\log \frac{N}{p})$. The control program is very simple and takes constant memory. However, $O(\frac{N}{p^{3/2}})$ control words are used for control information, which can be stored in data memory.

3 An Extended Columnsort Algorithm

In this section, we present an extension of the well-known Columnsort algorithm [5]. This extended Columnsort algorithm will be implemented in our architectural model and will be invoked repeatedly when sorting a large number of elements. There are two known versions of Columnsort [5], [6]: One involves eight steps, the other seven. We provide an extension of the 8-step Columnsort because the 7-step version does not map well to our architecture.

Columnsort was designed to sort, in column-major order, a matrix of r rows and s columns. The "classic" Columnsort contains eight steps. The odd-numbered steps involve sorting each of the columns of the matrix independently. The even-numbered steps permute the elements of the matrix in various ways. The permutation of Step 2 picks up the elements in column-major order and lays them down in row-major order. The permutation of Step 4 is just the reverse of that in Step 2. The permutation of Step 6 amounts to a $\lfloor \frac{r}{2} \rfloor$ shift of the elements in each column. The permutation of Step 8 is the reverse of the permutation in Step 6. The 8-step Columnsort works under the assumption that $r \geq 2(s-1)^2$. In [5], Leighton poses as an open problem to extend the range of applicability of Columnsort without changing the algorithm "drastically." We provide such an extension. We show that one additional sorting step is necessary and sufficient to complete the sorting in case $r \geq s(s-1)$. Our extension can be seen as trading one additional sorting step for a larger range of applicability of the algorithm.

Fig. 2 shows a matrix of r rows and s columns with $r = s(s-1)$ for which the condition $r \geq 2(s-1)^2$ is not satisfied. The first eight steps of this example correspond to the 8-step Columnsort algorithm which does not produce a sorted matrix. By adding one more step, Step 9, in which the elements in each column are sorted, we obtain an extended Columnsort algorithm. We assume a matrix M of r rows and s columns, numbered from 0 to $r-1$ and from 0 to $s-1$, respectively. Our arguments rely, in part, on the

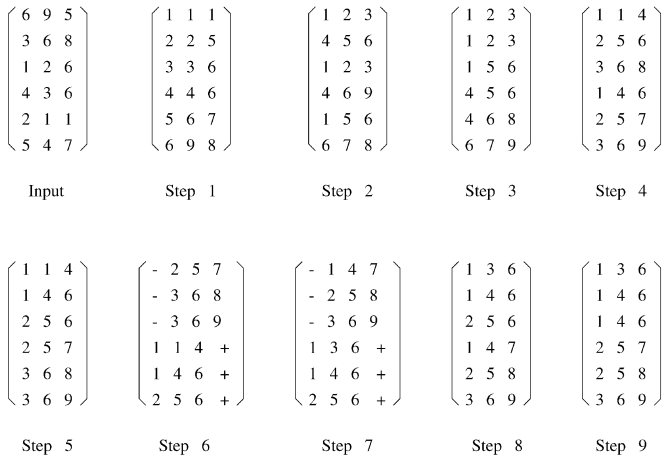

Fig. 2. Step-by-step application of the extended Columnsort algorithm. The first eight steps correspond to the classic 8-step Columnsort.

following well-known gem of computer science mentioned by Knuth [4]:

Proposition 1. *Let M be a matrix whose rows are sorted. After sorting the columns, the rows remain sorted.*

The following result was proven in [5].

Lemma 1. *If some element x ends up in position $M[i,j]$ at the end of Step 3, then x has rank at least $si + sj - (s-1)^2$.*

The following result was mentioned without proof in [5].

Lemma 2. *If element x ends up in position $M[i,j]$ at the end of Step 3, then its rank is at most $si + sj$.*

Proof. We are interested in determining a lower bound on the number of elements known to be larger than or equal to x. For this purpose, we note that, since at the end of Step 3, element x was in position $M(i,j)$, $r-i$ elements in column j are known to be larger than or equal to x. Among these, at most s are known to be smaller than or equal to $s-j$ elements in their columns at the end of Step 1. The remaining $r-i-s$ elements must be smaller than or equal to s other elements in their column at the end of Step 1. Consequently, x is known to be smaller than or equal to at least

$$s(s-j) + (r-i-s)s = rs - (si+sj) = n - (si+sj)$$

elements of M. It follows that the rank of x is at most $si + sj$, as claimed. □

For later reference, we now choose r such that

$$s(s-1) \leq r. \tag{2}$$

Lemma 3. *If some element x ends up in column c at the end of Step 4, then the correct position of x in the sorted matrix is in one of the pairs of columns $(c-1,c)$ or $(c,c+1)$.*

Proof. Consider, again, a generic element x that ended up in position $M(i,j)$ at the end of Step 3. The permutation specific to Step 4 guarantees that x will be moved, in Step 4, to a position that corresponds, in the sorted matrix, to the element of rank $si + j$. In general, this is not the correct position of x. However, as we shall prove, x is "close" to its correct position in the following sense: If x is in column c at the end of Step 4, then, in the sorted matrix, x must be in one of the pairs of columns $(c-1,c)$ or $(c,c+1)$.

Recall that, by virtue of Lemmas 1 and 2, combined, x has rank no smaller than $si + sj - (s-1)^2$ and no larger than $si + sj$. Moreover, simple algebraic manipulations show that

$$si + sj - (s-1)^2 \leq si + j \leq si + sj. \tag{3}$$

Now, consider the elements y and z of ranks $si + sj - (s-1)^2$ and $si + sj$, respectively. The number $N(y,z)$ of elements of the matrix M lying between y and z, in sorted order, is:

$$N(y,z) = si + sj - si - sj + (s-1)^2 + 1 = (s-1)^2 + 1$$

and, so, by (2), we have

$$N(y,z) = (s-1)^2 + 1 \leq r. \tag{4}$$

Observe that (4) implies that y and z must lie in *adjacent* columns of the sorted matrix. As we saw, at the end of Step 4, x lies in the position corresponding to the element of rank $is + j$ in the sorted matrix. Now, (3) confirms that x lies somewhere between y and z. Assume that x lies in column c at the end of Step 4. Thus, the correct position of x is in one of the columns $c-1$ or c in case z is in the same column as x and in one of the columns c or $c+1$ if y is in the same column as x. □

Lemma 4. *The rows of M are sorted at the end of Step 4.*

Proof. Consider an arbitrary column k ($0 \leq k \leq s-1$) at the end of Step 3. The permutation specified in Step 4 guarantees that the first $\frac{r}{s}$ elements in column k will appear in positions $k, k+s, k+2s, \cdots, k+r-s$ in column 0; the next group of $\frac{r}{s}$ elements will appear in positions $k, k+s, k+2s, \cdots, k+r-s$ of column 1, and so on. Since the columns were sorted at the end of Step 3, it follows that all the rows $k, k+s, k+2s, \ldots, k+r-s$ of M are sorted at the end of Step 4. Since k was arbitrary, the conclusion follows. □

Lemma 5. *If some element x is in the bottom half of column c at the end of Step 5, then its correct position in the sorted matrix is in one of the columns c or $c+1$.*

Proof. By Lemma 3, we know that the correct position of x is in one of the pairs of columns $(c-1,c)$ or $(c,c+1)$. Thus, to prove the claim we only need to show that x cannot be in column $c-1$. For this purpose, we begin by observing that by Proposition 1 and by Lemma 4, combined, the rows and columns are sorted at the end of Step 5. Now, suppose that element x ends up in row t, $t \geq \lfloor \frac{r}{2} \rfloor$, at the end of Step 5. If x belongs to column $c-1$ in the sorted matrix, then all the elements of the matrix in columns $c-1$ and c belonging to rows $0, 1, \ldots, t$ must belong to column $c-1$ or below. By Lemma 3, all elements that are already in columns $0, 1, \ldots, c-2$ must belong to columns $0, 1, \ldots, c-1$ in the sorted matrix.

Thus, at least $2(\lfloor \frac{r}{2} \rfloor + 1) > r$ additional elements must belong to column $c-1$ or below, a contradiction. □

In a perfectly similar way, one can prove the following result.

Lemma 6. *If some element is in the top half of column c at the end of Step 5, then its correct position in the sorted matrix is in one of the columns $c-1$ or c.*

Now, suppose that we find ourselves at the end of Step 8 of the 8-step Columnsort.

Lemma 7. *Every item x that is in column c at the end of Step 8 must be in column c in the sorted matrix.*

Proof. We begin by showing that

$$\text{no element in column } c \text{ can be in column } c-1. \tag{5}$$

We proceed by induction on c. The basis is trivial: No element in column 0 can lie in the column to its left. Assume that (5) is true for all columns less than c. In other words, no element that ends up in one of the columns $0, 1, \ldots, c-1$ at the end of Step 8 can lie in the column to its left. We only need to prove that the statement also holds for column c. To see that this must be the case, consider first an element u that lies in the bottom half of column c at the end of Step 8. At the end of Step 5, u must have been either in the bottom half of column c or in the top half of column $c+1$. If u belonged to the bottom half of column c, then, by Lemma 6, it must belong to columns c or $c+1$ in the sorted matrix. If u belonged to the top half of column $c+1$, then, by Lemma 5, it must belong to columns c or $c+1$ in the sorted matrix. Therefore, in either case, u cannot belong to column $c-1$.

Next, consider an element v that lies in the top half of column c at the end of Step 8. If v belonged to column $c-1$, then all the elements in the bottom half of column $c-1$, as well as the elements occurring above v in column c, must belong to column $c-1$. By the induction hypothesis, no element that lies in column $c-1$ at the end of Step 8 can lie in column $c-2$. By Lemmas 5 and 6 combined, no element that lies in the top half of column $c-1$ can belong to column c. But now, we have reached a contradiction: Column $c-1$ must contain more than r elements. Thus, (5) must hold.

What we just proved is that no element in a column can belong to the column to its left. A symmetric argument shows that no element belongs to the column immediately to its right, completing the proof. □

By Lemma 7, one more sorting step completes the task. Thus, we have obtained a 9-step Columnsort that trades an additional sorting step for a larger range of r versus s.

Theorem 1. *The extended 9-step Columnsort algorithm correctly sorts an $r \times s$ matrix such that $r \geq s(s-1)$.*

4 The Basic Algorithm

In this section, we show how to sort, in row-major order, m, $1 \leq m \leq p^{\frac{1}{2}}$, memory rows using our architectural model while enforcing conflict-free memory accesses. The resulting algorithm, referred to as the *basic algorithm*, will turn out to be the first stepping stone in the design of our time-optimal sorting algorithm. The basic algorithm is an implementation of the extended Columnsort discussed in Section 3 with $m = s$ and $p = r$.

Our presentation will focus on the efficient use of a generic sorting device of I/O size p. With this in mind, we shall keep track of the following two parameters that will become key ingredients in evaluating the running time of the algorithm:

- the number of calls to the sorting device, and
- the amount of time required by all the data movement tasks that do not involve sorting.

Assume that we have to sort, in row-major order, the elements in $m = p^{\frac{1}{2}}$ memory rows. The case $1 \leq m < p^{\frac{1}{2}}$ is perfectly similar. We assume, without loss of generality, that the input is placed, in some order, in memory rows $a+1$ through $a+p^{\frac{1}{2}}$ for some integer $a \geq 0$. The sorted elements will be placed in memory rows $b+1$ through $b+p^{\frac{1}{2}}$ such that the ranges $[a+1, a+p^{\frac{1}{2}}]$ and $[b+1, b+p^{\frac{1}{2}}]$ do not overlap.

Step 1: Sort all the rows independently.

This step consists of the following loop:

for $i = a+1$ **to** $a+p^{\frac{1}{2}}$ **do**
  read the ith memory row and sort it in nondecreasing order using the sorting device;
  let $x_1 \leq x_2 \leq \cdots \leq x_p$ be the resulting sorted sequence;
  for all j, $1 \leq j \leq p$ **do in parallel**
    store x_j in the ith word of memory module M_j
  endfor
endfor

Step 1 requires $p^{\frac{1}{2}}$ calls to the sorting device and $O(p^{\frac{1}{2}})$ time for data movement not involving sorting.

Step 2: Permuting rows.

The permutation specific to Step 2 of Columnsort prescribes picking up the elements in each memory row and laying them down column by column. For an illustration, consider the case $p = 9$, with the initial element distribution featured in the following matrix:

$$\begin{array}{ccccccccc} M_1 & M_2 & M_3 & M_4 & M_5 & M_6 & M_7 & M_8 & M_9 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1' & 2' & 3' & 4' & 5' & 6' & 7' & 8' & 9' \\ 1'' & 2'' & 3'' & 4'' & 5'' & 6'' & 7'' & 8'' & 9'' \end{array} \tag{6}$$

At the end of Step 2, the permuted matrix reads:

$$\begin{array}{ccccccccc} M_1 & M_2 & M_3 & M_4 & M_5 & M_6 & M_7 & M_8 & M_9 \\ 1 & 4 & 7 & 1' & 4' & 7' & 1'' & 4'' & 7'' \\ 2 & 5 & 8 & 2' & 5' & 8' & 2'' & 5'' & 8'' \\ 3 & 6 & 9 & 3' & 6' & 9' & 3'' & 6'' & 9'' \end{array}$$

A careful examination of the permuted matrix reveals that consecutive elements in the same memory row will end up in the same memory module (e.g., elements 1, 2, 3 will occur in memory module M_1). Therefore, in order to achieve the desired permutation without memory-access conflicts, one has to devise a different way of picking up the

elements in various memory rows. For this purpose, we find it convenient to view each element x stored in a memory module as an ordered triple $\langle x, row(x), module(x)\rangle$, where $row(x)$ and $module(x)$ stand for the identity of the memory row and of the memory module, respectively, containing x. Further, we let $row(x)|module(x)$ denote the binary number obtained by concatenating the binary representations of $row(x)$ and $module(x)$. The details are spelled out in the following procedure.

procedure *PERMUTE*
begin
  for $i = 0$ **to** $p^{\frac{1}{2}} - 1$ **do**
    for all j, $1 \leq j \leq p$, **do in parallel**
      read the $((i + j - 1) \bmod p^{\frac{1}{2}} + a + 1)$th word of memory module M_j
    endfor
    using the sorting device, sort the p elements in nondecreasing order of $row(x)|module(x)$;
    let $x_1 \leq x_2 \leq \cdots \leq x_p$ be the resulting sorted sequence;
    for all j, $1 \leq j \leq p$, **do in parallel**
      store x_j in the $((module(x_j) - 1) \bmod p^{\frac{1}{2}} + b + 1)$th word of memory module M_j
    endfor
  endfor
end

Clearly, this procedure involves $p^{\frac{1}{2}}$ iterations. In each iteration, p words are read, one from each memory module, sorted, and then written back into memory, one word per module, with no read and write memory access conflicts. It would seem as though each memory module requires an arithmetic unit to compute the address of the word to be accessed in each iteration. In fact, as we now point out, such arithmetic capabilities are not required. Specifically, we can use $p^{\frac{1}{2}}$ memory rows to store "offsets" used for memory access operations. For the above example, the offsets are:

$$\begin{array}{ccccccccc} M_1 & M_2 & M_3 & M_4 & M_5 & M_6 & M_7 & M_8 & M_9 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 0 \\ 2 & 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 \end{array} \quad (7)$$

At the beginning of Step 2, all the address registers contain $a + 1$. In the first iteration, the entries in the first row of the offset matrix are added to the contents of the address registers, guaranteeing that the correct word in each memory module is being accessed. As an illustration, referring to (7), we note that the offsets in the first row indicate that the words involved in the read operation will be found at address $a + 1 + 0$ in memory module M_1, address $a + 1 + 1 = a + 2$ in memory module M_2, address $a + 1 + 2 = a + 3$ in memory module M_3, and so on.

The key observation for understanding what happens in all the iterations is that, in any column of the offset matrix (7), once the entry in the first row is available, the subsequent elements in the same column can be generated by modulo $p^{\frac{1}{2}}$ arithmetic. In our architecture, this computation can be performed by the adder associated with each address register. In turn, this observation implies that, in fact, the offset matrix need not be stored at all, as its entries can be generated on the fly.

Yet another important point to note is that each ordered triple $\langle x, row(x), module(x)\rangle$ is a composed word with three fields and that the composed words are sorted using the combination of two fields, namely, $row(x)$ and $module(x)$. Clearly, $module(x)$ has $\log p$ bits, but it seems that, in order to represent $row(x)$, we need $\log \frac{N}{p}$ bits. Actually, we can replace $row(x)$ with the address offset contained in the offset matrix discussed above. Since the entries in that matrix are integers no larger than $p^{\frac{1}{2}}$, $0.5 \log p$ bits are sufficient. Therefore, the concatenation $row(x)|module(x)$ involves $1.5 \log p$ bits.

From the above discussion, it is clear that Step 2 requires $p^{\frac{1}{2}}$ calls to the sorting device and that the time spent on data movement operations not involving sorting is bounded by $O(p^{\frac{1}{2}})$.

Step 3: Same as Step 1.

Step 4: The permutation of Step 2 is performed in reverse; the permuted set of words are stored in rows $a + 1, a + 2, \ldots, a + p^{\frac{1}{2}}$.

Step 5: Same as Step 1.

Step 6: Shifting rows.

We shall permute the elements slightly differently from the way specified by Columnsort. However, it is easy to verify that the elements supposed to end up in a given row, indeed end up in the desired row. Since Step 7 sorts the rows, the order in which the elements are placed in the row in Step 6 is immaterial. The permutation of Step 6 is best illustrated by considering a particular example. Specifically, the permutation specified by Step 6 of Columnsort involving the three rows shown in (6) is:

$$\begin{array}{ccccccccc} M_1 & M_2 & M_3 & M_4 & M_5 & M_6 & M_7 & M_8 & M_9 \\ -\infty & -\infty & -\infty & -\infty & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 1' & 2' & 3' & 4' & 5' \\ 6' & 7' & 8' & 9' & 1'' & 2'' & 3'' & 4'' & 5'' \\ 6'' & 7'' & 8'' & 9'' & +\infty & +\infty & +\infty & +\infty & +\infty \end{array}$$

Our permutation is a bit different:

$$\begin{array}{ccccccccc} M_1 & M_2 & M_3 & M_4 & M_5 & M_6 & M_7 & M_8 & M_9 \\ 1 & 2 & 3 & 4 & 5 & ? & ? & ? & ? \\ 1' & 2' & 3' & 4' & 5' & 6 & 7 & 8 & 9 \\ 1'' & 2'' & 3'' & 4'' & 5'' & 6' & 7' & 8' & 9' \\ 1'' & 2'' & 3'' & 4'' & 5'' & 6'' & 7'' & 8'' & 9'' \end{array}$$

Assume that the $p^{\frac{1}{2}}$ consecutive input rows are stored in memory starting from memory row $a + 1$. In addition, we assume that memory row a is available to us. Some of its contents are immaterial and will be denoted by "?"s. The motivation is anchored in the observation that, in Step 7, we do not have to sort memory rows a and $a + p^{\frac{1}{2}}$: The elements in these rows will be sorted in Step 9. Consequently, the only rows that have to be sorted in Step 7 are rows $a + 1$ through $a + p^{\frac{1}{2}} - 1$. The details follow.

procedure *ROW_SHIFT*
begin
  for $i = a + 1$ **to** $a + p^{\frac{1}{2}} + 1$ **do**

for all j, $1 \leq j \leq \lceil \frac{p}{2} \rceil$, **do in parallel**
read the ith word of memory module M_j and
store it in the $(i-1)$th word of memory module M_j
endfor
endfor
end

It is important to note that, in our implementation, Step 6 does not involve sorting. However, $O(p^{\frac{1}{2}})$ time is spent on data movement operations that do not involve sorting.

Step 7: Same as Step 1.

Step 8: This is simply the reverse of the data movement in Step 6.

Step 9: Same as Step 1.

To summarize, we have proven the following result.

Theorem 2. *A set of $p^{\frac{3}{2}}$ elements stored in $p^{\frac{1}{2}}$ memory rows can be sorted, in row-major order, without memory-access conflicts, in at most $7p^{\frac{1}{2}}$ calls to a sorting device of I/O size p and in $O(p^{\frac{1}{2}})$ time for data movement not involving sorting.*

In essentially the same way, one can prove the following companion result to Theorem 2.

Theorem 3. *The task of sorting, in row-major order, a set of mp elements stored in m, $(1 \leq m \leq p^{\frac{1}{2}})$ memory rows can be performed, without memory-access conflicts, in at most $7m$ calls to a sorting device of I/O size p and in $O(m)$ time for data movement operations not involving sorting.*

In the remainder of this section, we present an important application of the basic algorithm. Suppose that we wish to merge two sorted sequences $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$. Our algorithm for merging A and B relies on the following technical result.

Lemma 8. *Assume that $a_{\lceil \frac{n}{2} \rceil} \leq b_{\lfloor \frac{n}{2} \rfloor + 1}$ and let $C = c_1 \leq c_2 \leq \cdots \leq c_n$ be the sequence obtained by merging $b_1, b_2, \ldots, b_{\lceil \frac{n}{2} \rceil}$ and $a_{\lceil \frac{n}{2} \rceil + 1}, a_{\lceil \frac{n}{2} \rceil + 2}, \ldots, a_n$. Then, no element in the sequence $D = a_1, a_2, \ldots, a_{\lceil \frac{n}{2} \rceil}, c_1, c_2, \ldots, c_{\lfloor \frac{n}{2} \rfloor}$ is strictly larger than any element in the sequence*

$$E = b_{\lceil \frac{n}{2} \rceil + 1}, b_{\lceil \frac{n}{2} \rceil + 2}, \ldots, b_n, c_{\lfloor \frac{n}{2} \rfloor + 1}, c_{\lfloor \frac{n}{2} \rfloor + 2}, \ldots, c_n.$$

Proof. We begin by showing that no a_i, $(1 \leq i \leq \lceil \frac{n}{2} \rceil)$, is strictly larger than any element in E. The assumption that $a_{\lceil \frac{n}{2} \rceil} \leq b_{\lfloor \frac{n}{2} \rfloor + 1}$ guarantees that if the claim is false, then some element a_i, $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$, is strictly larger than some element c_k, $(k \geq \lfloor \frac{n}{2} \rfloor + 1)$, in E.

To evaluate the position of the element c_k in the sorted sequence C, observe that all the $\lfloor \frac{n}{2} \rfloor$ elements in C that come from A are known to be larger than or equal to a_i and, therefore, strictly larger than c_k. Consequently, if n is even, then $\lfloor \frac{n}{2} \rfloor$ elements in C are larger than c_k, implying that $k \leq \lfloor \frac{n}{2} \rfloor$, a contradiction. On the other hand, if n is odd, then $\lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor + 1$ and, by assumption, $b_{\lceil \frac{n}{2} \rceil}$ is larger than a_i and, therefore, strictly larger than c_k. In this case, at least $\lceil \frac{n}{2} \rceil$ elements in C are strictly larger than c_k. It follows that $k \leq \lfloor \frac{n}{2} \rfloor$, contradicting that c_k belongs to E.

Next, we claim that no c_i, $(1 \leq i \leq \lfloor \frac{n}{2} \rfloor)$, is larger than any element in E. Since C is sorted, if the statement is false, then $c_i > b_k$ for some k, $(k \geq \lfloor \frac{n}{2} \rfloor + 1)$. Notice that all elements of C that come from B are smaller than or equal to b_k and, therefore, strictly smaller than c_i. It follows that $i \geq \lfloor \frac{n}{2} \rfloor + 1$, contradicting that c_i belongs to D. This completes the proof of the lemma. □

A mirror argument proves the following companion result to Lemma 8.

Lemma 9. *Assume that $a_{\lceil \frac{n}{2} \rceil} > b_{\lfloor \frac{n}{2} \rfloor + 1}$ and let $C = c_1 \leq c_2 \leq \cdots \leq c_n$ be the sequence obtained by merging $a_1, a_2, \ldots, a_{\lceil \frac{n}{2} \rceil}$ and $b_{\lceil \frac{n}{2} \rceil + 1}, b_{\lceil \frac{n}{2} \rceil + 2}, \ldots, b_n$. Then, no element in the sequence $D' = b_1, b_2, \ldots, b_{\lceil \frac{n}{2} \rceil}, c_1, c_2, \ldots, c_{\lfloor \frac{n}{2} \rfloor}$ is strictly larger than any element in the sequence*

$$E' = a_{\lceil \frac{n}{2} \rceil + 1}, a_{\lceil \frac{n}{2} \rceil + 2}, \ldots, a_n, c_{\lfloor \frac{n}{2} \rfloor + 1}, c_{\lfloor \frac{n}{2} \rfloor + 2}, \ldots, c_n.$$

It is worth noting that Lemmas 8 and 9, combined, show that given two sorted sequences, each of size n, the task of merging them can be handled as follows: We begin by splitting the two sequences into two sequences of size n each such that no element in the first one is strictly larger than any element in the second one. Once this "separation" is available, all that remains to be done is to sort the two sequences independently. The noteworthy feature of this approach is that it fits extremely well our architecture.

Let $1 \leq m \leq p^{\frac{1}{2}}$ and consider a sorted sequence $A = a_1 \leq a_2 \leq \cdots \leq a_{mp}$ stored in m memory rows $r_A, r_A + 1, \ldots, r_A + m - 1$ and a sorted sequence $B = b_1 \leq b_2 \leq \cdots \leq b_{mp}$ stored in m memory rows $r_B, r_B + 1, \ldots, r_B + m - 1$. The goal is to merge these two sequences and to store the resulting sorted sequence in memory rows

$$r_A, r_A + 1, \ldots, r_A + m - 1, r_B, r_B + 1, \ldots, r_B + m - 1.$$

The details follow.

procedure *MERGE_TWO_GROUPS*
begin
if $a_{\lceil \frac{mp}{2} \rceil} \leq b_{\lfloor \frac{mp}{2} \rfloor + 1}$ **then**
use the basic algorithm to sort
$b_1, b_2, \ldots, b_{\lceil \frac{mp}{2} \rceil}, a_{\lceil \frac{mp}{2} \rceil + 1}, a_{\lceil \frac{mp}{2} \rceil + 2}, \ldots, a_{mp}$ in nonincreasing order as $c_{mp} \geq c_{mp-1} \geq \cdots \geq c_1$ and store the result in memory rows $r_C, r_C + 1, \ldots, r_C + m - 1$
else
use the basic algorithm to sort
$a_1, a_2, \ldots, a_{\lceil \frac{mp}{2} \rceil}, b_{\lceil \frac{mp}{2} \rceil + 1}, b_{\lceil \frac{mp}{2} \rceil + 2}, \ldots, b_{mp}$ in nonincreasing order as $c_{mp} \geq c_{mp-1} \geq \cdots \geq c_1$ and store the result in memory rows $r_C, r_C + 1, \ldots, r_C + m - 1$
for $i = 1$ **to** $\lfloor \frac{m}{2} \rfloor$ **do**
copy memory row $r_B + i - 1$ into memory row $r_A + i - 1$;
copy memory row $r_A + m - i$ into memory row $r_B + m - i$
endfor
if m is odd **then**
copy the leftmost $\lceil \frac{p}{2} \rceil$ elements in row $r_B + \lceil \frac{m}{2} \rceil$ into the leftmost $\lceil \frac{p}{2} \rceil$ positions of row $r_A + \lceil \frac{m}{2} \rceil$;
copy the rightmost $\lfloor \frac{p}{2} \rfloor$ elements in row $r_A + \lfloor \frac{m}{2} \rfloor$ into the rightmost $\lfloor \frac{p}{2} \rfloor$ positions of row $r_B + \lfloor \frac{m}{2} \rfloor$;
endif
endif

if m is odd **then**
  copy the leftmost $\lceil \frac{p}{2} \rceil$ elements in memory row $r_C + \lfloor \frac{m}{2} \rfloor$ into the leftmost $\lceil \frac{p}{2} \rceil$ positions of row $r_B + \lfloor \frac{m}{2} \rfloor$;
  copy the rightmost $\lfloor \frac{p}{2} \rfloor$ elements in row $r_C + \lfloor \frac{m}{2} \rfloor$ into the rightmost $\lfloor \frac{p}{2} \rfloor$ positions of row $r_A + \lfloor \frac{m}{2} \rfloor$
endif
for $i = 1$ **to** $\lfloor \frac{m}{2} \rfloor$ **do**
  copy memory row $r_C + i - 1$ into memory row $r_B + i - 1$;
  copy memory row $r_C + m - i$ into memory row $r_A + m - i$
endfor
use the basic algorithm to sort memory rows $r_A, r_A + 1, \ldots, r_A + m - 1$ in nondecreasing order;
use the basic algorithm to sort memory rows $r_B, r_B + 1, \ldots, r_B + m - 1$ in nondecreasing order
end

It is obvious that procedure *MERGE_TWO_GROUPS* can be implemented directly in our architectural model. One point is worth discussing, however. Specifically, the task of sorting a sequence in nonincreasing order can be performed in our architecture as follows: The signs of all the elements to be sorted are flipped and the resulting sequence is then sorted in nondecreasing order. Finally, the signs are flipped back to their original value. The correctness of the procedure follows from Lemmas 8 and 9. Moreover, the procedure requires three calls to the basic algorithm.

Consider the task of sorting a collection of $2mp$ memory rows, with m as above. Having partitioned the input into two subgroups of m consecutive memory rows each, we use the basic algorithm to sort each group. Once this is done, we complete the sorting using procedure *MERGE_TWO_GROUPS*. Thus, we have the following result.

Theorem 4. *The task of sorting $2mp$, $1 \leq m \leq p^{\frac{1}{2}}$, elements stored in $2m$ memory rows can be performed in five calls to the basic algorithm and $O(m)$ time for data movement operations not involving sorting.*

5 An Efficient Multiway Merge Algorithm

Consider a collection $A =< A_1, A_2, \ldots, A_m >$ of m, $(2 \leq m \leq p^{\frac{1}{2}})$, sorted sequences, each of size $p^{\frac{i}{2}}$, for some $i \geq 3$. We assume that A is stored, top-down, in the order $A_1, A_2, \ldots, A_m$ in $mp^{\frac{i-2}{2}}$ consecutive memory rows. The *multiway merge problem* is to sort these sequences in row-major order. The goal of this section is to propose an efficient algorithm *MULTIWAY_MERGE* for the multiway merge problem, and to show how it can be implemented on our architecture.

procedure *MULTIWAY_MERGE*(*A, m, i*);
{Input: m, $(2 \leq m \leq p^{\frac{1}{2}})$, sorted sequences
$A =< A_1, A_2, \ldots, A_m >$ each of size $p^{\frac{i}{2}}$, for $i \geq 3$;
Output: the resulting sorted sequence stored in row-major order in $mp^{\frac{i-2}{2}}$ contiguous memory rows.}

Step 1. Select a sample S of size $mp^{\frac{i-2}{2}}$ from A by retaining every pth element in each sequence A_j $(1 \leq j \leq m)$ and move S to its own $\lceil mp^{\frac{i-4}{2}} \rceil$ memory rows[2] as discussed below;
Step 2.
  if $i = 3$ **then**
    sort S by one call to the sorting device
  else if $i = 4$ **then**
    sort S by one call to the basic algorithm
  else
    $MULTIWAY\_MERGE(S, m, i-2)$; {recursively multiway merge S}
  endif
  let $s_1 \leq s_2 \leq \cdots \leq s_{mp^{(i-2)/2}}$ be the sorted version of S;
Step 3. Partition A into $p^{\frac{i-2}{2}}$ buckets $B_1, B_2, \ldots, B_{p^{(i-2)/2}}$, each containing at most $2mp$ elements, as discussed below, and move the elements of A to their buckets without memory access conflicts;
Step 4. Sort all the buckets individually using the basic algorithm and procedure *MERGE_TWO_GROUPS*;
Step 5. Coalesce the sorted buckets into the desired sorted sequence.

The remainder of this section is devoted to a detailed implementation of this procedure on our architecture.

5.1 Implementing Step 1 and Step 2

For convenience, we view A as a matrix of size $mp^{\frac{i-2}{2}} \times p$, with the tth element of memory row j being denoted by $A[j, t]$. The element $A[j, p]$ is termed the *leader* of memory row j.

The goal of Step 1 is to extract a sample S of A by retaining the leader s of every memory row in A, along with the identity k of the subsequence A_k to which the leader belongs. In this context, k is referred to as the *sequence index* of s. Two disjoint groups of $\lceil mp^{\frac{i-4}{2}} \rceil$ consecutive memory rows each are set aside to store the sample S and the corresponding set I of sequence indices. In the remainder of this subsection, we view the memory rows allocated to S and I as two matrices of size $\lceil mp^{\frac{i-4}{2}} \rceil \times p$. The intention is that, at the end of Step 1, $S[x, y]$ and $I[x, y]$ store the $((x-1)p + y)$th leader of A and its sequence index, respectively.

To see how Step 1 can be implemented without memory access conflicts, notice that, in each memory row, the leader to be extracted is stored in memory module M_p. For a generic memory row j, the CU interchanges temporarily the elements $A[j, p]$ and $A[j, d(j)]$, where $d(j) = 1 + (j-1) \bmod p$. (This interchange will be undone at the end of Step 1). Next, $\lceil mp^{\frac{i-4}{2}} \rceil$ parallel read operations are performed, each followed by two parallel write operations. The jth parallel read operation picks up the $((j-1)p + k)$th word of memory module M_k, $(1 \leq k \leq p)$, and these p elements are stored in the jth memory row allocated to S. The second parallel write operation stores the sequence indices of these p elements in the jth memory row allocated to I. Thus, Step 1 can be implemented in $O(mp^{\frac{i-2}{2}})$ time for data movement and no calls to the sorting device.

2. Notice that if $i = 3$, the sample S will be stored in one memory row.

The sampling process continues, recursively, until a level is reached where procedure *MULTIWAY_MERGE* is invoked with either $i = 3$, in which case the corresponding sample set is stored in one memory row and will be sorted in one call to the sorting device, or with $i = 4$, in which case the sample set is stored in m memory rows, and will be sorted in one call to the basic algorithm. Since the operation of sorting one row is direct, we only discuss the way the basic algorithm operates in this context.

Conceptually, the process of sorting the samples benefits from being viewed as one of sorting the concatenation $s|k$, where s is a sample element and k its sequence index. Recall that, as described in Section 2, our design assumes that the sorting device provides data paths of size $w + 1.5 \log p$ from its inputs to its outputs. This implies that Steps 1, 3, 5, 7, and 9 of the extended Columnsort can be executed directly. To sort a row r of S and the corresponding row r of I, the CU loads, in two parallel read operations, the element field and the short auxiliary field of data register R_j $(1 \leq j \leq p)$ with $S[r, j]$ and $I[r, j]$ and the long auxiliary field with 0.

Let $s_{r,j}$ and $k_{r,j}$ be the element and its sequence index stored in register R_j and let $s_{r,j}|k_{r,j}$ denote their concatenation. Next, the contents of the data registers are supplied as input to the sorting device. Let $s_{r,j'}|k_{r,j'}$ be received by R_j after sorting, with $s_{r,j'}$ and $k_{r,j'}$ stored, respectively, in the element and short auxiliary field of R_j. In two parallel write operations, the CU stores the element field and the short auxiliary field of each register R_j $(1 \leq j \leq p)$ into $S[r, j]$ and $I[r, j]$, respectively.

Steps 2, 4, 6, and 8 of the basic algorithm perform permutations. The implementation of Steps 6 and 8 does not involve sorting. In this case, the data movement involving the sample elements and that of the corresponding sequence indices will be performed in two companion phases. Specifically, viewing the sample set S and its corresponding sequence index set I as two matrices, the same permutation is performed on S and I. Steps 2 and 4 of the basic algorithm involve both data movement operations and sorting. The data movement operations in these steps are similar to those in Steps 6 and 8 and will not be detailed any further. Recall that the sorting operations in Steps 2 and 4 of the basic algorithm are performed on the concatenation of the two auxiliary fields storing the relative row number and the column number of the element. Hence, we perform two companion sorting phases, one for permuting the sample elements and the other for permuting sequence indices. Clearly, this can be implemented with the same time complexity.

It is easy to confirm that, at the end of Step 2 of procedure *MULTIWAY_MERGE*, the sample set S is sorted in row-major order. Furthermore, viewed as matrices, $I[x, y]$ is the sequence index of the sample element $S[x, y]$. Let the sorted version of S be

$$S = s_1 \leq s_2 \leq \cdots \leq s_{mp^{(i-2)/2}}. \tag{8}$$

Equation (8) will be used in Step 3 to partition the elements of A into buckets. In order to do so, the leader of each row in A needs to learn its rank in (8).

Our next goal is to associate with every memory row in A the rank $(x - 1)p + y$ of its leader $s = S[x, y]$ in S. This task will be carried out in two stages. In the first stage, using the sequence index and the rank of s in S, the CU assigns to s a row number $row(s)$ in A. For every s in S, $row(s)$ is either the *exact* row number from which s was extracted in Step 1 or, in case the leaders of several rows are equal, $row(s)$ achieves a possible reassignment of leaders to rows. The details of the first stage are spelled out in procedure *ASSIGN_ROW_NUMBERS* presented below. For convenience, we use the matrix representation of S and I. These operations can be easily implemented using the addresses of words corresponding to $S[x, y]$ and $I[x, y]$. Initially, I contains the sequence indices of samples in S. When the procedure terminates, $I[x, y]$ contains $row(s)$ corresponding to $s = S[x, y]$.

procedure *ASSIGN_ROW_NUMBERS*
begin
  for $k = 1$ **to** m **do**
    $r_k :=$ the row number of the first memory row storing the sequence A_k
  endfor
  for $x = 1$ **to** $\lceil mp^{\frac{i-4}{2}} \rceil$ **do**
    for $y = 1$ **to** p **do**
      $k := I[x, y]$; $I[x, y] := r_k$; $r_k := r_k + 1$
    endfor
  endfor
end

In the second stage, the CU assigns the rank $(x - 1)p + y$ of $s = S[x, y]$ with the memory row $row(s)$ contained in $I[x, y]$. The operations performed on the matrix representations of S and I can be easily implemented using the addresses of words corresponding to $S[x, y]$ and $I[x, y]$. Since only read/write operations are used in the procedure described, the total time spent on these operations is bounded by $O(mp^{\frac{i-2}{2}})$.

5.2 Implementing Step 3 and Step 4

Once the rank of each leader in A is known, we are ready to partition A into buckets. Our first objective is to construct a collection $B_1, B_2, \ldots, B_{p^{(i-2)/2}}$ of buckets such that the following conditions are satisfied:

b1. Every element of A belongs to exactly one bucket;
b2. No bucket contains more than $2mp$ elements;
b3. For every i and j $(1 \leq i < j \leq p^{\frac{i-2}{2}})$, no element in B_i is strictly larger than any element in B_j.

Before presenting our bucket partitioning scheme, we need a few definitions. Let $S = s_1 \leq s_2 \leq \cdots \leq s_{mp^{(i-2)/2}}$ be as in (8). The memory row with leader s_b is said to be *regular* with respect to bucket B_j $(1 \leq j \leq p^{\frac{i-2}{2}})$ if

$$(j - 1)m < b \leq jm. \tag{9}$$

Notice that (9) guarantees that every memory row in A is regular with respect to exactly one bucket and that the identity of this bucket can be determined by the CU in constant time. Conversely, with respect to each bucket, there are *exactly* m regular memory rows.

A memory row r with leader s_b in some sequence A_k $(1 \leq k \leq m)$ is termed *special* with respect to bucket B_t if,

with s_a standing for the leader of the preceding memory row in A_k, if any, we have

$$a \leq tm < b. \tag{10}$$

Let the memory rows with leaders s_a and s_b be regular with respect to buckets $B_{j'}$ and B_j, respectively, such that $j' < j$. It is very important to note that (10) implies that the memory row whose leader is s_b is special with respect to *all* the buckets $B_{j'}, B_{j'+1}, \ldots, B_{j-1}$.

Conceptually, our bucket partitioning scheme consists of two stages. In the first stage, by associating all regular and special rows with respect to a generic bucket B_j ($1 \leq j \leq p^{\frac{i-2}{2}}$), we obtain a set C_j of *candidate elements* for bucket B_j. In the second stage, we assign the elements of A to buckets in such a way that the actual elements assigned to bucket B_j form a subset of the candidate set C_j.

Specifically, an element x of a memory row regular with respect to bucket B_j is *assigned* to B_j if one of the conditions below is satisfied:

$$s_{(j-1)m} < x \leq s_{jm} \text{ whenever } s_{(j-1)m} < s_{jm} \tag{11}$$

or

$$s_{(j-1)m} \leq x \leq s_{jm} \text{ whenever } s_{(j-1)m} = s_{jm}. \tag{12}$$

The elements of A that have been assigned to a bucket by virtue of (11) or (12) are no longer eligible for being assigned to buckets in the remainder of the assignment process.

Consider, further, an element x that was not assigned to the bucket with respect to which its memory row is regular. Element x will be assigned to exactly one of the buckets with respect to which the memory row containing x is special. Assume that the memory row containing x is special with respect to buckets $B_{j_1}, B_{j_2}, \ldots, B_{j_{l(x)}}$ with $j_1 < j_2 < \ldots < j_{l(x)}$. Let j_n be the smallest index, $1 \leq n \leq l(x)$, for which one of (11) or (12) holds, with j_n in place of j. Now, x is assigned to bucket B_{j_n}. The next result shows that the buckets we just defined satisfy the conditions b1-b3.

Lemma 10. *Every bucket B_j ($1 \leq j \leq p^{\frac{i-2}{2}}$) satisfies conditions b1, b2, and b3.*

Proof. Clearly, our assignment scheme guarantees that every element of A gets assigned to some bucket and that no element of A gets assigned to more than one bucket. Thus, condition b1 is verified.

Further, notice that, by (9) and (10), combined, for every j ($1 \leq j \leq p^{\frac{i-2}{2}}$) the candidate set C_j with respect to bucket B_j contains at most $2m$ memory rows and, therefore, at most $2mp$ elements of A. Moreover, as indicated, the elements actually assigned to bucket B_j are a subset of C_j, proving that b2 is satisfied.

Finally, (11) and (12) guarantee that if an element x belongs to some bucket b_j, then it cannot be strictly larger than any element in a bucket B_k with $j < k$. Thus, condition b3 holds as well. □

It is worth noting that the preceding definition of buckets works perfectly well even if all the input elements are identical. In fact, if all elements are distinct, one can define buckets in a simpler way. Moreover, in the case of distinct elements, Steps 1-3 of procedure *MULTIWAY_MERGE* can be further simplified.

We now present the implementation details of the assignment of elements to buckets. Write $s_0 = -\infty$ and denote, for every j ($1 \leq j \leq p^{\frac{i-2}{2}}$), the ordered pair $(s_{(j-1)m}, s_{jm})$ as the jth bounding pair. Notice that (11) and (12) amount to testing whether a given element lies between a bounding pair.

By b2, no bucket contains more than $2mp$ elements from A. This motivates us to set aside $2m$ memory rows for each bucket B_j. Out of these, we allocate the first m memory rows to elements assigned to B_j coming from regular memory rows with respect to B_j; we allocate the last m memory rows to elements assigned to bucket B_j that reside in special memory rows with respect to B_j. In addition, we find it convenient to initialize the contents of the $2m$ memory rows allocated to B_j to all $+\infty$s.

It is important to note that the regular memory rows with respect to a bucket B_j are naturally ordered from 1 to m by the order of the corresponding leaders in S. To clarify this last point, recall that, by (9), the m leaders belonging to bucket B_j are

$$s_{(j-1)m+1},\ s_{(j-1)m+2},\ \ldots,\ s_{jm}.$$

Accordingly, the memory row whose leader is $s_{(j-1)m+1}$ is the first regular row with respect to B_j, the memory row whose leader is $s_{(j-1)m+2}$ is the second regular row with respect to B_j, and so on. Similarly, the fact that each sequence A_k is sorted guarantees that it may contain *at most* one special memory row with respect to bucket B_j. Now, in case such a special row exists, it will be termed the kth special memory row with respect to B_j to distinguish it from the others.

In order to move the elements to their buckets, the CU scans the memory rows in A one by one. Suppose that the current memory row being scanned is row r in some sequence A_k. We assume that the leader of row r is s_b and that the leader of row $r-1$ is s_a. Using (9), the CU establishes that row r is regular with respect to bucket B_j, where $j = \lceil \frac{b}{m} \rceil$ and, similarly, that the previous memory row is regular with respect to bucket $B_{j'}$, where $j' = \lceil \frac{a}{m} \rceil \leq j$. In case row r is the first row of A_k, j' is set to 1.

Next, the elements in memory row r are read into the element fields of the data registers; the CU broadcasts to these registers the bounding pair $(s_{(j-1)m}, s_{jm})$. Using compare-and-set, each register stores in the short auxiliary field a 1 if the corresponding element is assigned to bucket B_j by virtue of (11) or (12) and a 0 otherwise. We say that an element x in some data register is *marked* if the value in the short auxiliary field is 1; otherwise, x is *unmarked*.

Clearly, every element x that is marked at the end of this first broadcast has been assigned to bucket B_j. In a parallel write operation, the CU copies all the marked elements to the corresponding words of the $((b-1) \bmod m + 1)$th memory row allocated to bucket B_j. Once this is done, using compare-and-set, all the marked elements in the data registers are set to $+\infty$ and the short auxiliary fields are cleared.

Further, the CU broadcasts to the data registers, in *increasing order*, the bounding pairs of the buckets $B_{j'}, B_{j'+1}, \ldots, B_{j-1}$. Let us follow the processing specific to bucket $B_{j'}$. Having received the bounding pair $(s_{(j'-1)m}, s_{j'm})$, each data register determines whether the value x stored in its element field satisfies (11) or (12) with j' in place of j and marks x accordingly. In a parallel write operation, the CU copies all the marked elements to the corresponding words of the next available memory row allocated to bucket B_j. Next, using compare-and-set, all the marked elements in the data registers are set to $+\infty$ and the short auxiliary fields are cleared. The same process is then repeated for all the remaining buckets with respect to which row r is special.

The reader will not fail to note that, when the processing of row r is complete, each of its elements has been moved to the bucket to which it has been assigned. Moreover, by (9) and (10), there are, altogether, at most $mp^{\frac{i-2}{2}}$ regular rows and at most $mp^{\frac{i-2}{2}}$ special rows and, so, the total time involved in assigning the elements of A to buckets is bounded by $O(mp^{\frac{i-2}{2}})$ and no calls to the sorting device. In summary, Step 3 can be implemented in $O(mp^{\frac{i-2}{2}})$ time for data movement and no calls to the sorting device.

In Step 4, the buckets are sorted independently. If a bucket has no more than $p^{\frac{1}{2}}$ memory rows, it can be sorted in one call to the basic algorithm. Otherwise, the bucket is partitioned in two halves, each sorted in one call to the basic algorithm. Finally, the two sorted halves are merged using procedure *MERGE_TWO_GROUPS*. By Theorem 4, the task of sorting all the buckets individually can be performed in $O(mp^{\frac{i-2}{2}})$ calls to the sorting device and in $O(mp^{\frac{i-2}{2}})$ time for data movement not involving sorting.

5.3 Implementing Step 5

To motivate the need for the processing specific to Step 5, we note that, after sorting each bucket individually in Step 4, there may be a number of $+\infty$s in each bucket. We refer to such elements as *empty*; memory rows consisting entirely of empty elements will be termed *empty rows*. A memory row is termed *impure* if it is partly empty. It is clear that each bucket may have at most one impure row. A memory row that contains no empty elements is referred to as *pure*.

The task of coalescing the nonempty elements in the buckets into $mp^{\frac{i-2}{2}}$ consecutive memory rows will be referred to as *compaction*. For easy discussion, we assume that all sorted buckets are stored in consecutive rows. That is, the nonempty rows of B_2 follow the nonempty rows of B_1, the nonempty rows of B_3 follow the nonempty rows of B_2, and so on, assuming that all empty rows have been removed. The compaction process consists of three phases.

Phase 1: Let C be the row sequence obtained by concatenating nonempty rows of B_js obtained in Step 4 of *MULTIWAY_MERGE* in the increasing order of their indices. We partition sequence C into subsequences $C_1, C_2, \ldots, C_x$ such that each C_j contains $p^{\frac{1}{2}}$ consecutive rows of C, except the last subsequence C_x, which may contain fewer rows. Clearly, $x \leq 2mp^{\frac{i-3}{2}}$. We use the basic algorithm to sort these subsequences independently. Let the sorted subsequence corresponding to C_i be C'_i with empty rows eliminated for future consideration. Let D be the row sequence obtained by concatenating rows of C'_js in the increasing order of their indices. We partition sequence D into subsequences $D_1, D_2, \ldots, D_y$ such that each D_j contains $p^{\frac{1}{2}}$ consecutive rows of D, except the last subsequence D_y, which may contain fewer rows. We then use the basic algorithm to sort these subsequences independently. Let the sorted subsequence corresponding to D_i be D'_i with empty rows eliminated. Let E be the row sequence obtained by concatenating rows of D'_js in the increasing order of their indices.

Lemma 11. *The preceding row of every impure row, except the last row, of E is a pure row.*

Proof. We notice the following fact: Except for the last row of D, every row of D either contains at least $p^{\frac{1}{2}}$ nonempty elements or if it contains fewer than $p^{\frac{1}{2}}$ nonempty elements, then its preceding row must be a pure row. This is because each row of C contains at least one nonempty element. An impure row of D can be generated under one of two conditions: 1) if C_j contains fewer than p nonempty elements, then C'_j contains only one row, an impure row, with its nonempty elements coming from $p^{\frac{1}{2}}$ impure rows of C_j, and 2) if a C_j contains more than p nonempty elements, then C'_j contains only one impure row and its preceding row is a pure row. The lemma directly follows from this fact. □

Phase 2: This phase computes a set of parameters which will be used in the next phase. Let w be the total number of (nonempty) rows in E. Assume that the rows of E are located from row 1 through row w. For every j $(1 \leq j \leq p^{\frac{i-2}{2}})$, we let n_j stand for the number of nonempty elements in the impure memory row c_j. The first subtask of Phase 2 is to determine $n_1, n_2, \ldots, n_{p^{\frac{i-2}{2}}}$. Consider a generic impure row c_j. To determine n_j, the CU reads the entire row c_j into the data registers $R_1, R_2, \ldots, R_p$ such that, for every k., $(1 \leq k \leq p)$, the c_jth word of memory module M_k is read into register R_k. The long auxiliary field of data register R_k is set to k. By using the compare-and-set feature, the CU instructs each register R_k to reset this auxiliary field to $-\infty$ if the element it holds is $+\infty$ (i.e., empty). Next, the data registers are loaded into the sorting device and sorted in increasing order of their long auxiliary fields. It is easy to confirm that, after sorting, the largest such value k_j is precisely the position of the rightmost nonempty element in memory row c_j. Therefore, the CU sets $n_j = p + 1 - k_j$. Consequently, the task of computing all the numbers $n_1, n_2, \ldots, n_{p^{\frac{i-2}{2}}}$ involves $O(p^{\frac{i-2}{2}})$ calls to the sorting device and $O(p^{\frac{i-2}{2}})$ read/write operations and does not involve sorting. Once the numbers $n_1, n_2, \ldots, n_w$ are available, the CU computes the prefix sums

$$\sigma_1 = n_1, \sigma_2 = n_1 + n_2, \ldots, \sigma_j = n_1 + n_2 + \ldots, +n_j, \ldots,$$
$$\sigma_w = n_1 + n_2 + \ldots, +n_w.$$

This, of course, involves only additions and can be performed by the CU in $O(mp^{\frac{i-2}{2}})$ time without any call to

the sorting device. Let $g = \lceil \frac{w}{p^{\frac{1}{2}}-2} \rceil$. Define $\alpha_0 = \alpha_g = 0$ and $\alpha_k = \sigma_{k(p^{\frac{1}{2}}-2)} \bmod p$, for $1 \leq k < g$. Define

$$\beta_k = n_{k(p^{\frac{1}{2}}-2)} + n_{k(p^{\frac{1}{2}}-2)-2}.$$

Phase 3: Construct row group E_k, $1 \leq k \leq g$, of consecutive rows as follows: If $\alpha_{k-2} > 0$, then row $k(p^{\frac{1}{2}}-2)-2$ is the starting row of E_k, else row $k(p^{\frac{1}{2}}-2)$ is the starting row of E_k; the ending row of E_k, $k < g$, is row $k(p^{\frac{1}{2}}-2)$ and the ending row of E_g is row w. Note that E_k and E_{k+1} may share at most two rows. By Lemma 11, for $1 \leq k < g$, each E_k contains at least $\frac{(p+1)p^{\frac{1}{2}}}{2}$ elements and the last two rows of E_k contains at least $p+1$ elements. For each E_k, $1 \leq k < g$, perform the following operations:

1. Sort using the basic algorithm;
2. Replace the $\beta_{k-1} - \alpha_{k-1}$ smallest elements by $+\infty$s;
3. Sort using the basic algorithms; and
4. If $\alpha_k > 0$ and $k < g$, eliminate the last row.

For E_g, perform 1, 2, and 3 only. Let E'_k be the row group obtained from E_k and let F be the row sequence obtained by concatenating rows of E'_js in the increasing order of their indices. F is the compaction of C.

Setting selected elements in a row to $+\infty$s can be done in $O(1)$ time by a compare-and-set operation. For example, setting the leftmost s elements of a row to $+\infty$s can be carried out as follows: Read the row into R_is, then CU broadcast s to all R_is and each R_i compare i with s and set its content to $+\infty$ if $i \leq s$; then, the modified row is written back to the memory array.

Based on Lemma 10, it is easy to verify that elements in F are in sorted order after Step 5, which can be implemented in $O(mp^{\frac{i-2}{2}})$ calls to the sorting device and $O(mp^{\frac{i-2}{2}})$ data movement not involving sorting.

5.4 Complexity Analysis

With the correctness of our multiway merge algorithm being obvious, we now turn to the complexity. Specifically, we are interested in assessing the total amount of data movement, not involving sorting, that is required by procedure *MULTIWAY_MERGE*. Specifically, let $J(mp^{\frac{i}{2}})$ stand for the time spent on data movement tasks that do not involve the use of the sorting device. If $i = 3$, Step 2 takes $O(1)$ time. In case $i = 4$, Step 2 takes $O(m)$ time (refer to Theorem 3). Finally, if $i > 4$, our previous discussion shows that each of Step 1, Step 3, Step 4, and Step 5 require at most $O(mp^{\frac{i-2}{2}})$ time, while Step 2 requires, recursively, $J(mp^{\frac{i-2}{2}})$ time. Thus, we obtain the following recurrence system:

$$\begin{cases} J(mp^{\frac{i}{2}}) \in O(mp^{\frac{i-2}{2}}) & \text{if } i = 3 \text{ or } 4 \\ J(mp^{\frac{i}{2}}) \leq J(mp^{\frac{i-2}{2}}) + O(mp^{\frac{i-2}{2}}) & \text{if } i > 4. \end{cases}$$

It is easy to confirm that, for $p \geq 4$, the solution of the above recurrence satisfies $J(mp^{\frac{i}{2}}) \in O(mp^{\frac{i-2}{2}})$. A similar analysis, that is not repeated, shows that the total number of calls to a sorting device of I/O size p performed by procedure *MULTIWAY_MERGE* for merging m $(2 \leq m \leq p^{\frac{1}{2}})$ sorted sequences, each of size $p^{\frac{i}{2}}$, is bounded by $O(mp^{\frac{i-2}{2}})$. To summarize our discussion, we state the following important result:

Theorem 5. *Procedure MULTIWAY_MERGE performs the task of merging m $(2 \leq m \leq p^{\frac{1}{2}})$ sorted sequences, each of size $p^{\frac{i}{2}}$, in our architecture, using $O(mp^{\frac{i-2}{2}})$ calls to the sorting device of I/O size p and $O(mp^{\frac{i-2}{2}})$ time for data movement not involving sorting.*

6 The Sorting Algorithm

With the basic algorithm and the multiway merge at our disposal, we are in a position to present the details of our sorting algorithm using a sorting device of fixed I/O size p. The input is a set Σ of N items stored, as evenly as possible, in p memory modules. Dummy elements of value $+\infty$ are added, if necessary, to ensure that all memory modules contain $\lceil \frac{N}{p} \rceil$ elements: These dummy elements will be removed after sorting. Our goal is to show that, using our architecture-algorithm combination, the input can be sorted in $O(\frac{N \log N}{p \log p})$ time and $O(N)$ data space. We assume that $p \geq 16$, which, along with (1), implies that

$$\log^2 p \leq p \leq \frac{N}{p}. \tag{13}$$

Equation (13) will be important in the analysis of this section, as our discussion will focus on the case where a sorting network of I/O size p and depth $O(\log^2 p)$ is used as the sorting device.[3] A natural candidate for such a network is Batcher's classic bitonic sorting network [1] that we shall tacitly assume.

Recall that, by virtue of (1), we have, for some t, $t \geq 4$,

$$p^{\frac{t}{2}} \leq N < p^{\frac{t+1}{2}}. \tag{14}$$

In turn, (14) guarantees that

$$t = \left\lfloor \frac{\log N}{\log p^{\frac{1}{2}}} \right\rfloor. \tag{15}$$

At this point, we note that (14) and (15), combined, guarantee that

$$\log^2 p \leq \frac{N}{p^{\frac{k}{2}}} \text{ for all } k \leq t-2. \tag{16}$$

Write

$$q = \left\lceil \frac{N}{p^{\frac{t}{2}}} \right\rceil \tag{17}$$

and observe that, by (14),

$$1 \leq q \leq p^{\frac{1}{2}}. \tag{18}$$

For reasons that will become clear later, we pad Σ with an appropriate number of $+\infty$ elements in such a way that,

3. As it turns out, the same complexity claim holds if the sorting device used is, instead, a p-sorter.

with N' standing for the length of the resulting sequence Σ', we have

$$N' = q * p^{\frac{t}{2}}. \tag{19}$$

It is important to note that (14), (17), and (19), combined, guarantee that

$$N \leq N' \leq 2N, \tag{20}$$

suggesting that the number of memory rows used by the sorting algorithm is bounded by $O(\frac{N}{p})$. Later, we will show that this is, indeed, the case.

In order to guarantee an overall running time of $O(\frac{N \log N}{p \log p})$, we ensure that each iteration can be performed in $O(\frac{N}{p})$ time. As we will see shortly, the sorting network will be used in the following three contexts:

1. to sort, individually, M memory rows;
2. to sort, individually, M groups, each consisting of m consecutive memory rows, where $m \leq p^{\frac{1}{2}}$;
3. to sort, individually, M groups, each consisting of $2m$ consecutive memory rows, where $m \leq p^{\frac{1}{2}}$.

For an efficient implementation of 1, we use *simple pipelining*: The M memory rows to sort are input to the sorting network, one after the other. After an initial overhead of $O(\log^2 p)$ time, each subsequent time unit produces a sorted memory row. Clearly, the total sorting time is bounded by $O(\log^2 p + M)$.

Our efficient implementation of 2 uses *interleaved pipelining*. Let $G_1, G_2, \cdots, G_M$ be the groups we wish to sort. In the interleaved pipelining, we begin by running Step 1 of the basic algorithm in pipelined fashion on group G_1, then on group G_2, and on so. In other words, Step 1 of the basic algorithm is performed on all groups using simple pipelining. Then, in a perfectly similar fashion, simple pipelining is used to carry out Step 2 of the basic algorithm on all the groups $G_1, G_2, \cdots, G_M$. The same strategy is used with all the remaining steps of the basic algorithm that require the use of the sorting device. Consequently, the total amount of time needed to sort all the groups using interleaves pipelining is bounded by $O(\log^2 p + Mm)$.

An efficient implementation of 3 relies on *extended interleaved pipelining*. Let $G_1, G_2, \cdots, G_M$ be the groups we want to sort. Recall that Theorem 4 states that sorting a group of $2m$ consecutive memory rows requires five calls to the basic algorithm. The extended interleaved pipelining consists of five interleaved pipelining steps, each corresponding to one of the five calls to the basic algorithm. Thus, the task of sorting all groups can be performed in $O(\log^2 p + Mm)$ time. We now discuss each of the iterations of our sorting algorithm in more detail.

Iteration 1. The input is Partitioned into $\frac{N'}{p}$ groups, each involving $p^{\frac{1}{2}}$ memory rows. By using interleaved pipelining with $m = p^{\frac{1}{2}}$, each such group is sorted individually. As discussed above, the running time of Iteration 1 is bounded by $O(\log^2 p + \frac{N'}{p}) = O(\frac{N}{p})$.

Iteration $k, 2 \leq k \leq t-2$. Let $i_k = k+1$. The input to Iteration k is a collection of $\frac{N'}{p^{\frac{i_k}{2}}}$ sorted sequences each of size $p^{\frac{i_k}{2}}$, stored in $p^{\frac{i_k-2}{2}}$ consecutive memory rows. The output of iteration k is a collection of $\frac{N'}{p^{\frac{i_k+1}{2}}}$ sorted sequences, each of size $p^{\frac{i_k+1}{2}}$, stored in $p^{\frac{i_k-1}{2}}$ consecutive memory rows.

Having partitioned these sorted sequences into $\frac{N'}{p^{\frac{i_k+1}{2}}}$ groups $G(k,1), G(k,2), \ldots, G(k, \frac{N}{p^{\frac{i_k+1}{2}}})$ of $p^{\frac{1}{2}}$ consecutive sequences each, we proceed to sort each group $G(k,j)$ by the call

$$MULTIWAY\_MERGE(S_1(k,j), p^{\frac{1}{2}}, i_k),$$

where $S_1(k,j) = G(k,j)$. We refer to the call

$$MULTIWAY\_MERGE(S_1(k,j), p^{\frac{1}{2}}, i_k)$$

as a *MULTIWAY_MERGE* call of the first level. Observe that, since there are $\frac{N}{p^{\frac{i_k+1}{2}}}$ groups, there will be, altogether, $\frac{N}{p^{\frac{i_k+1}{2}}}$ *MULTIWAY_MERGE* calls of the first, one for each group. In Step 1 of a *MULTIWAY_MERGE* call of the first level, we extract a sample $S_2(k,j)$ of $S_1(k,j)$ consisting of $p^{\frac{1}{2}}$ sorted sequences, each of size $p^{\frac{i_k-2}{2}}$, stored in $p^{\frac{i_k-4}{2}}$ consecutive memory rows. In turn, for every j $(1 \leq j \leq \frac{N}{p^{\frac{i_k+1}{2}}})$, the sample $S_2(k,j)$ is sorted by invoking $MULTIWAY\_MERGE(S_2(k,j), p^{\frac{1}{2}}, i_k - 2)$, which is referred to as a *MULTIWAY_MERGE* call of the second level. Step 1 of a *MULTIWAY_MERGE* call of the second level extracts a sample $S_3(k,j)$ of $S_2(k,j)$ and so on.

For every u, $1 \leq u \leq \lfloor \frac{i_k-1}{2} \rfloor$, a *MULTIWAY_MERGE* call of level u is of the form

$$MULTIWAY\_MERGE(S_u(k,j), p^{\frac{1}{2}}, i_k - 2(u-1)).$$

In Step 2 of the call

$$MULTIWAY\_MERGE(S_u(k,j), p^{\frac{1}{2}}, i_k - 2(u-1)),$$

we perform a *MULTIWAY_MERGE* call of level $u+1$, which is of the form

$$MULTIWAY\_MERGE(S_{u+1}(k,j), p^{\frac{1}{2}}, i_k - 2u).$$

Let $r_{k,u}$ denote the total number of rows in all samples $S_u(k,j)$ of level u. Clearly, we have $r_{k,u} = \frac{N'}{p^u}$. By (13), $r_{k,u} \geq qp$ and $r_{k,u} = qp$ only when t is even and $k = t-2$. The recursive calls to *MULTIWAY_MERGE* end at level $\lfloor \frac{i_k-1}{2} \rfloor$, the last call being of the form

$$MULTIWAY\_MERGE\Big(S_{\lfloor \frac{i_k-1}{2} \rfloor}(k,j), p^{\frac{1}{2}}, i_k - 2\Big(\Big\lfloor \frac{i_k-1}{2} \Big\rfloor - 1\Big)\Big).$$

Note that $i_k - 2(\lfloor \frac{i_k-1}{2} \rfloor - 1) = 3$ or $i_k - 2(\lfloor \frac{i_k-1}{2} \rfloor - 1) = 4$, depending on whether or not i_k is odd.

We proceed to demonstrate that, for $2 \leq k \leq t-2$, Iteration k takes $O(r_{k,1})$ time. We will do this by showing that the total time required by each of the five steps of the

MULTIWAY_MERGE calls of each level u is bounded by $O(r_{k,u})$.

Consider a particular level u. Step 1 of all *MULTIWAY_MERGE* calls of level u is performed on the samples $S_u(k,j)$, in increasing order of j, so that all the samples $S_{u+1}(k,j)$ are extracted one after the other. Clearly, the total time for these operations is $O(r_{k,u})$.

We perform Step 3 of all the *MULTIWAY_MERGE* calls of level u, in increasing order of j, to partition into buckets each of the samples $S_u(k,j)$ using the corresponding $S_{u+1}(k,j)$. By Lemma 10, each sample $S_u(k,j)$ is partitioned into $p^{\frac{i_k-2u}{2}}$ buckets and no bucket contains more than $2p^{\frac{3}{2}}$ elements. As discussed in Section 5.2, the task of moving all the elements of each $S_u(k,j)$ to their buckets can be carried out in $O(p^{\frac{i_k-2u+1}{2}})$ time without using the sorting device. Thus, the total time for partitioning the samples $S_u(k,j)$ in all the *MULTIWAY_MERGE* calls of level u is bounded by $O(\frac{N'}{p^{\frac{i_k+1}{2}}} \cdot p^{\frac{i_k-2u+1}{2}}) = O(\frac{N'}{p^u}) = O(r_{k,u})$.

Step 4 of a *MULTIWAY_MERGE* call of level u sorts the buckets (involving the elements of $S_u(k,j)$) obtained in Step 3. We perform Step 4 of all *MULTIWAY_MERGE* calls of level u in increasing order of j and use extended interleaved pipelining with $m = p^{\frac{1}{2}}$ to sort all buckets of each $S_u(k,j)$. There are, altogether, $\frac{N'}{p^{\frac{2u+1}{2}}}$ buckets in all the $S_u(k,j)$s. Thus, the total time for sorting all buckets is bounded by $O(\log^2 p + p^{\frac{1}{2}} \cdot \frac{N'}{p^{\frac{2u+1}{2}}}) = O(\log^2 p + r_{k,u})$. By (13) and 4, the total time for sorting the buckets in all *MULTIWAY_MERGE* calls of level u is $O(r_{k,u})$.

Step 5 of a *MULTIWAY_MERGE* call of level u has three phases. As discussed in Section 5.3, the operations of Phase 1 and Phase 3 that involve the sorting device can be carried out using interleaved pipelining. The operations of Phase 2 that involve the sorting device can be carried out using simple pipelining. Clearly, the time complexity of Step 5 for all *MULTIWAY_MERGE* calls of level u is bounded by $O(\log^2 p + r_{k,u}) = O(r_{k,u})$.

We now evaluate the time needed to perform Step 2 of all the *MULTIWAY_MERGE* calls of level u. First, consider the call of level $\lfloor \frac{i_k-1}{2} \rfloor$,

$$MULTIWAY\_MERGE\Big(S_{\lfloor \frac{i_k-1}{2} \rfloor}(k,j), p^{\frac{1}{2}}, i_k - 2\Big(\Big\lfloor \frac{i_k-1}{2} \Big\rfloor - 1\Big)\Big).$$

The sample $S_{\lfloor \frac{i_k-1}{2} \rfloor+1}(k,j)$ extracted in Step 1 of this call has p elements if i_k is odd, and $p^{\frac{3}{2}}$ elements if i_k is even. If i_k is odd, we use simple pipelining to sort all the samples $S_{\lfloor \frac{i_k-1}{2} \rfloor+1}(k,j)$ in $O(\log^2 p + \frac{N'}{p^{\frac{i_k+1}{2}}})$ time; if i_k is even, we use interleaved pipelining with $m = p^{\frac{1}{2}}$ to sort all the samples $S_{\lfloor \frac{i_k-1}{2} \rfloor+1}(k,j)$ in $O(\log^2 p + \frac{N'}{p^{\frac{i_k}{2}}})$ time. In either case, the time required is bounded by $O(\frac{N'}{p^{\lfloor \frac{i_k-1}{2} \rfloor+1}})$, which is no more than $O(\frac{N'}{p^{\lfloor \frac{i_k-1}{2} \rfloor}}) = O(r_{k,\lfloor \frac{i_k-1}{2} \rfloor})$. Thus, the total time for Steps 1 through 5 of all the *MULTIWAY_MERGE* calls of level $\lfloor \frac{i_k-1}{2} \rfloor$) is no more than $O(r_{k,\lfloor \frac{i_k-1}{2} \rfloor})$. Next, the time to perform Step 2 of all *MULTIWAY_MERGE* calls of level u is inductively derived as $O(r_{k,u})$ using our claim that the total time for Steps 1, 3, 4, and 5 of all *MULTIWAY_MERGE* calls of level u is no more than $O(r_{k,u})$ and the hypothesis that Step 2 of all *MULTIWAY_MERGE* calls of level $u+1$ is $O(r_{k,u+1})$. This, in turn, proves that the total time required for all the *MULTIWAY_MERGE* calls of level u is bounded by $O(r_{k,u})$.

Having shown that the time required for all the *MULTIWAY_MERGE* calls of level u of Iteration k is $O(r_{k,u})$, we conclude that the total time to perform Iteration k is $O(r_{k,1})$, which is $O(\frac{N}{p})$.

Iteration $t-1$. If $N = p^{\frac{t}{2}}$ the N input elements are sorted at the end of $t-2$ iterations. Assume that the algorithm does not terminate in $t-2$ iterations. The input to Iteration $t-1$ is a collection of $q = \frac{N'}{p^{\frac{t}{2}}}$ sorted sequences, where $2 \le q < p^{\frac{1}{2}}$. Each such sequence is of size $p^{\frac{t}{2}}$, stored in $p^{\frac{t-2}{2}}$ consecutive rows. To complete the sorting, we need to merge these q sequences into the desired sorted sequence. This task is performed by the call $MULTIWAY\_MERGE(\Sigma', q, t)$. The detailed implementation of $MULTIWAY\_MERGE(\Sigma', q, t)$ using a sorting network as the sorting device and the analysis involved are almost the same as that of Iteration 2 to Iteration $t-2$, except that different parameters are used. If the interleaved pipelining with $m = p^{\frac{1}{2}}$ is used in a step of *MULTIWAY_MERGE* for iterations 2 to $t-2$, then the corresponding step of *MULTIWAY_MERGE* for iteration $t-1$ uses the interleaved pipelining with $m = q$. Similarly, if the extended interleaved pipelining with $m = p^{\frac{1}{2}}$ is used in a step of *MULTIWAY_MERGE* for iterations 2 to $t-2$, then the corresponding step of *MULTIWAY_MERGE* for iteration $t-1$ uses the extended interleaved pipelining with $m = q$. The *MULTIWAY_MERGE* call of level $\lfloor \frac{t-1}{2} \rfloor$ is $MULTIWAY\_MERGE(S_{\lfloor \frac{t-1}{2} \rfloor}(t-1), q, t-2(\lfloor \frac{t-1}{2} \rfloor - 1))$.

If t is odd, then $t-2(\lfloor \frac{t-1}{2} \rfloor - 1) = 3$, and if t is even, then $t-2(\lfloor \frac{t-1}{2} \rfloor - 1) = 4$. The recursion stops at the $(\lfloor \frac{t-1}{2} \rfloor)$th level. The sample set $S_{\lfloor \frac{t-1}{2} \rfloor+1}(t-1)$ obtained in Step 1 of the *MULTIWAY_MERGE* call of level $\lfloor \frac{t-1}{2} \rfloor$ has $qp^{\frac{1}{2}}$ elements if t is odd and it has qp elements if t is even.

Let $r_{t-1,u}$ be the total number of memory rows in $S_{t-1}(u)$. Clearly, $r_{t-1,u} = qp^{\frac{t-2u}{2}}$. By a simple induction, we conclude that the *MULTIWAY_MERGE* call of level u, $1 \le u < \lfloor \frac{t-1}{2} \rfloor$, takes no more than $O(r_{t-1,u})$ time. The running time of Iteration $t-1$ is the running time of the

MULTIWAY_MERGE call of the first level and it takes $O(r_{t-1,1}) = O(qp^{\frac{t-2}{2}}) = O(\frac{N}{p})$.

We have shown that each of the $t-1$ iterations of *MULTIWAY_MERGE* can be implemented with time $O(\frac{N}{p})$. By (15), we conclude that the running time of our sorting algorithm is $O(\frac{N \log N}{p \log p})$. Since a p-sorter can be abstracted as a sorting network of I/O size p and depth $O(1)$, this time complexity stands if the sorting device used is a p-sorter. The working data memory for each iteration is $O(N)$ simply because the sample size of a *MULTIWAY_MERGE* call of level u is p times the sample size of a *MULTIWAY_MERGE* call of level $u+1$. Since the working data memory of one iteration can be reused by another iteration, the total data memory required by our sorting algorithm remains $O(N)$. Summarizing all our previous discussions, we have proven the main result of this work.

Theorem 6. *Using our simple architecture, a set of N items stored in $\frac{N}{p}$ memory rows can be sorted in row-major order, without any memory access conflicts, in $O(\frac{N \log N}{p \log p})$ time and $O(N)$ data space, by using either a p-sorter or a sorting network of I/O size p and depth $O(\log^2 p)$ as the sorting device.*

ACKNOWLEDGMENTS

This work was supported by US Office of Naval Research grant N00014-97-1-0526, US National Science Foundation grants CCR-9522093 and ECS-9626215, and by Louisiana grant LEQSF(1996-99)-RD-A-16.

REFERENCES

[1] K.E. Batcher, "Sorting Networks and Their Applications," *Proc. Am. Federation Inforimation Processing Socs. Conf.*, pp. 307-314, 1968.
[2] R. Beigel and J. Gill, "Sorting n Objects with a k-Sorter," *IEEE Trans. Computers*, vol. 39, pp. 714-716, 1990.
[3] J. Jang and V.K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *J. Parallel and Distributed Computing*, vol. 25, pp. 31-41, 1995.
[4] D.E. Knuth, *The Art of Computer Programming*, vol. 3. Reading, Mass.: Addison Wesley, 1973.
[5] F.T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Computers*, vol. 34, pp. 344-354, 1985.
[6] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. San Mateo, Calif.: Morgan Kaufmann, 1992.
[7] R. Lin, S. Olariu, J. Schwing, and J. Zhang, "Sorting in O(1) Time on a Reconfigurable Mesh of Size $n \times n$," *Parallel Computing: From Theory to Sound Practice, Proc. European Workshop Parallel Computing (EQPC '92)*, plenary address, pp. 16-27, 1992.
[8] M. Nigam and S. Sahni, "Sorting n Numbers on $n \times n$ Mesh with Buses," *Proc. Int'l Parallel Processing Symp.*, pp. 174-181, 1993.
[9] S. Olariu, M.C. Pinotti, and S.Q. Zheng, "How to Sort N Items Using a Sorting Network of Fixed I/O Size," *IEEE Trans. Parallel and Distributed Systems*, vol. 10, no. 5, pp. 487-499, May 1999.
[10] Y. Zhang and S.Q. Zheng, "An Efficient Parallel Sorting Architecture," *VLSI Design*, vol. 11, no. 2, pp. 137-147, 2000.

Stephan Olariu received the MSc and PhD degrees in computer science from McGill University, Montreal, in 1983 and 1986, respectively. In 1986, he joined the Computer Science Department at Old Dominion University, where he is now a 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.

Dr. Olariu serves on the editorial board of several archival journals including the *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*. He is a member of the IEEE.

M. Cristina Pinotti received the Dr degree cum laude in computer science from the University of Pisa, Italy, in 1986. From 1987-1999, she was a researcher with the National Council of Research at the Istituto di Elaborazione dell'Informazione, Pisa. Currently, she is an associate professor at the University of Trento. In 1994 and 1995, she was a research associate in the Department of Computers Sciences, University of North Texas, Denton. In 1997, she visited the Department of Computer Science at Old Dominion University, Norfolk, Virginia. Her research interests include computer arithmetic, residue number systems, VLSI special purpose architectures, design and analysis of parallel algorithms, parallel data structures, distributed data structures, multiprocessor interconnection networks, and wireless networks. She is a member of the IEEE Computer Society.

Si Qing Zheng received his 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 and served as the program committee chairman of numerous international conferences and an editor of several professional journals. He is a senior member of the IEEE.