A2-55
(1998)

Proceedings of the Tenth
IASTED International Conference

IASTED

A2-55
(1998)

PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS

Las Vegas, Nevada, USA
October 28-31, 1998

Editors: Yi Pan, Selim G. Akl, and Keqin Li

A Publication of
The International Association of Science and Technology for Development - IASTED

ISBN: 0-88986-231-1
ISSN: 1027-2658

IASTED / ACTA Press
ANAHEIM * CALGARY * ZURICH

Proceedings of the 10th IASTED International Conference
PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS
October 28-31, 1998, Las Vegas, Nevada, USA

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

Stephan Si Qi
S. Olariu[†], M. C. Pinotti[‡], S. Q. Zheng[§]

[†] Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, USA
[‡] Istituto di Elaborazione dell'Informazione, C.N.R., Pisa 56126, Italy
[§] Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, USA

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 $\Theta\left(\frac{N\log N}{p\log p}\right)$ time for all ranges of N. 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 sort network.

Key Words: Special-purpose architectures, hardware-algorithms, sorting networks, bitonic sort, 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.

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, 5, 6, 7]. Beigel and Gill [2] showed that the task of sorting N elements, $N \geq p$, requires $\Omega\left(\frac{N\log N}{p\log p}\right)$ 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 $\Theta\left(\frac{N\log N}{p\log p}\right)$, 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.

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 $\Theta\left(\frac{N\log N}{p\log p}\right)$ 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 sort 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.

Due to space constraint, the details of our algorithm implementation and proofs of all our claims are omitted. Interested readers may refer to [8] for more details.

*Work supported by ONR grant N00014-97-1-0526, NSF grants CCR-9522093 and ECS-9626215, and by Louisiana grant LEQSF(1996-99)-RD-A-16.

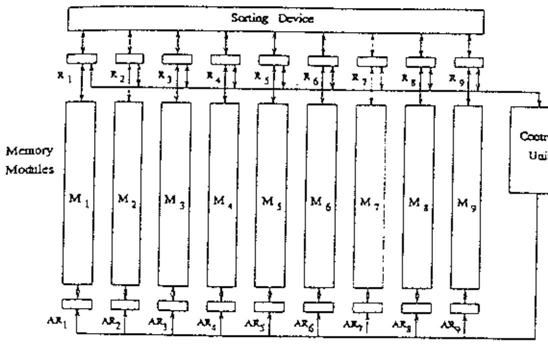

Figure 1: *The proposed architecture for $p = 9$.*

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. A noteworthy feature of our design is that the total additional VLSI area for hardware, other than the data memory, the sorting device, and the control unit does not exceed the area of any VLSI implementation of a sorting device of I/O size p. We consider that a sequential sorting algorithm is adequate for the case $N < p^2$. Consequently, from now on, we assume that

$$N \geq p^2. \qquad (1)$$

This assumption implies that just for addressing purposes we need at least $2\log p$ bits.[1] For the reader's convenience, Figure 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:

(i) 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).

(ii) 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.

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

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 i-th output of the sorting device, or by a broadcast from the CU. The output of register R_i is connected to the i-th 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 0's 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.

(iii) 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.

(iv) 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.

3 The Basic Algorithm – An Extension of Columnsort

The main goal of this section is to provide an extension of the well known Columnsort algorithm [4]. This extended

$$\underset{\text{Input}}{\begin{pmatrix} 6 & 9 & 5 \\ 3 & 6 & 8 \\ 1 & 2 & 6 \\ 4 & 3 & 6 \\ 2 & 1 & 1 \\ 5 & 4 & 7 \end{pmatrix}} \quad \underset{\text{Step 1}}{\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 5 \\ 3 & 3 & 6 \\ 4 & 4 & 6 \\ 5 & 6 & 7 \\ 6 & 9 & 8 \end{pmatrix}} \quad \underset{\text{Step 2}}{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \\ 4 & 6 & 9 \\ 1 & 5 & 6 \\ 6 & 7 & 8 \end{pmatrix}} \quad \underset{\text{Step 3}}{\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 5 & 6 \\ 4 & 5 & 6 \\ 4 & 6 & 8 \\ 6 & 7 & 9 \end{pmatrix}} \quad \underset{\text{Step 4}}{\begin{pmatrix} 1 & 1 & 4 \\ 2 & 5 & 6 \\ 3 & 6 & 8 \\ 1 & 4 & 6 \\ 2 & 5 & 7 \\ 3 & 6 & 9 \end{pmatrix}}$$

$$\underset{\text{Step 5}}{\begin{pmatrix} 1 & 1 & 4 \\ 1 & 4 & 6 \\ 2 & 5 & 6 \\ 2 & 5 & 7 \\ 3 & 6 & 8 \\ 3 & 6 & 9 \end{pmatrix}} \quad \underset{\text{Step 6}}{\begin{pmatrix} - & 2 & 5 & 7 \\ - & 3 & 6 & 8 \\ - & 3 & 6 & 9 \\ 1 & 1 & 4 & + \\ 1 & 4 & 6 & + \\ 2 & 5 & 6 & + \end{pmatrix}} \quad \underset{\text{Step 7}}{\begin{pmatrix} - & 1 & 4 & 7 \\ - & 2 & 5 & 8 \\ - & 3 & 6 & 9 \\ 1 & 3 & 6 & + \\ 1 & 4 & 6 & + \\ 2 & 5 & 6 & + \end{pmatrix}} \quad \underset{\text{Step 8}}{\begin{pmatrix} 1 & 3 & 6 \\ 1 & 4 & 6 \\ 2 & 5 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}} \quad \underset{\text{Step 9}}{\begin{pmatrix} 1 & 3 & 6 \\ 1 & 4 & 6 \\ 1 & 4 & 6 \\ 2 & 5 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}}$$

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

Columnsort algorithm will be implemented in our architectural model and will be invoked repeatedly when sorting a large number of elements.

Columnsort was designed to sort, in column-major order, a matrix of r rows and s columns. The "classic" Columnsort contains 8 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 [4], Leighton poses as an open problem to extend the range of applicability of Columnsort without changing the algorithm "drastically". We provide such an extension. More precisely, 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.

Figure 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.

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

The extended 9-step Columnsort algorithm can be implemented using our architectural model to sort m, $(1 \leq m \leq p^{\frac{1}{2}})$, memory rows in row-major order. Interested readers may refer to [8] for the implementation of the extended Columnsort on our architecture. We call the resulting hardware-algorithm the *basic algorithm.*

Theorem 2 *Using the basic algorithm, 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.*

We present an important application of the basic algorithm. Let *MERGE_TWO_GROUPS* be a procedure which receives two sorted sequences $A = a_1 \leq a_2 \leq \cdots \leq a_{mp}$ and $B = b_1 \leq b_2 \leq \cdots \leq b_{mp}$, stored each in m consecutive memory rows, with $1 \leq m \leq p^{\frac{1}{2}}$, and merge these two sequences into a sorted sequence.

Theorem 3
Procedure MERGE_TWO_GROUPS performs the task of merging two sorted sequences of mp, $(1 \leq m \leq p^{\frac{1}{2}})$, elements each, stored in $2m$ memory rows, in three calls to the basic algorithm and $O(m)$ time for data movement operations not involving sorting.

Based on Theorem 2 and Theorem 3, we have the following claim:

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.*

4 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. In outline, our multiway merge algorithm proceeds as follows. We begin by selecting a sample S of size $mp^{\frac{i-2}{2}}$ from A by retaining every p-th element in each sorted sequence A_j, $(1 \leq j \leq m)$. Next, having sorted S recursively (using multiway merge), we set up a collection of buckets, in a way reminiscent of the bucketing scheme of Olariu and Schwing [7]. As it turns out, the resulting buckets are surprisingly well balanced, each of them containing at most $2mp$ elements. Finally, each bucket is sorted using procedure *MERGE_TWO_GROUPS* and the sorted buckets are then coalesced into the desired sorted sequence. The details are spelled out as follows.

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 p-th 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 {recursively multiway merge $S$}
        $MULTIWAY\_MERGE(S, m, i-2)$;
    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, 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.

For convenience, we view A as a matrix of size $mp^{\frac{i-2}{2}} \times p$, with the t-th 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. 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. At the end of Step 2 of procedure *MULTIWAY_MERGE* the sample set S is sorted in row-major order. Let the sorted version of S be

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

Equation (2) will be used in Step 3 to partition the elements of A into buckets.

In Step 3, elements in A are partitioned into buckets. Our objective is to use $s_1, s_2, \cdots, s_{mp^{(i-2)/2}}$ as delimiters 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.

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

Using our architectural model and some special techniques, this task can be accomplished so that $2m$ memory rows are allocated to each B_j, and furthermore, in case that B_j has less than $2mp$ elements, dummy elements of value $-\infty$ are included into B_j.

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*.

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 preceding the elements 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 buckets into $mp^{\frac{i-2}{2}}$ consecutive memory rows will be referred to as *compaction*. In outline, the compaction proceeds in the following four phases. In Phase 1, we perform a preliminary compaction by removing all the empty rows; in Phase 2, we remove all empty elements from impure rows. It is the case that in Phase 2 new empty rows may be created. In other words, at the end of Phase 2 all the memory rows are either pure or empty. The task of Phase 3 is to finish the compaction by removing the empty rows created in Phase 2. Finally, in Phase 4 the rows are sorted individually to ensure that the entire sequence is sorted. For details of these phases, refer to [8].

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 2). 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 $i \geq 4$, 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.*

5 The Sorting Algorithm

With the basic algorithm and the multiway merge at our disposal, we are in a position to present our sorting algorithm that uses 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 $O\left(\frac{N \log N}{p \log p}\right)$ time and $O(N)$ data space. We assume that $p \geq 16$, which along with (1) implies that

$$\log^2 p \leq p \leq \frac{N}{p}. \tag{3}$$

Equation (3) 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 sort network [1] that we shall tacitly assume.

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

$$p^{\frac{t}{2}} \leq N < p^{\frac{t+1}{2}}. \tag{4}$$

In turn, equation (4) guarantees that

$$t = \left\lfloor \frac{\log N}{\log p^{\frac{1}{2}}} \right\rfloor. \tag{5}$$

At this point we note that (4) and (5), combined, guarantee that

$$\log^2 p \leq \frac{N}{p^{\frac{k}{2}}} \text{ for all } k \leq t-2. \tag{6}$$

Write

$$q = \left\lceil \frac{N}{p^{\frac{t}{2}}} \right\rceil \tag{7}$$

and observe that by (4),

$$1 \leq q \leq p^{\frac{1}{2}}. \tag{8}$$

For reasons that will become clear later, we pad Σ with an appropriate number of $+\infty$ elements in such a way that, with N' standing for the length of the resulting sequence Σ', we have

$$N' = q * p^{\frac{t}{2}}. \tag{9}$$

It is important to note that (4), (7), and (9), combined, guarantee that

$$N \leq N' \leq 2N, \tag{10}$$

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.

The sorting algorithm consists of a number of iterations. In the first iteration, we partition the $\frac{N'}{p}$ memory rows into $\frac{N'}{p^{\frac{3}{2}}}$ groups, each involving $p^{\frac{1}{2}}$ consecutive memory rows. The basic algorithm is applied to each group and, as a result, we obtain $\frac{N'}{p^{\frac{3}{2}}}$ sorted sequences, each of size $p^{\frac{3}{2}}$. From this point on, each iteration corresponds, conceptually, to the task of using procedure *MULTIWAY_MERGE* to create longer and longer sorted sequences. This process is continued until the entire sequence is sorted. Since in each such iteration the number of sorted sequence decreases by a factor of $p^{\frac{1}{2}}$, (4) implies that, with t specified in (5), the input sequence is sorted at the end of $t-2$ iterations if $N = p^{\frac{t}{2}}$ and at the end of $t-1$ iterations if $N > p^{\frac{t}{2}}$.

We discuss our algorithm under the assumption that the sorting device used in our architecture is a sorting network of I/O size p and depth $O(\log^2 p)$. In order to guarantee an overall running time of $O(\frac{N \log N}{p \log p})$, we ensure that each iteration of our sorting algorithm can be performed in $O(\frac{N}{p})$ time. The sorting network will be used in the following three contexts:

(i) to sort, individually, M memory rows;

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

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

For an efficient implementation of (i) 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 (ii) 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 (iii) 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: two calls for sorting the two halves, and three additional calls for merging the two halves using procedure *MERGE_TWO_GROUPS*. The extended interleaved pipelining consists of five interleaved pipelining

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

each corresponding to one of the five calls to the ba-
algorithm. Thus, the task of sorting all groups can be formed in $O(\log^2 p + Mm)$ time. We now discuss each the iterations of our sorting algorithm in more detail.

Iteration 1

Partition the input 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 level, 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 of *MULTIWAY_MERGE*$(S_u(k,j), p^{\frac{1}{2}}, i_k - 2(u-1))$. 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*$(S_{\lfloor\frac{i_k-1}{2}\rfloor}(k,j), p^{\frac{1}{2}}, i_k - 2(\lfloor\frac{i_k-1}{2}\rfloor - 1))$. To clarify this point, 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. Let $r_{k,u}$ denote the total number of rows in all samples $S_u(k,j)$ of level u in Iteration k. Clearly, we have $r_{k,u} = \frac{N'}{p^u}$. By (3), $r_{k,u} \geq qp$, and $r_{k,u} = qp$ only when t is even and $k = t-2$.

We want 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 defer the assessment of the running time of Step 2 until we evaluate the combined running times of Step 3 to Step 5 of all *MULTIWAY_MERGE* calls of level 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 $S_{u+1}(k,j)$. Without using the sorting device, 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 (3), the total time for sorting the buckets in all *MULTIWAY_MERGE* calls of level u is $O(r_{k,u})$.

The four phases of Step 5 of a *MULTIWAY_MERGE* call of level u ,as proved in [8], require $O(\log^2 p + r_{k,u}) = O(r_{k,u})$ time. Thus far, Steps 1, 3, 4, and 5 of all the *MULTIWAY_MERGE* calls of level u can be carried out in $O(r_{k,u})$ time. 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*$(S_{\lfloor\frac{i_k-1}{2}\rfloor}(k,j), p^{\frac{1}{2}}, i_k - 2(\lfloor\frac{i_k-1}{2}\rfloor - 1))$. 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 $O(r_{k,\lfloor\frac{i_k-1}{2}\rfloor})$. Thus, the total time required for all the *MULTIWAY_MERGE* calls of level u is bounded by $O(r_{k,u})$. We conclude that the total time to perform Iteration k is $O(r_{k,1})$ which is $O(\frac{N}{p})$. To summarize our findings we state the following important result.

Theorem 6 *The output of Iteration k, $(2 \leq i \leq t-2)$, is a collection of $\lceil \frac{N'}{p^{\frac{k+2}{2}}} \rceil$ sorted sequences each of size $p^{\frac{k+2}{2}}$ stored in $p^{\frac{k}{2}}$ consecutive memory rows. Furthermore, Iteration i runs in time bounded by $O(\frac{N}{p})$.*

Note that if $N = p^{\frac{t}{2}}$ the N input elements are sorted at the end of $t-2$ iterations.

Iteration $t-1$

Assume that the algorithm does not terminate in $t-2$ iterations. Then, by (4) and Theorem 6, we know that the input to Iteration $t-1$ is a collection of $q = \frac{N'}{p^{\frac{t}{2}}}$ sorted sequences, where $2 \leq q < p^{\frac{1}{2}}$. 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*(Σ', q, t). The detailed implementation of *MULTIWAY_MERGE*(Σ', 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 $m = q$.

Let $r_{t-1,u}$ be the total number of memory rows in $S_{t-1}(u)$. Clearly, $r_{t-1,u} = qp^{\frac{t-2u}{2}}$. We want to show that *MULTIWAY_MERGE* call at level u for Iteration $t-1$ takes no more than $O(r_{t-1,u})$ time if $u < \lfloor \frac{t-1}{2} \rfloor$. First, we estimate the running time of the *MULTIWAY_MERGE* call of level $\lfloor \frac{t-1}{2} \rfloor$. In its Step 2, the sample elements are sorted. If t is odd, $S_{\lfloor \frac{t-1}{2} \rfloor + 1}(t-1)$ is sorted by one call to the sorting device in $O(\log^2 p)$ time, and if t is even, $S_{\lfloor \frac{t-1}{2} \rfloor + 1}(t-1)$ is sorted by one call to the basic algorithm in $O(\log^2 p + q)$ time (refer to Theorem 2). However, the running time of *MULTIWAY_MERGE*$(S_{\lfloor \frac{t-1}{2} \rfloor}(t-1), q, t - 2(\lfloor \frac{t-1}{2} \rfloor - 1))$ is dominated by the total time for Steps 1, 3, 4, and 5 of *MULTIWAY_MERGE*$(S_{\lfloor \frac{t-1}{2} \rfloor}(t-1), q, t - 2(\lfloor \frac{t-1}{2} \rfloor - 1))$, which is $O(\log^2 p + qp^{\frac{1}{2}})$ if t is odd, or $O(\log^2 p + qp) = O(qp)$ if t is even. Now, consider the *MULTIWAY_MERGE* call of level $\lfloor \frac{t-1}{2} \rfloor - 1$. It is easy to verify that the total time for Steps 1, 3, 4 and 5 is no more than $O(\log^2 p + qp^{\frac{t-2(\lfloor \frac{t-1}{2} \rfloor - 1)}{2}}) = O(qp^2)$. As we just showed that Step 2 in this level is the *MULTIWAY_MERGE* call of level $\lfloor \frac{t-1}{2} \rfloor$, and it takes no more than $O(qp)$ time. Thus, *MULTIWAY_MERGE* call of level $\lfloor \frac{t-1}{2} \rfloor - 1$ takes no more than $O(qp^2) = O(r_{t-1, \lfloor \frac{t-1}{2} \rfloor - 1})$ time. By a simple induction, we conclude that the *MULTIWAY_MERGE* call of level u, $(1 \leq u < \lfloor \frac{t-1}{2} \rfloor)$, 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 (5), we conclude that the running time of our sorting algorithm is $O\left(\frac{N \log N}{p \log p}\right)$. Since a p-sorter can be considered 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. It is not difficult to see that the working data memory for each iteration is $O(N)$. Since the working data memory of one iteration can be reused by another iteration, the total data memory required by our sorting algorithm remains to be $O(N)$. Summarizing all our previous discussions, we have proved the main result of this work.

Theorem 7 *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 $O\left(\frac{N \log N}{p \log p}\right)$ 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.*

References

[1] K. E. Batcher, Sorting networks and their applications, *Proc. of AFIPS Conference*, 1968, 307–314.

[2] R. Beigel and J. Gill, Sorting n objects with a k-sorter, *IEEE Transactions on Computers*, C-39, (1990), 714–716.

[3] J. Jang and V.K. Prasanna, An optimal sorting algorithm on reconfigurable mesh, *Journal of Parallel and Distributed Computing*, 25, (1995), 31–41.

[4] F. T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Transactions on Computers*, C-34, (1985), 344–354.

[5] 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, Proceedings of EWPC'92*, Plenary Address, IOS Press, Amsterdam, 1992, 16–27.

[6] M. Nigam and S. Sahni, Sorting n numbers on $n \times n$ mesh with buses, *Proc. International Parallel Processing Symposium*, April 1993, 174–181.

[7] S. Olariu and J. Schwing, A new deterministic sampling scheme with applications to broadcast efficient sorting on the reconfigurable mesh, *Journal of Parallel and Distributed Computing*, 32, (1996), 215–222.

[8] S. Olariu, S.Q. Zheng and M.C. Pinotti, *An Optimal Hardware-Algorithm for sorting Using a Fixed-Size Parallel Sorting Device*, Tech Rep. #97-008, Dept. of Comput. Sci., Louisiana State University, Baton Rouge 1997.