A Novel Deterministic Sampling Scheme with Applications to Broadcast-Efficient Sorting on the Reconfigurable Mesh *

Stephan Olariu James L. Schwing

Department of Computer Science, Old Dominion University

Norfolk, VA 23529, U.S.A.

Abstract

The main contribution of this work is to present a simple deterministic sampling strategy that, when used for bucket sorting, yields buckets that are remarkably well balanced, making costly balancing unnecessary. To the best of our knowledge this is the first instance of a deterministic sampling strategy featuring this performance. Although the strategy is perfectly general, we illustrate its power by devising a VLSI-optimal, O(1) time sorting algorithm for the reconfigurable mesh. As a byproduct of the inherent simplicity of our sampling and bucketing scheme we show that our sorting algorithm can be implemented using only 35 broadcast operations, a substantial improvement over the previously best known algorithm that requires 59 broadcasts.

Keywords: deterministic sampling, bucket sort, reconfigurable meshes, sorting, VLSI optimal algorithms, constant-time algorithms

1 Introduction

Sorting is, unquestionably, one of the fundamental operations in computer science. A natural sorting strategy known as bucket sort proceeds in the following sequence of steps: (1) a sample consisting of $s-1$ items of the input sequence is chosen; (2) the sample is sorted thus delimiting s buckets in the obvious way; (3) the remaining items are sent to the appropriate buckets, and (4) the items in each bucket are sorted.

The obvious difficulty with the strategy we just outlined is that the resulting buckets are, as a rule, far from being balanced. A standard way around this difficulty is to use randomized sampling – resulting in buckets that are well balanced with high probability [3, 5, 6, 8, 24, 25]. Typically, by balanced we mean that the buckets differ from the average by at most a constant factor. To date, no deterministic sampling strategy is known to yield balanced buckets.

Our main contribution is to propose a simple deterministic sampling strategy that results in surprisingly well balanced buckets, making subsequent rebalancing unnecessary. We illustrate the power of our sampling by using it to devise a broadcast-efficient VLSI-optimal constant time sorting algorithm for the reconfigurable mesh.

*Work supported by NASA grant NCC1-99, by ONR grant N00014-95-1-0779, and by NSF grant CCR-9407180

The mesh-connected architecture has emerged as one of the most popular choices for solving a multitude of computational tasks in image processing, robotics, pattern recognition, computational geometry, and computer vision. This is due, in part, to its simple interconnection topology and to the fact that many problems feature input data that maps naturally onto the mesh. In addition, the mesh topology is particularly well suited for VLSI implementation [1]. At the same time, due to its large communication diameter, the mesh tends to be slow when it comes to handling data transfer operations over long distances.

In an attempt to alleviate this problem, mesh-connected machines have been enhanced by the addition of various bus systems. In particular, a number of bus systems whose configuration can change, under program control, have been proposed in the literature [10]. Examples include the *bus automaton* [26], the *reconfigurable mesh* [18], the *content addressable array processor* [30], the *reconfigurable network* [2], the *reconfigurable bus with shift switching* [12], the *polymorphic torus* [9, 17], and the *polymorphic processor array* [14, 15].

Among these, the reconfigurable mesh has emerged as a very attractive and versatile architecture. The reconfigurable mesh combines two desirable features of massively parallel architectures, namely, nearly constant diameter and a dynamically reconfigurable bus system.

It is not surprising, therefore, that the sorting problem has received a great deal of attention in the context of reconfigurable architectures [4, 7, 9, 11, 13, 16, 17, 19, 20, 22, 23, 27, 29]. In an early paper, Wang *et al.* [29] have proposed a constant-time sorting algorithm on a three-dimensional reconfigurable mesh, using $O(n^3)$ processors. Later Nakano *et al.* [19] have proposed an algorithm to sort n items in constant time on a reconfigurable mesh of size $n \times n \log^2 n$. More recently and independently Lin, Olariu, and Schwing [13] as well as Jang and Prasanna [7] have obtained for the first time VLSI-optimal sorting algorithms for the reconfigurable mesh. Both their algorithms sort n items stored in one row of a reconfigurable mesh of size $n \times n$ in constant time. The algorithms of [13] and [7] are vastly different. Specifically, the algorithm of Lin *et al.* [13] sorts by multi-selection, while the algorithm of Jang and Prasanna [7] implements Leighton's well-known column-sort strategy. Essentially at the same time Schuster [27] obtained a VLSI-optimal sorting algorithm on a model similar to the reconfigurable mesh and consisting of layered $n \times n$ reconfigurable meshes. Later, Nigam and Sahni [20] have proposed a more efficient implementation of column-sort that requires only 59 broadcasts.

Our second main contribution is to show that the proposed deterministic sampling technique yields a VLSI-optimal, constant time, broadcast-efficient sorting algorithm for the reconfigurable mesh. Rather remarkably, our sorting algorithm requires only 35 broadcasts, a substantial improvement over the 59 broadcasts required by the algorithm of Nigam and Sahni [20].

The remainder of this paper is organized as follows: Section 2 discusses the model of computation used throughout the paper; Section 3 reviews basic data movement results; Section 4 presents the deterministic sampling technique; Section 5 discusses implementation details; finally, Section 6 offers concluding remarks and open problems.

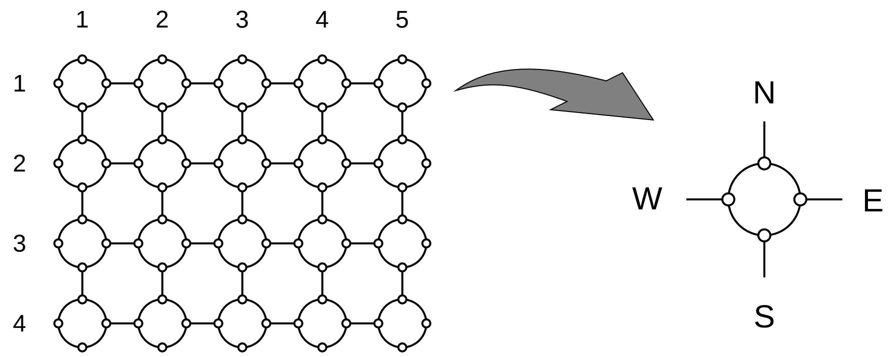

Figure 1: *A reconfigurable mesh of size* 4×5

2 The Computational Model

The computational model used throughout this work is the *reconfigurable mesh.*[1] A reconfigurable mesh of size $M \times N$ consists of MN identical processors positioned on a rectangular array with M rows and N columns. As usual, it is assumed that every processor knows its coordinates within the mesh. We let $P(i,j)$ denote the processor placed in row i and column j, with $P(1,1)$ in the north-west corner of the mesh. Every processor $P(i,j)$ is connected to its four neighbors $P(i-1,j)$, $P(i+1,j)$, $P(i,j-1)$, and $P(i,j+1)$, provided they exist, and is endowed with 4 ports denoted by N, S, E, and W as illustrated in Figure 1. Local connections between these ports can be established, under program control, creating a powerful bus system that changes dynamically to accommodate various computational needs. Our computational model allows at most two connections to be set in each processor at any one time. Furthermore, these two connections must involve disjoint pairs of ports, as illustrated in Figure 2. It is worth noting that in the absence of local connections, the reconfigurable mesh is functionally equivalent to the mesh connected computer. For practical reasons, at any given time, only one processor can broadcast a value onto a bus. Processors, if instructed to do so, read the bus: if no value is being transmitted on the bus, the read operation has no result.

We assume that the processors have a constant number of registers of $O(\log MN)$ bits and a very basic instruction set. Each instruction can consist of setting local connections, performing a simple arithmetic or boolean operation, broadcasting a value on a bus, or reading a value from a specified bus. More sophisticated operations including floor and modulo computation are specifically excluded. As it turns out, these operations can, in fact, be added to the repertoire with very little overhead, thus setting the stage for a very powerful computing environment [21]. We assume a SIMD model: in each time unit the same instruction is broadcast to all processors,

[1]When no confusion is possible a reconfigurable mesh will be referred to, simply, as a mesh.

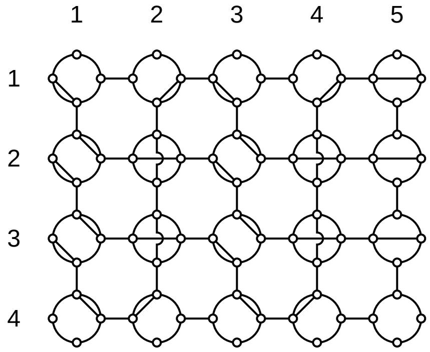

Figure 2: *Illustrating possible connections*

which execute it and wait for the next instruction. Its low wiring cost and regular structure make the reconfigurable mesh suitable for VLSI implementation [9, 14, 15]. In addition, it is not hard to see that the reconfigurable mesh can be used as a universal chip capable of simulating any equivalent-area architecture without loss of time.

It is worth mentioning that at least three recent VLSI implementations have demonstrated the feasibility and benefits of the reconfigurable mesh: these are YUPPIE chip [9, 14], GCN chip [28], and the PPA [15]. These implementations suggested that the broadcast delay, although not constant, is very small. For example, only 16 machine cycles are required to broadcast on a 10^6-processor YUPPIE. The GCN has further shortened the broadcast delay by adopting pre-charged circuits. Recently, it has been shown that the broadcast delay can be reduced even further if the reconfigurable bus system is implemented using optical fibers as the underlying global bus system and using electrically controlled directional coupler switches for connecting or disconnecting two fibers. Due to these new developments, the reconfigurable mesh is likely to become commercially available in the near future. Consequently, algorithms developed on this platform will become of practical relevance.

3 Preliminaries

Data movement operations constitute the basic building blocks that lay the foundations of many efficient algorithms for parallel machines constructed as an interconnection network of processors. The purpose of this section is to review a number of data movement results for the reconfigurable mesh that will be instrumental in the design of our sorting algorithm.

To begin, given a binary sequence $b_1, b_2, \ldots, b_n$ the prefix sum problem involves computing all the sums $b_1 + b_2, b_1 + b_2 + b_3, \ldots, b_1 + b_2 + \ldots + b_n$. Recently, Olariu *et al.* [22] proved the following result.

Proposition 3.1. On a reconfigurable mesh of size $m \times n$ with $1 \leq m \leq n$ the prefix sums of an n-element binary sequence can be computed in $O(\log n)$ time if $m = 1$, and in $O(\frac{\log n}{\log m})$ time if $m > 1$. □

Proposition 3.1 implies the following result that we state for further reference.

Proposition 3.2. On a reconfigurable mesh of size $n \times n$ the prefix sums of an n-element binary sequence can be computed in $O(1)$ time. □

Our sorting algorithm relies, in part, on the following result proved in Olariu *et al.* [22] and also very easy to obtain from scratch.

Proposition 3.3. On a reconfigurable mesh of size $mn \times n$ with $1 \leq m \leq n$ an n-element sequence chosen from a totally ordered universe can be sorted in $O(\log n)$ time if $m = 1$, and in $O(\frac{\log n}{\log m})$ time if $m > 1$. □

4 The Bucket Sort Algorithm – an Overview

The main goal of this section is to discuss our novel sorting algorithm in sufficient detail to convince the reader that the running time is bounded by a constant. A careful implementation, with the stated goal of minimizing the number of broadcasts, will be presented in the next section.

The novelty of our approach resides in a simple and elegant sampling and bucketing scheme. The main feature of the sampling scheme is that the resulting buckets are well balanced, making costly rebalancing unnecessary. Throughout this section we assume a reconfigurable mesh $\mathcal{R}$ of size $n \times n$ and an input sequence $S = a_1, a_2, \ldots, a_n$ stored in the first row of the platform one item per processor. For simplicity of exposition we assume that all the items in S are distinct. The algorithm proceeds in the following sequence of computational steps.

Step 1. Partition $\mathcal{R}$ into $n^{3/8}$ submeshes of size $n \times n^{5/8}$ as illustrated in Figure 3. For later reference, we shall refer to these submeshes as $R_1, R_2, \ldots, R_{n^{3/8}}$. Using Proposition 3.3 we sort the items stored in the first row of every submesh R_i ($1 \leq i \leq n^{3/8}$) in $O(1)$ time.

Step 2. Partition the items in the first row of every submesh R_i ($1 \leq i \leq n^{3/8}$) into $n^{3/8}$ groups of $n^{1/4}$ items each. We extract a *sample* A of S by retaining the largest item in every such group. This item will be subsequently referred to as the *leader* of the group. (It is easy to confirm that A contains $n^{3/4}$ items.)

Step 3. Our next goal is to rank the items in A: this amounts to computing, for every item in A, the number of items in A smaller than it. To perform this operation efficiently, we now perceive $\mathcal{R}$ as a collection of submeshes $T_1, T_2, \ldots, T_{n^{3/4}}$ each of size $n^{1/4} \times n$ as illustrated in Figure 4. The intention is to rank the first sample item in T_1, the second in T_2, and so on. This task can be solved by reducing it, in a fairly obvious way, to computing the number of 1's in a binary sequence. By Proposition 3.1 the running time is bounded by a constant.

Comment: At this point, we specify the sample A in sorted order as $q_1, q_2, \ldots, q_{n^{3/4}}$. It is important to observe that every sample point knows its position in this enumer-

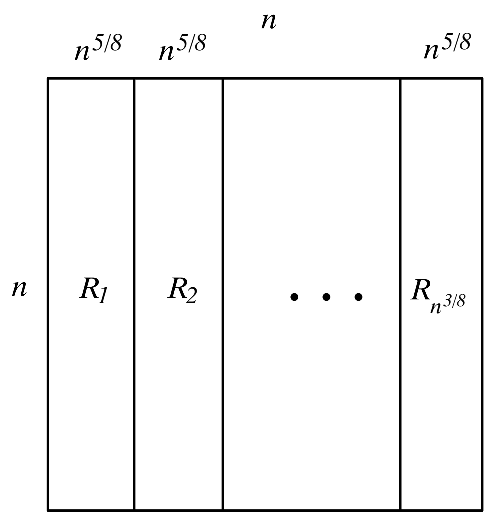

Figure 3: *Illustrating the partition into R_i's*

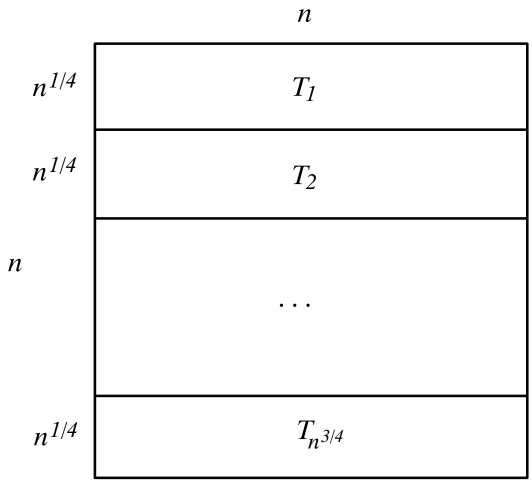

Figure 4: *Illustrating the partition into T_i's*

ation, without A being actually sorted.

Step 4. We are now in a position to specify our bucketing scheme. We define a set of $n^{3/8}$ buckets B_1, B_2, ..., $B_{n^{3/8}}$, by setting

$$B_1 = \{x \in S \mid x \leq q_{n^{3/8}}\} \tag{1}$$

and by setting for all i $(2 \leq i \leq n^{3/8})$,

$$B_i = \{x \in S \mid q_{(i-1)n^{3/8}} < x \leq q_{in^{3/8}}\}. \tag{2}$$

By (1) and (2), the leaders $q_{(i-1)n^{3/8}+1}$ through $q_{in^{3/8}}$ belong to bucket B_i. This observation motivates us to call a group defined above *regular* with respect to bucket B_i if its leader belongs to B_i. Similarly, a group is said to be *special* with respect to bucket B_i if its leader belongs to a bucket B_t with $t > i$, while the leader of the preceding group, if any, belongs to a bucket B_s with $s \leq i$. For an illustration of these concepts the reader is referred to Figure 5.

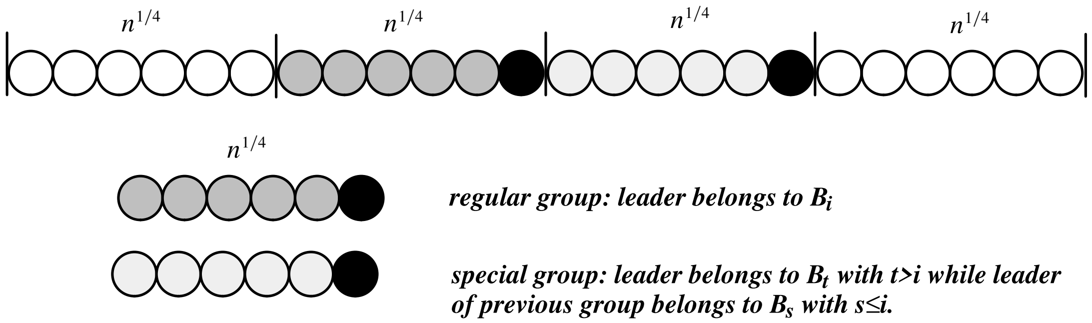

Figure 5: *Illustrating regular and special groups with respect to a bucket*

It is easy to see that our definition of buckets implies that all items that belong to bucket B_i must either be in regular groups with respect to B_i or in special groups with respect to B_i.

Somewhat surprisingly, this simple bucketing scheme results in buckets that are well balanced, in the sense that none of them contains more than $2n^{5/8}$ of the input items. The details are spelled out by the following two results.

Lemma 4.1. With respect to every bucket B_i, there exist exactly $n^{3/8}$ regular groups and at most $n^{3/8}$ special groups.

Proof. The number of regular groups follows directly from our bucketing scheme specified in (1) and (2). The claim concerning the number of special groups follows from the sortedness of each of the $n^{5/8}$ items stored in the R_j's. Specifically, the sortedness guarantees that every R_j contains *at most* one special group with respect to an arbitrary bucket B_i. Consequently, there are at most $n^{3/8}$ special groups with respect to any given bucket. □

Lemma 4.2. Every bucket B_i $(1 \leq i \leq n^{3/8})$ contains at most $2n^{5/8}$ items.

Proof. By Lemma 4.1, with respect to any bucket B_i there are $n^{3/8}$ regular groups and at most $n^{3/8}$ special groups. Since every group contains $n^{1/4}$ items, it follows there there are at most $2n^{5/8}$ items in B_i, as claimed. □

Step 5. Partition the original mesh $\mathcal{R}$ into submeshes $G_1, G_2, \ldots, G_{n^{3/8}}$ each of size $n^{5/8} \times n$ as illustrated in Figure 6.

Our next goal is to move the items in every bucket into the corresponding submesh, with the items in bucket B_i $(1 \leq i \leq n^{3/8})$ being routed to G_i at most two items per processor. The first task to perform, in this direction, is to inform each group of the identity of the bucket with respect to which the group is regular. This task is performed trivially in one broadcast. Identifying the buckets with respect to which a group is special is done similarly. Now in two broadcasts the items of every bucket B_i are moved, at most two items per processor, in the corresponding submesh G_i. The reader is referred to Section 5 for the details of this data movement.

Step 6. The goal now becomes to rank the items with respect to the bucket to which they belong. We proceed as in Proposition 3.3, with the exception of the final data movement that routes the items to their position in sorted order. This routing operation is not necessary here since we are only interested in ranks. This task can

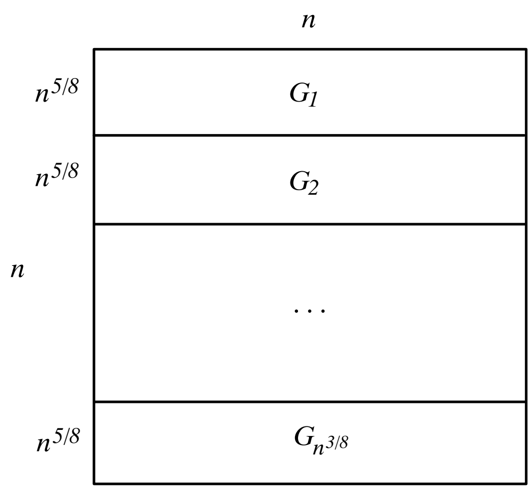

Figure 6: *Illustrating the partition into G_i's*

be performed, therefore, in O(1) time.

Step 7. Last, we compute the final rank of each of the items in the sorted version of S. This task is performed in two stages. First, we compute the final rank of the largest item in each bucket. As we shall argue in the next section, this task can be reduced to an instance of the binary prefix sums problem discussed in Section 3. Once this partial solution is available, it is easy to compute the final rank of every item in S: all that is needed is a simple broadcast operation. Finally, two more broadcasts are sufficient to route the items to their final destination in the first row of $\mathcal{R}$.

Since all the steps in this algorithm run in time bounded by a constant, we have established the following result.
Theorem 4.3. The task of sorting n items stored one per processor in the first row of a reconfigurable mesh of size $n \times n$ can be performed in O(1) time. □

5 The Bucket Sort Algorithm – Implementation Details

The purpose of this section is to provide implementation details for the sorting algorithm specified in Section 4. Although the running time of the algorithm is O(1), a more accurate measure of its complexity is provided by the number of broadcast operations required by the algorithm [20]. The best performance of a sorting algorithm known to date involves 59 broadcasts [20]. As we shall see, our algorithm can be implemented to run in only 35 broadcasts, a considerable improvement over the state of the art.

We begin our discussion by taking a closer look at the number of broadcasts involved in each of the steps of the sorting algorithm.
Step 1. The items in a generic R_i will be sorted using enumeration sort. In other words, we count for every item in the first row of R_i, the number of items smaller than it. For this purpose, having replicated the first row in all the other rows of

the mesh, we allocate to every item of R_i a submesh of size $n^{3/8} \times n^{5/8}$. We further subdivide this mesh into two submeshes of size $\frac{n^{3/8}}{2} \times n^{5/8}$ referred to as Odd and Even as illustrated in Figure 7. By virtue of the previous data movement, the sequence of items is available in the first row of each of these submeshes. Let x be the item that we want to rank. We assume, without loss of generality that x is the leftmost item in the first row.

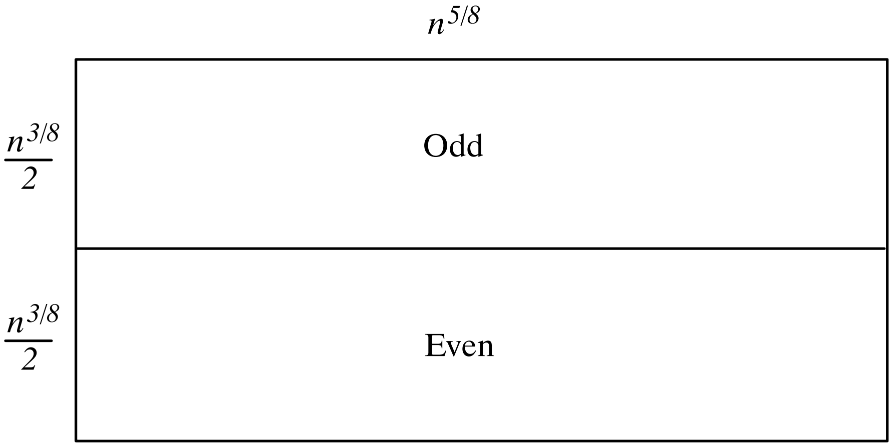

Figure 7: *Illustrating the partition into Odd and Even*

To begin, construct a binary sequence in the first row of Odd and Even by broadcasting x along the first row and by mandating every processor to write a 1 or a 0 in a local register depending on whether or not the item it stores is smaller than x. It is clear that the rank of x is precisely the number of 1's in the resulting binary sequence. In fact, we shall compute the rank of x among the odd-indexed bits of the sequence in the submesh Odd and the rank of x among the even-indexed bits in the submesh Even. This ranking will be done in parallel in both submeshes. Setting up the binary sequence in the first row of Odd and Even requires 1 broadcast. As in the prefix sums algorithm of Olariu *et al.* [22], the task of ranking x is performed in $\left\lceil \frac{\log n^{5/8}}{\log \frac{n^{3/8}}{2}} \right\rceil = \left\lceil \frac{5 \log n}{3 \log n - 8} \right\rceil$ iterations as described below. Clearly, for sufficiently large n the number of iterations is 2. The computation performed involves the following operations (in each case, the number of broadcasts is shown in parentheses):

- set up the local connections for the first iteration in the corresponding submesh (Odd and Even) (1 broadcast);
- broadcast the signal of the first iteration and record the result by some processor in the last column (1 broadcast);
- set up the local connections for the second iteration. This is done by local communication from the adjacent "wrap-around column" [22] (0 broadcasts);
- broadcast the signal of the second iteration and record the result by some processor in the last column (1 broadcast);

- combine the results of the first and second iterations and return the value to the processor storing x (2 broadcasts).

- combine the ranks of x among the odd bits and even bits (0 broadcasts);

- route x to the position in the first row of R_i corresponding to its rank (2 broadcasts).

Consequently, the task of ranking x (including replicating the input as described above) can be performed in 9 broadcasts. Therefore, Step 1 of the algorithm requires 9 broadcasts altogether.

Step 2. Requires no broadcast.

Step 3. The main task is to rank the sample A of size $n^{3/4}$. For this purpose, we allocate to each sample item one of the submeshes T_j ($1 \leq j \leq n^{3/4}$). The key observation for an efficient implementation is that since the sample items are sufficiently far apart, we do not need to handle odd and even bits separately. The computation is done, essentially, as in Step 1 with the exception that there will be 3 iterations. The details follow (in each case, the number of broadcasts is shown in parentheses):

- replicate the sequence in all rows of $\mathcal{R}$; this has already been done in Step 1 (0 broadcasts);

- set up the binary sequence for the first iteration (1 broadcast);

- broadcast the signal of the first iteration and record the result by some processor in the last column (1 broadcast);

- set up the local connections for the second iteration. This is done by local communication from the adjacent "wrap-around column" (0 broadcasts);

- broadcast the signal of the second iteration and record the result by some processor in the last column (1 broadcast);

- set up the local connections for the third iteration. This is done by local communications only (0 broadcasts);

- broadcast the signal of the third iteration and record the result by some processor in the last column (1 broadcast);

- combine the results and return the rank (2 broadcast).

Consequently, the task of ranking the samples in A can be performed in 6 broadcasts.

Steps 4 and 5. These two steps have for principle goal the setup of the buckets and, at the same time, the routing of the data items to the corresponding buckets.

The first operation is to inform each group of the identity of the unique bucket with respect to which the group is regular. This can be done trivially in one broadcast.

Next, to determine the identity of the bucket(s) with respect to which the group is special, the leader of the previous group, if any, broadcasts westbound. Thus, in 2 broadcast operations the items in all groups have the information about the regular and special groups of interest to them.

The routing operation is done in three more broadcasts: the first broadcast is along vertical buses, the second and third along horizontal buses. Although the data movement is trivial, one subtle detail is worth mentioning, however. Recall that Lemma 4.2 guarantees that no bucket contains more than $2n^{5/8}$ items. In the light of this result, we wish to route the items such that every processor in the first column of every G_j contains at most two items, one from a regular group, the other from a special group with respect to bucket B_j.

To give a reader an idea of how this can be achieved, recall that the definition of a generic bucket B_j in (1) and (2) implies that every leader of a group regular with respect to B_j knows its rank among the other regular groups with respect to the bucket. Our intention is to route the items in the first regular group in the topmost $n^{1/4}$ positions, the items in the second regular group in the next $n^{1/4}$ positions, and so on. Thus, every processor in the first column of G_j will contain exactly one item from a regular group. From now on, we shall refer to items from regular groups as *regular* items, while the items in special groups will be termed *special items*. Special groups with respect to bucket B_j are handled similarly. To see how this is done, recall that the sortedness of the items in every R_i at the end of Step 1 guarantees that every R_i contains at most one special group with respect to a given bucket B_j. This provides every group special with respect to a bucket a unique rank. Now the groups are placed in the first column of G_j in the order of their ranks. Clearly, three broadcasts suffice, as we claimed above.

In fact, one can do better. Since the first row of the $\mathcal{R}$ has been replicated in all the rows and since the ranks of the sample items have been broadcast along vertical buses, the whole information available to the processors in the first row is available in all other rows. As a result of this observation, the vertical broadcast mentioned above is not really necessary. It was included to make the presentation more transparent. Thus, Steps 4 and 5, combined, require 4 broadcasts altogether.

Step 6. We proceed to sort the items in a generic bucket B_j using enumeration sort, that is, we count for every item in the bucket the number of items smaller than it. For this purpose, we allocate to every item in B_j a submesh of size $n^{5/8} \times \frac{n^{3/8}}{2}$. Further, every such submesh will be partitioned into four submeshes ROdd, REven, SOdd and SEven each of size $n^{5/8} \times \frac{n^{3/8}}{8}$. The intention is to rank a given item x with respect to the regular items of G_j in ROdd and REven, much as we did in Step 1. At the same time, x will be ranked with respect to the special items in G_j in SOdd and SEven, in parallel. By Proposition 3.3 the number of iterations involved is $\left\lceil \frac{\log n^{5/8}}{\log \frac{n^{3/8}}{8}} \right\rceil = \left\lceil \frac{5 \log n}{3 \log n - 24} \right\rceil \leq 2$ for sufficiently large n. It is now a straightforward but slightly tedious task to confirm that the total number of broadcasts required by this step is 8. We leave the details to the reader.

Step 7. The last stage of our algorithm involves computing the final rank of each item in S and routing the items to the first row of $\mathcal{R}$ in sorted order.

The task at hand is naturally decomposed into two subtasks: first, we determine the final rank of the largest item in each bucket and then we extend this to all the remaining items.

We shall reduce the task of ranking the largest item in each G_j ($1 \leq j \leq n^{3/8}$) to that of computing the prefix sums of n bits in a reconfigurable mesh of size $n \times n$. This is trivially done in O(1) time. To begin, consider the largest item in G_j. By definition, this item is $q_{jn^{3/8}}$. Moreover, by Lemma 4.2 the rank of this item with respect to B_j is at most $2n^{5/8}$.

The processor storing $q_{jn^{3/8}}$ will write the corresponding rank r as $r = a * n^{5/8} + b$, with $0 \leq a \leq 1$ and $0 \leq b \leq n^{5/8}$. The intention is to convert the rank to two binary sequences, one corresponding to a the other to b. Specifically, if $a = 1$, the processor storing $q_{jn^{3/8}}$ broadcasts the packet (1,b) along the bus in the first column of G_j. If $a = 0$, the processor broadcasts the packet (0,b). The packet transmitted will provide two binary sequences in two successive iterations as follows.

Iteration 1. Every processor in the first column of G_j writes the value of the first component of the packet (i.e. 0 or 1) in a local register. The resulting binary sequence is then broadcast horizontally through the rows of $\mathcal{R}$. In response to this, every processor sets its local connections as in [22]. Now $P(1,1)$ broadcasts a signal. The processor in the last row of every G_i that receives the signal from its S port marks itself.

Iteration 2. Every processor in the first column of G_j whose row number (in G_j) is no larger than b writes a 1 in a local register. The remaining processors write a 0. The resulting binary sequence is then broadcast horizontally through the rows of $\mathcal{R}$. In response to this, every processor sets its local connections as in [22]. Now $P(1,1)$ broadcasts a signal. The processor in the last row of G_i that receives the signal from its S port marks itself.

It is easy to see that at the end of the two iterations described above the largest item in every G_j knows its final rank in S. In fact, the two iterations above can be executed in parallel, one performed as described, the other performed, at the same time, in a mirror fashion with the last column of G_j as a basis and with the processor $P(1,n)$ sending the signal. Clearly, there is sufficient bandwidth such that the two signals do not meet at any processor. With this simple observation, this subtask requires 5 broadcasts.

The second subtask, namely that of computing the final rank of every item in S is achieved by a simple broadcasting operation. Specifically, the largest item in G_{j-1}, if any, broadcasts its final rank to all the processors in the first column of G_j. It is now a simple matter to compute the final rank of every item.

Finally, in two more broadcasts, every item is routed to the position corresponding to its rank in the first row of $\mathcal{R}$. To see this, note that a generic processor in the first column of G_j will broadcast horizontally a packet consisting of the (at most) two items it stores along with their ranks. The processors in the columns corresponding to these ranks retain the packet and retransmit vertically the item whose rank corresponds to the column number.

Therefore, Step 7 in the sorting algorithm requires 8 broadcasts altogether. We

have proved the following result.
Theorem 5.1. The task of sorting n items stored one per processor in the first row of a reconfigurable mesh of size $n \times n$ can be performed in the time of 35 broadcasts. □

6 Concluding Remarks

Sorting is, unquestionably, one of the fundamental operations in computer science. The main contribution of this work is to present a simple and broadcast-efficient sorting algorithm for reconfigurable meshes.

At the heart of our algorithm is a new and elegant sampling and bucketing scheme. The main feature of our sampling scheme is that the resulting buckets are well balanced, making costly rebalancing unnecessary.

As a byproduct of the inherent simplicity of our sampling and bucketing scheme we show that the resulting sorting algorithm can be implemented using only 35 broadcast operations, a substantial improvement over the algorithm of Nigam and Sahni [20] that requires 59 broadcasts.

As it turns out, our novel sampling and bucketing scheme can be used to obtain a time- and VLSI-optimal sorting algorithm with a reduced number of broadcasts in a much more general setting. Specifically, suppose that one is given m $(1 \leq m \leq n)$ items pretiled in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns of a reconfigurable mesh of size $\sqrt{n} \times \sqrt{n}$. The authors have recently shown that the items can be sorted in row-major (resp. column-major) order in the first $\lceil \frac{m}{\sqrt{n}} \rceil$ columns in $\Theta(\frac{m}{\sqrt{n}})$ time, achieving both the time lower bound for this platform as well as the theoretical VLSI lower bound for sorting m items on a chip of area n.

It would be of interest to see if our technique yields broadcast-efficient algorithms for other fundamental problems as well. Preliminary results show that this is, indeed, the case for several problems in computational geometry and image processing including the tasks of computing the convex hull, the diameter, the width, the smallest enclosing rectangle of a set of points. Constant time algorithms for these problems have been derived before. What we have shown is that the number of broadcasts can be dramatically reduced.

A number of other problems can also benefit from a similar approach. We are looking at selection, multi-selection, and multiple-searching. All these problems are known to have a vast array of applications to query processing.

Acknowledgement:
The authors are indebted to Koji Nakano and three anonymous referees for many useful comments on our deterministic sampling scheme. Their constructive criticism and suggestions have greatly contributed to a more lucid presentation of our results. We would like to thanks Professors Sahni and Dekel for their timely processing of our submission.

References

[1] K. E. Batcher, Design of massively parallel processor, *IEEE Transactions on Computers*, C-29, (1980), 836–840.

[2] Y. Ben-Asher, D. Peleg, R. Ramaswani, and A. Schuster, The power of reconfiguration, *Journal of Parallel and Distributed Computing*, 13, (1991), 139–153.

[3] G. Bleloch, C. E. Leiserson, B. M. Maggs, S. G. Plaxton, S. J. Smith, and M. Zagha, A comparison of sorting algorithms for the Connection Machine CM-2, *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, South Carolina, 1991, 3–16.

[4] Y.-C. Chen and W.-T. Chen, Constant time sorting on reconfigurable meshes, *IEEE Transactions on Computers*, C-43, (1994), 749–751.

[5] R. E. Cypher and C. G. Plaxton, Deterministic sorting in nearly logarithmic time on the hypercube and related computers, *Proc. 22nd Annual ACM Symposium on Theory of Computing*, 1990, 193–203.

[6] J. S. Huang and Y. C. Chow, Parallel sorting and data partitioning by sampling, *Proc. of IEEE Computer Society's Seven International Computer Software and Applications Conference*, 1983, 627–631.

[7] J. Jang and V. K. Prasanna, An optimal sorting algorithm on reconfigurable meshes, *Proc. International Parallel Processing Symposium*, 1992, 130–137.

[8] C. Kaklamanis, D. Krizanc, L. Narayanan, and T. Tsantilas, randomized sorting and selection on mesh-connected processor arrays, *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, South Carolina, 1991, 17–28.

[9] H. Li and M. Maresca, Polymorphic-torus network, *IEEE Transactions on Computers*, 38, (1989), 1345–1351.

[10] H. Li and Q. F. Stout, Eds, *Reconfigurable massively parallel computers*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[11] R. Lin and S. Olariu, A practical constant-time sorting network, *Proc. of International IEEE Conference on Application-Specific Array Processors*, Venice, Italy, October 1993, 380–391.

[12] R. Lin and S. Olariu, Reconfigurable buses with shift switching – concepts and applications, *IEEE Transactions on Parallel and Distributed Systems*, 6, (1995), 93–102.

[13] R. Lin, S. Olariu, J. L. 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.

[14] M. Maresca, Polymorphic processor arrays, *IEEE Transactions on Parallel and Distributed Systems*, 4, (1993), 490–506.

[15] M. Maresca, H. Li, and P. Baglietto, Hardware support for fast reconfigurability in processor arrays, *Proc. International Conference on Parallel Processing*, St. Charles, Illinois, I, 1993, 282–289.

[16] M. S. Merry and J. W. Baker, A constant time sorting algorithm for a three dimensional mesh and reconfigurable network, *Parallel Processing Letters*, in press, 1995.

[17] M. Maresca and H. Li, Connection autonomy in SIMD computers: a VLSI implementation, *Journal of Parallel and Distributed Computing*, 7, (1989), 302–320.

[18] R. Miller, V. K. P. Kumar, D. Reisis, and Q. F. Stout, Parallel computations on reconfigurable meshes, *IEEE Trans. on Computers*, C-42, (1993), 678–692.

[19] K. Nakano, T. Masuzawa, and N. Tokura, A sub-logarithmic time sorting algorithm on a reconfigurable array, *IEICE Transactions*, E-74, (1991), 894–901.

[20] M. Nigam and S. Sahni, Sorting n numbers on $n \times n$ reconfigurable mesh with buses, *Proc. 7th International Parallel Processing Symposium*, April 1993, 174–181.

[21] S. Olariu, J. L. Schwing, and J. Zhang, Fundamental data movement on reconfigurable meshes, *International Journal of High Speed Computing*, 6, (1994) 311–323.

[22] S. Olariu, J. L. Schwing, and J. Zhang, Fundamental algorithms on reconfigurable meshes, *Proc. 29–th Annual Allerton Conference on Communications, Control, and Computing*, 1991, 811–820.

[23] S. Olariu, J. L. Schwing, and J. Zhang, Integer problems on reconfigurable meshes, with applications, *Journal of Computer and Software Engineering*, 1, (1993), 33–45.

[24] S. Rajasekaran and J. H. Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM Journal on Computing*, 18, (1989), 594–607.

[25] J. H. Reif and L. G. Valiant, A logarithmic time sort for linear size networks, *Journal of the ACM*, 34, (1987), 60–76.

[26] J. Rothstein, Bus automata, brains, and mental models, *IEEE Transactions on Systems, Man, and Cybernetics*, 18, (1988), 522–531.

[27] A. Schuster, Doctoral Dissertation, 1991.

[28] D. B. Shu and J. G. Nash, The gated interconnection network for dynamic programming, S. K. Tewsburg *et al.* (Eds.), *Concurrent Computations*, Plenum Publishing, New York, 1988.

[29] B. F. Wang, G.-H. Chen, and F.C. Lin, Constant time sorting on a processing array with a reconfigurable bus system, *Information Processing Letters*, 34, (1990), 187–192.

[30] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, J. G. Nash, and D. B. Sheu, The image understanding architecture, *International Journal of Computer Vision*, 2, (1989), 251–282.